UNIVERSITY OF CALIFORNIA

Los Angeles

Co-optimizing High-Level Synthesis and Physical Design for Rapid Timing Closure of Large-Scale FPGA Designs

A dissertation submitted in partial satisfaction
of the requirements for the degree
Doctor of Philosophy in Computer Science

by

Licheng Guo

2022

ABSTRACT OF THE DISSERTATION

Co-optimizing High-Level Synthesis and Physical Design for Rapid Timing Closure of Large-Scale FPGA Designs

by

Licheng Guo
Doctor of Philosophy in Computer Science
University of California, Los Angeles, 2022
Professor Jingsheng Jason Cong, Chair

High-level synthesis (HLS) tools simplify the FPGA design processes by allowing users to express their designs in high-level languages such as C/C++ or OpenCL. In this way, users could focus on algorithmic optimization with less concern for the cycle-by-cycle details at the register-transfer level (RTL). However, FPGA development flows still have two major limitations that hinder the adoption of FPGAs:

- Limited achievable frequency. There still exists a considerable gap between the quality-of-result (QoR) of an HLS-generated design and what is achievable by an RTL expert, especially the maximum operating frequency of the design. With the designs being scaled up in size, the final achievable frequency will be even lower. Unfortunately, a frequency degradation will directly lead to a proportional performance drop.

- Prolonged compilation time. In the current FPGA CAD flow, the RTL generated by the HLS compiler will be passed to the traditional synthesis and implementation tool. Although the C-to-RTL compilation is relatively quick, the RTL-to-bitstream implementation process will take much longer. With the designs becoming increasingly complex and the FPGA devices larger, the compile time surges from hours to days. Such an overlong process will seriously limit the working efficiency of engineers, especially when compared to software compilation that only takes seconds or minutes.

We observe that the existing FPGA CAD flows have not taken full advantage of HLS for further timing optimization and compilation reduction. Currently, the synthesis, placement, and routing tools are implemented and optimized to handle arbitrary RTL inputs. Those tools will adhere to the cycle-accurate behavior of the input design to ensure the correctness of the output. However, HLS-generated RTL is highly flexible and may tolerate additional pipeline registers without causing functional errors. Such latency-insensitive properties could significantly help the downstream compilation with timing closure. However, in the current toolchains, the HLS compilation is a standalone step, and the HLS-generated RTL will be treated in the same way as manually-written RTL by the logic synthesis tool. As a result, the information on pipeline flexibility in HLS designs will be lost, and the downstream physical implementation process cannot insert pipeline registers for timing closure.

Based on this observation, we propose methods to co-optimize the HLS compilation and the physical design process, which will enable frequency improvement and speed up the hardware accelerator development process simultaneously. Different from the conventional compilation stacks that separate the HLS compilation from the downstream physical implementation process, we propose to bridge the gap between HLS and physical design organically. By facilitating placement and routing with the latency-insensitive information of HLS, and in turn by guiding the HLS compilation with the physical layout information, we could achieve significant improvement in QoR and reduction in compile time.

Centered around this core idea, my thesis consists of three major parts. First, we explore how to improve the inherent timing quality of the RTL generated by HLS. Next, we couple HLS scheduling with coarse-grained floorplanning to improve the achievable frequency. Finally, we take one step further by partitioning the design for parallel placement and routing, then efficiently stitch them together without losing timing quality.

First, the thesis addresses the timing-closure challenge by improving the inherent timing quality of the machine-generated RTL. This chapter studies the timing issues in a diverse set of realistic and complex FPGA HLS designs, including two of my previously-published accelerator designs for genome sequencing. We observe that in almost all cases, the frequency degradation is caused by the broadcast structures generated by the HLS compiler. We

classify three major types of broadcasts and propose a set of effective yet easy-to-implement approaches. Our experimental results show that our methods can improve the maximum frequency of a set of nine representative HLS benchmarks by 53% on average.

In addition to optimizing the QoR of HLS by itself, the thesis further pushes up the final frequency by coupling HLS compilation with floorplanning. We propose AutoBridge, an automated framework that couples a coarse-grained floorplanning step with pipelining during HLS compilation. Since pipelining may introduce additional latency, we further present analysis and algorithms to ensure the added latency will not compromise the overall throughput. In our experiments with a total of 43 design configurations, we improve the average frequency from 147 MHz to 297 MHz (a 102% improvement) with no loss of throughput and a negligible change in resource utilization. Notably, in 16 experiments, we make the originally unroutable designs achieve 274 MHz on average. AutoBridge was recognized with the Best Paper Award in FPGA 2021.

Finally, we take one step further to enable parallel physical implementation on top of our HLS-floorplan co-design methodology. We propose a split compilation approach based on the pipelining flexibility at the HLS level. The pipeline flexibility allows us to partition designs for parallel placement and routing without timing degradation. 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. When tested on the AMD/Xilinx U280 FPGA, we observed a 5-7× compile time reduction and a 1.3× frequency increase. RapidStream was recognized with the Best Paper Award in FPGA 2022.

In conclusion, my thesis targets two of the most challenging problems for modern EDA tools: timing closure and agile compilation. We first study the fanout optimization at the HLS level. Next, we explore the co-optimization of HLS and floorplanning, which has been used by at least eight other accelerator design projects. Finally, we enable the split compilation of HLS designs to reduce the compile time significantly. At the end of the thesis, we discuss future directions, including extending the methodology to support the compilation of RTL designs, multi-FPGA designs, and ASIC designs.

The dissertation of Licheng Guo is approved.

Zhiru Zhang

George Varghese

Anthony John Nowatzki

Jingsheng Jason Cong, Committee Chair

University of California, Los Angeles

2022

This dissertation is dedicated to my family, my teachers, my friends, and all who have helped and inspired me.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

This dissertation would not have been possible without the support, encouragement, advice, and help from my family, teachers, mentors, friends, and all who have inspired me and helped me along the way to get to this point both academically, professionally, and personally.

First and foremost, I would like to thank my advisor and committee chair, Prof. Jason Cong, for his advice, support, and guidance throughout the last 5 years. I am fortunate to have the opportunity to study at and grow with the VAST lab and learn a lot more than just EDA. I would like to thank Jason for allowing me to explore topics I found interesting, for trusting me with highly risky projects, for providing invaluable guidance and advice, for connecting me to domain experts outside the lab, and for his unconditional support and constant availability throughout my time at UCLA. I am grateful to Jason for my fruitful Ph.D. experience like no other.

I would also like to thank Prof. Zhiru Zhang for participating in every one of my projects and providing insightful guidance. Zhiru is always available to discuss my problems and responds quickly when I need help. In addition, Zhiru has helped with my writing significantly. I enjoy discussing with him and learning valuable lessons from him.

I appreciate the advice from Prof. George Varghese and Prof. Tony Nowatzky. I want to thank Prof. George Varghese for teaching me the engineering principles that I will always remember. I learned important lessons about computer architecture and domain-specific accelerators from Prof. Tony Nowatzki. George and Tony have brought up invaluable suggestions on extending my research projects to broader applications; I feel lucky to have them on my committee.

I want to thank Pongstorn Maidee, Chris Lavin, Eddie Hung, and Alireza Kaviani for their crucial support for the RapidStream project. The RapidStream project will never succeed without their devotion and support.

I would also like to thank all of my collaborators, without whom our projects would

never have succeeded. First, I want to thank Jason Lau, Yuze Chi, Weikang Qiao, Jie Wang, Linghao Song, Yun Zhou, Peng Wei, Zhenyuan Ruan, Tianhe Yu, and Cody Yu, for working together with me. I want to thank Prof. Zhenman Fang, Prof. Peipei Zhou, Prof. Young-Kyu Choi, Prof. Po-Tsang Huang, Prof. Luciano Lavagno, Prof. David Pan, Jianyi Cheng, Atefeh Sohrabizadeh, Karl Marrett, Danial Tan, Suhail Basalama, Stephane Pouget, Chengdi Cao, Neha Prakriya, Jason Kimko, Lorenzo Ferretti, Di Wu, DJ Wang, Gai Liu, Wuxi Li, Zhengrong Wang, Sihao Liu, Jian Weng, Yuanlong Xiao, Xingyu Tian, Alec Lu, Moazin Khatti, Shaojie Xiang, Yi-Hsiang Lai, Ecenur Ustun, Wenqi Cao, Andreas Nowatzyk, Zeyu Kuang, Yue Zhong, Guangcan Li, Haowen Chen, Wenping Wang, Sida Peng, Zixuan Jiang, Pengfei Li, Qiuxiao Chen, Feiyu Chen, Junpei Zhou and many others for their help, guidance and suggestions.

I deeply appreciate the efforts by Jason Lau and Yuze Chi to maintain our server system and voluntarily provide technical support. I want to thank Alexandra Luong and Joseph Brown for all the help on the administrative side.

I am fortunate that my Ph.D. journey turns out to be the best years of my life. Although I was half a world away from home, I have never felt lonely, and I could not be more grateful for my friends. I want to thank Bing Han, Jason Lau, Weikang Qiao, Jie Wang, Zhengrong Wang, and many others for being an irreplaceable part of my Ph.D. journey.

Besides my academic advisors, I also want to thank all my teachers and friends since I was young. Specifically, I want to thank Prof. Peiyong Zhang and Prof. Zheng Shi at Zhejiang University, who led me into the fascinating world of EDA and FPGA. I want to thank my flute teacher, Erika Andres, for helping me find great joy in music through her professional and inspiring lessons. I want to thank my middle school math teacher, Yibo Ji, who helped me build confidence and find joy in learning. I want to thank my middle school Chinese teacher, Qin Li, who played an important role in my cultivating a positive outlook on the world, life, and values. She also taught me important lessons on academic writing that I still find helpful to this day. I want to thank Wei Xie, who was always there to answer my questions and discuss tricky puzzles with me for the entire middle school and high school. I want to thank Feng Xiao, for his encouragement and support, especially in my most difficult

days before and after the college entrance exam. I want to thank Ding Zhang for his support and accompany throughout the years. I want to thank Yike Li, we have worked side by side on many of the most important projects, and I have learned a lot from him; I always feel that he is like-minded, and we had much fun together. I want to thank Yifan Yuan, Yifan is one of my most visionary peers, and I appreciate all his advice and guidance. I want to thank Lijun Wang, his perseverance and optimism toward life have inspired me a lot.

I want to thank UCLA and Zhejiang University for providing a free, encouraging, tolerant, vibrant, warm, and supportive environment for their students. I want to thank the Advanced Class of Engineering Education (ACEE) at Zhejiang University, I am honored to be a part of the group, and it has broadened my perspective and connected me to so many good friends.

Most importantly, I thank my parents, Yingtong Xiong and Ling Guo, and my grandparents, Xiaojing Guo, Lamei Zhu, Zhongyin Xiong, and Aifen Ni, for all their extremely hard work that made me who I am, for always being understanding and supportive no matter the situation, and for creating an environment of love, warmth, tolerance, trust, and equality at home. I deeply thank my wife, Xinyi Li, for always being supportive and understanding and for all the love and happiness that we have shared together.

The contents of Chapter 3 is from a collaboration between Jason Lau and me. We made equal contributions to our DAC '20 paper [GLC20], which is the basis of Chapter 3. Specifically, the problem classification part (Section 3.2) is accredited to me, and the solution part (Section 3.3) is accredited to Jason Lau.

This thesis is partially supported by CRISP, one of the six centers in JUMP, a Semiconductor Research Corporation (SRC) program, member companies under the Center for Domain-Specific Computing (CDSC) Industrial Partnership Program, the Intel/NSF CAPA program, the NSF NeuroNex Award No. DBI-1707408 and the NIH Award No. U01MH117079. The authors acknowledge the valuable support of the Xilinx Adaptive Compute Clusters (XACC) Program. We thank Gurobi and GNU Parallel for their support to academia.

VITA

2014–2018 B.S., Electronic Engineering,
Zhejiang University, Hangzhou, China.

2018–2021 Master, Computer Science,
University of California, Los Angeles, U.S.A.

PUBLICATIONS

Licheng Guo, Pongstorn Maidee, Yun Zhou, Chris Lavin, Jie Wang, Yuze Chi, Weikang Qiao, Alireza Kaviani, Zhiru Zhang, Jason Cong, RapidStream: Parallel Physical Implementation of FPGA HLS Designs, *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA 2022)*

Yuze Chi, Licheng Guo, Jason Cong, RapidStream: Accelerating SSSP for Power-Law Graphs, *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA 2022)*

Linghao Song, Yuze Chi, Licheng Guo, Jason Cong, Serpens: a high bandwidth memory based accelerator for general-purpose sparse matrix-vector multiplication, *Proceedings of the 59th ACM/IEEE Design Automation Conference (DAC 2022)*

Licheng Guo, Yuze Chi, Jie Wang, Jason Lau, Weikang Qiao, Ecenur Ustun, Zhiru Zhang, Jason Cong, AutoBridge: Coupling Coarse-Grained Floorplanning and Pipelining for High-Frequency HLS Design on Multi-Die FPGAs, *Proceedings of the 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA 2021)*

Jie Wang, Licheng Guo, Jason Cong, AutoSA: A Polyhedral Compiler for High-Performance Systolic Arrays on FPGA, *Proceedings of the 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA 2021)*

Yuze Chi, Licheng Guo, Young-kyu Choi, Jie Wang, Jason Cong, Extending High-Level Synthesis for Task-Parallel Programs, *Proceedings of the IEEE 2021 Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM 2021)*

Weikang Qiao, Jihun Oh, Licheng Guo, Mau-Chung Frank Chang, Jason Cong, FANS: FPGA Accelerated Near-Storage Sorting Solution, *Proceedings of the IEEE 2021 Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM 2021)*

Licheng Guo, Jason Lau, Yuze Chi, Jie Wang, Cody Hao Yu, Zhe Chen, Zhiru Zhang, Jason Cong, *Proceedings of the 2020 Annual Design Automation Conference (DAC 2020)*

Licheng Guo, Jason Lau, Zhenyuan Ruan, Peng Wei, Jason Cong, Hardware Acceleration of Long Read Pairwise Overlapping in Genome Sequencing: A Race Between GPU And FPGA, *Proceedings of the IEEE 2019 Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM 2019)*

Jason Cong, Licheng Guo, Po-Tsang Huang, Peng Wei, Tianhe Yu, SMEM++: Pipelined and Time-Mux SMEM Seeding Accelerator for Genome Sequencing, *Proceedings of the IEEE 28th International Conference on Field Programmable Logic and Applications (FPL 2018)*

CHAPTER 1

Introduction

The slowdown in transistor scaling and the end of Moore's law are pushing us to new computing paradigms. Compared to general computing, specialized hardware devices, such as Field Programmable Gate Arrays (FPGAs), provide a promising solution to achieve intensive computation and energy efficiency [CWA22]. FPGA is a type of reconfigurable integrated circuit designed to be configured by designers. Because of their lower non-recurring engineering costs, reconfigurability, and short time-to-market, FPGAs are becoming increasingly attractive. Beyond the success in traditional applications like fast prototyping for application-specific integrated circuits (ASIC), FPGAs have also demonstrated their applicability as hardware accelerators in modern applications, such as machine learning [FOP18, WLC19, WGC21], genome sequencing [CGH18, GLR19], compression [QDF18, QFC19] and network processing [CCP16, FSP20] and many other fields.

The advancement of FPGA-based design methodologies is inseparable from the support of FPGA computer-aided design (CAD). A typical modern FPGA CAD flow is illustrated in Figure 1.1. Users first describe the architecture of their designs in certain software languages (e.g., OpenCL and C/C++). After that, the high-level synthesis (HLS) tool compiles the program into a cycle-accurate representation in a hardware description language (e.g., Verilog and VHDL). The logic synthesis and the technology mapping step further translate the architectural design into a gate-level netlist. Given the results of the logic design steps, the physical design process will map these abstract logical representations onto physical function units and interconnects on the FPGA fabric. The placement step determines the physical locations of each operator, and the routing step computes the physical interconnects among the function units.

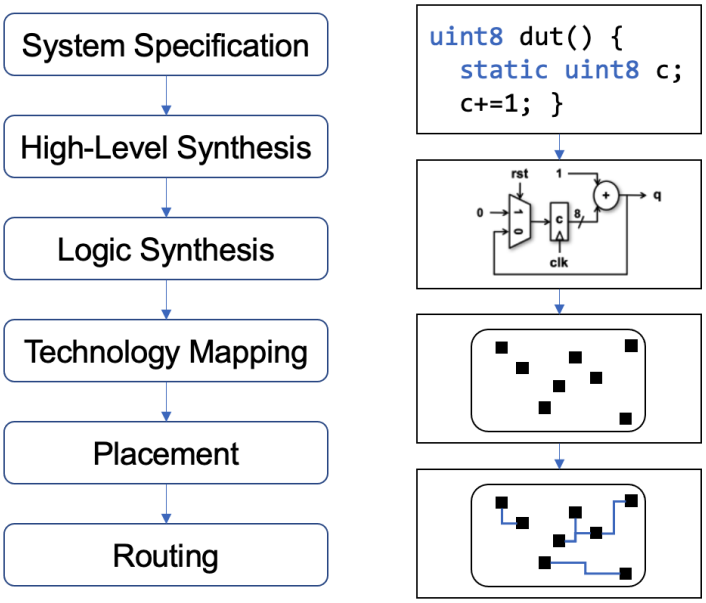


Figure 1.1: A typical FPGA CAD flow.

While RTL design methodology used to be the dominant choice, HLS has emerged as an increasingly popular and promising alternative to cope with the design productivity gap as the complexity of applications and hardware platforms continues to scale up. The key advantage of HLS is that the compiler will automatically convert the *untimed* high-level software programs into cycle-accurate RTL hardware implementations because the process of manual RTL optimization is becoming unequivocally difficult. By raising the level of abstraction from cycle-accurate hardware to untimed software, HLS reduces design effort while optimizing over a larger solution space regarding performance, area, and timing. Such benefits have led to growing development and adoption of both commercial and open-source HLS tools, including AMD/Xilinx Vitis HLS [Xil20b], LegUp HLS [HLS20a], Intel OpenCL [Exa20], Mentor Catapult HLS [HLS20b], and Cadence Stratus HLS [Cad20].

1.1 Current Challenges in FPGA CAD Flow

Despite the increasing adoption of HLS because of its design productivity advantage, the overall FPGA CAD flows still have unsatisfactory quality-of-results (QoR) and productivity. Such limitations continue to curb the adoption of FPGA accelerators.

- First, it is challenging to achieve a satisfactory frequency for HLS designs. The downside of a higher abstract level is the lack of control over low-level details. Regarding the frequency bottleneck, current HLS tools work like a black box when generating RTL from the user input. Unfortunately, the compiler does not provide helpful feedback or guidelines on improving the clock frequency at the source level or using additional tool options. It is also challenging for regular HLS users to reverse engineer the synthesized RTL code to identify the timing bottlenecks and optimize the corresponding critical paths in the source program.

- Second, the long end-to-end compilation still takes significant time and hinders productivity. While it only takes a few minutes for the HLS compilation, the generated RTL has to be synthesized into a physical netlist, then goes through the physical design process, including placement and routing. With the ever-increasing size of FPGA designs, the scalability issue of the physical design process becomes more serious, which results in tens of hours or even days of compilation.

Figure 1.2 shows a motivating example that exposes the limitation of the current FPGA CAD stack. We measure the compilation time and final frequency of a set of systolic array designs of varying sizes. Each processing element of the systolic array is the same. Although this type of design is highly regular in shape, the final frequency is still relatively low. Worse, scaling up the array size leads to a significant increase in compilation time.

1.2 Thesis Overview

This thesis targets two of the most challenging tasks for modern EDA tools: improving the timing quality and reducing the compile time. While these two goals are orthogonal and potentially conflicting[1] to some extent, we manage to address both of them using the same methodology.

The core idea throughout the thesis is to bridge the gap between physical design steps (e.g.,

[1]Generally, spending a longer time on more optimization iterations may lead to better QoR and vice versa.

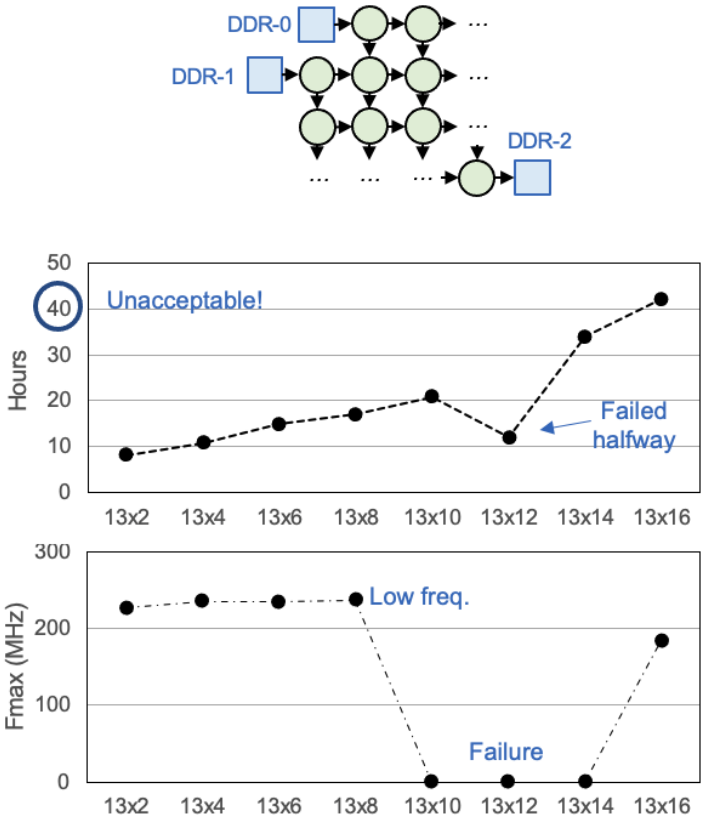


Figure 1.2: A motivating example showing the compile time and the achievable frequency of a set of systolic array designs.

global floorplanning and global routing) and the HLS compilation steps (e.g., scheduling and binding). Since HLS compiles the untimed source code into the cycle-accurate RTL code, it can introduce additional pipelining registers without breaking the overall functionality. With the help of HLS, we can rely less on manual timing optimization, where the designer must carefully adjust the surrounding logic of inserted pipeline registers, which is an error-prone and ad-hoc process.

Built on this important idea, we propose effective and scalable methods to facilitate timing closure and parallelize the physical design process. We start by optimizing the QoR of the code generation of HLS by itself. This step improves the timing quality of the generated RTL regardless of the underlying FPGA fabric. Next, we break the abstraction boundaries by coupling HLS scheduling with floorplanning. This combination significantly helps the placer determine a much better overall distribution of the logic by utilizing the high-level

topology of the design; in turn, it helps HLS to identify and pipeline the global interconnects accurately instead of relying on inaccurate modeling. To further extend the idea, we partition an input design at latency-insensitive boundaries to enable parallel placement and routing, dramatically reducing the end-to-end compilation time. Specifically, we made the following contribution.

1.2.1 Broadcast-Aware Optimization of HLS Code Generation.

This part of the thesis improves the QoR of the HLS code generation by itself (without interaction with other steps in the FPGA CAD flow). We study the timing issues in a diverse set of realistic and complex FPGA HLS designs and observe that in almost all cases, the frequency degradation is caused by the *broadcast* structures generated by the HLS compiler. We classify three major types of broadcasts in HLS-generated designs, including high-fanout data signals, pipeline flow control signals, and synchronization signals for concurrent modules. We further reveal several limitations of the current HLS tools that result in those broadcast-related timing issues. Finally, we propose a set of effective yet easy-to-implement approaches, including broadcast-aware scheduling, synchronization pruning, and skid-buffer-based flow control. Our experimental results show that our methods can improve the maximum frequency of a set of nine representative HLS benchmarks by 53% on average. In some cases, the frequency gain is more than 100 MHz.

1.2.2 Coupling Global Floorplanning with HLS Pipelining.

This part of the thesis shifts the global floorplanning step into the HLS compilation to improve the final QoR. First, our approach provides HLS with a view of the global physical layout of the design, allowing HLS to more easily identify and pipeline the long wires, especially those crossing the die boundaries. Second, by exploiting the flexibility of HLS pipelining, the floorplanner can distribute the design logic across multiple dies on the FPGA device without degrading clock frequency. This prevents the placer from aggressively packing the logic on a single die which often results in local routing congestion that eventually degrades timing.

Since pipelining may introduce additional latency, we further present analysis and algorithms to ensure the added latency will not compromise the overall throughput.

Our framework, AutoBridge, can be integrated into the existing CAD flow for Xilinx FPGAs. In our experiments with a total of 43 design configurations, we improve the average frequency from 147 MHz to 297 MHz (a 102% improvement) with no loss of throughput and a negligible change in resource utilization. Notably, in 16 experiments, we make the originally unroutable designs achieve 274 MHz on average. AutoBridge was recognized with the Best Paper Award in FPGA 2022.

1.2.3 HLS-Based Partitioning and Stitching Methodology for Parallel Physical Design

The aforementioned idea of combining HLS compilation with global floorplanning could be further extended to support parallel placement and routing. Conventionally, a major challenge in separately implementing different design parts is achieving a good timing quality on boundary nets that span different partitions. This problem could be addressed by utilizing the pipeline flexibility of HLS to fix the critical paths on the partition boundaries while maintaining the overall functionality and the throughput. In this way, the design can be partitioned at latency-insensitive boundaries into decoupled sub-designs, which could be separately placed and routed in parallel with minimal synchronization. The partial bitstream of all sub-designs will be later stitched together, and the tool will add additional pipeline registers to boundary nets for timing closure. In the thesis, we outline a number of technical challenges of such a split compilation approach. To address the challenges, we break the conventional boundaries between different stages in a typical CAD flow, and we reorganize the steps 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 5-7× reduction in compile time and up to 1.3× increase in frequency when compared to 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 5-7× compile time reduction and 1.3× frequency increase. RapidStream was recognized with the Best Paper Award in FPGA 2022.

CHAPTER 2

Background

As the FPGA architecture evolves and its complexity increases, CAD software has advanced significantly as well. Nowadays, FPGA vendors provide a comprehensive set of design tools that allow automatic synthesis and compilation from design specifications in hardware specification languages all the way down to a bitstream to program FPGA chips. Fig 2.1 shows a typical FPGA design flow.

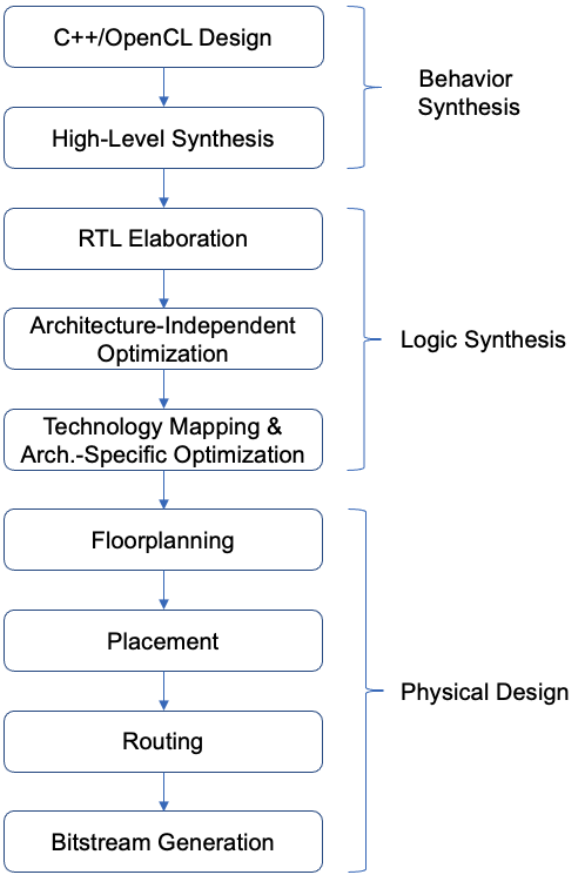


Figure 2.1: A typical FPGA design flow starting from behavior-level specifications.

2.1 High-Level Synthesis

The rapid increase of complexity in FPGA design has pushed the industry and academia to raise the design abstractions with better productivity than register transfer level (RTL). High-level synthesis (HLS) plays a crucial role by enabling the automatic compilation of high-level, untimed, or partially timed specifications (e.g., C/C++ or OpenCL) to low-level cycle-accurate RTL specifications for efficient implementation in field programmable gate arrays (FPGAs) or application-specific integrated circuits (ASICs). This synthesis can be optimized considering the performance, power, and cost requirements of a particular system [CLN11]. The typical flow of modern FPGA HLS tools usually consists of three core steps: (1) scheduling, (2) binding, and (3) RTL generation, as shown in Figure 2.2.

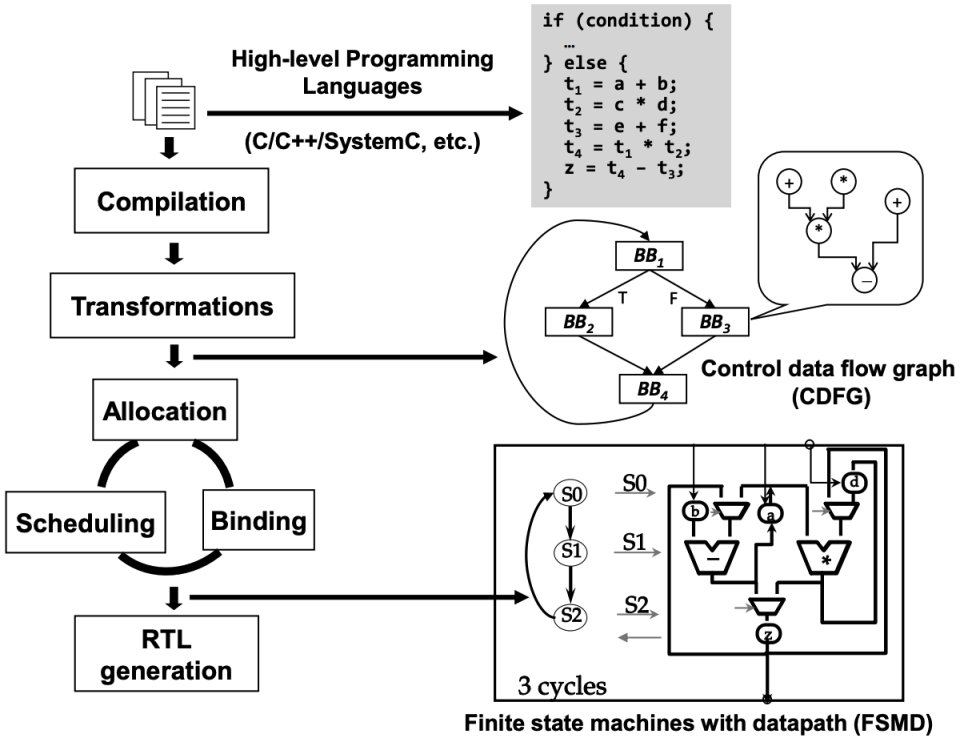


Figure 2.2: A typical FPGA HLS flow. [CLN11]

- The *scheduling* phase inserts clock boundaries into the original untimed specification. It takes in the control data flow graph (CDFG) generated by a compiler front-end from the high-level description, for example, C/C++, and then maps the operations in the

CDFG to the states and the control flow to state transitions specified by a finite-state machine (FSM). In each clock cycle, the controller will be in a state in the corresponding FSM.

- The *binding* process maps high-level operations and variables to RTL-level resources, such as functional units and registers. It maps variables to registers and links wires from registers to functional units as operands of operations. The result of a functional unit is then wired to another functional unit or a register to store the computed value.

- The *RTL generation* phase creates concrete RTL based on the results of the scheduling and the binding step. The key in this step is to properly create the control logic to orchestrate the datapath, controlling each stage to execute at its scheduled cycle.

One general limitation of today's HLS is that the HLS scheduler relies on a high-level estimation of the operation delay. This estimation is usually based on pre-characterized statistics, which include the delay of common components for computing, storage, and interconnects (e.g., adders, multipliers, registers, BRAMs, multiplexers, etc). However, such a delay model is inherently inaccurate. HLS tools lack consideration of the additional net delay in broadcast structures. The predicted delay by HLS tools for a certain operator is fixed regardless of the actual environment. Thus, the actual value for operators near broadcasts is usually larger than the predicted value. The situation is worse for global signals because HLS has no layout information on how the designs will be floorplanned and placed, thus it can hardly determine which signals will span a long distance.

2.2 Logic Synthesis

Logic synthesis is the process of transforming an RTL-specified design into a gate-level representation [Rud89, Sas93, DGK94]. The development and maturity of logic synthesis came before that of HLS, and RTL-based development used to be the dominant way and is still irreplaceable in many situations today. Logic synthesis is a highly automated procedure bridging the gap between high-level synthesis and the physical design process. Given an

RTL digital design, logic synthesis transforms it into a gate-level or transistor-level network. This process explores different potential mappings of a logic function optimally under certain desired design constraints. After logic synthesis, the physical positions of primitive elements and the shapes and positions of the interconnections of the gate layouts are then further determined at the time of physical design.

Given an RTL design, a set of design constraints, and the target FPGA device, the overall FPGA synthesis process goes through the following steps:

- RTL elaboration. This step identifies and/or infers datapath operations (e.g. additions, multiplications, floating point arithmetic operations, register files, memory blocks, etc) and the corresponding control logic. Then they are elaborated into a set of finite-state machines or generic Boolean networks. Since most of the datapath elements have dedicated architectural support in modern FPGAs, it is important to recognize those elements such as adders with dedicated fast-carry chains and embedded DSP.

- Architecture-independent optimization. This step includes both datapath optimization (e.g., constant propagation, strength reduction, operation sharing, expression optimization, etc.) and control optimization. The control logic optimization includes sequential optimization such as finite-state machine encoding/minimization and retiming, and combinational logic optimization such as constant propagation, redundancy removal, logic network restructuring, etc.

- Technology mapping and architecture-specific optimization. This step maps the previously generated control logic and datapath to physical units of the target FPGA device. Specifically, this step maps the optimized datapath to on-chip dedicated circuit structures, such as DSPs, adders with dedicated carry-chains, embedded block RAMs, or the basic LUTs. It maps the optimized control logic to LUTs and registers.

2.3 Placement

The task of placement is to assign exact locations for each functional unit in the netlist to a physical component within the FPGA layout [LJG20, SM91, CCS05]. To a great extent, placement determines the overall quality of the final FPGA implementation. An inferior placement solution can hinder the downstream routing step by producing excessive wire length, which not only hampers the design performance and may even cause routing failure. Modern placement tools optimize toward various objectives, typically including wire length, routability, timing, and power. Minimizing the total wire length is often a major optimization goal because it is a reasonable approximation of routability, timing, and power. In addition to minimizing the wire length, another important factor to consider is the local routing feasibility. The placement tool must predict and distribute routing demand to avoid excessive local routing congestion. To meet a given frequency target, the placement tool has to ensure the total signal delay of the critical path is no greater than the according delay requirement.

Given the complexity of modern large-scale designs and the NP-hard nature of various algorithms involved, the FPGA placement problem is usually decomposed into several easier sub-problems, including global placement, packing, legalization, and detailed placement. Global placement targets producing a coarse-grained placement solution while globally optimizing the aforementioned placement objectives. Packing aims at grouping LUTs and FFs into architectural-legal and placement-friendly CLBs. Legalization produces a feasible placement solution by locally perturbing the result of global placement and packing. Detailed placement further conducts local refinement while maintaining the problem's feasibility.

2.4 Routing

Given a placed netlist where the physical location of each element in the netlist has been determined, the routing problem is to assign each net to the physical wire segments and switches such that all nets are connected while an objective function is optimized. Typical objectives for routing include total wirelength and segments [ZML21, CZ05, AR95].

Since modern FPGAs have limited and discrete interconnects and switch boxes, the routing step is a complex combinatorial optimization problem. Modern routing algorithms often abstract an FPGA as a weighted graph whose topology can represent the routing architecture of the target FPGA fabric. Each edge is associated with a weight that corresponds to the congestion level. During the routing process, the router will dynamically update the edge weights based on the available resource and applies graph-search techniques to look for desired connections.

Most FPGA routing algorithms will separate the routing task into two stages: the global routing stage and the detailed routing stage. First, global routing splits the available routing area into coarse-grained routing regions or channels to reduce the problem size. Then the router determines the coarse routing topology of each net regarding which routing regions the net is assigned to. The global routing step typically has the objectives of minimizing and balancing the overall estimated congestion and satisfying the timing constraints of critical nets. Next, the detailed routing stage produces the exact routing geometry to map every net onto each individual routing channel or region.

Due to the NP-hard nature of the routing problem, such a hierarchical approach has the advantage of being more computationally scalable. Otherwise, it will be extremely complex to determine the exact routing details directly in one step, considering that there are millions of nets and the design sizes have been increasing explosively in the last decade. However, the downside of such a two-step approach is the potential disconnection between the two stages. Since the global router relies on an estimated model for the distribution of available routing resources, the model's accuracy is critical. However, the early stage estimation often lacks consideration of the details of routing obstacles, and the final timing quality may be degraded.

2.5 Related Works

There are two major goals in our proposal: improving the achievable frequency and reducing the compilation time. Correspondingly, we present the previous works related to the two

topics.

2.5.1 Physical-Aware HLS Timing Optimization

- Co-optimize HLS and logic synthesis. Zhao *et al.* [ZTD15] and Tan *et al.* [TDG15] show that HLS typically has an over-conservative prediction of the signal delay of logic operations (e.g., `AND`, `OR`, `NOT`, etc). As a result, they propose methods to improve HLS scheduling by considering the technology mapping process.

- Co-optimize HLS and physical design. Zheng *et al.* [ZGR14] propose to iteratively run placement and routing for fine-grained calibration of the delay estimation of wires. The long runtime of placement and routing prohibits their methods from benefiting large-scale designs, and their experiments are all based on small examples (1000s of registers and 10s of DSPs in their experiments). Cong *et. al.* [CFH04] presented placement-driven scheduling and binding for multi-cycle communications in an island-style reconfigurable architecture. Xu *et. al.* [XK97] proposed to predict a register-level floorplan to facilitate the binding process. Some commercial HLS tools [Cad20, Syn20] have utilized the results of logic synthesis to calibrate HLS delay estimation, but they do not consider the interconnect delays.

However, the major issue with the previous works is scalability. The previous approaches share the common aspect of focusing on the fine-grained interaction between HLS and physical design, where individual operators and the associated wires and registers are all involved during the delay prediction and iterative HLS-layout co-optimization. While such a fine-grained method can be effective on relatively small HLS designs and FPGA devices, it is too expensive (if not infeasible) for today's large designs targeting multi-die FPGAs, where each implementation iteration from HLS to bitstream may take days to complete.

2.5.2 Physical-Independent HLS Timing Optimization

Cong *et al.* [CWY18b] propose to pipeline the data transfer logic from the external interfaces to the many individual processing elements. As we will introduce later in Chapter 3, this is a special case of *data broadcast*. They attempted to alleviate the critical path by accessing large buffers, which may be mapped to scattered BRAM units. However, they require explicit user intervention and iterative tuning to explore the best topology. They do not consider the ultimate limitation of the HLS. Moreover, they can only re-arrange the data interconnect between the external port and each explicitly-defined processing element, but not fine-grained datapath. Lau *et al.* [LSZ20] propose a new dynamic invariant analysis and automated refactoring technique, which reduces BRAM by 83% and increases frequency by 42% for recursive programs. The techniques used by HeteroRefactor have inspired our work, and its source-to-source transformation has motivated the implementation of the data broadcast optimization in Chapter 3.

2.5.3 Accelerating the FPGA CAD Flow

Overlay-Based Compilation Flow. One major direction to speed-up FPGA compilation is to implement an overlay on top of the FPGA device and compile the source design onto the overlay [PXM18, XAD20, XPB19, MAA16, WS19, XMB22, XD22, PXD22]. Usually, an overlay divides the FPGA fabric into disjoint slots, which are connected through dedicated interconnect logic, such as a Network-on-Chip (NoC) [PH12, HD12, KMD06, KG17, Kap17, VGK17]. Since each slot only communicates with the NoC, the slots could be implemented in parallel. However, such a method introduces nontrivial area overhead (due to the dedicated interconnect logic and resource under-utilization in each slot), performance degradation (limited bandwidth of the interconnect logic), and may require human intervention to modify the design to fit the overlay (since each slot has fixed size).

Specifically, [PXM18, XPB19] proposed to manually decompose a set of HLS designs into small, separate sub-parts for parallel implementation and used a packet-switched NoC to connect them. They experience performance degradation of as much as 12.5× with an area

overhead of as much as 5×. [XAD20] further replaces the NoC by direct wire connection, reducing the area overhead and throughput degradation. However, they have to design the overlay structure based on the target designs manually, make nontrivial modifications to the designs to fit into the overlay, and manually assign each sub-part to specific slots. In addition, [MAA16] provides an overlay that connects all the reconfigurable tiles similar to the switch boxes, but it only routes word-wide data. [WS19] provided an FPGA overlay consisting of HLS-generated circuits, an execution manager, and a soft processor function unit. The execution manager can move some functionality from HLS-generated RTL circuits to the soft-processor incrementally without hardware re-compilation but at the cost of performance degradation and significant area overhead. In comparison, we have negligible area overhead or performance degradation as we use direct wire connection, and we achieve one-click automation for users. Instead of modifying the design to suit the size of the slots, we choose the proper slot sizes according to the given design. We propose methods to automatically partition the design into suitable sub-parts and floorplan them accordingly.

[LPL11] present a CAD flow that allows users to reuse pre-implemented hard macros, but the reuse flow is purely manual. Instead, we propose methods to automatically identify repetitive patterns in the given design and reuse the placement and routing results.

Acceleration of Physical Design Algorithms. Previous efforts have attempted to parallelize the numerical optimization process of placement [LBP08, CZ09, LCW15, LLW17, LJG20, ASB14] and routing [Sto17, SL15, GA10, GA11]. However, as these algorithms are generally serial, only a limited degree of parallelism can be exploited. There is frequent synchronization between the job workers, which limits the speed-up. In addition, heuristics to parallelize the original serial algorithm often come with compromised quality of results (QoR). As a result, the placement and routing steps remain the bottleneck.

Customizable FPGA CAD Flow FPGA tools, such as Vivado from AMD/Xilinx and Quartus Prime from Intel/Altera, have become highly complex software systems that must perform the entire CAD flow for all devices in the vendor's product portfolio. However, the necessity of breadth coverage and priority to generality by commercial tools often make the

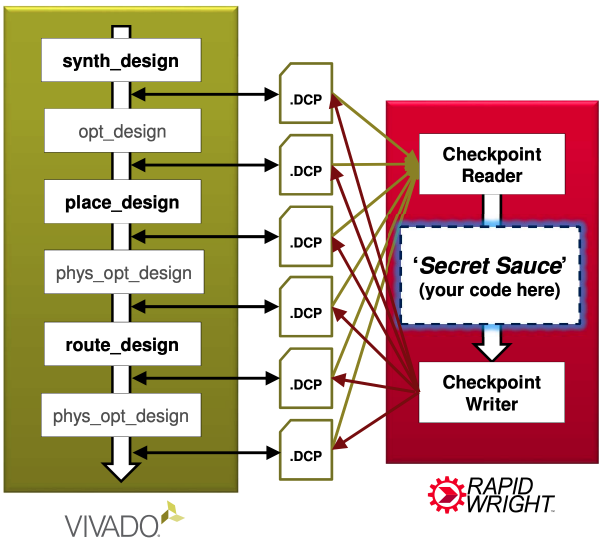


Figure 2.3: Overview of how RapidWright and Vivado interacts [LK18].

commercial tools less suitable for domain-specific situations. For example, Vivado performs poorly in partial placement or partial routing problems. Even if the majority of the cells in the design have already been placed, Vivado still takes a long time for initialization and for optimization iterations over the already fixed cells.

To address this issue, AMD/Xilinx rolls out the RapidWright framework [LK18], an open-source platform that provides a gateway to Vivado's back-end implementation tools (see Figure 2.3) to realize the full potential of advanced FPGA silicon. RapidWright works synergistically with Vivado to produce highly tuned, custom implementations for emerging applications. It can read the checkpoint of a Vivado project at any stage and correctly expose the logical and physical netlist to users for customized processing. Then RapidWright could generate another checkpoint that includes the user modification, which can be seamlessly inserted back into the Vivado tool for further optimization.

2.5.4 Latency Insensitive Design

Latency Insensitive Design (LID) [CMS01, CS00] has been proposed as a design methodology where design components are made insensitive to the latency of the communication logic among them. LID enables the flexible pipelining of communication channels between computation

components while ensuring that the correctness of the design will be preserved. Doing so allows the designer to add adequate pipeline stages between components to meet the desired clock frequency without breaking the functionality of the design.

The LID methodology has been used as a design methodology for designers so that they could easily fix critical paths or port the accelerator design to different FPGAs [AB18, CSN14, HKK16, MOD20, KVC18, XMB22]. For example, Abbas et al. [AB18] present a modified LID solution suitable for modern FPGAs. Chacko et al. [CSN14] develops a latency-insensitive hardware accelerator of the physical layer for cognitive wireless communication systems that use orthogonal frequency division multiplexing (OFDM) schemes. Hofmann et al. [HKK16] present a highly parametrized FPGA implementation of Semi-Global Matching (SGM). By using a latency-insensitive design style and high-level synthesis, an automated design-space exploration flow can effectively examine many implementation alternatives with high productivity. However, there is limited research on how to adopt the LID methodology into the EDA tools and automate the pipeline insertion for timing optimization.

One challenge of automatic pipelining at latency-insensitive links is that the tool needs to derive the high-level meaning of certain modules, such as FIFOs. Due to the difficulty in extracting high-level semantics from an RTL program, existing tools take a conservative approach and strictly preserve the cycle-accurate specification. Given an RTL input, it remains an open question of how to automatically differentiate the cycle-by-cycle timing behavior of RTL from the functional behavior of a design. Fleming et al. [FAP12] proposed to introduce a special primitive to the programming language that represents a latency-insensitive unit, and then the compiler could automatically synthesize the actual logic. However, no timing optimization based on pipelining was proposed in their work.

As will be shown later, in this thesis, we propose to obtain the latency-insensitive information at the HLS abstract level, where we can easily identify interfaces that can tolerate extra latency. More importantly, we utilize such flexibility to automatically fix the potential critical paths to improve the achievable frequency by 2X over a set of benchmarks. In our future work section, we will also discuss methods to extend our tool to RTL designs.

2.5.5 Other Related Works

In this section, we present some other general topics that are related to the proposal. First, since we will discuss methods to reduce the broadcast structures generated by HLS in Chapter 3, we include the previous works on RTL-based fanout optimization techniques. Second, as we will present the HLS-level floorplanning method in Chapter 4, we include some previous works on floorplanning. Finally, we present related works on latency-insensitive designs, which are related to our proposal to additionally pipeline the global interconnects.

- **General high-fanout optimization.** fanout optimization has been extensively studied in logic-synthesis [PB91, HKP84, SS90] and physical design [OC97, BL03, WMP03, Wea08]. However, optimization approaches at these levels are restricted by the cycle-accurate timing specification of the RTL input. For example, they cannot arbitrarily divide the broadcast delay into two or more clock cycles; retiming [Wea08, VHB] will not work without enough registers on the path, etc. In contrast, eliminating high fan-out structures at the behavior level is more effective as we can change the schedule of the broadcast. Even though the original designs are implemented with modern backend broadcast optimization, our behavior-level optimizations still bring huge frequency gain.

- **Floorplanning Algorithms.** Floorplanning has been extensively studied [AMS08, CW06, BSB09, MB15]. Conventionally, floorplanning consists of 1) feasible topology generation and 2) determining the aspect ratios for goals such as minimal total wire length. In Chapter 4, we propose to perform coarse-grained floorplanning during the HLS step to help gain layout information for the HLS tool. Similar to [Lau88, LD86, MK87], our algorithm adopts the idea of the partitioning-based approach. As our problem size is relatively small, so we use ILP for each partitioning.

- **Throughput Analysis of Dataflow Designs.** In Chapter 4, we will discuss how to preserve throughput after pipelining the global data transfer logic. This is related to the study of throughput analysis of dataflow designs. Various dataflow models have been proposed in other literature, such as the Kahn Process Network (KPN) [Gil74],

Synchronous Data Flow (SDF) [LM87], among many others. The more simplified the model is, the more accurately we can analyze its throughput. In the SDF model, it is restricted that the number of data produced or consumed by a process for each firing is fixed and known. Therefore, it is possible to analytically compute the influence of additional latency on throughput [GGS06]. The latency insensitive theory (LIT) [CS00, LK03, LK06, CC07, AB18] also enforces similar restrictions as SDF. [VG14] proposes methods to insert delays when composing IP blocks of different latency. [JSG20] studies the buffer placement problem in dataflow circuits [JGI18, CJC20]. In our situation, each function will be compiled into an FSM that can be arbitrarily complex, thus it is difficult to quantitatively analyze the effect of the added latency on the total execution cycles. Therefore, we adopt a conservative approach to balance the added latency on all reconvergent paths. Other works have studied how to map dataflow programs to domain-specific coarse-grained reconfigurable architectures [WLD19, WLK22, DLN22, LWV21]

CHAPTER 3

Analysis and Optimization of the Implicitly Broadcasts by HLS

High-level synthesis (HLS) tools simplify the process of implementing new applications on FPGAs. However, there still exists considerable room to the timing qualities of the HLS-synthesized designs. Unfortunately, current HLS tools do not provide helpful feedback or guidelines on how to improve the clock frequency at the source level or use additional tool options. It is also challenging for regular HLS users to reverse engineer the synthesized RTL code to identify the timing bottlenecks and optimize the corresponding critical paths in the source program.

Designs generated by HLS tools typically achieve a lower frequency compared to manual RTL designs. As the first step in our journey to improve the achievable frequency, we study the timing issues in a diverse set of realistic and complex FPGA HLS designs. We observe that in almost all cases, the frequency degradation is caused by the ***broadcast*** structures generated by the HLS compiler. Based on our observation, we classify three major types of broadcasts in HLS-generated designs: high-fanout data signals, pipeline flow control signals, and synchronization signals for concurrent modules. Next, we reveal a number of limitations of the current HLS tools that result in those broadcast-related timing issues.

In collaboration with Jason Lau, we propose a set of effective yet easy-to-implement approaches, including broadcast-aware scheduling, synchronization pruning, and skid-buffer-based flow control. The solution part is accredited to Jason Lau. We briefly present them here for the sake of completeness. Our experimental results show that our methods can improve the maximum frequency of a set of nine representative HLS benchmarks by 53% on

average. In some cases, the frequency gain is more than 100 MHz.

In this chapter, the proposed optimization is *architecture-independent*, where we improve the inherent quality of the produced RTL to be passed to downstream tools. In comparison, we will present techniques to co-optimize HLS compilation and the downstream tools for further optimization.

3.1 Introduction

In this work, we analyze the timing issues of a diverse set of real-world HLS designs that are implemented and optimized using state-of-the-art commercial tools. To our surprise, in most cases, the frequency degradation is related to signal broadcasts. The signal broadcasts are automatically inferred or created by the HLS compiler, either in the datapath or the control logic. These broadcast structures are typically not explicitly presented in the source code, thereby often overlooked by HLS users. However, they will result in high-fanout interconnects that pose challenges for downstream physical design tools to close timing.

Here, we briefly discuss two case studies to motivate the importance of optimizing the implicit broadcasts in FPGA HLS. One is the genome sequencing accelerator [GLR19], where we identify a data signal broadcast that sends the output of one register to tens of targets. We observe that the tool underestimates the delay of the broadcast operation, which leads to sub-optimal scheduling results. Fixing the problem boosts the final frequency from 264 MHz to 341 MHz when implemented on an Amazon F1 instance. Another example is the streaming Jacobi accelerator [CCW18], where the pipeline control signal broadcast becomes the critical path. By optimizing the control strategy and removing its unnecessary broadcast, we improve the maximum operating frequency from 120 MHz to 253 MHz (2.1×).

Motivated by these encouraging results, we conduct a systematic analysis of the timing-critical broadcast structures in HLS, which naturally fall into two major categories:

- **Data broadcast** refers to a high-fanout signal in the datapath, which is typically formed after loop unrolling or array partitioning during the HLS compilation process.

The wire delay of an operator will increase as the broadcast factor increases. Such varying delays may cause trouble to the HLS scheduler, which generally relies on static pre-characterized delay estimation.

- **Control broadcast** refers to a high-fanout control signal which typically originates from an FSM (or controller) and reaches numerous datapath components such as registers or multiplexers. In our study, we particularly focus on two critical classes of control broadcast: (1) synchronization signal broadcast and (2) pipeline control signal broadcast. These structures commonly cause timing degradation in deeply pipelined and/or highly parallelized designs. Compared to the data broadcasts, the control broadcasts are less studied in HLS.

The timing issues caused by such broadcasts are extremely hard to debug. On the one hand, data broadcast structures are hard to notice in the source code. It is difficult for HLS users to realize and understand the fact that certain "innocent-looking" software code has negative implications on the timing of the synthesized hardware. On the other hand, since most of the control signals are created by the HLS tool, they are much more challenging (if not impossible) to optimize through source code changes. Hence, a sub-optimal control broadcast may completely offset the performance gains from other sophisticated HLS optimizations. Moreover, data and control broadcasts often entangle with each other. As we will see, an HLS design as innocent as a simple buffer can suffer from both of these two broadcasts. Both data and control broadcasts must be eliminated to achieve frequency improvements.

Although the broadcasts cause serious problems in the current HLS tools, we manage to find concise and easy-to-integrate solutions. First, we use synthetic designs to capture the relationship between the increased net delay versus the broadcast factor, which serves as an effective approximation. Second, we utilize a different pipeline control methodology to trade area for a lower broadcast factor, while we further minimize the area overhead. Third, we propose to prune redundant synchronization signals to simplify the design. Our experimental results based on Vivado HLS show that (1) the timing problems caused by broadcast are indeed widespread, and (2) our proposed methods can improve the frequency of a set of

representative HLS benchmarks significantly. In some cases, the gain is more than 100 MHz.

Our main technical contributions are as follows:

- We are the first to identify that the implicit signal broadcast is a major cause of the frequency degradation in highly-optimized designs synthesized using industrial-strength HLS tools. We further provide a classification of the timing-critical data and control broadcast structures.
- We propose a set of simple but effective techniques to optimize the timing of the implicit broadcasts in HLS automatically, which includes broadcast-aware scheduling, redundant synchronization pruning, and skid-buffer-based pipeline control.
- We apply our approaches to a set of nine real-world HLS benchmarks, and improve their frequency by 53% on average, with a marginal area overhead.

3.2 Classification of HLS Broadcasts

3.2.1 Data Signal Broadcast

By implicit data broadcast, we broadly refer to signal broadcasts in the HLS-synthesized datapath. These broadcasts are specified by the source code and directives, though they are less obvious to the users. As is explained in Section 2.1, current HLS tools have a fixed delay estimation for a certain operator. However, such estimation is no longer accurate with large data broadcasts.

We create two synthetic examples of common HLS design patterns to show how data signal broadcast structures are formed, and what limitations in current HLS tools lead to this problem:

1) Loop unrolling, as in Figure 3.1. The variable `source` is defined outside the loop body and is loop-invariant. Since it is accessed in each iteration, in the corresponding hardware shown in Figure 3.2, the register for `source` is connected to 1024 instances of the loop body, resulting in a data signal broadcast.

Obviously, in this case, the actual delay of the `add` operator in `"source + foo"` (line 5) includes the additional wire delay between the `source` register and the `add` operators. However, the current HLS delay model does not consider such a broadcast cost. Therefore, the scheduler still views the delay of this 1024-broadcast-`add` the same as a normal `add` without broadcast.

For example, in Figure 3.2, assume the delay of a simple `add` or `sub` operator is 1.5ns, while the actual delay for the 1024-broadcast-`add` is 2.5ns. If the timing target is 3ns, the HLS tool will schedule the `add` and `sub` to be performed within the same cycle, while they should have been separated to meet the timing constraint.

```
1 data_t source = ...;          // loop-invariant variable
2 for (size_t i = 0; i < 1024; i++) {
3 #pragma HLS unroll
4   foo = ...i...; bar = ...i...;  // loop-dependent
5   dest[i] = source + foo - bar; /* ... */ }
```

Figure 3.1: Code of data broadcast - Example #1: loop unrolling.

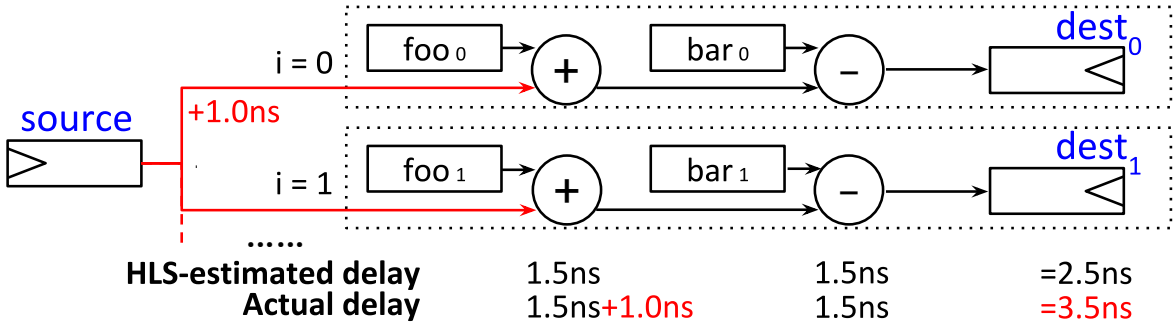


Figure 3.2: HLS-generated architecture of Fig. 3.1.

2) Large buffer and memory arrays, as in Figure 3.3. On FPGAs, a large on-chip buffer will be implemented as multiple block RAMs (BRAMs). Thus, the data `source` will fan out to many physically scattered memories, though they jointly form a single logical entity.

When the buffer size increases, the `load` and `store` operations will also suffer extra wire delays. However, most existing HLS tools do not take them into consideration either. The predicted delay remains the same regardless of the size of the buffer. This results in inadequate pipelining between the BRAM units and the data source/sink. For example, the

```
1 data_t buffer[737280]; // mapped to multiple BRAM units
2 buffer[idx] = source; // ‘source‘ connects to every BRAM unit
```

Figure 3.3: Code of data broadcast - Example #2: large array.

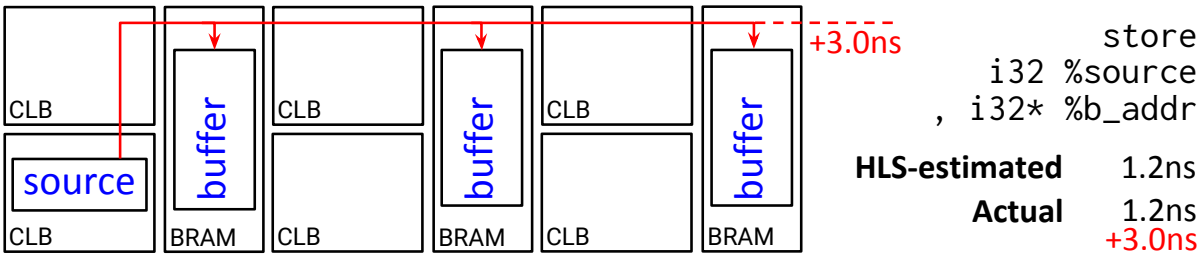


Figure 3.4: HLS-generated architecture of Fig. 3.3

double-buffer technique requires distributing data to the local buffers of multiple parallel processing elements (PEs) [CWY18a], which tend to be inadequately pipelined; the HLS support for dynamic data structures also requires large buffers [LSZ20], where their accesses degrades the maximum frequency.

3.2.2 Control Signal Broadcast - Synchronization

The synchronization logic originates from the parallelization of the sequential source code. The HLS scheduler automatically infers parallelism and schedules independent functions and operations to the same state for concurrent execution. To guarantee correctness, the HLS strictly follows the original semantics and generates control logic to wait for all parallel modules to complete before proceeding to the following operations. However, this fixed synchronization template may not be optimal for large designs and may introduce critical paths if the degree of parallelism scales up. Meanwhile, the FSM proceeds to the next stage only when all concurrent modules at the current stage signal their completion to the controller. This aggregated condition of `done`s is used as the next `start` signal and will be broadcast to the parallel modules in the next stage.

Figure 3.5 shows one scenario of this broadcast. For streaming designs, users describe the dataflow graph in sequential C++ code and let the HLS tools infer the parallelism. However, if multiple streaming kernels are defined in the same loop, the HLS will pedantically

synchronize them at the granularity of one iteration. As a result, independent flows are glued together and form a broadcast of the synchronization signal. Figure 3.7a visualizes this situation.

Figure 3.6 shows another example, where multiple independent instances of `PE_*()` are called, and they execute in parallel. The controller waits for all of them to finish, then reads their outputs together and proceeds to the next FSM stage. Figure 3.7b shows the corresponding logic generated by HLS.

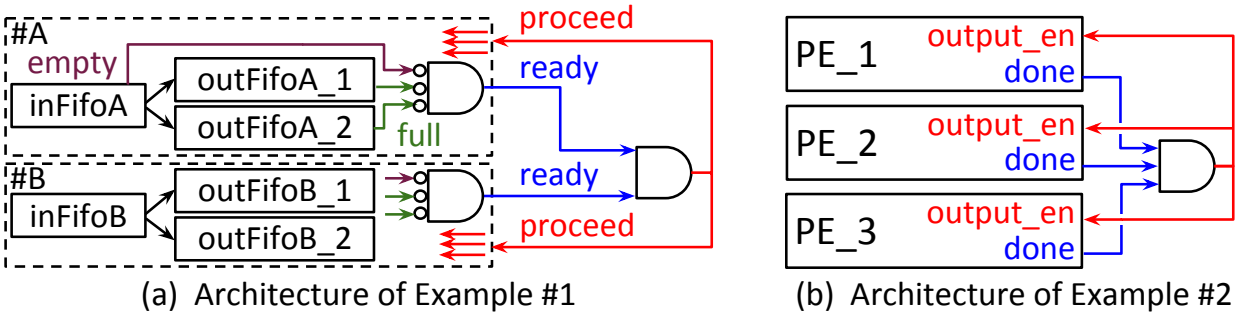
Although functionally correct, such a synchronization strategy is not scalable. The complexity of routing such "reduce-broadcast" signals will soon explode with increasing degrees of parallelism. Optimization of the synchronization logic is necessary.

```
1 #pragma HLS dataflow
2 while (1) {
3   /* --- inferred parallelization --- */
4   inFifoA.read(&a);
5     outFifoA1.write(a.foo); outFifoA2.write(a.bar); // #A
6   inFifoB.read(&b);
7     outFifoB1.write(b.foo); outFifoB2.write(b.bar); // #B
8   /* --- HLS infers excessive synchronization --- */ }
```

Figure 3.5: Code of synchronization - Example #1.

```
1 data_t kernel( ..... ) {
2   /* --- inferred parallelization --- */
3   aOut = PE_1(aIn); bOut = PE_2(bIn); cOut = PE_3(cIn); // ...
4   /* --- inferred synchronization --- */
5   return aOut + bOut + cOut /* ... */; }
```

Figure 3.6: Code of synchronization - Example #2.



(a) Architecture of Example #1 (b) Architecture of Example #2

Figure 3.7: HLS-generated architecture of the code above.

3.2.3 Control Signal Broadcast - Pipeline

For fully-pipelined [ZPF16] datapath, the `enable` signals for activation or the `stall` signals for flow control will be broadcast to every element of the pipeline to operate the datapath as a whole.

For a pipeline that interacts with modules with flow control interfaces (e.g., FIFOs), the most common approach of current HLS tools is to broadcast the back-pressure signals (e.g., `empty/full` and `valid/ready`) to control the flow. Figure 3.8 shows an example, and Figure 3.9 shows the corresponding inferred broadcast structure.

Previous works based on the theory of latency-insensitive design analyze the role of back-pressure in a theoretical way [Car06], but lack consideration in the aspect of circuit implementation. Though effective for small designs, this methodology will soon become the critical path with increasing pipeline sizes.

```
1 for (int i = 0; i < ITER; i++) {
2 #pragma HLS pipeline
3   input_fifo.read(&a); /* implicit "empty"-based stall */
4   b = inlined_datapath_foo(a);
5   output_fifo.write(b); /* implicit "full"-based stall */ }
```

Figure 3.8: Code example of pipeline control signal broadcast.

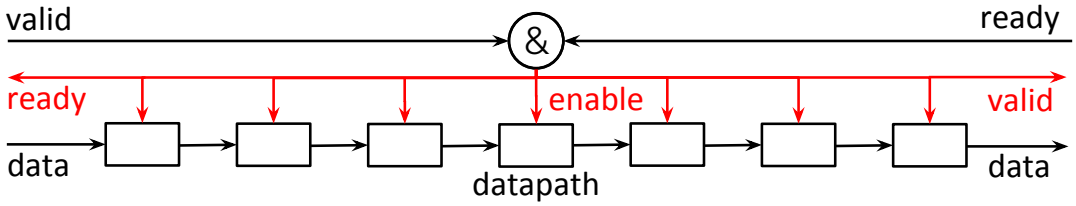


Figure 3.9: HLS-generated architecture of pipeline control.

3.3 Approaches

In the last section, we present specific examples of the implicit broadcasts in FPGA HLS, and why the current HLS tools fail to properly handle them. In this section, we present

corresponding automated solutions to address these limitations[1].

3.3.1 Broadcast-Aware Scheduling

As previously mentioned, the current HLS delay estimation does not consider the extra wire delay caused by the broadcast. Decades of research on HLS have shown that it is extremely hard to have an accurate delay estimation without the placement information [ZGR14]. However, here we propose a simple but effective method that can be used to approximate this extra delay.

We implement skeleton broadcast structures on an empty FPGA to obtain the post-routed delay. For example, in one skeleton design, we instantiate 64 adders, and one of the two input ports of every adder is connected to a common source register. For buffer access operations (`load`, `store`), we record the actual delays of different buffer sizes when targeting an empty FPGA. Although the placement results may differ from the real situation, it serves as an effective lower bound to the delay penalty. In this way, we collect reusable statistics of calibrated delays for each combination of operator, data type, and broadcast factor. When the broadcast factor is small, the delay obtained from our experiment is consistent with the predicted delay of the Vivado HLS tool. In fact, current HLS tools adopt a similar pre-characterized approach to build up their delay model, except that they do not characterize the effects of broadcasts as we do.

Figure 3.10 shows our measured delay of the `add` operation and the BRAM buffer access of the `int` type, and multiplication of the `float` type by Xilinx Vivado. For the add and buffer access operations, the delay values obtained by our experiments perfectly match with the Vivado-HLS-predicted values when the broadcast factor is small. For large broadcast factors, our measurement significantly surpasses the HLS-predicted values, which reveals the inaccuracy of current delay estimations under large broadcast factors. For the multiplication, the HLS-predicted delays are much higher than that in our experiments, possibly because

[1]The contents of Chapter 3 is from a collaboration between me and Jason Lau. We made equal contributions to our DAC '20 paper [GLC20], which is the basis of Chapter 3. Specifically, the problem classification part (Section 3.2) is accredited to me and the solution part (Section 3.3) is accredited to Jason Lau.

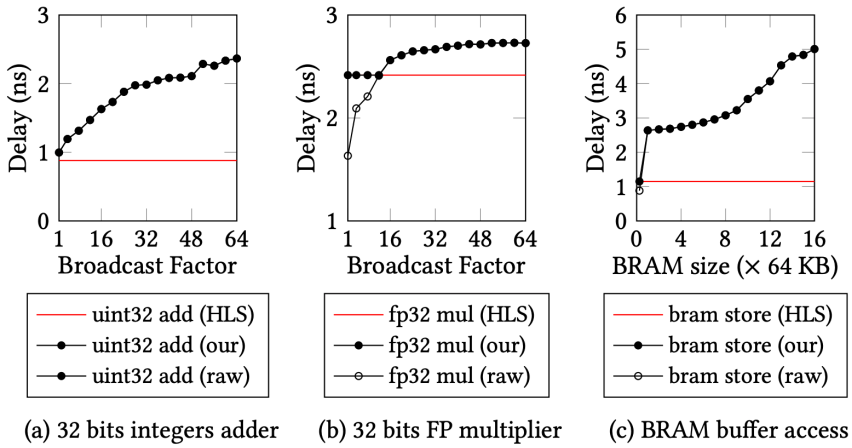


Figure 3.10: Vivado HLS estimated delay, our calibrated delay, and raw experimental delay on different operators.

the Vivado HLS tool is deliberately conservative about multiplication for floating points. Therefore, we choose the maximum between the HLS-predicted delay and our experimented results as our calibrated delay.

3.3.2 Synchronization Logic Pruning

The redundant synchronization logic may severely limit the maximum achievable frequency. From the perspective of the downstream logic synthesis tools, they cannot be optimized away; but with high-level information, we are able to identify and get rid of these synchronizations. For instance, [QDF18] removes the redundant synchronization logic by directly modifying the RTL design. We are motivated by their design and performance optimization at the HLS level.

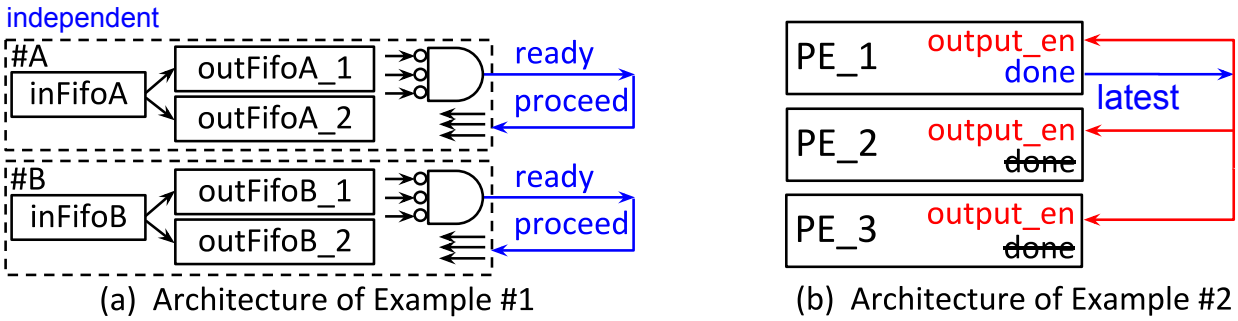


Figure 3.11: Pruned architecture corresponding to Figure 3.7.

For the first case—dataflow synchronization, as shown in Fig. 3.5, we propose to isolate the independent flow paths in the flow graph. We reconstruct the dataflow graph, not based on the user-defined streaming kernels, but at the granularity of the elementary flow control units. We identify the isolated sub-graphs within user-defined streaming kernels and split the independent flows explicitly into separate loops, which avoids unwanted synchronization from the HLS compiler. Figure 3.11a shows the optimized logic.

For the second case — synchronizing parallel modules, as shown in Figure 3.6, the key challenge is to properly pipeline the control signals. To do so, we need the layout information of the design. This problem is addressed by our later work that couples floorplanning with HLS compilation, which will be detailed in the next Chapter.

3.3.3 Skid-Buffer-Based Pipeline Control

We identify that the flow control broadcast can be avoided by a common practice of adopting additional *bounded-size* buffering called a skid buffer [Int22b], which is shown in Figure 3.12. We further improve this method by minimizing the area overhead.

Instead of switching the whole pipeline between two modes—active and stalled, we transform the control logic to keep the pipeline *always flowing*, and associate a `valid` bit with each data. The key to avoiding overflow is the skid buffer (an extra bypass FIFO) appended at the end of the pipeline. When the downstream is not ready, data will accumulate in the buffer. Then the buffer will become non-empty, and the pipeline will stop reading from the upstream so that the later pipeline inputs will be invalid bubbles. Assuming the length of the pipeline is N, as long as the depth of the buffer is no smaller than $N + 1$ ($+1$ since the `empty` signal will be reset one cycle after the first element is in), no overflow will happen. We refer to this practice as *skid-buffer-based pipeline control*. Note that this approach has the exact same throughput as the original stall-based back-pressure control.

However, this method introduces area overhead. For the original implementation, the area overhead will be:

$$BufferArea = (N + 1) \cdot w_\beta$$

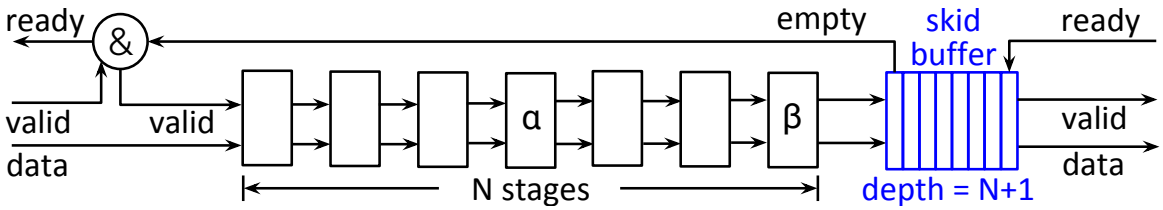


Figure 3.12: Skid-buffer-based pipeline control architecture.

where N is the depth of the pipeline and w_β is the width of the output data of stage β, as marked in Figure 3.12.

Observe that the skid buffer can be split and distributed into the datapath, as shown in Figure 3.13. Instead of an N-depth buffer of width w_β at the end of the whole pipeline, we can insert an $(M{+}1)$-depth buffer of width w_α after the M-th stage, and an $(N{-}M{+}1)$-depth buffer of width w_β after the final stage.

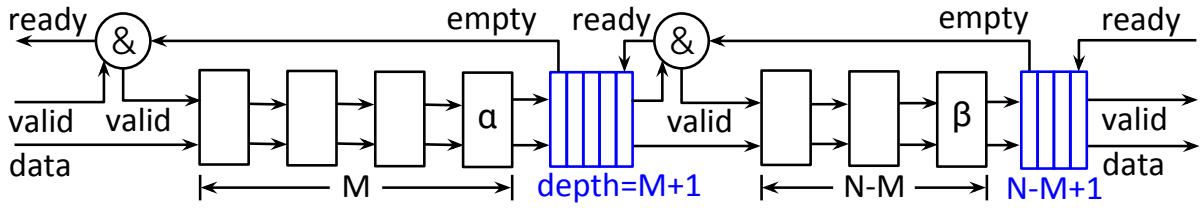


Figure 3.13: Multi-level skid-buffer-based pipeline control.

The new area overhead will be:

$$BufferArea' = (M+1) \cdot w_\alpha + (N-M+1) \cdot w_\beta$$

To obtain the data width between stages, we parse the schedule report and collect the definition location and usage location for each variable, thus obtaining the total data width passed between stages. Due to the engineering challenges to modify the HLS tool without access to the source code, our proof-of-concept implementation of the proposed solutions still involves some manual parts. We hope that the identified issues and corresponding solutions appeal to HLS vendors and can be integrated into their commercial tools.

3.4 Experiments

Our experiments are based on the Vivado HLS since most open-source HLS benchmarks are developed using it. We use the Vivado version 2018.2 with default mode. Retiming and fan-out optimization are enabled. The target FPGA chips are based on the choices of the original sources of the designs.

Table 3.1: Timing improvements and post-implementation resources on HLS designs using our proposed solutions.

Application	Broadcast type	Target FPGA	LUT (%)		FF (%)		BRAM (%)		DSP (%)		Freq (MHz)		
			Orig	Opt	Orig	Opt	Orig	Opt	Orig	Opt	Orig	Opt	Diff
Genome Sequencing [GLR19]	Data	UltraScale+ (AWS F1)	22	22	11	12	6	6	8	8	264	341	29%
LSTM Network [CHB18]	Data	UltraScale+ (AWS F1)	8	9	6	6	2	2	14	14	285	325	14%
Face Detection [SDM17]	Data	ZYNQ (ZC706)	21	22	14	15	16	16	9	9	220	273	24%
Matrix Multiply	Pipe. Ctrl. & Data	UltraScale+ (AWS F1)	23	23	24	27	25	25	74	74	202	299	48%
Stream Buffer	Pipe. Ctrl. & Data	UltraScale+ (AWS F1)	1	1	1	1	95	95	0	0	154	281	82%
Stencil [CCW18]	Pipe. Ctrl.	UltraScale+ (AWS F1)	40	40	41	41	30	29	83	83	120	253	111%
Vector Arithmetic	Pipe. Ctrl. & Sync.	UltraScale+ (AWS F1)	17	17	16	15	0	¡1	60	60	195	301	54%
HBM-Based Stencil [CCW18]	Pipe. Ctrl. & Sync.	UltraScale+ (Alveo U50)	21	23	23	23	34	31	37	37	191	324	70%
Pattern Matching [CWY18a]	Data & Sync.	Virtex-7 (Alpha-Data)	17	17	5	7	9	9	0	0	187	278	49%

3.4.1 Benchmarks

Our results are in Table 3.1. The genome sequencing design is from [GLR19]. We adjust the broadcast factor by changing `BACK_SEARCH_COUNT`. The LSTM inference network design is from [CHB18]. We adopt the `HLS_N-Node` part, change the data type to floating point, and set N to 256. The face detection design is from the Rosetta benchmark [ZGD18]. The matrix multiply and the pattern matching design are adapted from [CWY18a]. We further increase the parallelism of the matrix multiplication design to expose the problem. The Jacobi stencil kernel and its HBM version are generated by the SODA compiler [CCW18]. The streaming buffer design consists of two loops, which first write to a very large buffer and then read from the buffer.

3.4.2 Case Study for Broadcast-Aware Scheduling

We illustrate the experiment with [GLR19], an genome sequencing accelerator designed by the author, as a case study to present our broadcast-aware scheduling method.

In genome sequencing, it is a crucial but time-consuming task to detect potential overlaps between any pair of input reads, especially those that are ultra-long. The state-of-the-art overlapping tool Minimap2 outperforms other popular tools in speed and accuracy. It has a single computing hot-spot, chaining, that takes 70% of the time and needs to be accelerated.

There are several crucial issues for hardware acceleration because of the nature of chaining. First, the original computation pattern is poorly parallelizable and a direct implementation will result in low utilization of parallel processing units. We propose a method to reorder the operation sequence that transforms the algorithm into a hardware-friendly form. Second, the large but variable sizes of input data make it hard to leverage task-level parallelism. Therefore, we customize a fine-grained task dispatching scheme that could keep parallel PEs busy while satisfying the on-chip memory restriction. Based on these optimizations, we map the algorithm to a fully pipelined streaming architecture on FPGA using HLS, which achieves significant performance improvement. Compared to the multi-threading CPU baseline, our FPGA accelerator achieves 28× acceleration.

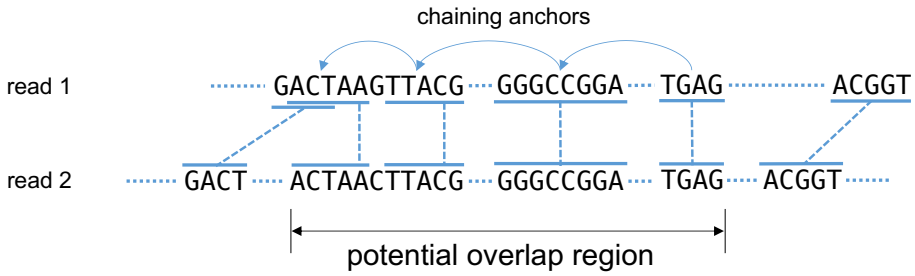


Figure 3.14: Finding estimated overlap region.

Figure 3.15 shows the micro-architecture of the accelerator and Figure 3.17 shows an operation chain scheduled by HLS. Since `curr.x` is consumed by 64 `sub` operators, in comparison to the HLS predicted delay, we adjust the predicted delay of the `sub` from 0.78ns to 2.08ns according to our measurement of the skeleton designs. Therefore, we insert a register module to force the splitting of the operation chain. Figure 3.18 shows the frequency gain.

Experiment results show that our method approximates the actual delay in a more reasonable way, while the HLS-estimated delay is invariant to broadcast factors. Although

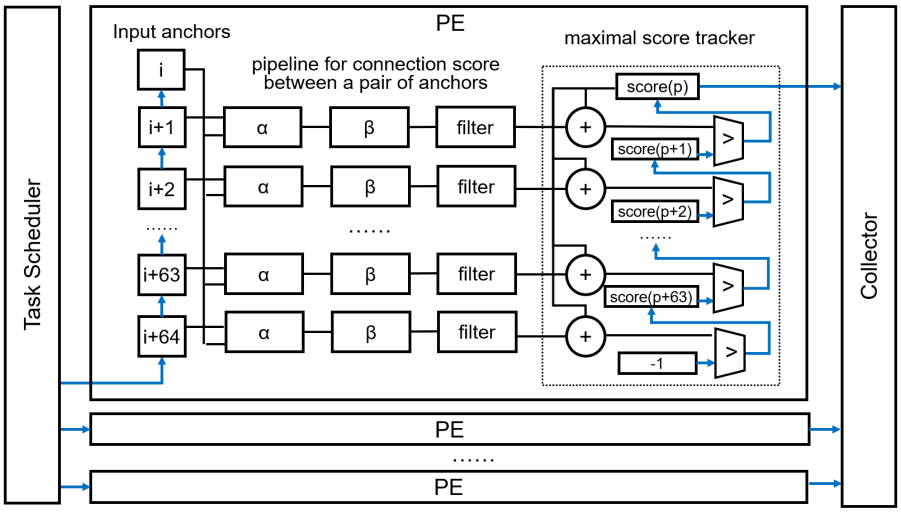


Figure 3.15: Overview of a processing element array.

our result does not match the actual perfectly, our frequency gain shows that this is helpful for broadcast operations, since less neighboring logic will be put in the same cycle of the broadcast, which will facilitate downstream retiming and fanout optimization. As for overhead, the length of the pipeline is 9 originally and 10 after optimization. Both have the same initiation interval of 1. There will be a fairly small overhead in the usage of flip-flops, which is generally negligible.

3.4.3 Synchronization Logic Pruning

We use the HBM-based (High-Bandwidth Memory) Jacobi stencil acceleration kernel generated by the SODA compiler [CCW18, CC20], which uses 28 independent memory ports of the HBM. The 512-bit data from each HBM port is scattered into 8 64-bit FIFOs, later to different streaming kernels. However, the SODA compiler expresses the 28 independent flows together in a single loop, forming a sync broadcast pattern similar to Figure 3.7a. Thus there is synchronization among all HBM ports and all destination FIFOs. We prune the unnecessary sync by splitting the independent parts into different loops. This boosts the frequency from 191 MHz to 324 MHz.

```
1 #pragma HLS pipeline
2 #define UNROLL_FACTOR 64
3 // ......
4 for (int j = 0; j < UNROLL_FACTOR; j++) {
5 #pragma HLS unroll
6     dist_x = prev[j].x - curr.x;
7     dist_y = prev[j].y - curr.y;
8
9     dd = dist_x > dist_y ? dist_x - dist_y : dist_y - dist_x;
10    min_d = dist_y < dist_x ? dist_y : dist_x;
11    log_dd = log2(dd); // a series of if-else
12    temp = min_d > prev[j].w ? prev[j].w : min_d;
13
14    dp_score[j]= temp - dd * avg_qspan - (log_dd>>1)
15    if((dist_x == 0 || dist_x > max_dist_x )||
16        (dist_y > max_dist_y || dist_y <= 0) ||
17        (dd > bw) || (curr.tag != prev[j].tag) ){
18        dp_score[j] = NEG_INF_SCORE;
19 }   } ......
```

Figure 3.16: Design code snippet from [GLR19]. The loop-invariant variables (broadcast sources) are marked blue.

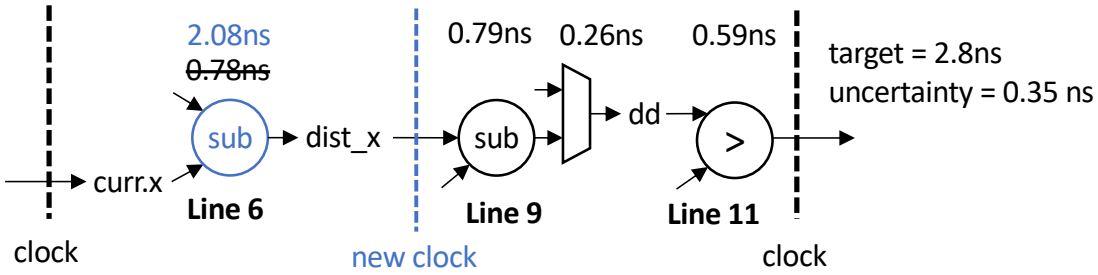
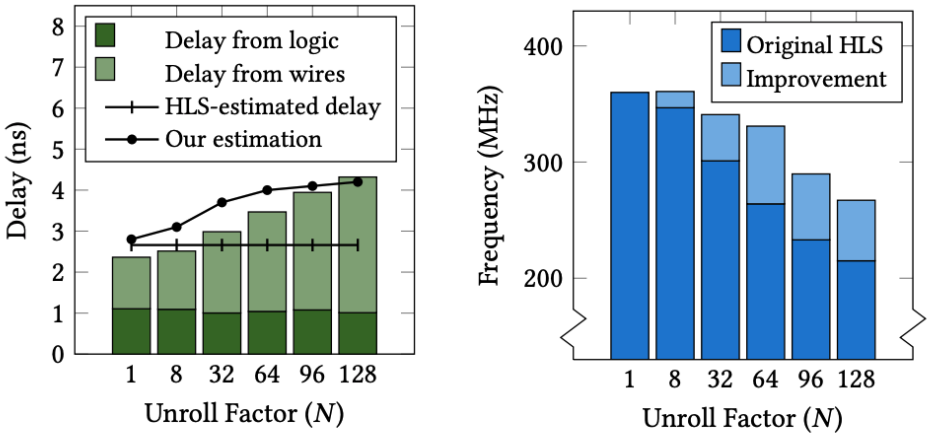


Figure 3.17: An operation chain with broadcast operators.

3.4.4 Skid-Buffer-Based Pipeline Control

We again experiment with the SODA compiler [CCW18], but this time generate the 2D Jacobi kernel as a whole pipeline. We concatenate different iterations of the kernel to change the size of the pipeline. Each iteration takes about 5% of LUT, 5% of Flip-Flop, 4% of BRAM, and 10% of DSP. Figure 3.19 shows the improvements by changing the pipeline control logic to the skid-buffer-based method. The super pipeline of eight Jacobi iterations has 370 datapath stages and produces 512-bit results. Since this pipeline has a spindle shape, the best strategy is to add the entire buffer at the end of the pipeline. The corresponding



(a) The delay estimations of HLS and our tool, and the actual delay of a critical path in the original [1] design.

(b) Achieved frequency of the [1] design using HLS's original schedule and our schedule on different unroll factors.

Figure 3.18: Optimization of data broadcast.

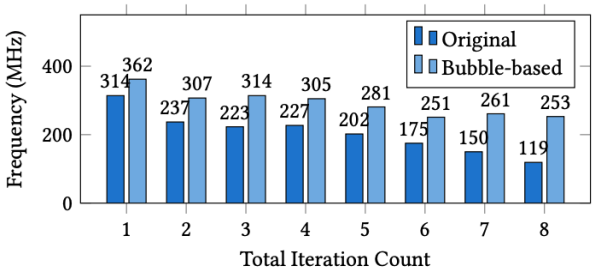


Figure 3.19: The achieved frequency of the Jacobi kernels.

buffer only costs about 23KB of BRAM resources.

We present a synthetic example to further demonstrate the benefit of our dynamic programming algorithm to minimize the area of the extra buffer. Assume the pipeline computes $(\mathbf{a}\cdot\mathbf{b})\mathbf{c}$, where the dot-product of vector $\mathbf{a}$ and $\mathbf{b}$ is scalar-multiplied with vector $\mathbf{c}$. A reduction tree is inferred for $\mathbf{a}\cdot\mathbf{b}$, and the output scalar is multiplied with $\mathbf{c}$. Figure 3.20 shows the case for a 32-wide vector of float numbers. Note that in stage #56 only one number (the result of $\mathbf{a}\cdot\mathbf{b}$) is passed through. Thus, the first stages #1 to #56 should be buffered separately from the stages after #56. Directly adding a buffer at the end results in $(61+1)\times 1024 = 63488$ bits while the optimized version costs $(56+1)\times 32 + (5+1)\times 1024 = 7968$ bits. Table 3.2 shows the results for the 512-wide vector product.

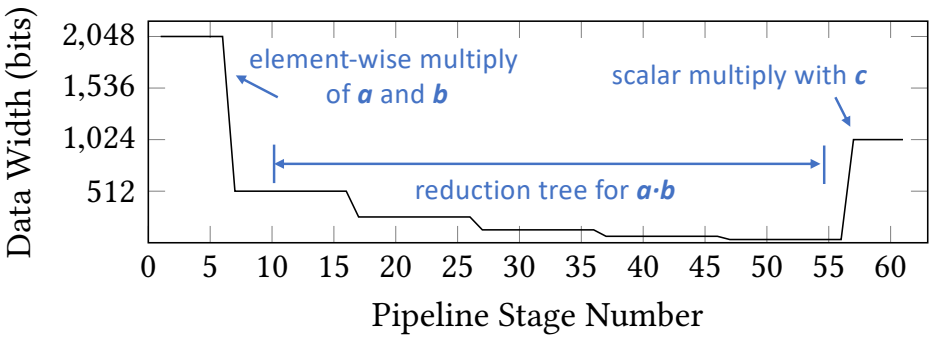


Figure 3.20: Bitwidth of the passed data between stages.

Table 3.2: Experiment results on 512-wide vector product.

Implementation	Frequency	LUT	FF	BRAM	DSP
Stall	195 MHz	17%	16%	0%	60%
Skid Buffer	299 MHz	18%	16%	12%	60%
Min-Area Skid Buf.	301 MHz	17%	15%	0.02%	60%

3.4.5 Combined Effect

In many real-world cases, we must combine these two aforementioned approaches to truly resolve the timing degradation. For example, Figure 3.21 shows a simple stream buffer with both data and control broadcasts. The source data register is connected to each of the BRAM units, forming an implicit data broadcast. Besides, the `enable` back-pressure signal is broadcast to all BRAM units.

```
1 loop1: for (int i = 0; i < BIG_SIZE; i++) {
2 #pragma HLS pipeline II=1
3     in_fifo.read(&buffer[i]); } // data into buffer
4 loop2: for (...) ... // data out of buffer
```

Figure 3.21: Code for the large buffer access example.

Based on the size of the array and the pipeline environment, additional latency is added to optimize the data broadcast. Meanwhile, the skid-buffer-based pipeline control is used to avoid the broadcast of `enable`. Figure 3.22 shows the achieved frequency of varying buffer sizes. Three batches of experiments are done: the original one; the version which only has the data broadcast optimized; the version with both the data and control broadcast optimized.

As is obvious, we need to optimize both the data broadcast and the control broadcast to achieve scalable performance.

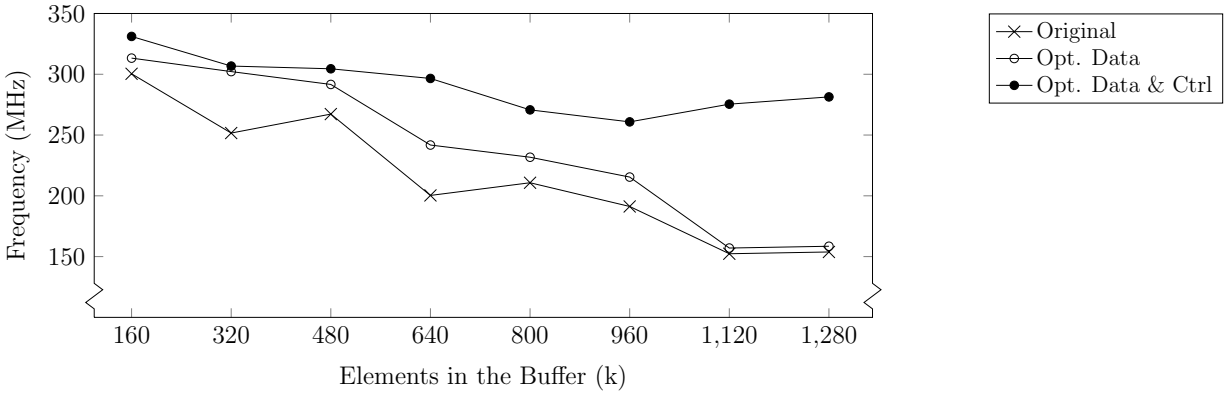


Figure 3.22: Achieved frequencies of the stream buffer design.

Another example is pattern matching from [CWY18a] with both data and sync control broadcast similar to Figure 3.7b. Addressing both of them lead to a large frequency improvement, as in Table 3.3.

Table 3.3: Experiment results on pattern matching.

Implementation	Frequency	LUT	FF	BRAM	DSP
Original	187 MHz	17%	5%	9%	0%
Opt. Data	208 MHz	18%	7%	9%	0%
Opt. Data & Ctrl	278 MHz	17%	7%	9%	0%

3.5 Conclusion

In this paper, we analyze the common types of broadcast in HLS. We present delay model calibration, synchronization pruning, and min-area skid-buffer-based pipeline control. We bring over 50% of frequency gain on real-world designs.

CHAPTER 4

AutoBridge: Coupling Coarse-Grained Floorplanning and Pipelining for High-Frequency HLS Design on Multi-Die FPGAs

In the previous chapter, we present techniques to improve the inherent timing quality of the RTL produced by HLS. Note that the proposed optimization targets the HLS compilation process alone, without involving other parts of the overall CAD tool stack.

In this chapter, we explore further improvements in the final frequency by considering HLS compilation and placement together. The key idea is that HLS has the flexibility to add additional pipelining without breaking the overall functionality, while the placer knows which connections are the critical paths and need additional pipelining. Coupling them together brings brand-new opportunities to improve the frequency even for the largest designs today.

4.1 Introduction

One major cause that leads to the unsatisfactory frequency is that HLS cannot easily predict the physical layout of the design after placement and routing. Current HLS tools typically rely on pre-characterized operation delays and a very crude interconnect delay model to insert clock boundaries (i.e., registers) into an untimed design to generate a timed RTL implementation [ZGR14, TDG15, GLC20]. Afterward, optimizations in RTL and physical synthesis such as retiming are expected to fix the potential critical paths due to inadequate pipelining. However, while retiming can redistribute the registers along a path, the total number of registers along each path or cycle must remain a constant [LS91], significantly

limiting the scope of improvement. Hence, as the HLS designs get larger, the timing quality of the synthesized RTLs usually further degrades.

This timing issue is worsened as modern FPGA architectures become increasingly heterogeneous [Xil20c]. The latest FPGAs integrate multiple dies using silicon interposers to pack more logic on a single device; however, the interconnects that go across the die boundaries will carry a non-trivial delay penalty. In addition, specialized IP blocks such as PCIe and DDR controllers are embedded in the programmable logic. These IP blocks usually have fixed locations near the dedicated I/O banks and will consume a large number of programmable resources nearby. As a result, these dedicated IPs often detour the nearby wires toward more expensive and/or longer routing paths. Further, modules interacting with such fixed-location IPs are also more constrained in their layout. This, in turn, results in long-distance communication with other modules. Together these factors tend to further lower the final clock frequency.

There are a number of prior attempts that couple the physical design process with HLS compilation [ZGR14, CFH04, XK97, Cad20, Syn20], as we discuss in Section 2.5.1. The previous approaches share the common aspect of focusing on the fine-grained interaction between HLS and physical design, where individual operators and the associated wires and registers are all involved during the delay prediction and iterative HLS-layout co-optimization. While such a fine-grained method can be effective on relatively small HLS designs and FPGA devices, it is too expensive (if not infeasible) for today's large designs targeting multi-die FPGAs, where each implementation iteration from HLS to bitstream may take days to complete.

In this paper, we propose *AutoBridge*, a *coarse-grained* floorplan-guided pipelining approach that addresses the timing issue of large HLS designs in a highly effective and scalable manner. Instead of coupling the entire physical design process with HLS, we guide HLS with a coarse-grained floorplanning step, as shown in Figure 4.1. Our coarse-grained floorplanning involves dividing the FPGA device into a grid of regions and assigning each HLS function to one region during HLS compilation. For all the inter-region connections we further pipeline them to facilitate timing closure while we leave the intra-region optimization to the default

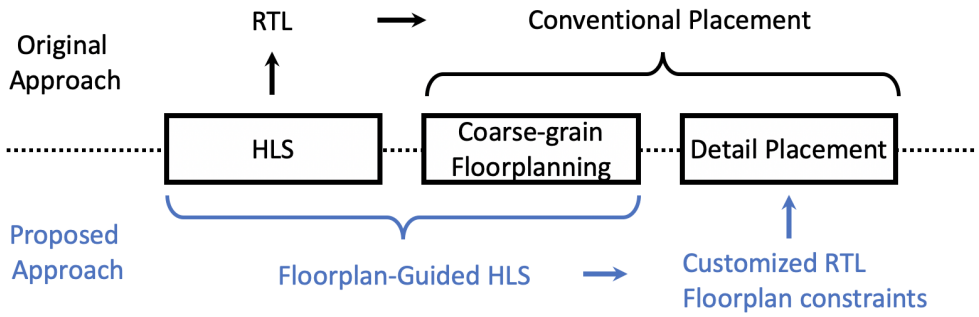


Figure 4.1: Core idea of the proposed methodology.

HLS tool.

Our methodology has two major benefits. First, the early floorplanning step provides HLS with a view of the global physical layout which helps HLS more accurately identify and pipeline the long wires, especially those crossing the die boundaries. Compared to retiming [LRS83], HLS-level pipelining creates more optimization opportunities for the downstream synthesis and physical design steps, thus potentially leading to higher performance. Second, pipeline-aware floorplanning can reduce local routing congestion by guiding the subsequent placement steps to better distribute logic across multiple dies, instead of attempting to pack the logic into a single die as much as possible.

While AutoBridge can improve the *frequency* with additional interconnect pipelining, we also need to ensure the added latency does not negatively impact the overall throughput of the design. To this end, we present analysis and latency balancing algorithms to guarantee the throughput of the resulting design is not negatively impacted.

Our specific contributions are as follows:

- To the best of our knowledge, we are the first to tackle the challenge of high-frequency HLS design on multi-die FPGAs by coupling floorplanning and pipelining.
- We design a coarse-grained floorplan scheme tailored for HLS which can distribute the design logic across multiple dies on an FPGA to effectively reduce local congestion and facilitate HLS to adequately pipeline global interconnects.
- We analyze how the additional latency may affect the throughput of the design and

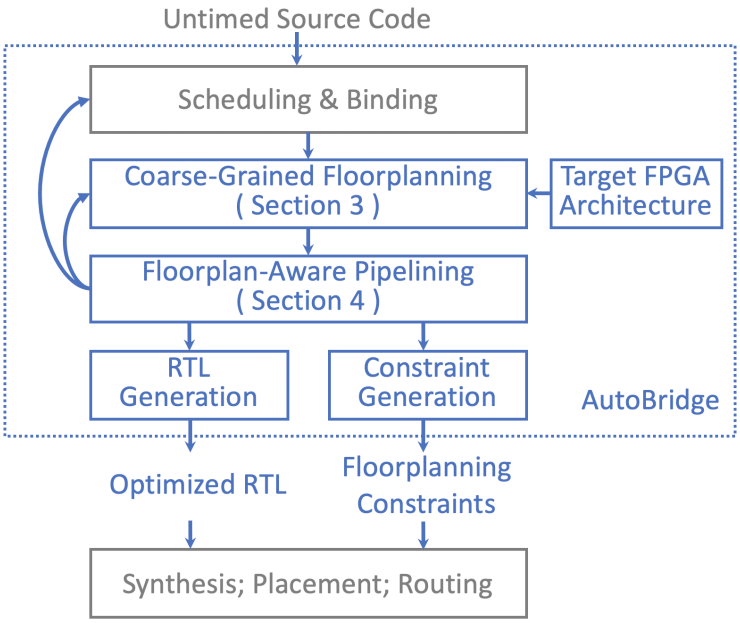


Figure 4.2: Overview of the AutoBridge Framework. Grey boxes represent the original software flow and blue boxes represent components of AutoBridge.

propose algorithms to offset the potential negative influence of the added latency.

- Our framework, AutoBridge, interfaces with the commercial FPGA design tool flow, with a compile-time overhead in the order of seconds. It improves the average frequency of 43 designs from 147 MHz to 297 MHz with a negligible area overhead.

Figure 4.2 shows the overall flow of our proposed methodology. The rest of the paper is organized as follows: Section 4.2 introduces background information on modern FPGA architectures and shows motivating examples; Section 4.3 details our coarse-grained floorplan scheme inside the HLS flow; Section 4.4 describes our floorplan-aware pipelining methods; Section 4.5 presents experimental results; Section 2.5 provides related work, followed by conclusion and acknowledgments.

4.2 Background and Motivating Examples

4.2.1 Multi-Die FPGA Architectures

Figure 4.3 shows three representative multi-die FPGA architectures, each of which is described in more detail as follows.

- The Xilinx Alveo U250 FPGA is one of the largest FPGAs with four dies. All the I/O banks are located in the middle column and the four DDR controller IPs are positioned vertically in a tall-and-slim rectangle in the middle. On the right lies the Vitis platform region [Xil20d], which incorporates the DMA IP, the PCIe IP, etc, and serves to communicate with the host CPU.

- The Xilinx Alveo U280 FPGA is integrated with the latest High-Bandwidth Memory (HBM) [CCW20, CCQ21, HBM20], which exposes 32 independent memory ports at the bottom of the chip. I/O banks are located in the middle columns. Meanwhile, there is a gap region void of programmable logic in the middle.

- The Intel Stratix 10 FPGA [Int20] also sets the DDR controller and I/O banks in the middle of the programmable logic. The embedded multi-die interconnect bridges and the PCIe blocks are distributed at the two sides of the chip, allowing multiple FPGA chips to be integrated together. Although this paper uses the Xilinx FPGAs to demonstrate the idea, our methodology is also applicable to Intel FPGAs and other architectures.

Compared to previous generations, the latest multi-die FPGA architectures are divided into disjoint regions, where the region-crossing naturally incurs additional signal delay. In addition, the large pre-located IPs consume significant programmable resources near their fixed locations which may also cause local routing congestion. These characteristics can hamper the existing HLS flows from achieving a high frequency.

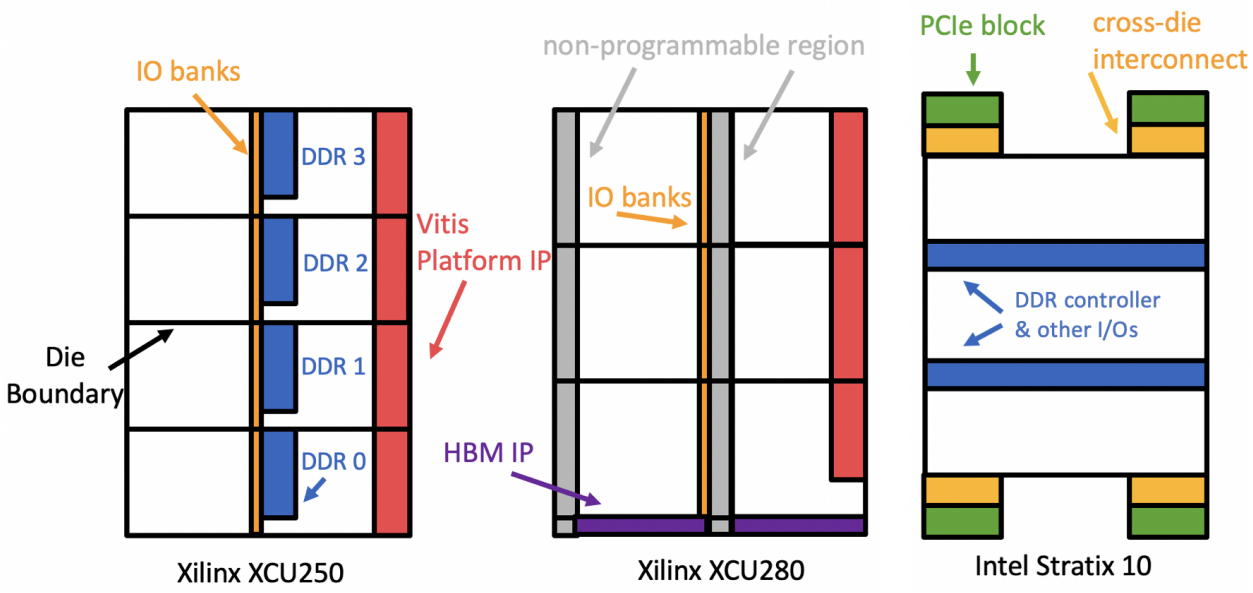


Figure 4.3: Block diagrams of three representative FPGA architectures: the Xilinx Alveo U250, U280 (based on the Xilinx UltraScale+ architecture), and the Intel Stratix 10.

4.2.2 Motivating Examples

We show two examples to motivate our floorplan-guided HLS approach. First, Figure 4.4 shows a CNN accelerator implemented on the Xilinx U250 FPGA. It interacts with three DDR controllers, as marked in grey, pink, and yellow blocks in the figure. In the original implementation result, the whole design is packed close together within die 2 and die 3. To demonstrate our proposed idea, we first manually floorplan the design to distribute the logic in four dies and to avoid overlapping the user logic with DDR controllers. Additionally, we pipeline the FIFO channels connecting modules in different dies as demonstrated in the figure. The manual approach improves the final frequency by 53%, from 216 MHz to 329 MHz.

Second, Figure 4.5 shows a stencil computation design on the Xilinx U280 FPGA. It consists of four identical kernels in linear topology with each color representing a kernel. In the original implementation, the tool's selection of die-crossing wires is sub-optimal and one kernel may be divided among multiple regions. Instead in our approach, we pre-determine all the die-crossing wires during HLS compilation and pipeline them, so the die boundaries will not cause any problems for the placement and routing tool. For this example, we achieve 297 MHz while the design is originally *unroutable*.

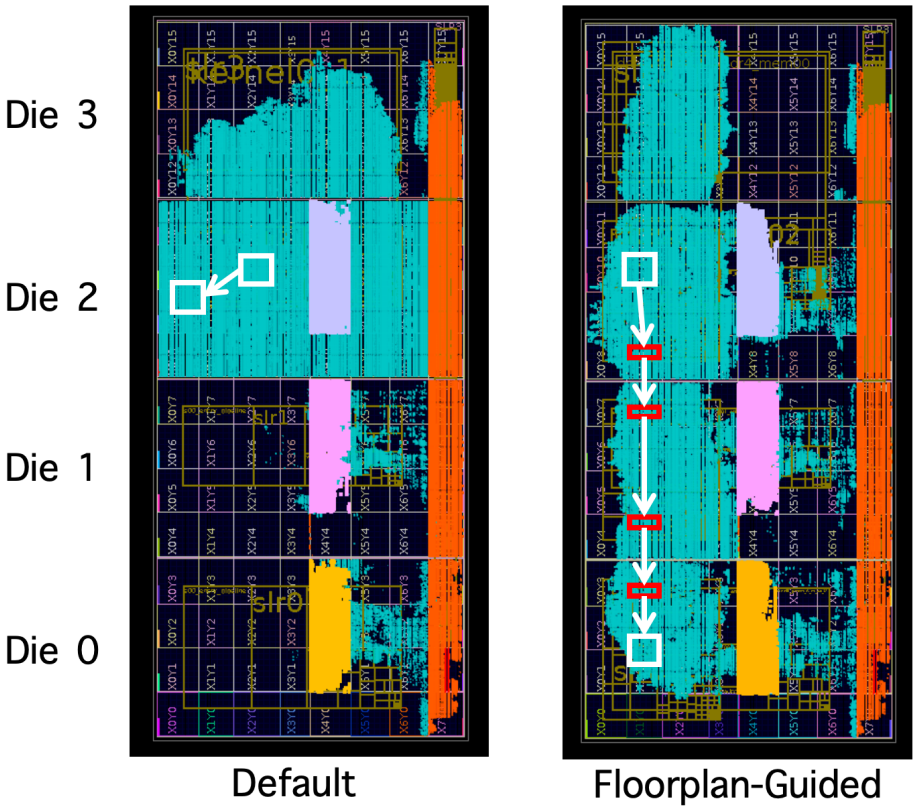


Figure 4.4: Implementation results of a CNN accelerator on the Xilinx U250 FPGA. Spreading the design across the device helps reduce local congestion, while the die-crossing wires are additionally pipelined.

4.3 Coupling HLS with Coarse-Grained Floorplanning

In this section, we present our coarse-grained floorplanning scheme that can be integrated with HLS. We assume that HLS preserves the hierarchy of the source code, and each ***function*** in the HLS source code will be compiled into an RTL ***module***.

Note that the focus of this work is not on improving floorplanning algorithms; instead, we intend to properly use coarse-grained floorplan information to guide HLS and placement.

4.3.1 Coarse-Grained Floorplanning Scheme

Instead of finding a dedicated region with a detailed aspect ratio for each module, we choose to view the FPGA device as a ***grid*** that is formed by the die boundaries and the large IP blocks. These physical barriers split the programmable fabric apart into a series of disjoint ***slots*** in

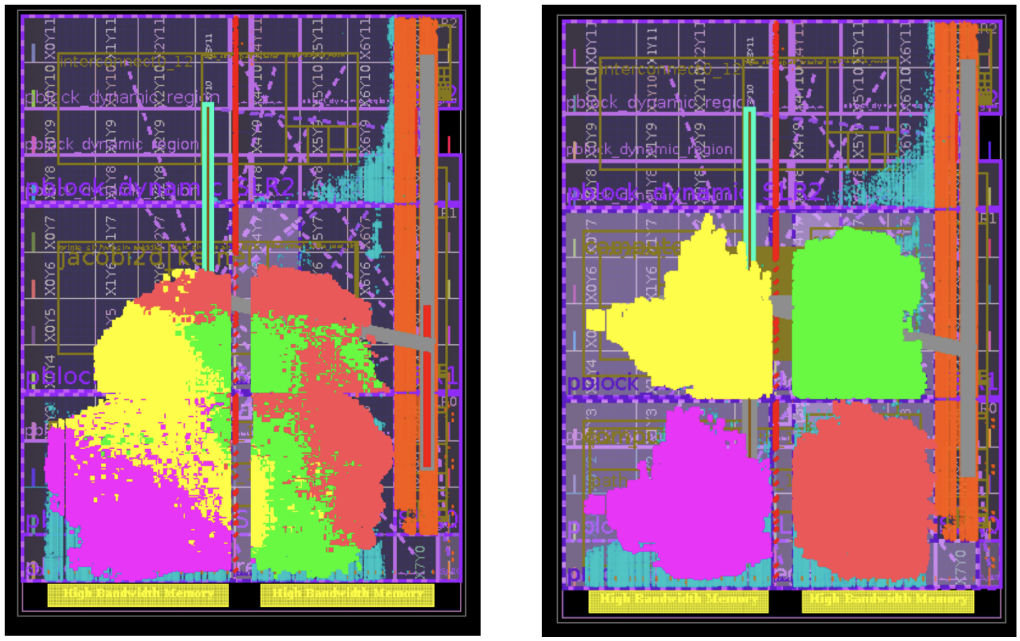


Figure 4.5: Implementation results of a stencil computing design on U280. Floorplanning during HLS compilation significantly benefits the physical design tools.

the grid where each slot represents a sub-region of the device isolated by die boundaries and IP blocks. Using our coarse-grained floorplanning, we will assign each function of the HLS design to one of these slots.

For example, for the Xilinx Alveo U250 FPGA, the array of DDR controllers forms a vertical split in the middle column; and there are three horizontal die boundaries. Thus the device can be viewed as a grid of 8 slots in 2 columns and 4 rows. Similarly, the U280 FPGA can be viewed as a grid of 6 slots in 2 columns and 3 rows.

In this scheme, each slot contains about 700 BRAM_18Ks, 1500 DSPs, 400K Flip-Flops, and 200K LUTs. Meanwhile, to reduce the resource contention in each slot, we set a maximum utilization ratio for each slot to guarantee enough blank space. Experiments show that such slot sizes are suitable, and HLS has a good handle on the timing quality of the local logic within each slot, as in Section 4.5.

4.3.2 Problem Formulation

We first assume the HLS design adopts a dataflow programming model, where each function corresponds to one dataflow process, and each function will be compiled into an RTL module. Functions communicate with each other through FIFO channels.

Given: (1) a graph $G(V, E)$ representing the HLS design where V represents the set of functions[1] of the dataflow design and E represents the set of FIFO channels between vertices; (2) the number of rows R and the number of columns C of the grid representation of the target device; (3) maximum resource utilization ratios for each slot; (4) location constraints such that certain IO modules must be placed nearby certain IP blocks. In addition, we may have constraints that certain vertices must be assigned to the same slot. This is for throughput concerns and will be explained in Section 4.4.

Goal: Assign each $v \in V$ to one of the slots such that (1) the resource utilization ratio[2] of each slot is below the given limit; (2) the cost function is minimized. We choose the total number of slot-crossings as the cost instead of the total estimated wire lengths. Specifically, the cost function is defined as

$$\sum_{e_{ij} \in E} e_{ij}.width \times (|v_i.row - v_j.row| + |v_i.col - v_j.col|) \tag{4.1}$$

where $e_{ij}.width$ is the bit width of the FIFO channel connecting v_i and v_j and module v is assigned to the $v.col$-th column and the $v.row$-th row. The physical meaning of the cost function is the sum of the number of slot boundaries that every wire crosses.

4.3.3 Solution

Our problem is small in size as HLS-level FPGA designs seldom have more than a few hundred functions. We adopt the main idea of top-down partitioning-based placement algorithms [Bre77, DK85, MAB03] to solve our problem. Meanwhile, due to the relatively

[1] Inlined functions will be merged accordingly in the C++ front-end processing.

[2] Based on the estimation of resource utilization by HLS.

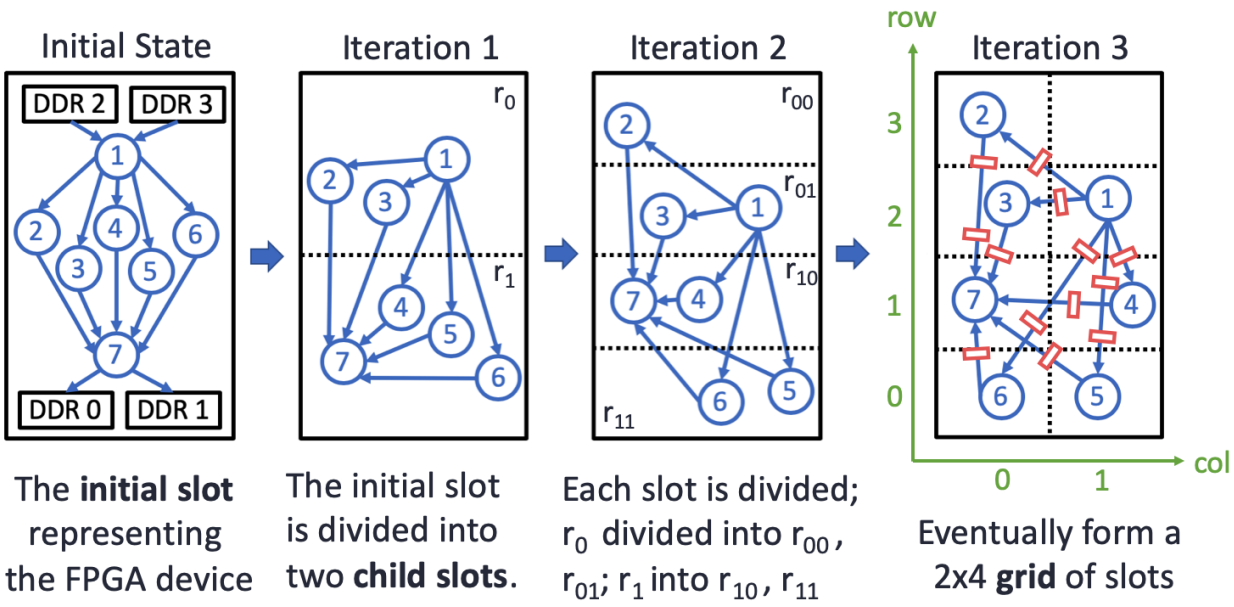


Figure 4.6: Generating the floorplan for a target 2×4 grid. Based on the floorplan, all the cross-slot connections will be accordingly pipelined (marked in red) for high frequency.

small problem size, we plan to pursue an exact solution for each partitioning.

Figure 4.6 demonstrates the floorplanning of an example design through three iterations of partitioning. The top-down partitioning-based approach starts with the initial state where all modules are assigned to the same slot, iteratively partitions the current slots in half into two ***child slots*** and then assigns the modules into the child slots. Each partitioning involves splitting all of the current slots in half either horizontally or vertically.

Since the problem size is relatively small, we formulate the partitioning process of each iteration using integer linear programming (ILP). In every partitioning iteration, all current slots need to be divided in half. Since some of the modules in a slot may be tightly connected to modules outside of the slot, ignoring such connections can adversely affect the quality of the assignment. Therefore our ILP formulation considers the partitioning of all slots together for an exact solution which is possible due to the small problem size. Experiments in Section 4.5 show that our ILP formulation is solvable within a few seconds or minutes for designs of hundreds of modules.

Performing an N-way partitioning is another potential method. However, compared to our iterative 2-way partitioning, experiments show that it is much slower than iterative 2-way

partitioning.

ILP Formulation of One Partitioning Iteration.

The formulation declares a binary decision variable v_d for each v to denote whether v is assigned to the left or the right child slot during a vertical partitioning (or to the upper or the lower child slot for a horizontal one). Let R denote the set of all current slots. For each slot $r \in R$ to be divided, we use r_v to denote the set of all vertices that r is currently accommodating. To ensure that the child slots have enough resources for all modules assigned to them, the ILP formulation imposes the resource constraint for each child slot r_{child} and for each type of on-chip resource.

$$\sum_{v \in r_v} v_d \times v_{area} < (r_{child})_{area}$$

where v_{area} is the resource requirement of v and $(r_{sub})_{area}$ represents the available resources in the child slot divided from r.

To express the cost function that is based on the coordinates of each module, we first need to express the new coordinates $(v.row, v.col)$ of v based on the previous coordinates $((v.row)_{prev}, (v.col)_{prev})$ and the decision variable v_d. For vertical partitioning, the new coordinates of v will be

$$v.col = (v.col)_{prev} \times 2 + v_d$$

$$v.row = (v.row)_{prev}$$

And for horizontal partitioning, the new coordinates will be

$$v.row = (v.row)_{prev} \times 2 + v_d$$

$$v.col = (v.col)_{prev}$$

Finally, the objective is to minimize the total slot-crossing shown in Formula (4.1) for each partitioning iteration.

For the example in Figure 4.6, Table 4.1 shows the *row* and *col* indices of selected vertices in each partitioning iteration.

Table 4.1: Coordinates of selected vertices in Figure 4.6.

	v_2	v_1	v_4	v_5
Init	row = 0; col = 0			
iter-1	$v_d = 1$; row = $0 \times 2 + 1 = 1$		$v_d = 0$; row = $0 \times 2 + 0 = 0$	
iter-2	$v_d = 1$; row = $1 \times 2 + 1$	$v_d = 0$; row = $1 \times 2 + 0$	$v_d = 1$; row = $0 \times 2 + 1$	$v_d = 0$; row = $0 \times 2 + 0$
iter-3	$v_d = 0$; col = $0 \times 2 + 0$	$v_d = 1$; col = $0 \times 2 + 1$		

In summary, we will create $|V|$ binary decision variables for each type of resource, and a typical Xilinx FPGA has 5 types of resources (Flip-Flop, LUT, BRAM, DSP, URAM). We will create $|E|$ variables to express the length of each edge. For each resource type, we will have 2 area constraints for each of the sub-slots in our partition problem. The optimization goal involves the weighted sum of all edges, thus it includes $|E|$ variables. Table 4.6 shows the actual number of vertices and edges of representative real-world designs.

4.4 Floorplan-Aware Pipelining

Based on the generated floorplan, we aim to *pipeline every cross-slot connection* to facilitate timing closure.

Although HLS has the flexibility to pipeline them to increase the final ***frequency***, the additional latency could potentially lead to a large increase of the ***execution cycles***, which we need to avoid. This section presents our methods to pipeline slot-crossing connections without hurting the overall throughput of the design.

We will first focus on pipelining the dataflow designs, then extend the method to other types of HLS design. In Section 4.4.1 we introduce our approach of pipelining with latency balancing; and Section 4.4.2 presents the detailed algorithm. In Section 4.4.3 we discuss pipelining other types of HLS designs.

4.4.1 Pipelining Followed by Latency Balancing for Dataflow Designs

In our problem, an HLS dataflow design consists of a set of concurrently executed functions communicating through FIFO channels, where each function will be compiled into an RTL module controlled by a finite-state machine (FSM) [PP91]. The rich expressiveness of FSM makes it difficult to statically determine how the additional latency will affect the total execution cycles. Note that our problem is different from other simplified dataflow models such as the Synchronous Data Flow (SDF) [LM87] and the Latency Insensitive Theory (LIT) [CMS01], where the firing rate of each vertex is fixed. Unlike SDF and LIT, in our problem, each vertex is an FSM and the firing rate is not fixed and can have a complex pattern.

Therefore, we adopt a conservative approach, where we first pipeline all edges that cross slot boundaries, then balance the latency of parallel paths based on the *cut-set pipelining* [Par07]. A cut-set is a set of edges that can be removed from the graph to create two disconnected sub-graphs; and if all edges in a cut-set are of the same direction, we could add an equal amount of latency to each edge and the throughput of the design will be unaffected. Figure 4.7 (a) illustrates the idea. If we need to add one unit of latency to e_{13} (marked in red) due to the floorplan results, we need to find a cut-set that includes e_{13} and *balance* the latency of all other edges in this cut-set (marked in blue).

Since we can choose different cut-set to balance the same edge, we need to minimize the area overhead. For example, for e_{13}, balancing the `cut-set 2` in Figure 4.7 (b) costs smaller area overhead compared to `cut-set 1` in Figure 4.7 (a), as the width of e_{47} is smaller than that of e_{14}. Meanwhile, it is possible that multiple edges can be included in the same cut-set. For example, the edges e_{27} and e_{37} are both included in the `cut-set 3`, so we only need to balance the other edges in `cut-set 3` once.

Cut-set pipelining is equivalent to balancing the total added latency of every pair of ***reconvergent paths*** [Par07]. A path is defined as one or multiple concatenated edges of the same direction; two paths are reconvergent if they have the same source vertex and destination vertex. When there are multiple edges with additional latency from the floorplanning step, we

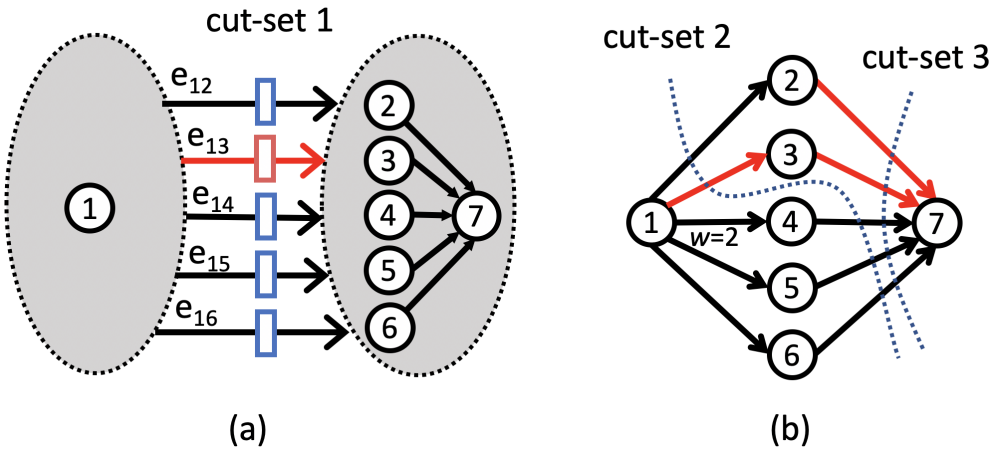


Figure 4.7: Assume that the edges e_{13}, e_{37} and e_{27} are pipelined according to some floorplan, and each of them carries 1 unit of inserted latency. Also, assume that the bit width of e_{14} is 2 and all other edges are 1. In the latency balancing step, the optimal solution is adding 2 units of latency to each of e_{47}, e_{57}, e_{67} and 1 unit of latency to e_{12}. Note that edge e_{27} and e_{37} can exist in the same cut-set.

need to find a globally optimal solution that ensures all reconvergent paths have a balanced latency, and the area overhead is minimized.

4.4.2 Latency Balancing Algorithm

Problem Formulation.

Given: A graph $G(V, E)$ representing a dataflow design that has already been floorplanned and pipelined. Each vertex $v \in V$ represents a function in the dataflow design and each edge $e \in E$ represents the FIFO channel between functions. Each edge $e \in E$ is associated with $e.width$ representing the bit width of the edge. For each edge e, the constant $e.lat$ represents the additional latency inserted to e in the previous pipelining step. We use the integer variable $e.balance$ to denote the number of latency added to e in the current latency balancing step.

Goal: (1) For each edge $e \in E$, compute $e.balance$ such that for any pair of reconvergent paths $\{p_1, p_2\}$, the total latency on each path is the same:

$$\sum_{e \in p_1} (e.lat + e.balance) = \sum_{e \in p_2} (e.lat + e.balance)$$

and (2) minimize the total area overhead, which is defined as:

$$\sum_{e \in E} e.balance \times e.width$$

Note that this problem is different from the min-cut problem [Cut20] for DAG. One naïve solution is to find a min-cut for every pipelined edge and increase the latency of the other edges in the cut accordingly. However, this simple method is suboptimal. For example in Figure 4.7, since edge e_{27} and e_{37} can be in the same cut-set, we only need to add one unit of latency to the other edges in the cut-set (e.g., e_{47}, e_{57} and e_{67}) so that all paths are balanced.

Solution.

We formulate the problem in a restricted form of ILP that can be solved in polynomial time. For each vertex v_i, we associate it with an integer variable S_i that denotes the maximum latency from pipelining between v_i and the sink vertex of the graph. In other words, given two vertices v_x and v_y, $(S_x - S_y)$ represents the maximum latency among all paths between the two vertices. Note that we only consider the latency on edges due to pipelining.

For each edge e_{ij}, we have

$$S_i \geq S_j + e_{ij}.lat$$

According to our definition, the additional balancing latency added to edge e_{ij} in this step can be expressed as

$$e_{ij}.balance = (S_i - S_j - e_{ij}.lat)$$

since we want every path from v_i to v_j have the same latency.

The optimization goal is to minimize the total area overhead, i.e. the weighted sum of the additional depth on each edge:

$$minimize \sum_{e_{ij} \in E} e_{ij}.balance \times e_{ij}.width$$

In total, for a graph $G(V, E)$, we introduce $|V|$ variables and $|E|$ constraints.

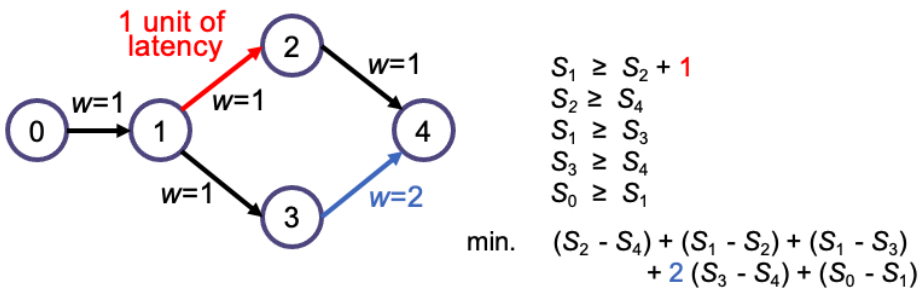


Figure 4.8: Example SDC formulation for the latency balancing problem.

For example, assume that there are two paths from v_1 to v_2 where path p_1 has 3 units of latency from pipelining while p_2 has 1 unit. Thus from our formulation, we will select the edge(s) on p_2 and add 2 additional units of latency to balance the total latency of p_1 and p_2 so that the area overhead is minimized. Figure 4.8 provides an example to illustrate the formulation. In this graph, assume that we need to add one unit of latency to edge e_{12} based on the floorplanning results. Therefore, for edge e_{12} we have the constraints $S_1 \geq S_2 + 1$. In the optimization target, since the edge e_{34} has a width of 2 while all other edges have a width of 1, the item $(S_3 - S_4)$ has the coefficient 2.

Our formulation is essentially a system of differential constraints (SDC), in which all constraints are in the form of $x_i - x_j \leq b_{ij}$, where b_{ij} is a constant and x_i, x_j are variables. Because of this restrictive form of constraint, we can solve SDC as a linear programming problem while the solutions are guaranteed to be integers. As a result, it can be solved in polynomial time [LS91, CZ06].

If the SDC formulation does not have a solution, there must be a dependency cycle in the dataflow graph [CZ06]. This means that at least one of the edges in the dependency cycle is pipelined based on the floorplan. In this situation, we will feedback to the floorplanner to constrain those vertices into the same region and then re-generate a new floorplan.

4.4.3 Extension to Non-Dataflow Designs

In the previous subsections, we focus on pipelining and latency balancing for dataflow designs as they can be easily pipelined. However, our methodology applies to other types of HLS

designs as well. Since most interface protocols of HLS-generated modules have pre-determined operation latency, we can accurately predict at compile time whether the additional latency on certain interfaces will cause throughput degradation, in which case we will adjust the constraints for the floorplanning step.

When a C++ function is compiled into an RTL module, the arguments to the function become ports of the module with IO protocols according to the type of C++ arguments. Here we discuss how to add latency to interfaces with Vivado HLS [Xil20b] designs, but the concept and implementation are similar in other HLS compilers. Besides the FIFO interface, there are four major types of ports on RTL modules generated by HLS:

- `Control signals`. For example, the `start`, `ready`, and `done` indicate when the module starts executing and whether it has finished. They can be directly pipelined without influencing functionality. We require that the function is not invoked inside a loop to prevent the added latency from increasing the initiation interval of the loop.

- `Scalar` or `input pointer`. By default, the pass-by-value input arguments and pointers are implemented as simple input wire ports. They can be directly pipelined, and the `start` should be pipelined accordingly.

- `Output pointers`. These are implemented with an associated output valid signal to indicate when the output data is valid. We can directly pipeline the output signals along with the valid signals.

- `Array` arguments. The compiler will compile them into a standard block RAM interface with data, address, chip-enable, and write-enable ports. For such an interface, the configuration option specifies the read or write latency of the RAM resource driving the interface, which is known at compile time. Adding pipelining to all signals of the RAM interface will change the latency of RAM access operations, thus we require that the array should only be accessed inside a pipelined loop, where increasing the latency of the RAM operation will not increase the initiation interval of the pipeline. Figure 4.9 visualizes the process. Assuming the module **foo** sends out an address and the module

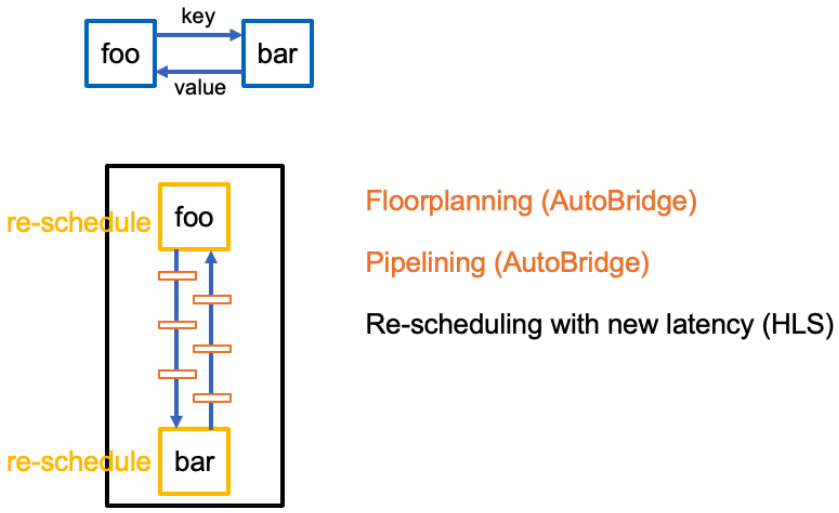


Figure 4.9: Example of AutoBridge on a key-value store.

bar returns the value, we could pipeline the two connections. Since the latency of the key-value store operation has changed, we could go back and redo the scheduling process to apply the change.

In addition, we must re-run HLS synthesis for each function after annotating the new interface latency in the source code. In comparison, this is not needed for dataflow designs with latency-insensitive interfaces.

4.5 Experiments

4.5.1 Implementation Details

We implement our proposed methods in Python interfaced with the CAD flow for Xilinx FPGAs, including Vivado HLS, Vivado, and Vitis (2019.2). We parse the scheduling and binding reports of dataflow HLS designs to create the graph representation of the design and obtain the resource utilization of each RTL module. We use the Python MIP package [ST20] coupled with Gurobi [Gur20] to solve the various ILP problems introduced in previous sections. We generate TCL constraint files to be used by Vivado to enforce our high-level floorplanning scheme. Our RTL generator parses the RTL from Vivado HLS using PyVerilog [Tak15], then

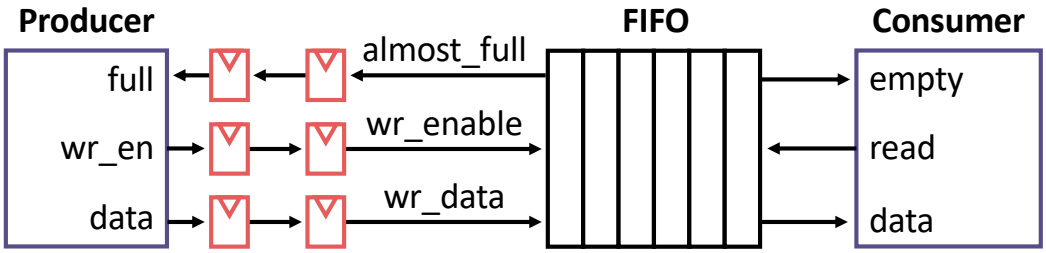


Figure 4.10: Pipelining FIFO interfaces using almost-full FIFOs.

traverses the AST to add the additional pipelining and regenerate the optimized RTL.

We mainly implement the AutoBridge prototype for Vivado HLS dataflow designs, where the top function instantiates all the dataflow processes and the FIFO connections. In addition, we support the TAPA compiler [CGC20, GCL22], which serves as a front-end to the existing HLS tools to enable more expressibility over task-level parallel programs [DLN21]. We also include tools to process non-dataflow designs and some manual help is necessary due to the limited access to the internals of the HLS compiler. A certain coding style is expected and we provide examples in our open-sourced repository.

Figure 4.10 shows how we add pipelining to a FIFO-based connection. We adopt FIFOs that assert their `full` pin before the storage actually runs out, so that we could directly register the interface signals without affecting the functionality.

Meanwhile, we turn off the hierarchy rebuild process during RTL synthesis [Xil20a] to prevent the RTL synthesis tool from introducing additional wire connections between RTL modules. The hierarchy rebuild step first flattens the hierarchy of the RTL design and then tries to rebuild the hierarchy. As a result, hierarchy rebuild may create unpredictable new connections between modules. As a result, if two modules are floorplanned far apart, these additional wires introduced during RTL synthesis will be under-pipelined as they are unseen during HLS compilation. Note that disabling this feature may lead to slight differences in the final resource utilization.

We test out designs on the Xilinx Alveo U250 FPGA[3] with 4 DRAMs and the Xilinx

[3]The U250 FPGA contains 5376 BRAM18K, 12288 DSP48E, 3456K FF, and 1728K LUT

Alveo U280 FPGA[4] with High-Bandwidth Memory (HBM). As the DDR controllers are distributed in the middle vertical column while the HBM controller lies at the bottom row, these two FPGA architectures present different challenges to the CAD tools. Thus it is worthwhile to test them separately.

To run our framework, users first specify how they want to divide the device. By default, we divide the U250 FPGA into a 2-column × 4-row grid and the U280 FPGA into a 2-column × 3-row grid, matching the block diagram of these two architectures shown in Figure 4.3. To control the floorplanning, users can specify the maximum resource utilization ratio of each slot. The resource utilization is based on the estimation by HLS. Users can also specify how many levels of pipelining to add based on the number of boundary crossings. By default, for each boundary crossing, we add 2 levels of pipelining to the connection. The processed design is integrated with the Xilinx Vitis (2019.2) infrastructure to communicate with the host. A snapshot of the original evaluated artifact is available [GCW21b].

4.5.2 Benchmarks

We use six representative benchmark designs with different topologies and change the parameter of the benchmarks to generate a set of designs with varying sizes on both the U250 and the U280 board. The six designs are all large-scale designs implemented and optimized by HLS experts. Figure 4.11 shows the topology of the benchmarks. Note that even for those benchmarks that seem regular (e.g., CNN), the location constraints from peripheral IPs can highly distort their physical layouts.

- The stencil designs created by the SODA [CCW18] compiler have a set of kernels in linear topologies.
- The genome sequencing design [GLR19] performing the Minimap2 overlapping algorithm [Li18] has processing elements (PE) in broadcast topology. This benchmark is based on shared-memory communication and all other benchmarks are dataflow designs.

[4]The U280 FPGA contains 4032 BRAM18K, 9024 DSP48E, 2607K FF and 434K LUT

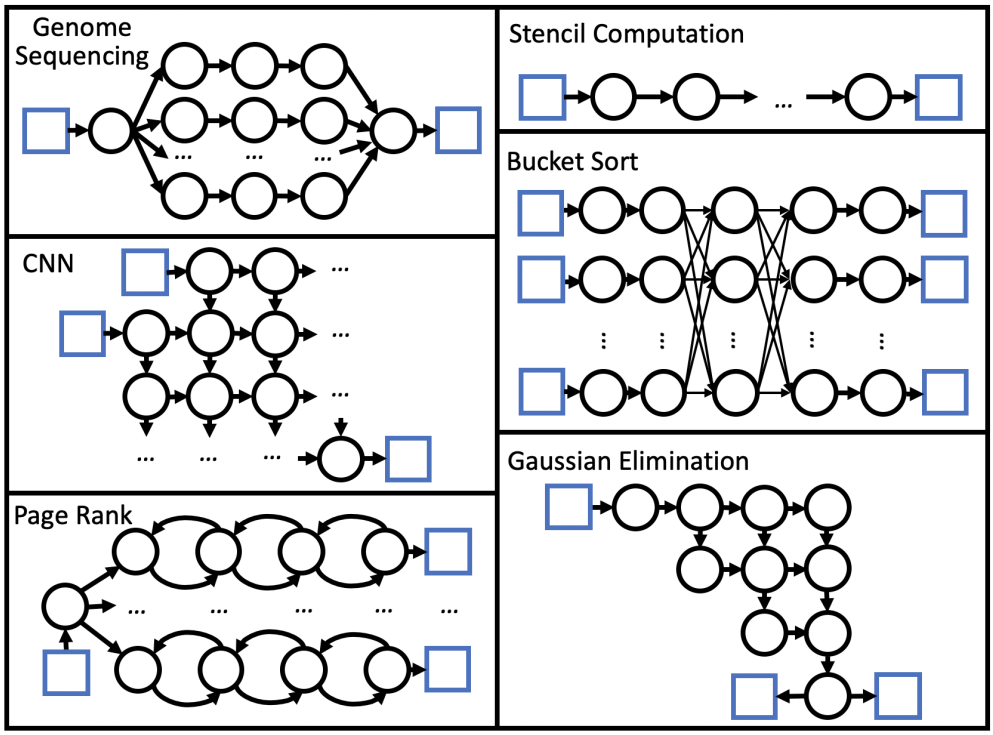


Figure 4.11: Topologies of the benchmarks. Blue rectangles represent external memory ports and black circles represent the computation kernels of the design. In the genome sequencing design, the arrows represent BRAM channels; in other designs, the arrows represent FIFO channels.

- The CNN accelerators created by the PolySA [CW18] compiler are in a grid topology.
- The HBM graph processing design [CGC20] performs the page rank algorithm. It features eight sets of processing units and one central controller. This design also contains dependency cycles, if viewed at the granularity of computing kernels.
- The HBM bucket sort design adapted from [SQA20, QOG21] which includes 8 parallel processing lanes and two fully-connected layers.
- The Gaussian elimination designs created by the AutoSA [WGC21] compiler are in triangle topologies.

4.5.3 Frequency Improvements

By varying the size of the benchmarks, in total, we have tested the implementation of 43 designs with different configurations. Among them, 16 designs failed in routing or placement with the baseline CAD flow, compared AutoBridge which succeeds in routing all of them and achieves an average of 274 MHz. For the other 27 designs, we improve the final frequency from 234 MHz to 311 MHz on average. In general, we find that AutoBridge is effective for designs that use up to about 75% of the available resources. We execute our framework on an Intel Xeon CPU running at 2.2GHz. Both the baseline designs and optimized ones are implemented using Vivado with the highest optimization level. The final checkpoints of all experiments are available in our open-sourced repository.

In some experiments, we may find that the optimized versions have even slightly smaller resource consumption. Possible reasons are that we adopt a different FIFO template and disable the hierarchy rebuild step during RTL synthesis. Also, as the optimization leads to very different placement results compared to those of the original version, we expect different optimization strategies will be adopted by the physical design tools. The correctness of the code is verified by cycle-accurate simulation.

Next, we present the detailed results of each benchmark.

Stencil Computation.

For the stencil computing design, the kernels are connected in a chain format through FIFO channels. By adjusting the number of kernels, we can vary the total size of the design. We test anywhere from 1 kernel up to 8 kernels, and Figure 4.12 shows the final frequency of the eight design configurations on both U250 and U280 FPGAs. In the original flow, many design configurations fail in routing due to routing resource conflicts. Those that are routed successfully still achieve relatively low frequencies. In comparison, with the help of AutoBridge, all design configurations are routed successfully. On average, we improve the timing from 86 MHz to 266 MHz on the U280 FPGA, and from 69 MHz to 273 MHz on the U250 FPGA.

Starting from the 7-kernel design, we observe a frequency decrease on the U280 FPGA.

This is because each kernel of the design is very large and uses about half the resources of a slot; thus starting from the 7-kernel design on the relatively small U280, two kernels have to be squeezed into one slot which will cause more severe local routing congestion. Based on this phenomenon, we recommend that users avoid designing very large kernels and instead split the functionality into multiple functions to allow the tool more flexibility in floorplanning the design.

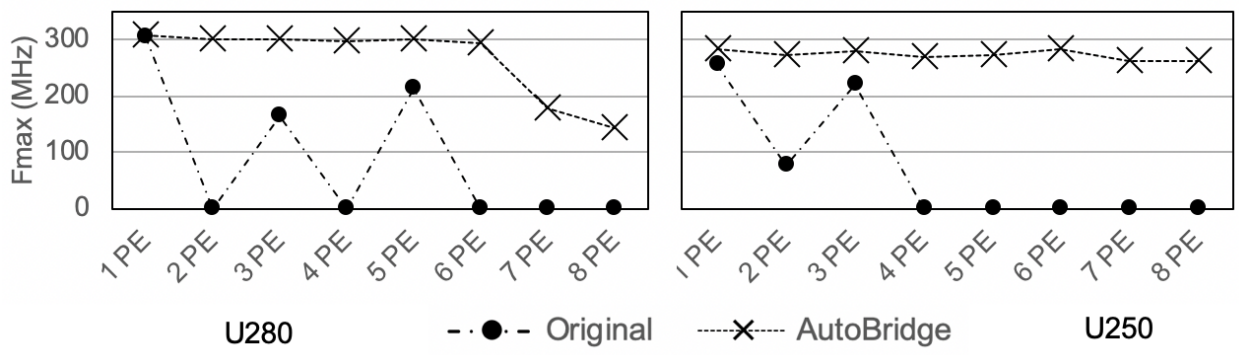


Figure 4.12: Results of the stencil computation designs.

CNN Accelerator.

The CNN accelerator consists of identical PEs in a regular grid topology. We adjust the size of the grid from a 2×13 array up to a 16×13 array to test the robustness of AutoBridge. Figure 4.13 shows the result on both U250 and U280 FPGAs.

Although the regular 2-dimensional grid structure is presumed to be FPGA friendly, the actual implementation results from the original tool flow are not satisfying. With the original tool flow, even small-size designs are bounded at around 220 MHz when targeting U250. Designs of larger sizes will fail in placement (13×12) or routing (13×10 and 13×14). Although the final frequency is high when the design is small for the original tool flow targeting U280, the timing quality is steadily dropping as the designs become larger.

In contrast, AutoBridge improves from 140 MHz to 316 MHz on U250 on average, and from 214 MHz to 328 MHz on U280. Table 4.2 lists the resource consumption and cycle counts of the experiments on U250. Statistics on U280 are similar and are omitted here.

Table 4.2: Post-placement results of the CNN designs on U250. The design point of 13×12 failed placement and 13×10 and 13×14 failed routing with the original tool flow.

Size	LUT(%)		FF(%)		BRAM(%)		DSP(%)		Cycle	
	orig	opt	orig	opt	orig	opt	orig	opt	orig	opt
13x2	17.82	17.90	14.11	14.25	21.69	21.67	8.57	8.57	53591	53601
13x4	23.52	23.59	18.98	19.04	25.74	25.73	17.03	17.03	68630	68640
13x6	29.26	29.24	23.86	23.80	29.80	29.78	25.50	25.50	86238	86248
13x8	34.98	34.90	28.72	28.56	33.85	33.84	33.96	33.96	103882	103892
13x10	40.71	40.48	33.58	33.25	37.91	37.89	42.42	42.42	121472	121491
13x12	-	46.18	-	38.06	-	41.95	-	50.89	139098	139108
13x14	52.10	51.92	43.28	42.93	46.02	46.00	59.35	59.35	156715	156725
13x16	57.82	57.61	48.13	47.70	50.07	50.06	67.81	67.81	174377	174396

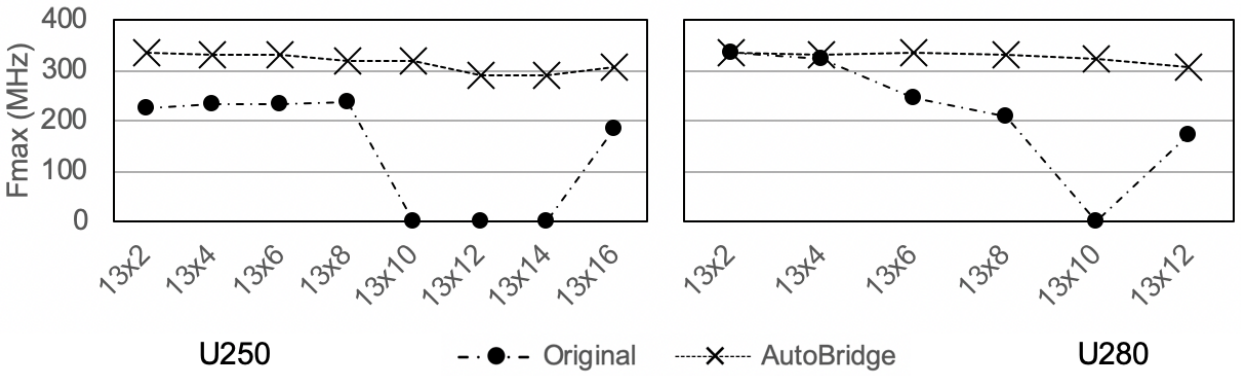


Figure 4.13: Results of the CNN accelerator designs.

Gaussian elimination.

The PEs in this design form a triangle topology. We adjust the size of the triangle and test on both U250 and U280. Table 4.3 shows the results. On average, we improve the frequency from 245 MHz to 334 MHz on U250, and from 223 MHz to 335 MHz on U280.

Table 4.3: Results of Gaussian elimination designs on U250.

Size	LUT(%)		FF(%)		BRAM(%)		DSP(%)		Cycle	
	orig	opt	orig	opt	orig	opt	orig	opt	orig	opt
12×12	18.58	18.69	13.05	13.14	13.24	13.21	2.79	2.79	758	781
16×16	26.62	26.68	17.36	17.30	13.24	13.21	4.99	4.99	1186	1209
20×20	38.55	38.28	23.46	23.38	13.24	13.21	7.84	7.84	1728	1738
24×24	54.05	53.59	32.16	32.06	13.24	13.21	11.34	11.34	2361	2375

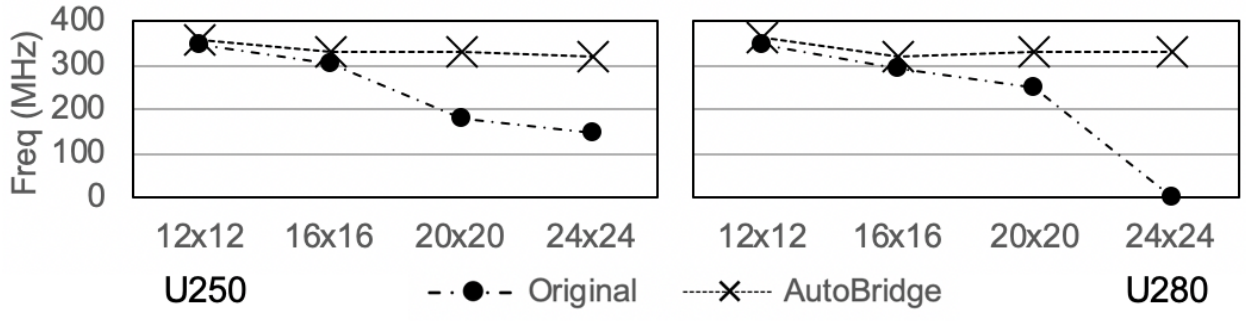


Figure 4.14: Results of the Gaussian elimination designs.

Genome Sequencing.

The genome sequencing design contains eight parallel PEs that communicate with the external memory through local buffers of a BRAM interface. The design provides parameters to adjust the computation accuracy of each PE and higher accuracy will result in a larger area. Thus, we test three design configurations where each PE is of 1×, 1.5×, and 2× the original size. For this non-dataflow design, AutoBridge first performs floorplanning and creates a wrapper for each PE to pipeline all I/O signals. Then we manually add pragmas to the source code to specify the modified latency on shared memory blocks and re-run HLS to update the internals of each PE. On average, we improve the frequency from 132 MHz to 248 MHz as in Table 4.4. When the original size of the PE is small Vivado performs well, but AutoBridge outperforms Vivado with larger PEs.

Table 4.4: Experiment result of genome sequencing on U250.

	Fmax (MHz)	LUT %	FF %	BRAM %	DSP %	Cycle (K)
Orig, Size=1	265	25.43	16.12	17.21	4.38	11710
Opt, Size=1	267	25.48	16.29	17.21	4.38	11830
Orig, Size=1.5	-	-	-	-	-	12350
Opt, Size=1.5	272	31.73	19.39	15.14	6.46	12470
Orig, Size=2	131	38.89	23.11	17.21	8.54	12990
Opt, Size=2	206	38.91	23.31	17.21	8.54	13110

HBM Bucket Sort.

The bucket sort design has two complex fully-connected layers. Each fully-connected layer involves an 8×8 crossbar of FIFO channels, with each FIFO channel being 256-bit wide. AutoBridge pipelines the FIFO channels to alleviate the routing congestion. Table 4.5 shows the frequency gain, where we improve from 255 MHz to 320 MHz on U280. As the design requires 16 external memory ports and U250 only has 4 available, the test for this design is limited to U280 only.

Because the original source code has enforced a BRAM-based implementation for some small FIFOs, which results in wasted BRAM resources, the results of AutoBridge have slightly lower BRAM and flip-flop consumption than the original implementation. In comparison, we use a different FIFO template that chooses the implementation style (BRAM-based or shift-register-based) based on the area of the FIFO. Cycle-accurate simulation has proven the correct functionality of our optimized implementation.

Table 4.5: Results of the bucket sort designs on U280.

	Fmax (MHz)	LUT %	FF %	BRAM %	DSP %	Cycle
Original	255	28.44	19.11	16.47	0.04	78629
Optimized	320	29.39	16.66	13.69	0.04	78632

4.5.4 Control Experiments

First, we test whether the frequency gain comes from the combination of pipelining and HLS-floorplanning, or simply pipelining alone. To do this, we set a control group where we perform floorplanning and pipelining as usual, but we do not pass the floorplan constraints to the physical design tools. The blue curve with triangle markers in Figure 4.15 shows the results. As can be seen, the control group has a lower frequency than the original design for small sizes and has limited improvements over the original designs for large sizes. In all experiments, the group with both pipelining and floorplan constraints (green curve with crossing markers) has the highest frequency. This experiment proves that the frequency gain

is not simply a result of more pipelining.

Meanwhile, if we only do floorplanning without pipelining, obviously the frequency will be much degraded, as visualized by Fig. 4.4.

Second, we test the effectiveness of setting a slot boundary based on the DDR controllers. We run a set of experiments where we only divide the FPGA into four slots based on the die boundaries, minus the division in the middle column. The yellow curve with diamond markers in Figure 4.15 shows the results. As can be seen, it achieves lower frequency compared to our default eight-slot scheme.

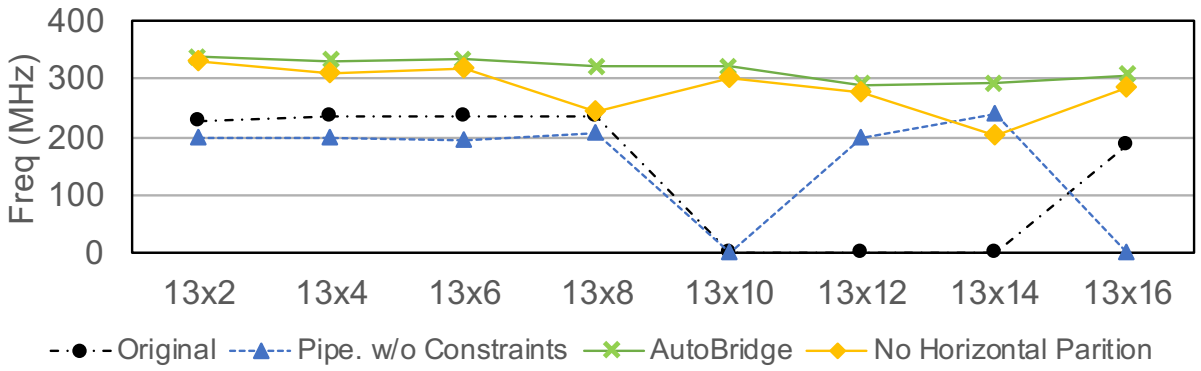


Figure 4.15: Control experiments with the CNN accelerators.

4.5.5 Scalability

To show that the tool works well on designs with large numbers of small functions, we utilize the CNN experiments to test the scalability of our algorithms, as the CNN designs have the most vertices (HLS functions) and edges. Table 4.6 lists The compile time overhead for the floorplanning and the latency balancing when using Gurobi as the ILP solver[5]. For the largest CNN accelerator that has 493 modules and 925 FIFO connections, the floorplan step only takes around 20 seconds and the latency balancing step takes 0.03s. Usually, FPGA designs are not likely to have this many modules and connections [Lib20] [Exa20], and our method is fast enough.

[5]Meanwhile, we observed that many open-sourced ILP solvers are much slower.

Table 4.6: Computing time for the CNN test cases targeting the U250 FPGA. *Div-1* and *Div-2* denote the first and the second vertical decomposition, and *Div-3* denotes the first horizontal decomposition. *Re-balance* denotes the delay balancing.

Size	# V	# E	Div-1	Div-2	Div-3	Re-balance
13 × 2	87	141	0.02 s	0.02 s	0.01 s	<0.01 s
13 × 4	145	253	0.05 s	0.02 s	0.20 s	<0.01 s
13 × 6	203	365	0.07 s	1.02 s	0.56 s	<0.01 s
13 × 8	261	477	0.07 s	1.07 s	3.58 s	0.01 s
13 × 10	319	589	3.17 s	1.61 s	2.63 s	0.01 s
13 × 12	377	701	3.42 s	1.43 s	9.84 s	0.01 s
13 × 14	435	813	3.54 s	1.55 s	6.18 s	0.03 s
13 × 16	493	925	4.95 s	2.02 s	12.56 s	0.03 s

4.6 Conclusions

We propose to couple coarse-grained floorplanning with pipelining to improve the frequency of the HLS designs on multi-die FPGAs. Our methodology has two key advantages: (1) it helps HLS identify and pipeline the long wires, especially those that will cross die boundaries; (2) it further reduces local routing congestion since early floorplanning can distribute the logic across multiple dies. According to our evaluation of 43 realistic benchmarks, our framework effectively improves the average frequency from 147 MHz to 297 MHz without compromising the throughput of the design.

We are extending our methodology to support latency-insensitive RTL designs as well. For example, automatic domain-specific generation framework like DSAGEN [WLD20, LWK22] opens a new research topic on building overlay for coarse-grain accelerators on FPGA. We adopt DSAGEN as an application driver to study the extension of AutoBridge to latency-insensitive RTL designs. The extensions of AutoBridge to support general-purpose RTL designs and ASIC remain as future work.

CHAPTER 5

Parallel Physical Implementation of HLS Designs for Fast Timing Closure.

FPGAs require a much longer compilation cycle than conventional computing platforms like CPUs.In this paper, 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 5-7× reduction in compile time and up to 1.3× increase in frequency when compared to 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 5-7× compile time reduction and 1.3× frequency increase.

5.1 Introduction

FPGA compilation techniques have traditionally been adopted from the EDA industry, where designers have a higher tolerance for a long turn-around time. However, this significantly impedes the adoption of FPGAs by the computing industry, where software programmers are used to a much shorter compile cycle [LUX21].

One general approach to speeding up FPGA compilation is to utilize multi-core CPUs or GPUs to parallelize the CAD algorithms, such as logic synthesis [DB94, DCR95], placement [LBP08, CZ09, LCW15, LLW17, DSI19, ASB14], and routing [Sto17, SL15, GA10, GA11, WDT17, HK18, ZVS20]. However, many important algorithms used in the FPGA CAD tool flow 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 5.1, 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.

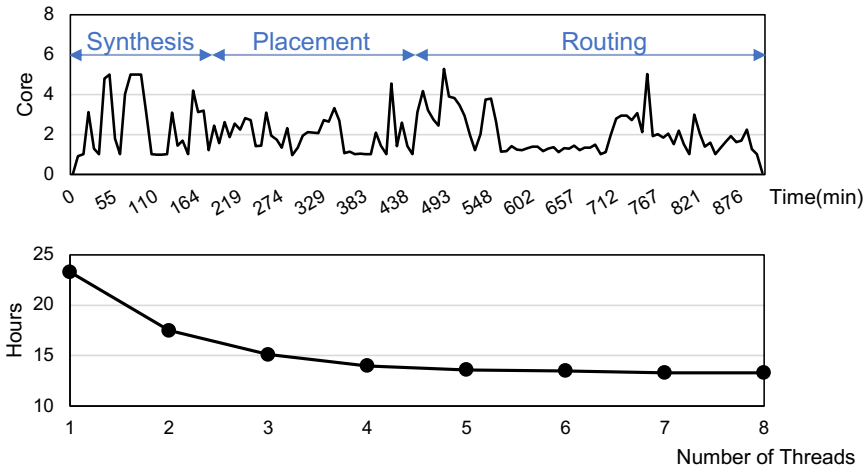


Figure 5.1: The upper figure shows the number of active CPU cores when implementing a CNN benchmark by Vivado (8 threads) on a 56-core server. The total implementation process takes about 14 hours, with an average CPU utilization of 2.1 cores. The lower figure displays the runtime as we increase the number of threads.

Another approach to fast FPGA compilation is splitting the whole application into several

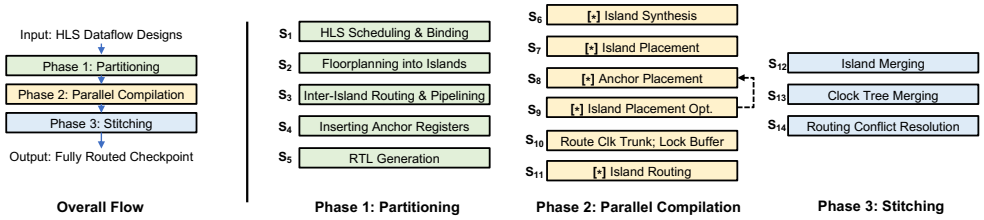


Figure 5.2: An overview of our RapidStream workflow. We use [∗] to denote a parallelized step.

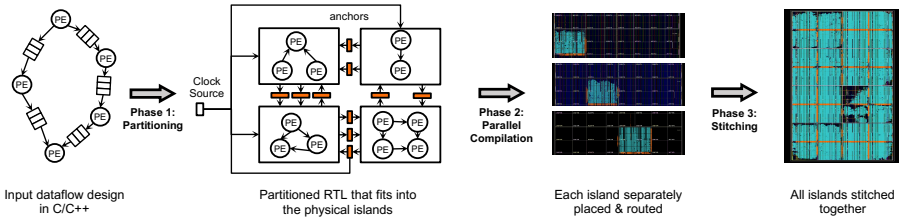


Figure 5.3: Illustration of results obtained in different phases. In the final output, the orange part shows the anchor registers, the cyan part shows the implemented partitions.

partitions and then compiling different parts in parallel. A new challenge naturally arises here — *how to achieve timing closure with many inter-partition nets*? 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 sacrifice the timing quality of inter-partition nets for runtime efficiency, rendering the acceleration less meaningful.

Our prior work, RapidStream 1.0 [GMZ22], 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 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.

As illustrated in Figure 5.3, 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 5-7× 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

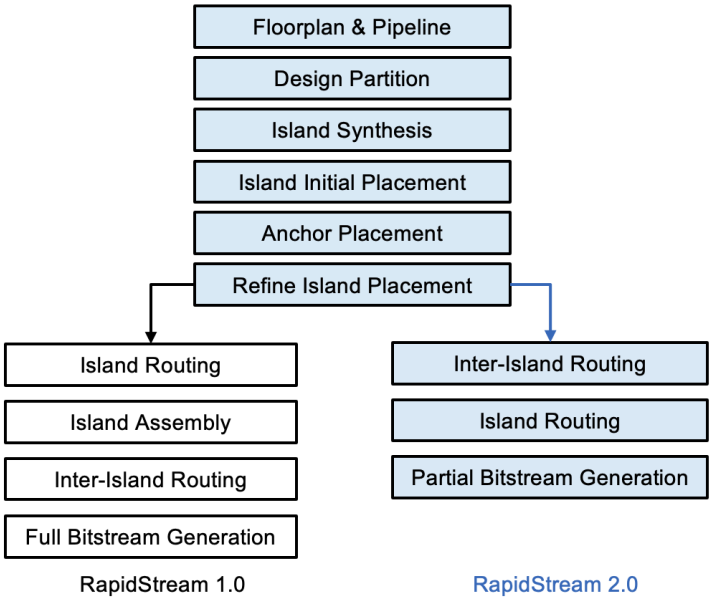


Figure 5.4: Comparison of RapidStream 1.0 and 2.0.

conflicts. While RWRoute can finish the routing task quickly and efficiently compared to Vivado, the open-source router lacks an accurate hold time model and the 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.

As an extension, we present RapidStream 2.0, which adopts a different approach to address the bottleneck in the routing step. Compared to 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, etc. 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 5-7X end-to-end speedup compared to a normal implementation process. Notably, our generated bitstream is fully functional and has passed onboard tests. Compared to RapidStream 1.0, we observe as much as 2× speedup in RapidStream 2.0.

This chapter is organized as follows. We first present the background information and the overview of our workflow in Section 5.2. Then we present the major steps that are shared between version 1.0 and 2.0, i.e. design partitioning (Section 5.3) parallel placement (Section 5.4) clock management (Section 5.5). Next, we present the routing solution of RapidStream 1.0 and 2.0, respectively. For RapidStream 1.0, we discuss island stitching and inter-island routing (Section 5.6). We present our efforts to accelerate further the inter-island routing step for RapidStream 1.0 (Section 5.6). Next, for RapidStream 2.0, we present how to set up the partial reconfiguration environment to enable parallel routing (Section 5.8). 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 the experiment results of RapidStream 1.0 in Section 5.10, and then we evaluate RapidStream 2.0 in Section 5.11. We also compare with related works in Section 2.5.

5.2 Preliminaries

5.2.1 Problem Scope

RapidStream focuses on latency-insensitive FPGA designs. By our definition, a latency-insensitive design consists of (1) a collection of *processing elements (PE)* 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.

5.2.2 Organization of the FPGA Fabric

To facilitate the split compilation, we divide the FPGA fabric into two types of regions. As illustrated in Figure 5.5, these regions include (1) large disjoint *islands* (in blue) that are equally 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. Meanwhile, 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 5.5).

5.2.3 Flow Overview

Figure 5.3 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 5.2). 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.

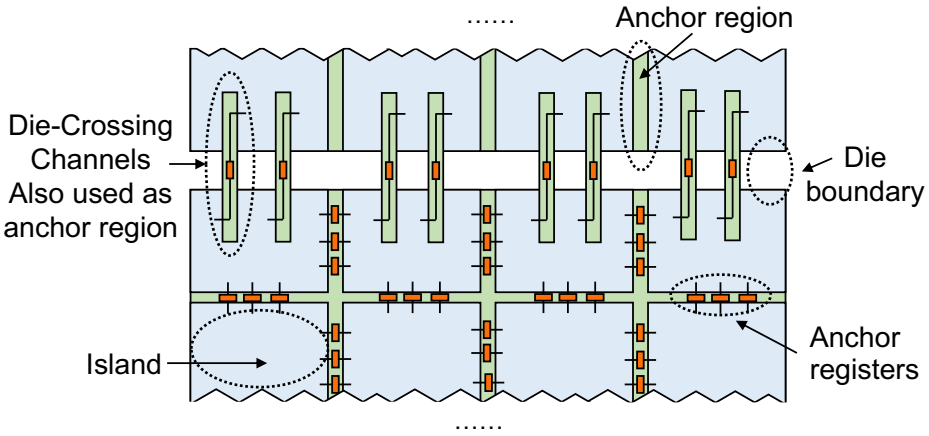


Figure 5.5: Organization of the FPGA device.

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 [LK18] to stitch the physical netlists of post-routing islands together (S_{12}, S_{13}). Although the nets inside each island remain legal 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 to the full-fledged commercial router, we achieve a 4× speedup on average while retaining nearly the same setup slacks.

5.3 Partitioning

This section describes steps S_1-S_5 of the partitioning phase of RapidStream, as shown in Figure 5.2.

5.3.1 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.

5.3.2 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:

$$\sum_{e_{ij} \in E} e_{ij}.width \times (|v_i.row - v_j.row| + |v_i.col - v_j.col|) \tag{5.1}$$

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 [GCW21a, Bre77, DK85, MAB03].

Global Planing & 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 5.6(A) shows the potential routes (P_1, P_2, P_3) for an connection between two non-adjacent islands.

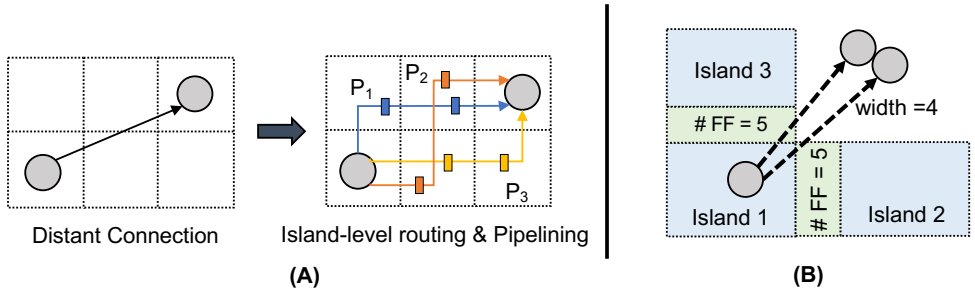


Figure 5.6: (A) three potential routes for a connection. (B) Each anchor region (in green) only has 5 Flip-Flops, so the two connections (both of width 4) cannot go through the same anchor region.

The main constraint in this routing problem is the number of available flip-flops (FFs) in the anchor regions. Recall in Figure 5.5 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 the participating anchor regions have sufficient FFs for pipelining all the nets passing through, as illustrated in Figure 5.6(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 5.7 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 the tolerance of the round-trip latency between adjacent islands, which allows us to add a pipeline register without causing an overflow.

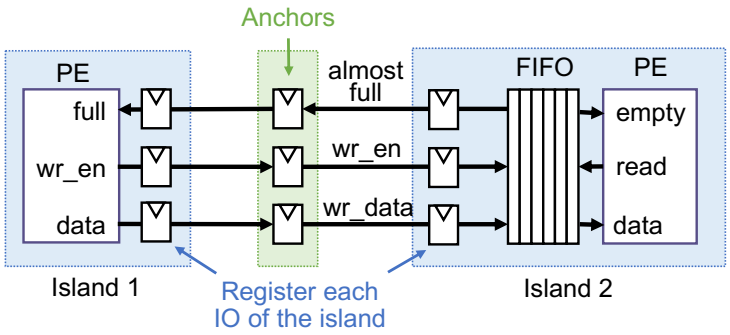


Figure 5.7: Inserting anchor registers.

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 5.10. For future FPGA designs that may become much larger, we can incorporate other well-known techniques such as multi-level placement [CCS05] and hierarchical routing [YVA07] to handle the increased complexity.

5.4 Parallel Placement

Phase 1 produces an optimized version of the RTL that is floorplanned to the island regions and anchor regions (Fig. 5.5). In step S_2, we determine which PEs are assigned to each island region; and 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).

5.4.1 Iterative Placement of Anchors and Islands

Compared to 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 5.8 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 the 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. So 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 5.4.2.

Iteration 3 (S_9). We fine-tune the placement of islands based on the exact anchor locations.

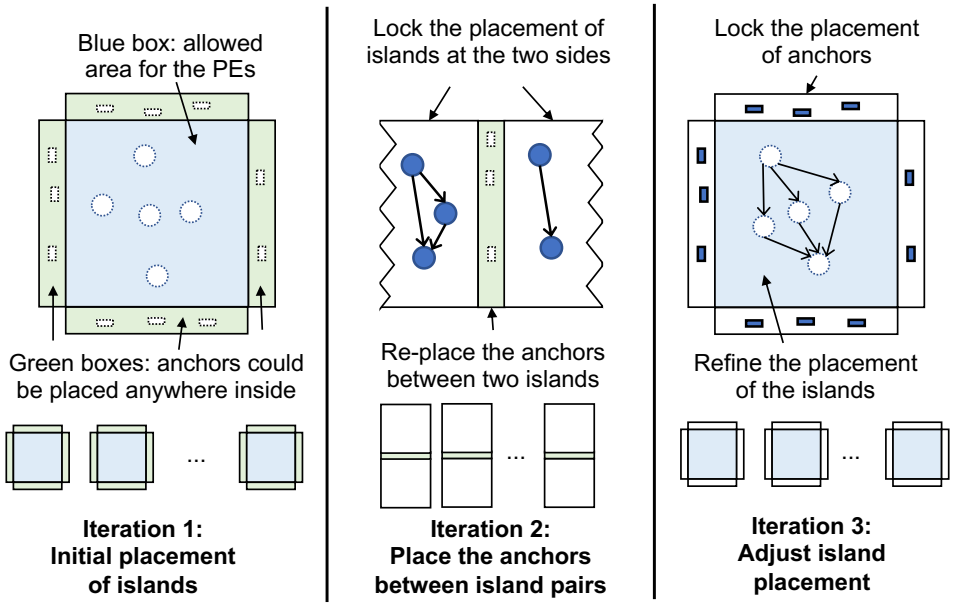


Figure 5.8: Demonstration of the iterative placement.

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.

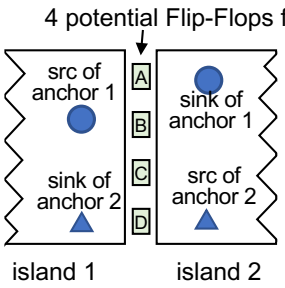
Through the three iterations, all islands are placed in a parallel manner. It is possible to repeat iteration 2 (S_8) and iteration 3 (S_9) to further improve the overall timing quality. However, our experiments indicate that applying them just once already achieves a post-placement frequency of 400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

5.4.2 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

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

to make sure that all anchors are exactly inside the feasible regions.



	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

Figure 5.9: Illustration of the anchor placement formulation.

Method. We propose a simple yet effective distance-driven placement formulation specifically for iteration 2 (S_8), which can achieve a similar timing quality compared to a standard placer but with a much shorter running time. 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 [Che10]. Specifically, we formulate the problem in linear programming (LP), which in this case guarantees integer solutions because the constraint matrix is totally unimodular [JZP08].

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; (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 5.9, 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.

5.5 Clock Routing

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

5.5.2 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 to 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 [Xil21b]. 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 on in the following subsections. Figure 5.10 visualizes the key concepts in our clock management scheme.

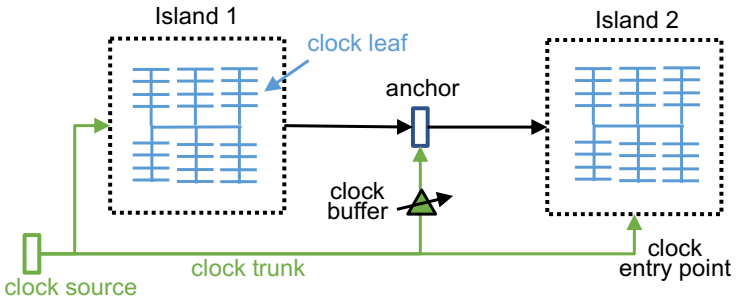


Figure 5.10: Route different segments of the clock separately and maintain a stable clock skew in one pass. Step 1: route the clock trunk. Step 2: lock the delay level of the clock buffers for anchors. Step 3: route each island and merge with the clock trunk.

5.5.3 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 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.

5.5.4 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 [Fis90, YM05, DL09] 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 [Xil21a, KS16]. The time-borrowing optimization can utilize the configurable buffers to redistribute the timing slack between consecutive pipeline stages, as demonstrated by Figure 5.11.

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

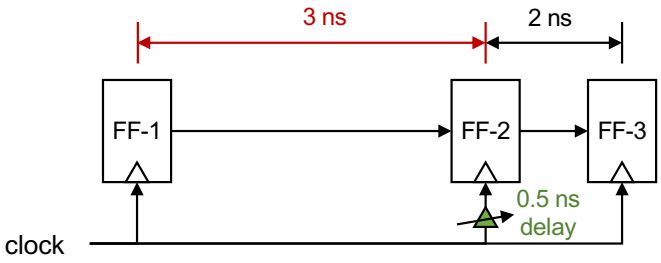


Figure 5.11: By introducing an artificial clock delay of 0.5 ns to FF-2, the critical path is reduced from 3 ns to 2.5 ns.

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; (2) the local placement optimization performed after fixing the anchor locations (S_8).

5.5.5 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 pre-determined 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

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

clock skew for inter-island timing paths is also stable. Section 5.10 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.

5.6 Stitching and Inter-Island Routing

5.6.1 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 off-the-shelf commercial tools. We utilize the open-source RapidWright framework [LK18] to edit the netlists and assemble the physical information of the island checkpoints. RapidWright is an open-source framework that enables netlist and implementation manipulation of modern AMD/Xilinx FPGA and SoC designs.

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 5.4), we can safely merge them without causing conflicts in anchor locations. Further, our clock routing scheme (Section 5.5) ensures that different islands are routed under the same clock trunk, thus the clock net can also be merged without conflicts (S_{13}).

5.6.2 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 5.12 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 5-10% of all the nets. Those conflicts will be exposed after we glue the post-routing checkpoints of the islands together.

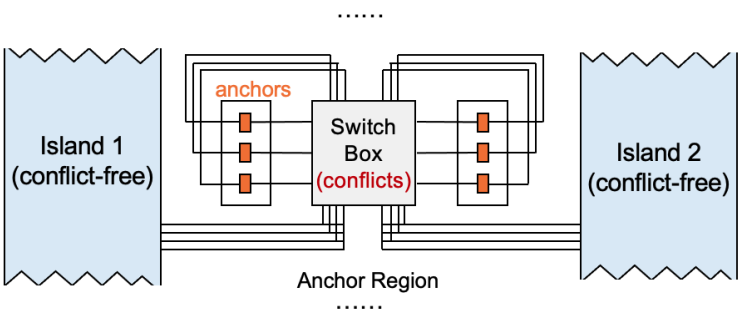


Figure 5.12: Detailed view of anchor region. Only one switch box is shown.

One potential solution is to resolve the inter-island conflicts pair by pair. Figure 5.13 illustrates why this will not work. In Figure 5.13 we could try to separately re-route the conflict nets between islands (1, 2) and between islands (2, 3). However, while pairwise re-routing resolves the anchor region conflicts, it will lead to new conflicts within the islands. In Figure 5.13, assume the black and the yellow net are separately routed by two router processes, conflicts may show up inside the islands (the red segment).

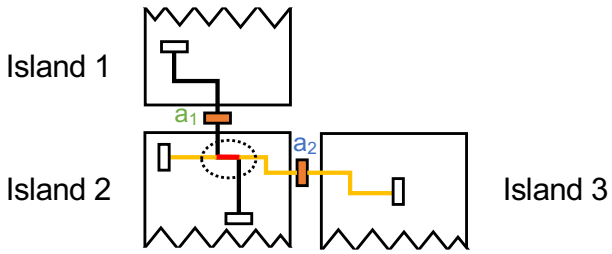


Figure 5.13: Pairwise inter-island routing will not work because it may cause conflicts inside the island.

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, while the other solution relies on a customized open-source routing solution for the best compile time.

Commercial routers can resolve 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 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, 90-95% 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.

5.7 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. Meanwhile, the partial router preserves other fully routed nets, i.e., 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; while a small bounding box limits the routability but also reduce 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 [GA11, LGW14, WDT17, HK18, ZVS20]. 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 5.14 shows a case where 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 where 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.

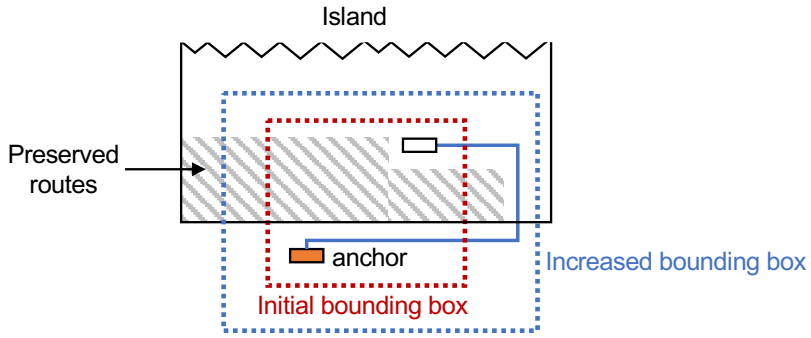


Figure 5.14: Required long routing detours outside of the initial net bounding box.

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.

We achieve the goal by customizing an open-source router called RWRoute [ZML21]. 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 to the Vivado router.

As of now, RWRoute relies on an open timing model [MNK19] 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, e.g., in Figure 5.7, 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.

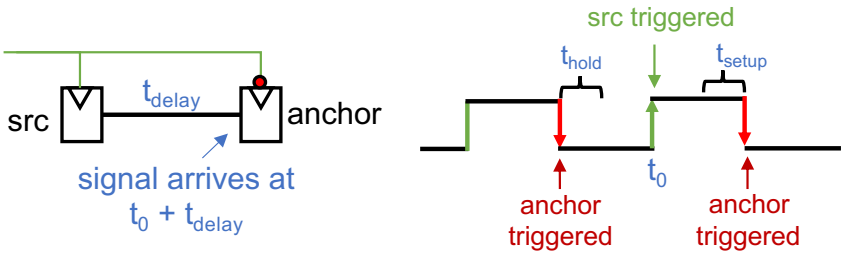


Figure 5.15: Make anchors trigger on negative clock edges.

Figure 5.15 depicts the idea 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 Equation (5.2) and (5.3) must be satisfied.

$$t_0 + t_{slow\_delay} < t_0 + t_{period}/2 - t_{setup} \tag{5.2}$$

$$t_0 + t_{fast\_delay} > t_0 - t_{period}/2 + t_{hold} \tag{5.3}$$

Equation (5.2) and (5.3) can be reduced to (5.4) and (5.5):

$$t_{period} > 2(t_{setup} + t_{slow\_delay}) \tag{5.4}$$

$$t_{period} > 2(t_{hold} - t_{fast\_delay}) \tag{5.5}$$

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 re-routes 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.

5.8 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 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 at the beginning. Figure 5.16 shows the process.

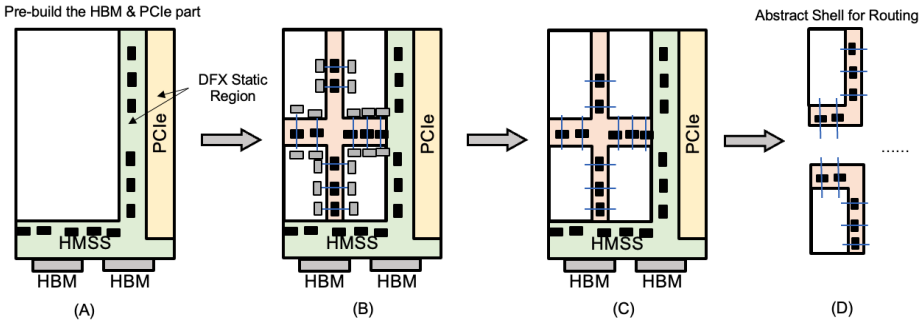


Figure 5.16: Partial routing of the inter-island nets using a skeleton design. We first do a complete routing of the nets from the anchor registers to the source/sink cells inside the island, then we prune away most routing nodes inside the island and leave the net in an antenna state. The endpoints 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.

Figure 5.16(A) shows the pre-built shell that includes the PCIe, DMA, and HBM subsystem IPs. Then in Figure 5.16(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 5.16(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 5.16(D), which prunes away all irrelevant elements and only preserves the cells that directly connect to an island.

5.8.1 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. For example, in AMD/Xilinx FPGAs, certain routing node has 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. Figure 5.17 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) can not be routed without causing conflicts. In a normal flow where the router has full knowledge of the design placement, the router will avoid using such nodes. But in our 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.

Another situation is shown in Figure 5.18. For a Super Long Logic (SLL) node used to cross the die boundary, it is shared among multiple connections. If a net occupies the SLL node in its routing, then certain Laguna FF will be made unreachable.

Problems like this will not occur in a conventional routing process where 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

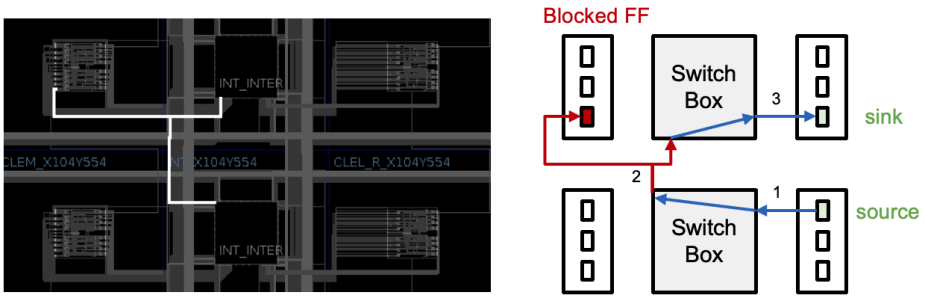


Figure 5.17: Example of a route with a multi-output node. The red FF is made unreachable by other nets since routing node 2 has been occupied.

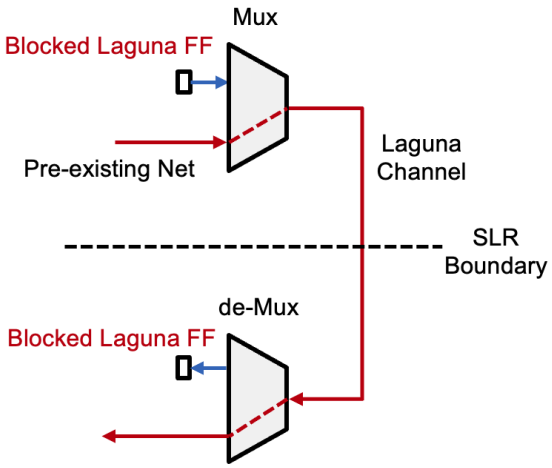


Figure 5.18: Example of a route with an SLL node. The red FFs are made unreachable since the SLL node is the only input/output connection to them.

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 [LK18] and mark the corresponding laguna registers as prohibited in placement.

5.9 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 too much 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 re-use 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 set up 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 we are able to integrate RapidStream 2.0 with the AMD/Xilinx Vitis accelerator workflow, so that we can re-use the host-device communication infrastructure to execute our generated bitstream on a real FPGA board.

5.10 Evaluation of RapidStream 1.0

5.10.1 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 2.40GHz 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 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.

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

[3]Please refer to our open source repository for more details.

uniform height of 120 CLBs.[4] Between adjacent islands, we reserve three empty columns (or ten 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 as that of an island. At die boundaries, we use all Laguna columns as the anchor region (see Figure 5.5).

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 re-routing. As shown in Figure 5.5, 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 [Xil20c], 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.

5.10.2 Benchmarks

To evaluate RapidStream, we use six large-scale dataflow designs listed in Table 5.1. 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 [WGC21]; the 2-D and 3-D stencil accelerators are from the SODA project [CCW18].

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

The benchmarks are mapped onto the target U250 FPGA, which contains 5376 BRAMs, 12288 DSPs, 3456K FFs, and 1728K LUTs. The mapped designs consume 60-70% of the available resources.

Table 5.1: Benchmarks for RapidStream evaluation.

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

5.10.3 Runtime Reduction

Figure 5.19 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 pipelined by RapidStream.

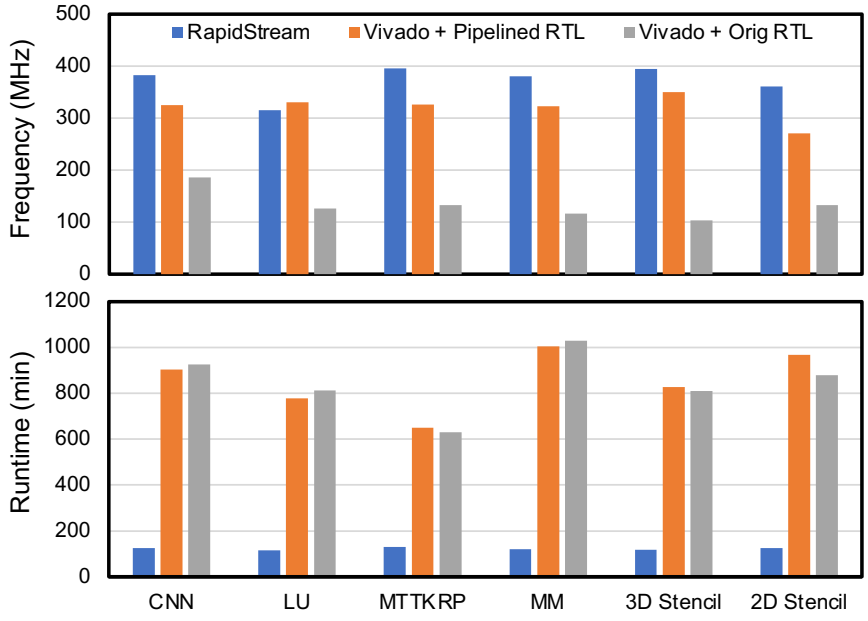


Figure 5.19: Comparison of the runtime and achievable frequency between RapidStream and Vivado.

By default, we use Vivado for inter-island routing (S_{13}) to pursue the best timing quality. In this case, we achieve a 5-7× speed up 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 exception is the LU benchmark, which has many division operations that become the critical paths in both flows.

Figure 5.20 shows the CPU and memory utilization when we use RapidStream to compile the same CNN design as in Figure 5.1. 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.

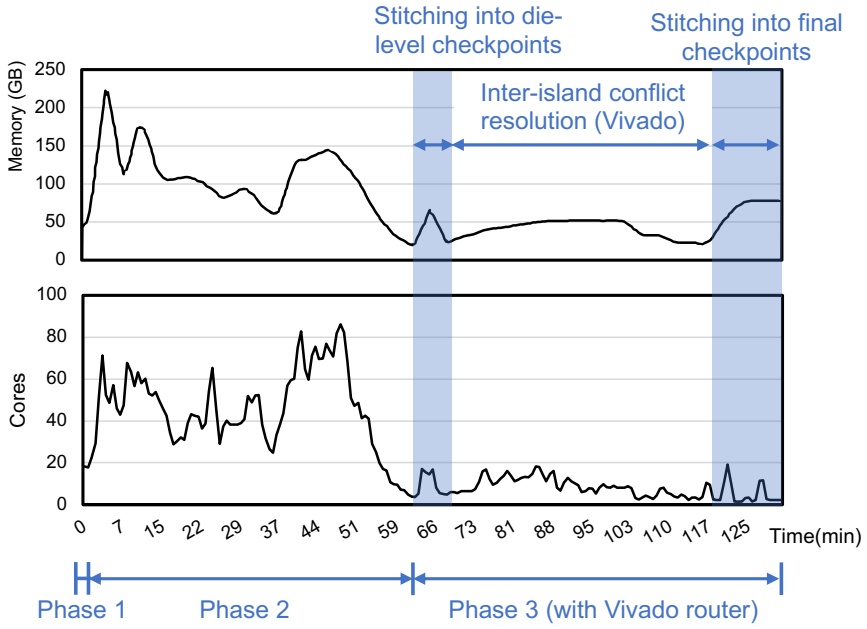


Figure 5.20: CPU and memory usage of the RapidStream run on the CNN design. No re-route is needed after die-level stitching (Sec. 5.10.1).

Figure 5.21 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.

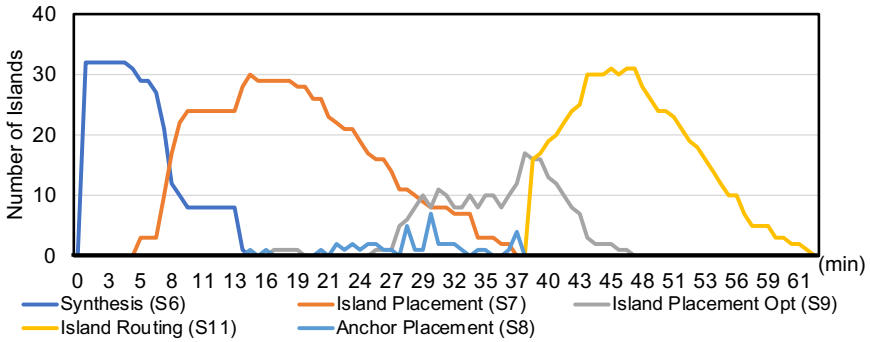


Figure 5.21: Number of active jobs in Phase 2.

5.10.4 Fast Inter-Island Routing

Figure 5.20 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 5.6.2, we customize the open-source RWRoute to further accelerate this step. Figure 5.22 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 relies on negatively-triggered anchors (Section 5.7) 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 5.10.3 down to ∼80 minutes, which is a 7-10× speedup over the Vivado flow.

5.10.5 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

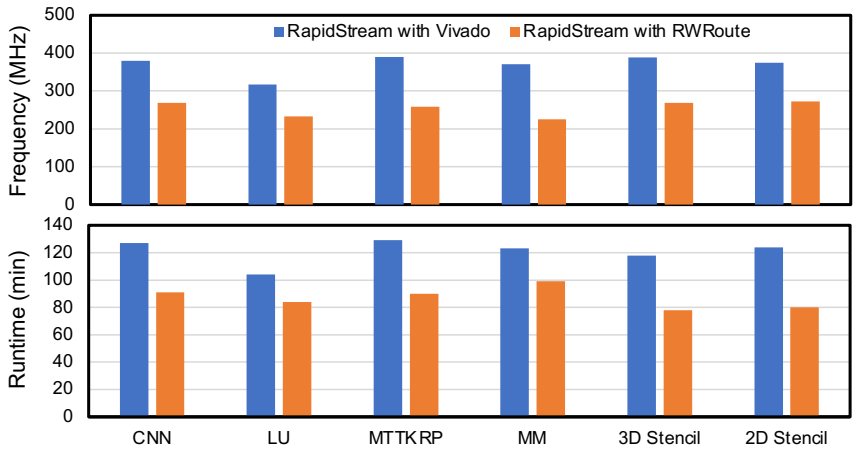


Figure 5.22: Runtime comparison in conflict resolution.

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, while 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 5.23. 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.

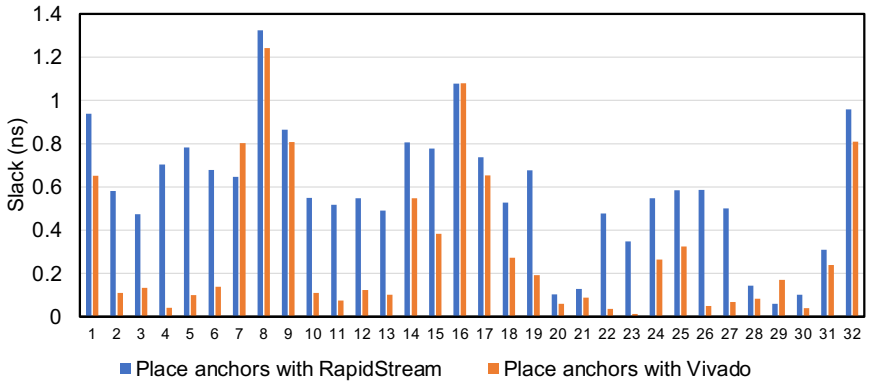


Figure 5.23: Post-placement slack between using the Vivado placer or the min-cost matching placer for anchor placement.

5.10.6 Clock Management

Here we demonstrate the advantages of preserving the clocking trunk using a number of experiments with the MM benchmark. Figure 5.24 shows the timing degradation when we stitch the islands 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.

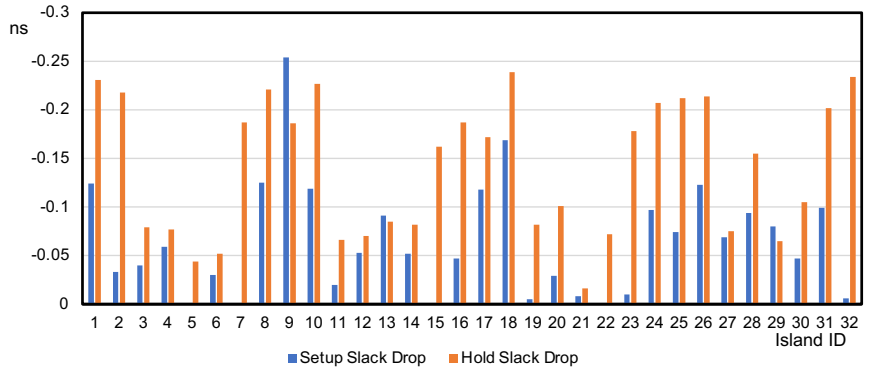


Figure 5.24: Timing loss after stitching w/o clock management.

Figure 5.25 shows the setup slack differences when an island is routed with preserved clock trunk. This is compared to the reference case used in Figure 5.24 without any clocking

constraints. The drop in setup slack is at most 0.15 ns, which is much smaller than that in Figure 5.24. The key takeaway is that we avoid the setup/hold loss during stitching by keeping the clock consistent.

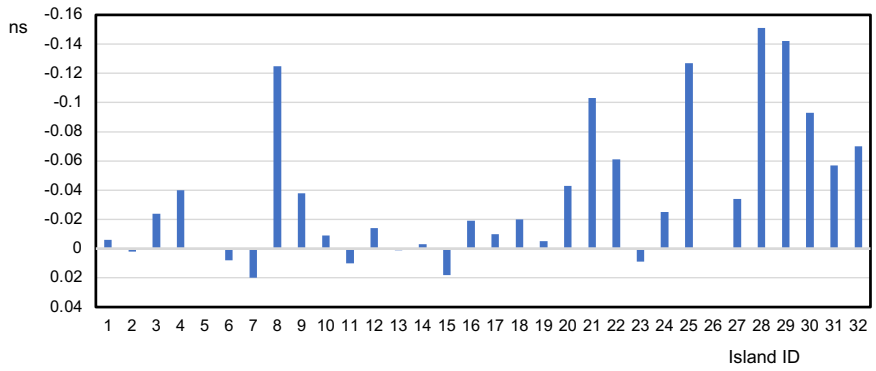


Figure 5.25: Clock preservation reduces timing degradation.

5.11 Evaluation of RapidStream 2.0

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

5.11.1 Implementation Details

RapidStream 2.0 takes a TAPA [GCW21a] dataflow program as input. Compared to Version 1.0, the benefit of using a TAPA program as input is that the TAPA compiler will help check if 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 5.26. 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 where 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.

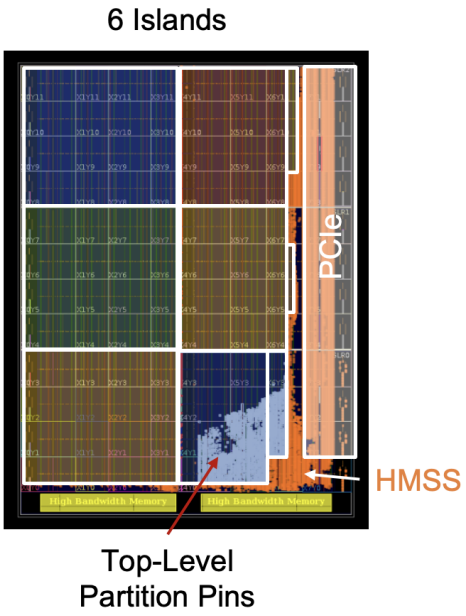


Figure 5.26: An example shell for RapidStream 2.0 corresponding to Figure 5.16(C).

Table 5.2: Benchmarks for RapidStream 2.0 evaluation.

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

5.11.2 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. Table 5.2 shows the details of the two designs. Table 5.3 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 [GCW21a], (3) compile by RapidStream 2.0. The table records the time of each major step and the final frequency.

Compared to 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

Table 5.3: 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

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.

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.

5.11.3 Profiling of RapidStream 2.0 Compilation

Figure 5.27 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 have a difference compared to 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 progress to update 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 and it remains future work to speed up this step with a customized logic pruning tool based on RapidWright.

Still, the routing overhead in RapidStream 2.0 is significantly lower than RapidStream 1.0. We test the gaussian-float design using the RapidStream 1.0 flow. RapidStream 1.0 needs almost three hours to fix the inter-island routing conflicts after routing the islands; while 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,

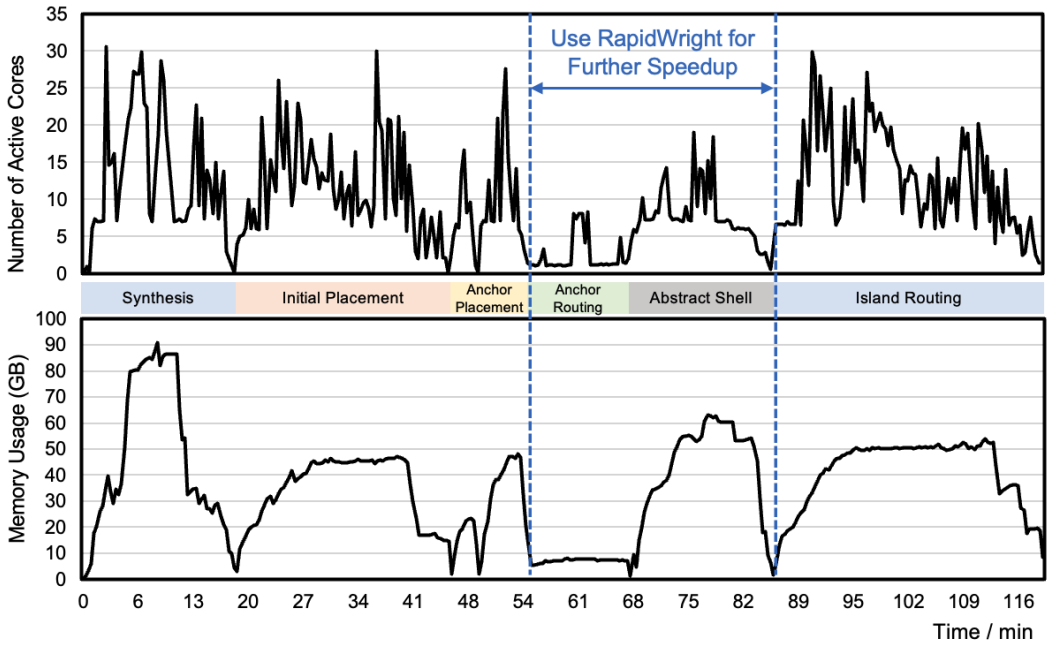


Figure 5.27: Profiling of the CPU and memory usage in RapidStream 2.0 for the gaussian-float benchmark.

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 toward local routing fixes.

5.12 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 a high timing quality. Compared to a commercial toolchain, RapidStream achieves about 5-7× reduction in compile time and up to 1.3× increase in frequency for HLS dataflow designs. In addition, our results show potential for up to an order of magnitude speed-up by leveraging customized open-source routers.

CHAPTER 6

Conclusion

Despite the increasing adoption of HLS for its design productivity advantage, it still has a long journey to achieve a high QoR out-of-box. Meanwhile, the prolonged compile time continues to curb the productivity of the FPGA development process. In this thesis, we identify that we can enable additional timing optimization and compile time reduction by marrying the high-level flexibility of HLS with the low-level physical information. In this section, we first summarize how we address the QoR and compile time challenge of FPGA EDA tools by co-optimization of HLS and physical design. Then we discuss the potential future research directions. Finally, we present the impacts our methods have achieved.

6.1 Thesis Summary and Contributions

Although (HLS) tools have simplified the design processes by allowing users to express their designs in high-level languages, existing FPGA development tools are still troubled by the limited achievable frequency and the long end-to-end compilation time. To address these two challenges, this thesis presents methods and explorations to reconsider how HLS should be organically composed with the backend physical implementation process.

Unlike the traditional methodology, where HLS compilation is completely decoupled from the downstream process, we propose to center the placement and routing steps around HLS tightly. The core insight is that the existing CAD flow does not utilize the flexibility of HLS to insert additional pipelining without affecting overall functionality. By guiding the lower-level placement and routing step with the high-level information of HLS and by utilizing HLS's pipeline flexibility to fix critical paths, we could have significant improvement in QoR

and compilation time. Built around such a core idea, the thesis includes three major parts. We first explore the timing optimization at the HLS stage alone; then we expand our scope to investigate the co-optimization of HLS pipelining and floorplanning; finally, we take one step forward to enable parallel compilation on top of our HLS-floorplan co-optimization.

First, the thesis proposes methods to improve the inherent timing quality of the HLS-generated RTL. We study and classify the timing issues observed in a diverse set of real-world FPGA HLS designs and we generalize three major types of broadcasts in HLS-generated designs. The common broadcasts include high-fanout data signals, pipeline flow control signals, and synchronization signals for concurrent modules. We propose a set of effective yet easy-to-implement approaches to address the issues. Our proposed approaches include broadcast-aware scheduling, synchronization pruning, and skid-buffer-based flow control. Our experimental results show that our methods can improve the maximum frequency of a set of nine representative HLS benchmarks by 53% on average. In some cases, the frequency gain is more than 100 MHz.

Next, we present techniques that couple HLS compilation with the floorplanning step. We propose AutoBridge, an automated framework that couples a coarse-grained floorplanning step with pipelining during HLS compilation. Through this co-design methodology, we ensure that global interconnects are adequately pipelined while the local congestion is alleviated as the logic is evenly distributed around the whole device. Since pipelining may introduce additional latency, we further present analysis and algorithms to ensure the added latency will not compromise the overall throughput. In our experiments with a total of 43 design configurations, we improve the average frequency from 147 MHz to 297 MHz (a 102% improvement) with no loss of throughput and a negligible change in resource utilization. Notably, in 16 experiments, we make the originally unroutable designs achieve 274 MHz on average. AutoBridge received the Best Paper Award in FPGA 2021.

Finally, this thesis addresses the technical challenges to enable parallel physical implementation on top of AutoBridge. 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 several 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 framework, RapidStream, takes in a latency-insensitive program in C/C++ and generates a fully placed and routed implementation. RapidStream is compatible and integrated with the Xilinx Vitis toolchain through partial reconfiguration. Tested on Xilinx U280 FPGA, we observed a 5-7× compile time reduction and 1.3× frequency increase. RapidStream was selected for the Best Paper Award in FPGA 2022.

6.2 Vision and Future Work

In this section, we discuss the future directions of the work presented in this thesis.

6.2.1 Extension to RTL designs

In this thesis, we present methods that utilize the latency-insensitive information of HLS designs to facilitate the timing closure of the physical design process. Since RTL is still the dominant programming model in the industry, a natural question is can we extend the methodology to manual RTL designs?

One challenge of automatic pipelining at latency-insensitive links is that the tool needs to derive the high-level meaning of certain modules, such as FIFOs. Due to the difficulty in extracting high-level semantics from an RTL program, existing tools take a conservative approach and strictly preserve the cycle-accurate behavior of the original design. In this thesis, we choose to obtain latency-insensitive information at the C/C++ level. By analyzing the connection between an RTL port and a C++ variable, we can infer which ports compose a latency-insensitive interface. While such information is not readily available in a manual RTL design, we could let the designer add pragmas to assist the compiler. We plan to design a set of pragmas that the designer could attach to their RTL design to guide the compiler to properly partition, floorplan, and pipeline the design.

Figure 6.1 shows a potential scheme for the proposed RTL annotation. In this example,

there is a module `foo` that has three ports, `my_valid`, `my_ready`, `my_data`. The programmer could add certain pragmas informing that these three ports together represent a latency-insensitive interface and the maximal number of extra latency allowed is 3 cycles. With this information, the compiler could search for an optimal floorplan that properly utilizes the latency-insensitive information here.

In this way, we can extend our framework to handle real-world designs that consist of a mix of HLS modules, manual RTL modules, and even encrypted IP modules. As long as the modules have latency-insensitive interfaces, our methods for floorplanning, pipelining, and partitioning will apply in the same way. Such an enhancement will significantly enlarge the application range as most industrial designs are mixtures of different sources.

6.2.2 Extension to Other FPGA Devices and ASIC

Although our experiments so far have only targeted Xilinx FPGA devices, the technique could be potentially extended to other FPGAs as well. Specifically, Intel FPGAs are known for their HyperFlex registers [Int22a] that are embedded in the routing segments, which can be utilized for more accurate and less intrusive pipelining. Further, we believe that the technology has a lot of potential even for ASICs as well. Regardless of the vendor or technology target, modern FPGA and ASIC designs are highly modularized and decoupled in general, giving us the opportunity to apply our techniques to partition and pipeline the designs accordingly for timing closure and parallelization.

```
1 module foo (
2   /* begin LI interface */
3   /* max_latency = 3 */
4   /* LI valid */ input my_valid,
5   /* LI ready */ output my_ready,
6   /* LI data */ input my_data,
7   /* end LI interface */
8   ...
9 )
10 ...
11 endmodule
```

Figure 6.1: Example annotation to an RTL module interface.

6.2.3 Efficient Emulation

We envision that our techniques could significantly benefit the hardware emulation flow. Conventionally, an emulation tool will map a large ASIC design onto a cluster of FPGAs for verification before taping out. Since emulation requires iterative re-compilation for debugging purposes, the overlong compilation will severely limit the working efficiency. Therefore, if we employ the techniques from this thesis, we can speed up the compilation process and improve the working efficiency. Although emulation requires maintaining the cycle-accurate netlist so we cannot insert pipeline registers, it also has a low timing requirement since emulation commonly runs at as low as 10 MHz [Syn20]. Therefore, we could employ the RapidStream partitioning methodology without inserting additional pipeline registers and still meet the timing target. To do that, we can replace the pipeline registers with LUTs that will not introduce extra latency.

6.2.4 Multi-FPGA Programming

The techniques discussed in this thesis mainly target single-FPGA compilation. An important future direction is to extend the methodology to map a unified program to execute on a system consisting of multiple interconnected FPGAs [DHC17, JSZ19, Hau95]. Currently, the major overhead and hurdle of designing a multi-FPGA system are implementing the cross-FPGA communication infrastructures. To address this issue, we propose to present a unified programming interface, where computation kernels communicate through latency-insensitive channels, and the underlying FPGA cluster is abstracted as one device. In this way, users are free from concerns about design partitioning and low-level details about inter-device communication. To implement the design, we can extend our floorplan tool to automatically partition the design into different FPGA devices and synthesize the corresponding inter-FPGA communication structure.

6.3 Thesis Impact

The technologies developed under this thesis have generated an impact in academia and the industry and have benefited dozens of peer researchers.

First, the proposed pipeline flow control using a skid buffer is an essential reference to the commercial Vitis HLS compiler. Starting from Version 2020.2, Vitis HLS provides an option to allow users to choose the skid-buffer-based pipeline flow control method instead of the broadcast-based flow control [Xil20b].

Second, the AutoBridge framework has been recognized with the Best Paper Award in FPGA 2021 and has been directly used in multiple other projects to improve the maximal frequency:

- A convolution neural network accelerator [WGC21] targeting Xilinx U250 is improved from routing failure to 300 MHz.
- A sparse matrix-matrix multiplication accelerator [SCS22] targeting Xilinx U280 is improved from 216 MHz to 245 MHz.
- A sparse matrix-vector multiplication accelerator [SCG21] targeting Xilinx U280 is improved from 193 MHz to 283 MHz.
- A stencil accelerator [TYL22] targeting Xilinx U280 is improved from routing failure to 250 MHz.
- A Bayesian inference accelerator [CSS22] is improved from 234 MHz to 300 MHz.
- By manually applying the methodology to an RTL latency-insensitive design, a merge sort accelerator [QGF22] targeting Xilinx U280 is improved from routing failure to 214 MHz. Specifically, the designer manually assigns sub-kernels to different regions of the device and adjusts the interconnects between the kernels. This example also motivates our extension to support manual RTL design with latency-insensitive interfaces.

Finally, the RapidStream framework reveals new opportunities to tackle the unbearably long compile time of large-scale FPGA designs. The paper is recognized with the Best Paper Award at FPGA 2022; the work is covered in a dedicated article on the EE Journal [Lei22]. Furthermore, some part of RapidStream has been directly implemented into the AMD/Xilinx RapidWright framework, specifically in the router module named "RapidStreamRoute" and the block stretching module, demonstrating the commercial applications of this work. Large industrial companies like Google, Samsung, and AMD have invited the author to present RapidStream to their research and development teams.

REFERENCES

[AB18] Mustafa Abbas and Vaughn Betz. "Latency insensitive design styles for FPGAs." In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 360–3607. IEEE, 2018.

[AMS08] Charles J Alpert, Dinesh P Mehta, and Sachin S Sapatnekar. *Handbook of algorithms for physical design automation*. CRC press, 2008.

[AR95] Michael J Alexander and Gabriel Robins. "New performance-driven FPGA routing algorithms." In *Proceedings of the 32nd annual ACM/IEEE Design Automation Conference*, pp. 562–567, 1995.

[ASB14] Matthew An, J Gregory Steffan, and Vaughn Betz. "Speeding up FPGA placement: Parallel algorithms and methods." In *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*, pp. 178–185. IEEE, 2014.

[BL03] Giancarlo Beraudo and John Lillis. "Timing optimization of FPGA placements by logic replication." In *DAC '03*, 2003.

[Bre77] Melvin A Breuer. "A class of min-cut placement algorithms." In *Proceedings of the 14th Design Automation Conference*, pp. 284–290, 1977.

[BSB09] Pritha Banerjee, Susmita Sur-Kolay, and Arijit Bishnu. "Fast unified floorplan topology generation and sizing on heterogeneous FPGAs." *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, **28**(5):651–661, 2009.

[Cad20] Cadence. "https://www.cadence.com/.", 2020.

[Car06] Luca P Carloni. "The role of back-pressure in implementing latency-insensitive systems." *Electronic Notes in Theoretical Computer Science*, **146**(2):61–80, 2006.

[CC07] Rebecca L Collins and Luca P Carloni. "Topology-based optimization of maximal sustainable throughput in a latency-insensitive system." In *Proceedings of the 44th annual Design Automation Conference*, pp. 410–415, 2007.

[CC20] Yuze Chi and Jason Cong. "Exploiting computation reuse for stencil accelerators." In *DAC '20*, 2020.

[CCP16] Adrian M. Caulfield, Eric S. Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, Daniel Lo, Todd Massengill, Kalin Ovtcharov, Michael Papamichael, Lisa Woods, Sitaram Lanka, Derek Chiou, and Doug Burger. "A cloud-scale acceleration architecture." In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1–13, 2016.

[CCQ21] Young-kyu Choi, Yuze Chi, Weikang Qiao, Nikola Samardzic, and Jason Cong. "HBM Connect: High-performance HLS interconnect for FPGA HBM." In *Proceedings of the 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2021.

[CCS05] Tony Chan, Jason Cong, and Kenton Sze. "Multilevel generalized force-directed method for circuit placement." In *Proceedings of the 2005 international symposium on physical design*, pp. 185–192, 2005.

[CCW18] Yuze Chi, Jason Cong, Peng Wei, and Peipei Zhou. "SODA: Stencil with optimized dataflow architecture." In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 1–8. IEEE, 2018.

[CCW20] Young-kyu Choi, Yuze Chi, Jie Wang, Licheng Guo, and Jason Cong. "When HLS meets FPGA HBM: benchmarking and bandwidth optimization." *arXiv preprint arXiv:2010.06075*, 2020.

[CFH04] Jason Cong, Yiping Fan, Guoling Han, Xun Yang, and Zhiru Zhang. "Architecture and synthesis for on-chip multicycle communication." *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, **23**(4):550–564, 2004.

[CGC20] Yuze Chi, Licheng Guo, Young-kyu Choi, Jie Wang, and Jason Cong. "Extending high-level synthesis for task-parallel programs." *arXiv preprint arXiv:2009.11389*, 2020.

[CGH18] Jason Cong, Licheng Guo, Po-Tsang Huang, Peng Wei, and Tianhe Yu. "SMEM++: A pipelined and time-multiplexed SMEM seeding accelerator for genome sequencing." In *FPL '18*, 2018.

[CHB18] Zhe Chen, Andrew Howe, Hugh T Blair, and Jason Cong. "CLINK: Compact LSTM inference kernel for energy efficient neurofeedback devices." In *ISLPED'18*, p. 2, 2018.

[Che10] Chandra Chekuri. "https://courses.engr.illinois.edu/cs598csc/sp2010/Lectures/Lecture11.pdf.", 2010.

[CJC20] Jianyi Cheng, Lana Josipovic, George A Constantinides, Paolo Ienne, and John Wickerson. "Combining dynamic & static scheduling in high-level synthesis." In *The 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 288–298, 2020.

[CLN11] Jason Cong, Bin Liu, Stephen Neuendorffer, Juanjo Noguera, Kees Vissers, and Zhiru Zhang. "High-level synthesis for FPGAs: From prototyping to deployment." *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, **30**(4):473–491, 2011.

[CMS01] Luca P Carloni, Kenneth L McMillan, and Alberto L Sangiovanni-Vincentelli. "Theory of latency-insensitive design." *IEEE Transactions on computer-aided design of integrated circuits and systems*, **20**(9):1059–1076, 2001.

[CS00] Luca P Carloni and Alberto L Sangiovanni-Vincentelli. "Performance analysis and optimization of latency insensitive systems." In *Proceedings of the 37th Annual Design Automation Conference*, pp. 361–367, 2000.

[CSN14] James Chacko, Cem Sahin, Danh Nguyen, Doug Pfeil, Nagarajan Kandasamy, and Kapil Dandekar. "FPGA-based latency-insensitive OFDM pipeline for wireless research." In *2014 IEEE high performance extreme computing conference (HPEC)*, pp. 1–6. IEEE, 2014.

[CSS22] Young-kyu Choi, Carlos Santillana, Yujia Shen, Adnan Darwiche, and Jason Cong. "FPGA acceleration of probabilistic sentential decision diagrams with high-Level synthesis." *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 2022.

[Cut20] Minimum Cut. "https://en.wikipedia.org/wiki/Minimum_cut.", 2020.

[CW06] Lei Cheng and Martin DF Wong. "Floorplan design for multimillion gate FPGAs." *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, **25**(12):2795–2805, 2006.

[CW18] Jason Cong and Jie Wang. "PolySA: Polyhedral-based systolic array auto-compilation." In *ICCAD '18*, 2018.

[CWA22] Yuze Chi, Qiao Weikang, Sohrabizadeh Atefeh, Wang Jie, and Jason Cong. "Democratizing domain-specific computing." *arXiv preprint arXiv:2209.02951*, 2022.

[CWY18a] Jason Cong, Peng Wei, Cody Hao Yu, and Peng Zhang. "Automated accelerator generation and optimization with composable, parallel and pipeline architecture." In *DAC '18*, 2018.

[CWY18b] Jason Cong, Peng Wei, Cody Hao Yu, and Peipei Zhou. "Latte: Locality aware transformation for high-level synthesis." In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 125–128. IEEE, 2018.

[CZ05] Jason Cong and Yan Zhang. "Thermal-driven multilevel routing for 3-D ICs." In *Proceedings of the 2005 Asia and South Pacific Design Automation Conference*, pp. 121–126, 2005.

[CZ06] Jason Cong and Zhiru Zhang. "An efficient and versatile scheduling algorithm based on SDC formulation." In *2006 43rd ACM/IEEE Design Automation Conference*, pp. 433–438. IEEE, 2006.

[CZ09] Jason Cong and Yi Zou. "Parallel multi-level analytical global placement on graphics processing units." In *2009 IEEE/ACM International Conference on Computer-Aided Design-Digest of Technical Papers*, pp. 681–688. IEEE, 2009.

[DB94] Kaushik De and Prithviraj Banerjee. "Parallel logic synthesis using partitioning." In *1994 International Conference on Parallel Processing Vol. 3*, volume 3, pp. 135–142. IEEE, 1994.

[DCR95] Kaushik De, LA Chandy, Sumit Roy, Steven Parkes, and Prithviraj Banerjee. "Parallel algorithms for logic synthesis using the MIS approach." In *Proceedings of 9th International Parallel Processing Symposium*, pp. 579–585. IEEE, 1995.

[DGK94] Srinivas Devadas, Abhijit Ghosh, and Kurt William Keutzer. "Logic synthesis." 1994.

[DHC17] Guohao Dai, Tianhao Huang, Yuze Chi, Ningyi Xu, Yu Wang, and Huazhong Yang. "ForeGraph: Exploring large-scale graph processing on multi-FPGA architecture." In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 217–226, 2017.

[DK85] Alfred E Dunlop, Brian W Kernighan, et al. "A procedure for placement of standard cell VLSI circuits." *IEEE Transactions on Computer-Aided Design*, **4**(1):92–98, 1985.

[DL09] Xiao Dong and Guy GF Lemieux. "PGR: Period and glitch reduction via clock skew scheduling, delay padding and GlitchLess." In *2009 International Conference on Field-Programmable Technology*, pp. 88–95. IEEE, 2009.

[DLN21] Vidushi Dadu, Sihao Liu, and Tony Nowatzki. "Polygraph: Exposing the value of flexibility for graph processing accelerators." In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pp. 595–608. IEEE, 2021.

[DLN22] Vidushi Dadu, Sihao Liu, and Tony Nowatzki. "Systematically understanding graph accelerator dimensions and the value of hardware flexibility." *IEEE Micro*, 2022.

[DSI19] Shounak Dhar, Love Singhal, Mahesh Iyer, and David Pan. "FPGA accelerated FPGA placement." In *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 404–410, 2019.

[Exa20] Intel OpenCL Examples. "https://www.intel.com/content/www/us/en/programmable/products/design-software/embedded-software-developers/opencl/support.html.", 2020.

[FAP12] Kermin Elliott Fleming, Michael Adler, Michael Pellauer, Angshuman Parashar, Arvind Mithal, and Joel Emer. "Leveraging latency-insensitivity to ease multiple FPGA design." In *Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays*, pp. 175–184, 2012.

[Fis90] John P. Fishburn. "Clock skew optimization." *IEEE transactions on computers*, **39**(7):945–951, 1990.

[FOP18] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi, Stephen Heil, Prerak Patel, Adam Sapek, Gabriel Weisz, Lisa Woods, Sitaram Lanka, Steven K. Reinhardt, Adrian M. Caulfield, Eric S. Chung, and Doug Burger. "A configurable cloud-scale DNN processor for real-time AI." In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pp. 1–14, 2018.

[FSP20] Alex Forencich, Alex C. Snoeren, George Porter, and George Papen. "Corundum: An open-source 100-Gbps NIC." In *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 38–46, 2020.

[GA10] Marcel Gort and Jason H Anderson. "Deterministic multi-core parallel routing for FPGAs." In *2010 International Conference on Field-Programmable Technology*, pp. 78–86. IEEE, 2010.

[GA11] Marcel Gort and Jason H Anderson. "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):61–74, 2011.

[GCL22] Licheng Guo, Yuze Chi, Jason Lau, Linghao Song, Xingyu Tian, Moazin Khatti, Weikang Qiao, Jie Wang, Ecenur Ustun, Zhenman Fang, et al. "TAPA: A scalable task-parallel dataflow programming framework for modern FPGAs with co-optimization of HLS and physical design." *arXiv preprint arXiv:2209.02663*, 2022.

[GCW21a] Licheng Guo, Yuze Chi, Jie Wang, Jason Lau, Weikang Qiao, Ecenur Ustun, Zhiru Zhang, and Jason Cong. "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*, pp. 81–92, 2021.

[GCW21b] Licheng Guo, Yuze Chi, Jie Wang, Jason Lau, Weikang Qiao, Ecenur Ustun, Zhiru Zhang, and Jason Cong. "https://doi.org/10.5281/zenodo.4412047.", 2021.

[GGS06] Amir Hossein Ghamarian, Marc CW Geilen, Sander Stuijk, Twan Basten, Bart D Theelen, Mohammad Reza Mousavi, Arno JM Moonen, and Marco JG Bekooij. "Throughput analysis of synchronous data flow graphs." In *Sixth International Conference on Application of Concurrency to System Design (ACSD'06)*, pp. 25–36. IEEE, 2006.

[Gil74] KAHN Gilles. "The semantics of a simple language for parallel programming." *Information processing*, **74**:471–475, 1974.

[GLC20] Licheng Guo, Jason Lau, Yuze Chi, Jie Wang, Cody Hao Yu, Zhe Chen, Zhiru Zhang, and Jason Cong. "Analysis and optimization of the implicit broadcasts in

FPGA HLS to improve maximum frequency." In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pp. 1–6, 2020.

[GLR19] Licheng Guo, Jason Lau, Zhenyuan Ruan, Peng Wei, and Jason Cong. "Hardware acceleration of long read pairwise overlapping in genome sequencing: A race between FPGA and GPU." In *FCCM '19*, 2019.

[GMZ22] Licheng Guo, Pongstorn Maidee, Yun Zhou, Chris Lavin, Jie Wang, Yuze Chi, Weikang Qiao, Alireza Kaviani, Zhiru Zhang, and Jason Cong. "RapidStream: Parallel physical implementation of FPGA HLS designs." In *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 1–12, 2022.

[Gur20] Gurobi. "https://www.gurobi.com/.", 2020.

[Hau95] Scott Alan Hauck. *Multi-FPGA systems*. University of Washington, 1995.

[HBM20] Xilinx HBM. "https://www.xilinx.com/products/silicon-devices/fpga/virtex-ultrascale-plus-hbm.html.", 2020.

[HD12] Yutian Huan and André DeHon. "FPGA optimized packet-switched NoC using split and merge primitives." In *2012 International Conference on Field-Programmable Technology*, pp. 47–52. IEEE, 2012.

[HK18] Chin Hau Hoo and Akash Kumar. "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, p. 67–76, New York, NY, USA, 2018. Association for Computing Machinery.

[HKK16] Jaco Hofmann, Jens Korinth, and Andreas Koch. "A scalable latency-insensitive architecture for FPGA-accelerated semi-global matching in stereo vision applications." In *2016 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pp. 1–8. IEEE, 2016.

[HKP84] H James Hoover, Maria M Klawe, and Nicholas Pippenger. "Bounding fan-out in logical networks." In *JACM*, volume 31, pp. 13–18, 1984.

[HLS20a] LegUp HLS. "https://www.legupcomputing.com/.", 2020.

[HLS20b] Mentor Catapult HLS. "https://eda.sw.siemens.com/en-US/ic/catapult-high-level-synthesis/.", 2020.

[Int20] Intel. "Intel Stratix 10 FPGA.", 2020.

[Int22a] Intel. "https://www.intel.com/content/www/us/en/docs/programmable/683353/21-3/fpga-architecture-introduction.html.", 2022.

[Int22b] Intel. *Intel Hyperflex architecture high-performance design handbook*. 2022.

[JGI18] Lana Josipović, Radhika Ghosal, and Paolo Ienne. "Dynamically scheduled high-level synthesis." In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 127–136, 2018.

[JSG20] Lana Josipović, Shabnam Sheikhha, Andrea Guerrieri, Paolo Ienne, and Jordi Cortadella. "Buffer placement and sizing for high-performance dataflow circuits." In *The 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 186–196, 2020.

[JSZ19] Weiwen Jiang, Edwin H-M Sha, Xinyi Zhang, Lei Yang, Qingfeng Zhuge, Yiyu Shi, and Jingtong Hu. "Achieving super-linear speedup across multi-fpga for real-time dnn inference." *ACM Transactions on Embedded Computing Systems (TECS)*, **18**(5s):1–23, 2019.

[JZP08] Wei Jiang, Zhiru Zhang, Miodrag Potkonjak, and Jason Cong. "Scheduling with integer time budgeting for low-power optimization." In *2008 Asia and South Pacific Design Automation Conference*, pp. 22–27. IEEE, 2008.

[Kap17] Nachiket Kapre. "Deflection-routed butterfly fat trees on FPGAs." In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 1–8. IEEE, 2017.

[KG17] Nachiket Kapre and Jan Gray. "Hoplite: A deflection-routed directional torus noc for fpgas." *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, **10**(2):1–24, 2017.

[KMD06] Nachiket Kapre, Nikil Mehta, Michael Delorimier, Raphael Rubin, Henry Barnor, Michael J Wilson, Michael Wrighton, and Andre DeHon. "Packet switched vs. time multiplexed FPGA overlay networks." In *2006 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 205–216. IEEE, 2006.

[KS16] Parivallal Kannan and Satish Sivaswamy. "Performance driven routing for modern FPGAs." In *Proceedings of the 35th International Conference on Computer-Aided Design*, pp. 1–6, 2016.

[KVC18] Brucek Khailany, Rangharajan Venkatesan, Jason Clemons, Joel S Emer, Matthew Fojtik, Alicia Klinefelter, Michael Pellauer, Nathaniel Pinckney, Yakun Sophia Shao, Shreesha Srinath, et al. "A modular digital VLSI flow for high-productivity SoC design." In *Proceedings of the 55th Annual Design Automation Conference*, pp. 1–6, 2018.

[Lau88] Ulrich Lauther. "A min-cut placement algorithm for general cell assemblies based on a graph representation." In *Papers on Twenty-five years of electronic design automation*, pp. 182–191. 1988.

[LBP08] Adrian Ludwin, Vaughn Betz, and Ketan Padalia. "High-quality, deterministic parallel placement for FPGAs on commodity hardware." In *Proceedings of the*

16th international ACM/SIGDA symposium on Field programmable gate arrays, pp. 14–23, 2008.

[LCW15] Tao Lin, Chris Chu, and Gang Wu. "POLAR 3.0: An ultrafast global placement engine." In *2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 520–527. IEEE, 2015.

[LD86] David P La Potin and Stephen W Director. "Mason: A global floorplanning approach for VLSI design." *IEEE transactions on computer-aided design of integrated circuits and systems*, **5**(4):477–489, 1986.

[Lei22] Steven Leibson. "Can HLS partitioning speed up placement and routing of FPGA designs? Yes, Oh Yes!" In *Electrical Engineering Journal*, pp. 1–1, 2022.

[LGW14] 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. "VTR 7.0: Next generation architecture and CAD system for FPGAs." **7**(2), 2014.

[Li18] Heng Li. "Minimap2: Pairwise alignment for nucleotide sequences." *Bioinformatics*, **34**(18):3094–3100, 2018.

[Lib20] Xilinx Vitis Library. "https://github.com/Xilinx/Vitis_Libraries.", 2020.

[LJG20] Yibo Lin, Zixuan Jiang, Jiaqi Gu, Wuxi Li, Shounak Dhar, Haoxing Ren, Brucek Khailany, and David Z Pan. "DREAMPlace: Deep learning toolkit-enabled GPU acceleration for modern VLSI placement." *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2020.

[LK03] Ruibing Lu and Cheng-Kok Koh. "Performance optimization of latency insensitive systems through buffer queue sizing of communication channels." In *ICCAD-2003. International Conference on Computer Aided Design (IEEE Cat. No. 03CH37486)*, pp. 227–231. IEEE, 2003.

[LK06] Ruibing Lu and Cheng-Kok Koh. "Performance analysis of latency-insensitive systems." *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, **25**(3):469–483, 2006.

[LK18] Chris Lavin and Alireza Kaviani. "Rapidwright: Enabling custom crafted implementations for fpgas." In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 133–140. IEEE, 2018.

[LLW17] Wuxi Li, Meng Li, Jiajun Wang, and David Z Pan. "UTPlaceF 3.0: A parallelization framework for modern FPGA global placement." In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 922–928. IEEE, 2017.

[LM87] Edward A Lee and David G Messerschmitt. "Synchronous data flow." *Proceedings of the IEEE*, **75**(9):1235–1245, 1987.

[LPL11] Christopher Lavin, Marc Padilla, Jaren Lamprecht, Philip Lundrigan, Brent Nelson, and Brad Hutchings. "HMFlow: Accelerating FPGA compilation with hard macros for rapid prototyping." In *2011 IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines*, pp. 117–124. IEEE, 2011.

[LRS83] Charles E Leiserson, Flavio M Rose, and James B Saxe. "Optimizing synchronous circuitry by retiming (preliminary version)." In *Third Caltech conference on very large scale integration*, pp. 87–116. Springer, 1983.

[LS91] Charles E Leiserson and James B Saxe. "Retiming synchronous circuitry." *Algorithmica*, **6**(1-6):5–35, 1991.

[LSZ20] Jason Lau, Aishwarya Sivaraman, Qian Zhang, Muhammad Ali Gulzar, Jason Cong, and Miryung Kim. "HeteroRefactor: Refactoring for heterogeneous computing with FPGA." In *ICSE '20*, 2020.

[LUX21] Yi-Hsiang Lai, Ecenur Ustun, Shaojie Xiang, Zhenman Fang, Hongbo Rong, and Zhiru Zhang. "Programming and synthesis for software-defined FPGA acceleration: status and future prospects." *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, **14**(4):1–39, 2021.

[LWK22] Sihao Liu, Jian Weng, Dylan Kupsh, Atefeh Sohrabizadeh, Zhengrong Wang, Licheng Guo, Jiuyang Liu, Maxim Zhulin, Rishabh Mani, Lucheng Zhang, et al. "OverGen: Improving FPGA usability through domain-specific overlay generation." In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 35–56. IEEE, 2022.

[LWV21] Sihao Liu, Jian Weng, Dadu Vidushi, and Tony Nowatzki. "Generality is the key dimension in accelerator design." **250**:300, 2021.

[MAA16] Sen Ma, Zeyad Aklah, and David Andrews. "Just in time assembly of accelerators." In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 173–178, 2016.

[MAB03] Pongstorn Maidee, Cristinel Ababei, and Kia Bazargan. "Fast timing-driven partitioning-based placement for island style FPGAs." In *Proceedings of the 40th annual design automation conference*, pp. 598–603, 2003.

[MB15] Kevin E Murray and Vaughn Betz. "HETRIS: Adaptive floorplanning for heterogeneous FPGAs." In *2015 International Conference on Field Programmable Technology (FPT)*, pp. 88–95. IEEE, 2015.

[MK87] H Modarres and A Kelapure. "An automatic floorplanner for up to 100,000 gates." *VLSI Systems Design*, **8**(13):38, 1987.

[MNK19] Pongstorn Maidee, Chris Neely, Alireza Kaviani, and Chris Lavin. "An open-source lightweight timing model for RapidWright." In *2019 International Conference on Field-Programmable Technology (ICFPT)*, pp. 171–178. IEEE, 2019.

[MOD20] Federico Montaño, Tarek Ould-Bachir, and Jean Pierre David. "A latency-insensitive design approach to programmable FPGA-based real-time simulators." *Electronics*, **9**(11):1838, 2020.

[OC97] Takumi Okamoto and Jason Cong. "Buffered Steiner tree construction with wire sizing for interconnect layout optimization." In *ICCAD '96*, 1997.

[Par07] Keshab K Parhi. *VLSI digital signal processing systems: design and implementation*. John Wiley & Sons, 2007.

[PB91] Massoud Pedram and Narasimha B Bhat. "Layout driven logic restructuring/decomposition." In *ICCAD'91*, 1991.

[PH12] Michael K Papamichael and James C Hoe. "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*, pp. 37–46, 2012.

[PP91] Wuxu Peng and S Puroshothaman. "Data flow analysis of communicating finite state machines." *ACM Transactions on Programming Languages and Systems (TOPLAS)*, **13**(3):399–442, 1991.

[PXD22] Dongjoon Park, Yuanlong Xiao, and André DeHon. "Fast and Flexible FPGA Development using Hierarchical Partial Reconfiguration." In *IEEE International Conference on Field-Programmable Technology*, 2022.

[PXM18] Dongjoon Park, Yuanlong Xiao, Nevo Magnezi, and André DeHon. "Case for fast FPGA compilation using partial reconfiguration." In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 235–2353. IEEE, 2018.

[QDF18] Weikang Qiao, Jieqiong Du, Zhenman Fang, Michael Lo, Mau-Chung Frank Chang, and Jason Cong. "High-throughput lossless compression on tightly coupled CPU-FPGA platforms." In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 37–44. IEEE, 2018.

[QFC19] Weikang Qiao, Zhenman Fang, Mau-Chung Frank Chang, and Jason Cong. "An FPGA-Based BWT accelerator for bzip2 data compression." In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 96–99. IEEE, 2019.

[QGF22] Weikang Qiao, Licheng Guo, Zhenman Fang, Mau-Chung Frank Chang, and Jason Cong. "TopSort: A high-performance two-phase sorting accelerator optimized on

HBM-based FPGAs." In *2022 IEEE 30th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 1–1. IEEE, 2022.

[QOG21] Weikang Qiao, Jihun Oh, Licheng Guo, Mau-Chung Frank Chang, and Jason Cong. "FANS: FPGA-accelerated near-storage sorting." In *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 106–114. IEEE, 2021.

[Rud89] Richard L Rudell. *Logic synthesis for VLSI design*. University of California, Berkeley, 1989.

[Sas93] Tsutomu Sasao. *Logic synthesis and optimization*, volume 2. Springer, 1993.

[SCG21] Linghao Song, Yuze Chi, Licheng Guo, and Jason Cong. "Serpens: A high bandwidth memory based accelerator for general-purpose sparse matrix-vector multiplication." *arXiv preprint arXiv:2111.12555*, 2021.

[SCS22] Linghao Song, Yuze Chi, Atefeh Sohrabizadeh, Young-kyu Choi, Jason Lau, and Jason Cong. "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*, pp. 65–77, 2022.

[SDM17] Nitish Kumar Srivastava, Steve Dai, Rajit Manohar, and Zhiru Zhang. "Accelerating face detection on programmable SoC using C-based synthesis." In *FPGA '17*, 2017.

[SL15] Minghua Shen and Guojie Luo. "Accelerate FPGA routing with parallel recursive partitioning." In *2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 118–125. IEEE, 2015.

[SM91] Khushro Shahookar and Pinaki Mazumder. "VLSI cell placement techniques." *ACM Computing Surveys (CSUR)*, **23**(2):143–220, 1991.

[SQA20] Nikola Samardzic, Weikang Qiao, Vaibhav Aggarwal, Mau-Chung Frank Chang, and Jason Cong. "Bonsai: High-performance adaptive merge tree sorting." In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture*, pp. 282–294. IEEE, 2020.

[SS90] Kanwar Jit Singh and Alberto Sangiovanni-Vincentelli. "A heuristic algorithm for the fanout problem." In *DAC '90*, 1990.

[ST20] H.G. Santos and T.A.M. Toffolo. "Python MIP (Mixed-Integer Linear Programming) tools.", 2020.

[Sto17] Mirjana Stojilović. "Parallel FPGA routing: Survey and challenges." In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 1–8. IEEE, 2017.

[Syn20] Synopsys. "https://www.synopsys.com/.", 2020.

[Tak15] Shinya Takamaeda-Yamazaki. "PyVerilog: A python-based hardware design processing toolkit for Verilog HDL." In *International Symposium on Applied Reconfigurable Computing*, pp. 451–460. Springer, 2015.

[TDG15] Mingxing Tan, Steve Dai, Udit Gupta, and Zhiru Zhang. "Mapping-aware constrained scheduling for LUT-based FPGAs." In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 190–199, 2015.

[TYL22] Xingyu Tian, Zhifan Ye, Alec Lu, Licheng Guo, Yuze Chi, and Zhenman Fang. "SASA: A scalable and automatic stencil acceleration framework for optimized hybrid spatial and temporal parallelism on HBM-based FPGAs." *arXiv preprint arXiv:2208.10770*, 2022.

[VG14] Girish Venkataramani and Yongfeng Gu. "System-level retiming and pipelining." In *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*, pp. 80–87. IEEE, 2014.

[VGK17] Kizhepatt Vipin, Jan Gray, and Nachiket Kapre. "Enabling partial reconfiguration and low latency routing using segmented FPGA NoCs." In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 1–8. IEEE, 2017.

[VHB] Babette Van Antwerpen, Michael D Hutton, Gregg Baeckler, and Richard Yuan. "Register retiming technique." US Patent 7,120,883.

[WDT17] Dekui Wang, Zhenhua Duan, Cong Tian, Bohu Huang, and Nan Zhang. "A runtime optimization approach for FPGA routing." *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, **37**(8):1706–1710, 2017.

[Wea08] Nicholas Weaver. *Retiming, Repipelining and C-Slow Retiming*, pp. 383–399. 2008.

[WGC21] Jie Wang, Licheng Guo, and Jason Cong. "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*, 2021.

[WLC19] Xuechao Wei, Yun Liang, and Jason Cong. "Overcoming data transfer bottlenecks in FPGA-based DNN accelerators via layer conscious memory management." In *DAC '19*, 2019.

[WLD19] Jian Weng, Sihao Liu, Vidushi Dadu, and Tony Nowatzki. "DAEGEN: A modular compiler for exploring decoupled spatial accelerators." *IEEE Computer Architecture Letters*, **18**(2):161–165, 2019.

[WLD20] Jian Weng, Sihao Liu, Vidushi Dadu, Zhengrong Wang, Preyas Shah, and Tony Nowatzki. "DSAGen: Synthesizing programmable spatial accelerators." In *2020*

ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA), pp. 268–281. IEEE, 2020.

[WLK22] Jian Weng, Sihao Liu, Dylan Kupsh, and Tony Nowatzki. "Unifying spatial accelerator compilation with idiomatic and modular transformations." *IEEE Micro*, **42**(5):59–69, 2022.

[WMP03] Nicholas Weaver, Yury Markovskiy, Yatish Patel, and John Wawrzynek. "Post-placement C-slow retiming for the Xilinx Virtex FPGA." In *FPGA '03*, 2003.

[WS19] David Wilson and Greg Stitt. "Seiba: An FPGA overlay-based approach to rapid application development." In *2019 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pp. 1–8. IEEE, 2019.

[XAD20] Yuanlong Xiao, Syed Tousif Ahmed, and André DeHon. "Fast linking of separately-compiled FPGA blocks without a NoC." In *2020 International Conference on Field-Programmable Technology (ICFPT)*, pp. 196–205. IEEE, 2020.

[XD22] Yuanlong Xiao and Andre DeHon. "HiPR: Fast, Incremental Custom Partial Reconfiguration for HLS Developers." In *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 155–155, 2022.

[Xil20a] Xilinx. "Vivado Design Suite.", 2020.

[Xil20b] Xilinx. "Vivado High-Level Synthesis.", 2020.

[Xil20c] Xilinx. "Xilinx UltraScale Plus Architecture.", 2020.

[Xil20d] Xilinx. "Xilinx Vitis Unified Platform.", 2020.

[Xil21a] Xilinx. "https://www.xilinx.com/support/documentation/user_guides/ug572-ultrascale-clocking.pdf.", 2021.

[Xil21b] Xilinx. "Vivado Design Suite User Guide: Hierarchical Design.", 2021.

[XK97] Min Xu and Fadi J Kurdahi. "Layout-driven RTL binding techniques for high-level synthesis using accurate estimators." *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, **2**(4):312–343, 1997.

[XMB22] Yuanlong Xiao, Eric Micallef, Andrew Butt, Matthew Hofmann, Marc Alston, Matthew Goldsmith, Andrew Merczynski-Hait, and André DeHon. "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*, pp. 933–945, 2022.

[XPB19] Yuanlong Xiao, Dongjoon Park, Andrew Butt, Hans Giesen, Zhaoyang Han, Rui Ding, Nevo Magnezi, Raphael Rubin, and André DeHon. "Reducing FPGA compile time with separate compilation for FPGA building blocks." In *2019 International Conference on Field-Programmable Technology (ICFPT)*, pp. 153–161. IEEE, 2019.

[YM05] Chao-Yang Yeh and Malgorzata Marek-Sadowska. "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*, pp. 33–40, 2005.

[YVA07] Zhen Yang, Anthony Vannelli, and Shawki Areibi. "An ILP based hierarchical global routing approach for VLSI ASIC design." *Optimization Letters*, **1**(3):281–297, 2007.

[ZGD18] Yuan Zhou, Udit Gupta, Steve Dai, Ritchie Zhao, Nitish Srivastava, Hanchen Jin, Joseph Featherston, Yi-Hsiang Lai, Gai Liu, Gustavo Angarita Velasquez, et al. "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*, pp. 269–278, 2018.

[ZGR14] Hongbin Zheng, Swathi T Gurumani, Kyle Rupnow, and Deming Chen. "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*, pp. 1–10, 2014.

[ZML21] Yun Zhou, Pongstorn Maidee, Chris Lavin, Alireza Kaviani, and Dirk Stroobandt. "RWRoute: An open-source timing-driven router for commercial FPGAs." *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, **15**(1):1–27, 2021.

[ZPF16] P. Zhou, H. Park, Z. Fang, J. Cong, and A. DeHon. "Energy efficiency of full pipelining: A case study for matrix multiplication." In *FCCM '16*, 2016.

[ZTD15] Ritchie Zhao, Mingxing Tan, Steve Dai, and Zhiru Zhang. "Area-efficient pipelining for FPGA-targeted high-level synthesis." In *Proceedings of the 52nd Annual Design Automation Conference*, pp. 1–6, 2015.

[ZVS20] Yun Zhou, Dries Vercruyce, and Dirk Stroobandt. "Accelerating FPGA routing through algorithmic enhancements and connection-aware parallelization." *ACM Trans. Reconfigurable Technol. Syst.*, **13**(4), August 2020.