Title

Enabling Heterogeneous Computing for Software Developers

Author

Lau, Jason

Publication Date

2024

Peer reviewed|Thesis/dissertation

UNIVERSITY OF CALIFORNIA
Los Angeles

Enabling Heterogeneous Computing for Software Developers

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

by

Jason Lau

2024

ABSTRACT OF THE DISSERTATION

Enabling Heterogeneous Computing for Software Developers

by

Jason Lau
Doctor of Philosophy in Computer Science
University of California, Los Angeles, 2024
Professor Jingsheng Jason Cong, Chair

The slowing of CMOS technology scaling mismatches the ever-increasing demand for computational power, leading to a rise in the use of heterogeneous systems, which pair scalar processors such as CPUs with specialized accelerators like FPGAs and GPUs. These systems enable continued performance and efficiency scaling for specialized tasks while retaining limited generality. This restricted generality inherent in heterogeneous platforms requires specialized knowledge of hardware architectures and low-level programming models, posing a substantial barrier to software developers.

This dissertation addresses the challenges software developers face in leveraging heterogeneous computing resources, particularly FPGA acceleration. We identify three major limitations: limited programmability support in domain-specific resources, difficulty in achieving high performance and efficiency, and time-consuming porting across diverse computational architectures. We present novel approaches and tools to bridge the gap between high-level software development and efficient hardware implementation, making heterogeneous computing more accessible to a broader range of developers.

In this dissertation, we introduce Heterosys, an end-to-end optimization framework simplifying heterogeneous hardware development. It decouples algorithmic descriptions from underlying fabrics and offers layout-driven and architecture-driven design generation, bridging the gap between high-level designs and hardware details.

The frontend of Heterosys is HeteroRefactor, which combines dynamic invariant analysis, automated refactoring, and selective offloading. HeteroRefactor optimizes software kernels onto accelerators for common-case inputs while maintaining correctness through CPU fallback mechanisms. HeteroRefactor automatically refactors software code to make it FPGA-compatible and hardware-friendly, reducing chip resource usage through bitwidth optimization and floating-point precision tuning.

From the individual synthesizable hardware kernels, Adroit optimizes them using a static approach to identify data and control broadcasts. It analyzes data and control dependencies in the source code and reports, trading off clock-cycle latency for higher frequency. By optimizing the FPGA architecture generated by high-level synthesis tools, Adroit relieves software developers from needing to understand the underlying fabric.

As the backend, Heterosys composes multiple kernels into an optimized FPGA system using RapidIR, a comprehensive infrastructure for high-level physical synthesis optimizations. RapidIR integrates coarse-grained floorplanning with high-level pipelining, supporting hierarchical composition of heterogeneous designs from diverse sources. It automates the exploration of various physical optimization strategies, freeing programmers from designing device-specific hardware layouts for each target device.

Our research demonstrates substantial performance improvements across diverse applications and benchmarks, including genomic sequencing and large language model accelerations. Our FPGA optimization techniques achieve operating frequency improvements of 30% to over 100% compared to state-of-the-art EDA tools, resource requirement reductions of 21% to over 90%, and 51% code reduction in porting between platforms.

This dissertation contributes a comprehensive set of methodologies and tools that significantly lower the barriers to entry for heterogeneous computing, particularly FPGA acceleration. By abstracting away much of the hardware complexity, our work paves the way for broader adoption of heterogeneous acceleration in software development practices, potentially driving research innovation and performance improvements across a wide range of applications and industries.

The dissertation of Jason Lau is approved.

Miryung Kim

Anthony John Nowatzki

Glenn D. Reinman

Jingsheng Jason Cong, Committee Chair

University of California, Los Angeles

2024

Peace for all.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

Pursuing my doctoral degree is a privilege born of fortune. I am blessed with the guidance of my advisor and professors, the unwavering support of family and friends, and the financial backing of funding bodies. Living in peaceful countries and enjoying both physical and mental support have provided me with the peace needed to complete my projects. I hope others may share such privileges, and I wish for peace for all.

I am deeply grateful to my advisor and committee chair, Professor Jason Cong, for his invaluable guidance and support throughout my academic journey. His mentorship has enabled me to pursue my research interests. It has been a privilege to join the UCLA VAST lab, where Prof. Cong has fostered an inspiring environment that encourages the exploration and advancement of high-level synthesis and customized computing.

I would like to thank my Ph.D. committee members, Professors Miryung Kim, Tony Nowatzki, and Glenn Reinman, for their invaluable guidance in refining this dissertation. I am also deeply indebted to Professors Miryung Kim and Zhiru Zhang for their ongoing support and crucial contributions to my various projects.

Thank you to my collaborators: Zhe Chen, Yuze Chi, Young-kyu Choi, Kristof Denolf, Yubo Du, Zhenman Fang, Muhammad Gulzar, Eddie Hung, Jingtong Hu, Shixin Ji, Alex Jones, Alireza Kaviani, Moazin Khatti, Chris Lavin, Wuxi Li, Gai Liu, Sihao Liu, Jack Lo, Michael Lo, Pongstorn Maidee, Stephen Neuendorffer, Peichen Pan, Weikang Qiao, Zhenyuan Ruan, Yiyu Shi, Aishwarya Sivaraman, Atefeh Sohrabizadeh, Linghao Song, Xingyu Tian, Ecenur Ustun, Kees Vissers, Jie Wang, Peng Wei, Shaojie Xiang, Yuanlong Xiao, Yutong Xie, Zhuoping Yang, Hanchen Ye, Cody Hao Yu, Qian Zhang, Peipei Zhou, Yun Zhou, Jinming Zhuang, and others for their invaluable contributions and insights.

Among the collaborators, special thanks to Licheng Guo for his crucial contributions to Chapter 4, particularly in identifying and categorizing broadcast issues. I am equally grateful to Aishwarya Sivaraman and Qian Zhang for their significant work on Chapter

3, implementing integer transformation and floating-point precision optimization approaches, as well as conducting key experiments. Yuanlong Xiao has contributed to the experiments of RapidIR in Chapter 5. Together with these co-first authors, we developed comprehensive end-to-end solutions, Adroit and HeteroRefactor.

I am grateful for my friends who have stood by me through difficult times. Although I cannot name everyone, I am particularly grateful for Yangyang Chen, Licheng Guo, Xiaohan Li, Miao Wang, and Peiran Yao, in alphabetical order. To those unlisted: Your support is no less valued; my brevity stems from discretion, not indifference.

My parents, Weihua Ke and Fujin Liu, provide unwavering emotional and financial support, nurturing my growth in a secure and encouraging environment.

Zhengmei Huang, my spouse and lifelong love, is my heartbeat. We have enjoyed each other's best and suffered the worst, only making us closer. While I would not say the dissertation owes its existence to her, or paint her as the perfect partner, which is common in such acknowledgments, she is the irreplaceable one who makes me alive.

In this dissertation, large language models serve dual roles: as subjects of FPGA accelerator optimization and as my writing assistants. While the models differ, GPT-4 and Claude 3.5 Sonnet have proven invaluable in correcting grammar and improving sentence readability. All AI-corrected texts incorporated herein have been double-checked to ensure they accurately and faithfully convey my original texts' intent and meaning.

This dissertation is partially supported by RapidStream Design Automation, Inc.; 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; NSF grants CCF-1764077, CCF-1527923, and CCF-1723773; ONR grant N00014-18-1-2037; and a Samsung grant.

VITA

2014–2018	Bachelor of Science, Computer Science, Tsinghua University, Beijing, China.
2018–2022	Master of Science, Computer Science, University of California, Los Angeles, U.S.A.

PUBLICATION HIGHLIGHTS

Underline indicates co-first authors.

Jason Lau, Yuanlong Xiao, Yutong Xie, Yuze Chi, Linghao Song, Sihao Liu, Shaojie Xiang, Michael Lo, Zhiru Zhang, Jason Cong, and Licheng Guo. 2024. RapidStream IR: Infrastructure for FPGA High-Level Physical Synthesis. In *Proceedings of the 43rd IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*.

Licheng Guo, Yuze Chi, **Jason Lau**, Linghao Song, Xingyu Tian, Moazin Khatti, Weikang Qiao, Jie Wang, Ecenur Ustun, Zhenman Fang, Zhiru Zhang, and Jason Cong. 2023. TAPA: A Scalable Task-parallel Dataflow Programming Framework for Modern FPGAs with Co-optimization of HLS and Physical Design. *ACM Trans. Reconfig. Technol. Syst.*

Jinming Zhuang, **Jason Lau**, Hanchen Ye, Zhuoping Yang, Yubo Du, Jack Lo, Kristof Denolf, Stephen Neuendorffer, Alex Jones, Jingtong Hu, Deming Chen, Jason Cong, and Peipei Zhou. 2023. CHARM: Composing Heterogeneous Accelerators for Matrix Multiply on Versal ACAP Architecture. In *Proceedings of the 2023 ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)*.

Jason Cong, **Jason Lau**, Gai Liu, Stephen Neuendorffer, Peichen Pan, Kees Vissers, and Zhiru Zhang. 2022. FPGA HLS Today: Successes, Challenges, and Opportunities. *ACM Trans. Reconfigurable Technol. Syst.*

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

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

Jason Lau, Aishwarya Sivaraman, Qian Zhang, Muhammad Ali Gulzar, Jason Cong, and Miryung Kim. 2020. HeteroRefactor: Refactoring for Heterogeneous Computing with FPGA. In *Proceedings of the 42nd International Conference on Software Engineering (ICSE).*

Licheng Guo, **Jason Lau**, Yuze Chi, Jie Wang, Cody Hao Yu, Zhe Chen, Zhiru Zhang, and Jason Cong. 2020. Analysis and Optimization of the Implicit Broadcasts in FPGA HLS to Improve Maximum Frequency. In *Proceedings of the 57th ACM/EDAC/IEEE Design Automation Conference (DAC).*

Licheng Guo, **Jason Lau**, Zhenyuan Ruan, Peng Wei, and Jason Cong. 2019. Hardware Acceleration of Long Read Pairwise Overlapping in Genome Sequencing: A Race Between GPU and FPGA. In *Proceedings of the 2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM).*

CHAPTER 1

Introduction

The landscape of computing has changed significantly in recent years, driven by the slowing of CMOS technology scaling and the ever-increasing demand for computational power. This surge in the need for computational resources is particularly evident given the exponential growth of data and increasingly complex machine learning models. Consequently, there has been a growing interest in heterogeneous and customizable architectures, where each component is specialized in different aspects and tailored for certain compute patterns to improve performance [CSG11, CMH10, CGG14, CSR10]. Such heterogeneous systems typically integrate scalar processors, such as CPUs, with domain-specific accelerators, including Field-Programmable Gate Arrays (FPGAs), Neural Processing Units (NPUs), AI Engines [Xil21b], and Graphics Processing Units (GPUs). By utilizing these customizable accelerators, heterogeneous systems could potentially achieve higher performance and energy efficiency across diverse domains of applications. For instance, FPGAs, when reconfigured to customize for specific tasks, often outperform general-purpose processors by orders of magnitude [CCP16, PCC14, GLR19, SCS22].

The complex nature of real-world end-to-end applications requires a heterogeneous approach to computing resources. Composing diverse computational elements into a single system enables sustained performance improvements in specialized tasks while preserving some programmability by integrating general-purpose processors [Car20, CHK21, GZK21, KSR18, DKR18, PSS18]. For instance, modern heterogeneous platforms such as AMD's Versal ACAP (Adaptive Compute Acceleration Platform) combine ARM processors, FPGA fabric, and specialized AI Engines on a single chip [Xil22a, Xil22d, Xil22c], offering a versatile computing environment capable of addressing a broad spectrum of computational challenges. By leveraging the strengths of different architectural paradigms, heterogeneous systems can efficiently handle diverse workloads, from traditional serial tasks to highly parallel computations and custom accelerated functions.

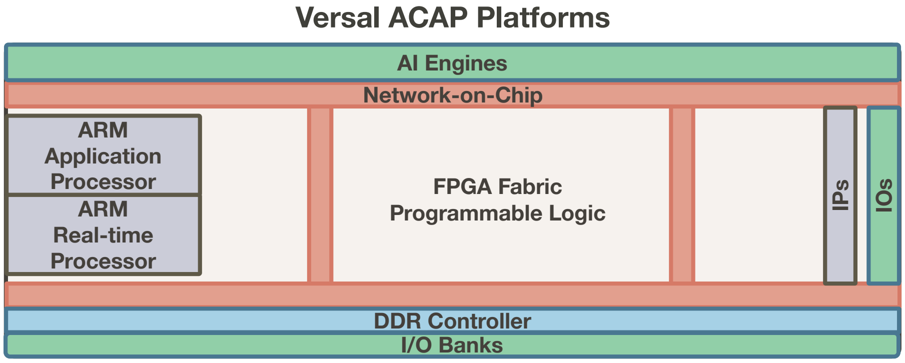


Figure 1.1: Schematic illustration of the Versal ACAP platform.

Figure 1.1 illustrates the architectural layout of the Versal ACAP platform. The chip integrates several components including ARM processors, AI Engines, memory controllers, and various control or computational intellectual properties (IPs), which handle tasks related to signal processing and cryptography. These components are interconnected through the FPGA fabric and the network-on-chip.

From this example, we can see that at the heart of this heterogeneous system, the FPGA fabric plays a central role due to its customizability. The FPGA serves as the central hub that connects and orchestrates various components while concurrently handling some customized, computation-intensive tasks for acceleration.

To support this evolution in hardware architecture and the complexity of FPGA programming, electronic design automation (EDA) tools have also advanced significantly. High-Level Synthesis (HLS) tools [Fei12, Xil21c, Int24, CGL21], for example, allow developers to describe FPGA designs using high-level languages like C/C++, OpenCL, or

Python, automating much of the low-level hardware design process. Similar advancements have been made for other components, such as the Adaptive Data Flow language for programming AI Engine arrays [Xil21b], and CUDA for NVIDIA GPU devices [Nvi11].

However, despite advancements in hardware architectures, design tools, and compilers, a significant challenge persists: making heterogeneous computing accessible to software developers. Efficiently utilizing diverse computational resources requires specialized knowledge of hardware architectures and multiple low-level programming paradigms and coding styles for each device, creating a barrier for those who lack the expertise or time to learn. The gap between the potential of heterogeneous systems and developers' ability to harness it creates a pressing need for innovative approaches and tools. These solutions must abstract most of the underlying complexity while still enabling developers to leverage the performance benefits of specialized hardware.

This dissertation develops methodologies to bridge the gap between high-level software development and the efficient utilization of heterogeneous resources. By improving the accessibility of heterogeneous computing for software developers, we aim to unlock the potential of these systems and foster innovation in the computing landscape.

1.1 Problem Statement

The use of heterogeneous computing platforms remains limited to a small subset of programmers with specialized knowledge of architectural details, driving up development costs [RLL11]. These challenges can be broadly categorized into three main areas: limited programmability support, difficulty in achieving high performance, and the time-consuming nature of porting across diverse computational architectures.

While this research focuses on the EAD tools from AMD [Xil21b, Xil21c], these challenges are also common across tools from other vendors, such as Intel and Mentor.

1.1.1 Limited Programmability Support in Domain-Specific Resources

HLS tools for FPGA accelerators aim to simplify hardware development by abstracting hardware complexity [CLN11]. Nevertheless, significant design constraints continue to burden software developers [CLL22]. These constraints often conflict with conventional software development practices, requiring substantial code modification to meet hardware-specific requirements. For instance, targeting FPGAs using HLS tools mandates several synthesizability requirements [Xil20, Xil21c], including:

1. **Pointer Restrictions.** HLS tools limit or prohibit the use of pointers, which are fundamental in software development for data structures and memory management.

2. **Static Memory Allocation.** Developers often need to pre-allocate overly conservative, static arrays rather than using dynamic memory management on heterogeneous resources, leading to resource inefficiencies and increased code complexity.

3. **Recursion Limitations.** Recursive algorithms, which are intuitive and efficient enough in software, are usually unsupported or require complex manual transformations into iterative forms when targeting heterogeneous resources.

4. **Physical Layout.** Software programmers typically create code using logical hierarchies, grouping related functionalities. However, FPGA design tools require modules to be grouped by physical affinity for quality of result or even successful routing [Con01]. Even worse, automated FPGA optimization tools aiming to solve the routing challenges often mandate a flat dataflow graph as input, requiring extensive code refactoring [GCW21, GCL23, DLS23, DLZ24, CGL21].

Similarly, programming for embedded systems, AI engines, or GPUs often requires learning specialized languages or APIs that diverge from traditional software development paradigms [Xil21b, Nvi11]. This divergence mandates substantial code rewrites to ensure functionality, often conflicting with software engineering best practices.

1.1.2 Difficulty in Achieving High Performance and Efficiency

While heterogeneous systems offer promising performance improvements, realizing these gains requires specialized hardware knowledge and expertise in optimization techniques specific to each architecture. Software developers often struggle to fully exploit diverse computational resources due to the following key challenges:

1. **Data Representation.** Optimizing for resource efficiency involves fine-tuning the bitwidth of data. Determining the minimal required bitwidth for each variable and arithmetic operation without compromising functionality requires a deep understanding of both the algorithmic and hardware implications of data representation.
2. **Micro-Architecture.** Subtle issues like high-fanout nets from data or control signal broadcasts can significantly degrade the maximum achievable clock frequency. Software developers usually lack the expertise to identify and mitigate these hardware-specific issues, since they are not apparent in high-level source code.
3. **Global Layout and Pipelining.** In large, complex designs, especially for multi-die FPGAs, physical layout is crucial for performance. Long interconnects can create critical paths that limit clock frequency. Effectively pipelining these critical paths requires an understanding of physical layout beyond software descriptions.

These challenges often require developers to think in terms of hardware architecture rather than software algorithms, a significant mindset shift for many software engineers.

1.1.3 Time-Consuming Porting Across Diverse Architectures

The heterogeneous nature of modern computing systems challenges code portability and maintenance, especially when porting code between target accelerator devices.

1. **Code Adaptation.** Ideally, existing software code should run on FPGAs with no modifications if not part of the kernel's bottleneck. In reality, developers must substantially rewrite CPU-targeted code for FPGA compatibility [LSZ20]. This refactoring is time-consuming, error-prone, and requires hardware-specific knowledge.

2. **Optimization for Hardware Fabrics.** Code targeting FPGAs often needs manual register insertion to meet timing constraints [GLC20], varying by hardware fabric. This demands detailed hardware knowledge. An automated low-level optimization approach would simplify code porting and improve design quality.

3. **Physical Layout Redesign.** Porting designs to newer FPGAs often requires substantial code changes so that they are optimized for new physical layouts [GCW21]. This change of physical layout may be due to increased die count, new built-in components (IPs), or changes in hardware interconnect. An automatic optimization solution from a single codebase targeting different FPGA devices would enhance portability and reduce development efforts.

The challenges of porting design code across target hardware devices create maintenance issues and require multiple code versions of an algorithm. Solutions automating software adaptation, handling low-level optimizations, and providing seamless portability across FPGA devices would lower entry barriers for developers.

1.2 Research Objectives

This dissertation address the challenges in our problem statement. We pursue the following research objectives (RO):

RO1 Improve the heterogeneous development experience for software programmers.

To make heterogeneous computing more accessible to software developers, we focus on: (a) Developing automated code transformation techniques that convert software-oriented code into hardware-synthesizable equivalents without manual effort; (b) Creating approaches that identify potential performance issues in the code and perform optimizations specific to target hardware; (c) Designing methods to adapt existing designs to different hardware fabrics, reducing the time and effort required for porting code across diverse computational architectures.

RO2 Bridge the gap between high-level description and efficient hardware implementation. To ensure that high-level descriptions can be efficiently implemented on heterogeneous hardware, we aim to: (a) Develop compiler frameworks that decouple hardware-specific optimizations from high-level software descriptions; (b) Create optimization techniques that identify and mitigate hardware-specific issues automatically; (c) Design strategies for automated global layout optimization and pipelining, particularly for large, complex designs on multi-die FPGAs.

We aim to create an end-to-end framework from software source code to bitstream that empowers software developers to harness FPGA-based heterogeneous computing systems without sacrificing development productivity or requiring extensive hardware design training. Our research aims to lower the barriers to heterogeneous computing, enabling a broader range of developers to leverage these resources efficiently.

1.3 Contribution Overview

In this dissertation, we present a comprehensive set of methodologies and tools designed to address the challenges faced by software developers when utilizing heterogeneous computing resources. Our contributions in this work support the research objectives of enhancing the heterogeneous development experience (*RO1*) and bridging the gap between high-level descriptions and efficient hardware utilization (*RO2*).

Our main contributions are collectively called Heterosys, a holistic framework incorporating our proposed methodologies and tools. Heterosys focuses on tackling the issues of limited programmability, performance optimization, and cross-architecture portability by offering (1) automated code transformation and optimization techniques, (2) architecture-driven design generation to improve abstraction over hardware-specific details, and (3) seamless layout-driven design optimization and porting across diverse FPGA targets. It comprises three components: HeteroRefactor, Adroit, and RapidIR.

Figure 1.2 presents the overall system of Heterosys, which begins with a software code input. This code undergoes dynamic analysis by HeteroRefactor, which refactors it into hardware-compatible code. Subsequently, the code is optimized by Adroit, which utilizes architecture information to optimize the hardware code into an architecture-aware kernel. RapidIR integrates multiple such optimized kernels with design libraries, orchestrating them into a cohesive system that is physically optimized. HeteroRefactor employs a selective offloading mechanism to ensure correctness, enabling the refactored software code to run on an FPGA-centric heterogeneous system.

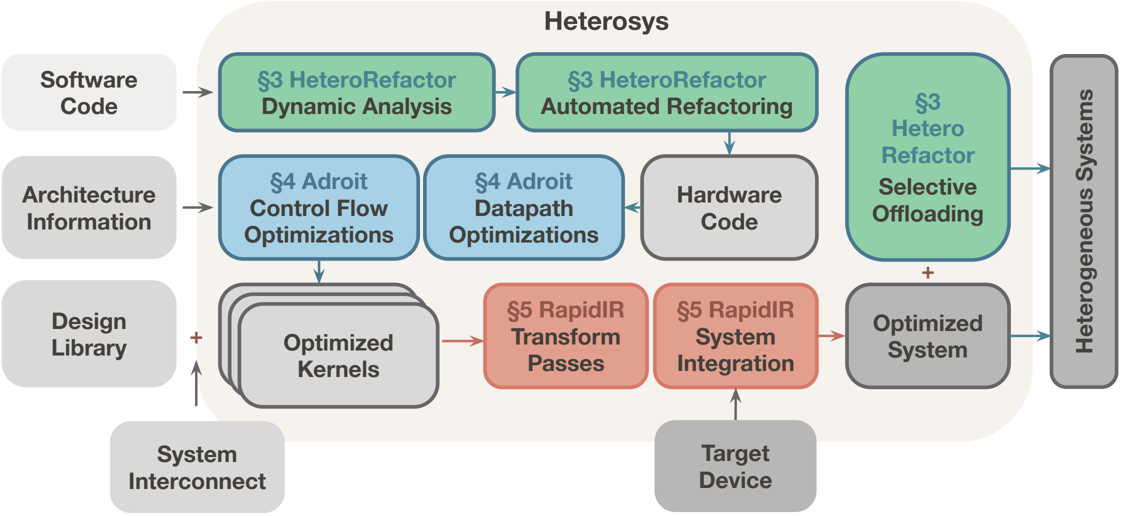

Heterosys
Software Code
§3 HeteroRefactor Dynamic Analysis
§3 HeteroRefactor Automated Refactoring
§3 Hetero Refactor Selective Offloading
Architecture Information
§4 Adroit Control Flow Optimizations
§4 Adroit Datapath Optimizations
Hardware Code
Design Library
+
Optimized Kernels
§5 RapidIR Transform Passes
§5 RapidIR System Integration
+
Optimized System
Heterogeneous Systems
System Interconnect
Target Device

Figure 1.2: System overview of the Heterosys framework, integrating HeteroRefactor, Adroit, and RapidIR to facilitate end-to-end optimization for heterogeneous systems.

1.3.1 HeteroRefactor: Dynamic Analysis and Automated Refactoring

To address the challenge of limited programmability support in domain-specific resources (*RO1*), we present HeteroRefactor, an automated refactoring tool. HeteroRefactor bridges the gap between software development practices and hardware synthesis requirements through a combination of dynamic analysis and code transformations. It ensures the correctness of the refactored code by leveraging selective offloading only when the observed dynamic behavior is consistent with the expected behavior.

The workflow of HeteroRefactor is as follows: First, a developer implements their kernel in a familiar high-level language such as C/C++. The kernel is then executed on existing test data or a representative subset of inputs to identify FPGA-specific dynamic invariants. Using this information, HeteroRefactor automatically transforms the kernel, converting pointer-based structures into synthesizable, flattened arrays and recursion into iterative equivalents. This process not only makes the code HLS-compatible but also optimizes resource usage by inferring the minimal required bitwidths for integers instead of int32 or int64, and the minimal precision of floating-point numbers, leading to reduced resource consumption and increased operating frequency at the FPGA level (*RO2*). To ensure the correctness of the transformed code, HeteroRefactor employs a selective offloading mechanism by monitoring if the observed dynamic behavior matches the assumptions during the initial dynamic analysis and refactoring process.

The impact of HeteroRefactor is fourfold:

1. **Code Complexity Reduction.** For an average recursive program of 175 lines of code (LOC), HeteroRefactor eliminates the need for an additional 185 LOC that an expert FPGA programmer would typically write to achieve HLS compatibility.
2. **Resource Efficiency Improvements.** By using tight bounds for recursive data structures (e.g., 2k depth instead of an overly conservative 16k), HeteroRefactor

achieves up to 83% reduction in BRAM usage and 42% increase in operating frequency.

3. **Integer Representation Optimization.** For integer-intensive programs, HeteroRefactor reduces bit usage by 76%, leading to 25% reduction in flip-flops, 21% in look-up tables, 41% in BRAM, and 52% in DSP resources.

4. **Floating-Point Precision Tuning.** In floating-point-intensive programs, with a specified acceptable precision loss of 10^{-4} at 95% confidence level, HeteroRefactor achieves up to 61% reduction in flip-flops, 39% in LUTs, and 50% in DSP usage.

By automating these hardware-specific code transformations, HeteroRefactor enables software developers to leverage the power of FPGAs without rewriting existing code or sacrificing their familiar programming paradigms.

1.3.2 Adroit: Architecture-Driven Optimization for Implicit Broadcasts

To address the challenge of achieving high performance in heterogeneous systems (*RO2*), we introduce Adroit, an innovative analysis methodology that targets a critical yet often overlooked aspect of HLS–implicit signal broadcasts. Our research reveals that these broadcasts, automatically inferred or created by HLS compilers in both datapath and control logic, are a major cause of frequency degradation in HLS-synthesized designs.

Adroit tackles this issue through three key contributions:

1. **Identification and Classification:** We provide the first comprehensive analysis of implicit signal broadcasts in highly-optimized designs synthesized using industrial-strength HLS tools, classifying them into data and control broadcast structures.

2. **Automated Optimization Techniques:** Adroit implements a set of simple yet effective techniques, including broadcast-aware scheduling, redundant synchronization

pruning, and skid-buffer-based pipeline control, to automatically optimize these broadcasts. These techniques have been integrated into commercial products[1].

3. **Performance Improvements:** Applied to nine real-world HLS benchmarks in experiments, Adroit achieves an average frequency improvement of 53% with minimal area overhead, with some cases showing gains of over 100 MHz.

Adroit employs an architecture-centric strategy focused on enhancing implicit broadcasts by analyzing the HLS design of a kernel and its resulting RTL. This examination helps pinpoint subtle yet impactful broadcast-related performance issues, which are then addressed using a set of optimization techniques. Adroit enables software developers to achieve significantly higher performance in heterogeneous systems without requiring deep hardware expertise. It further improves the heterogeneous development experience by providing an additional layer of abstraction over the hardware details (*RO1*).

1.3.3 RapidIR: Infrastructure for High-Level Physical Synthesis

To address the challenges of porting across diverse FPGA architectures (*RO1*) and enabling efficient hardware utilization (*RO2*), we introduce RapidIR, a comprehensive infrastructure for high-level physical synthesis. RapidIR bridges the gap between software-level design and layout-specific optimizations, providing a unified framework that supports an optimized hierarchical composition of FPGA designs from diverse sources.

RapidIR accepts multiple forms of input, including hardware kernels from the aforementioned tools, reusable design libraries, system interconnect descriptions, and target device specifications, generating a physically optimized system configured for implementation on the specified FPGA. Key features of RapidIR include:

[1] Starting from Xilinx Vivado HLS 2020.2, a prior version of AMD Vitis HLS, it allows users to manually choose the skid-buffer-based pipeline flow control method instead of the broadcast-based flow control.

1. **Intermediate Representation (IR).** RapidIR provides a flexible IR capturing both software semantics and hardware-specific information, allowing for the creation of reusable passes that support various design formats and device targets, requiring only the implementation of minimal information extractors.

2. **Transparent Integration.** RapidIR allows seamless hierarchical composition of FPGA designs from diverse sources, such as HLS-generated kernels, handcrafted RTL libraries, and vendor-specific IPs. It enables automated physical optimizations in complex designs with hybrid source formats, aiming to achieve high frequency while maintaining design productivity and code maintainability.

3. **Extensible Optimizations.** RapidIR implements automated exploration of physical synthesis optimizations, including coarse-grained partitioning, floorplanning, and pipelining. It provides an extensible architecture, supporting reusable optimization passes for various design formats and device targets.

RapidIR addresses critical limitations in current HLS approaches by automating optimizations typically performed manually by RTL experts. These include pipeline insertion to mitigate long wire delays across dies and balanced resource utilization through design partitioning. Experimental results demonstrate a state-of-the-art and consistent average frequency improvement of 40% across a range of well-researched and newly introduced FPGA architectures. Notably, RapidIR enables some previously unimplementable designs to reach a frequency of around 300 MHz.

By automating the effort required to optimize and port designs across different FPGA platforms, RapidIR enables software developers to achieve RTL-expert-level optimizations without requiring in-depth hardware knowledge of every FPGA target.

1.4 Dissertation Organization

This dissertation is organized into the following chapters:

Chapter 2. Background and Related Work. This chapter provides an overview of heterogeneous computing systems, focusing on FPGAs and their role in accelerating various applications. It discusses the current state of high-level synthesis tools, their limitations, and existing approaches to bridging the gap between software development and hardware acceleration. By demonstrating a real-world acceleration design example, this chapter motivates the work introduced in this thesis. The chapter also reviews related work in automated code transformation, performance optimization for heterogeneous systems, and cross-architecture portability.

Chapter 3: Dynamic Analysis and Automated Refactoring. This chapter presents the design and implementation of HeteroRefactor. It details the methodology for dynamic invariant analysis, the automated refactoring process for converting software constructs into hardware-synthesizable equivalents, and the optimization techniques for data representation. The chapter includes case studies demonstrating HeteroRefactor's effectiveness in reducing code complexity and improving resource efficiency for FPGA implementations.

Chapter 4: Architecture-Driven Optimization for Implicit Broadcasts. This chapter introduces Adroit, our architecture-driven approach to optimizing implicit signal broadcasts in HLS-generated designs. It provides an in-depth analysis of data and control broadcast structures, describes the optimization techniques proposed by Adroit, and presents experimental results showcasing the significant frequency improvements achieved across various benchmarks.

Chapter 5: Infrastructure for High-Level Physical Synthesis. This chapter details the design and implementation of RapidIR. It presents the flexible intermediate representation (IR), the automated exploration of physical synthesis optimizations, and the support for integrating diverse design sources. The chapter includes case studies demonstrating RapidIR's effectiveness in improving design frequency and portability across different FPGA architectures.

Chapter 6: Evaluation and Discussion. This chapter presents a comprehensive evaluation of the end-to-end frequency enhancements achieved by Heterosys when applied to a large language model (LLM) accelerator and on genome sequencing applications, domains where the challenges addressed in this dissertation are particularly salient. The chapter provides an in-depth analysis of the optimizations implemented within the overall design flow, elucidating their individual and collective contributions to performance improvement. A detailed examination of a synthetic design is conducted, offering insights into the Heterosys approach.

Chapter 7: Conclusion and Future Work. The final chapter summarizes the key contributions of this dissertation, discusses the broader implications of our work for the field of heterogeneous computing, and outlines promising directions for future research. It reflects on the progress made towards enabling software developers to effectively utilize heterogeneous computing resources and identifies remaining challenges and opportunities in this rapidly evolving field.

CHAPTER 2

Background and Related Work

The landscape of computing has undergone significant changes in recent years, with heterogeneous systems emerging as a solution to address the growing demands for computational power and energy efficiency. This chapter provides an overview of heterogeneous computing, with a particular focus on the challenges and opportunities it presents for software developers to leverage heterogeneous platforms. We begin by exploring FPGA-centric heterogeneous systems and their growing importance. The chapter then narrows to the specific challenges faced by software developers when working with these FPGA systems. We examine the current state of FPGA high-level synthesis tools, evaluating their capabilities and limitations, motivating our work. Furthermore, this chapter reviews related work in key areas relevant to our research objectives: automated code transformation techniques, performance optimization strategies for heterogeneous systems, and approaches to cross-architecture portability.

To ground our discussion in practical realities, we use the AMD Versal device VCK190 [Xil22d] and other multi-die FPGAs as exemplars of modern heterogeneous computing systems (Section 2.1). We then present a detailed analysis of existing compilers targeting these systems, highlighting their current limitations (Section 2.2). Based on a set of 12 real-world applications from a broad range of domains, we identify the deficiencies in existing solutions when implementing these applications on FPGA-centric heterogeneous systems (Section 2.3). To further illustrate the potential and challenges of heterogeneous computing, we conduct an in-depth case study on an FPGA large language model accelerator, underscoring the existing obstacles (Section 2.4). We conclude this chapter with an examination of the literature background and related work (Section 2.5).

2.1 Heterogeneous Computing Architecture

As Moore's Law began to decelerate, experts in computer architecture and systems engineering turned to specialized and parallel hardware accelerators to continue improving performance and efficiency. This shift was motivated by two key insights: firstly, different types of computations could be more efficiently executed on separate hardware specifically designed for those tasks, rather than relying solely on general-purpose processors; and secondly, that expanding to multiple computational units proves more feasible than increasing the clock speed of a single centralized component [CLL22, CLN11].

Modern heterogeneous computing systems typically integrate various computational units, including CPUs, Graphics Processing Units (GPUs), Field-Programmable Gate Arrays (FPGAs), and Application-Specific Integrated Circuits (ASICs). CPUs excel at general-purpose computing and complex control flow, GPUs are optimized for parallel processing of large datasets, FPGAs offer reconfigurable hardware for custom acceleration, and ASICs provide maximum efficiency for fixed functions. With the recent boost in the need for computational resources for artificial intelligence (AI) tasks, specialized Neural Processing Units (NPUs) or AI Engines (AIEs) [ASB19, Xil22a] are also becoming increasingly common in heterogeneous systems. The combination of these diverse components allows for optimized performance across a wide range of applications.

Take the AMD Versal VCK190 [Xil22a] as an example of a modern heterogeneous system, which integrates multiple computational resources into a single System-on-Chip (SoC). The overall architecture of the VCK190 is shown in Figure 2.1.

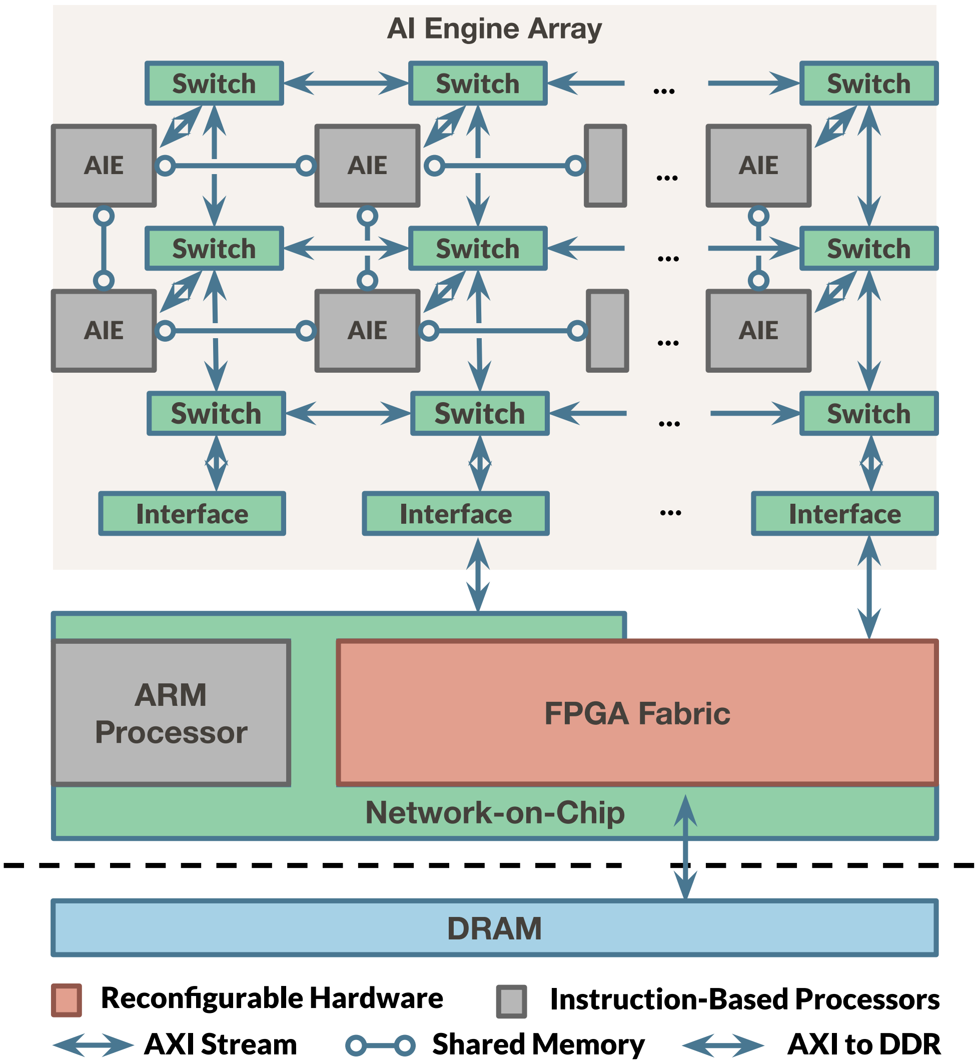


Figure 2.1: The architecture of a heterogeneous system, Versal VCK190 as an example.

The VCK190 features three main components: ARM CPU processors, FPGA fabric[1], and an AI Engine (AIE) array [ASB19, Xil22a]. The ARM processors can run Linux and handle general-purpose applications, providing a familiar environment for software developers. The FPGA fabric allows for the design of application-specific hardware, offering flexibility and customization. Distributed in the FPGA fabric are 1,968 Digital Signal Processing (DSP) elements for arithmetic operations. The AIE array consists of 8×50 VLIW processors operating at 1 GHz, supporting vector operations up to 1024 bits, which are useful for signal processing and AI applications [Xil22a, ZLY23].

At the core of the VCK190 is the FPGA fabric, acting as the central hub between the ARM processors, AIE array, DSP elements, I/O peripherals, and other intellectual property (IP) cores. It facilitates high-bandwidth communication between components, integrating ARM processors, AIE array, and I/O peripherals such as PCIe and DRAM controllers. The memory is designed to support the diverse needs of its components. The FPGA fabric includes both Ultra RAM (URAM) and Block RAM (BRAM), providing on-chip memory resources for custom logic designs. Each AIE processor tile has 32 KB of data memory and 16 KB of program memory, with the ability to access up to 128 KB of memory, including that of adjacent tiles. External DDR4 memory is also available, accessible by both the ARM processors and the FPGA fabric. Coordinated communication facilitated by the FPGA fabric helps to fully utilize the memory bandwidth between these heterogeneous components due to its customizable features [Xil21b].

This heterogeneous architecture, while powerful, creates challenges for software developers. Firstly, adapting existing code from the ARM cores to the FPGA fabric requires compatibility with HLS tools, as the FPGA fabric is configured as a circuit with specific physical locations for each logical component. This means common software constructs like dynamic memory allocation and recursion are not supported. Developers must completely rewrite code to move a workload from ARM processors to FPGA fabric. Secondly,

[1] Also referred to as Programmable Logic or PL.

optimizing for performance on heterogeneous systems involves hardware-specific skills such as clock cycle management, layout optimization to minimize routing delays, and pipelining for maximum throughput. These optimizations demand a deep understanding of the hardware architecture, including the physical location of the aforementioned components. Lastly, while reusable RTL and IP libraries provide high-performance acceleration for specific tasks on AMD FPGAs, they are often designed for specific hardware devices and lack easy portability, even across platforms from the same vendor.

Making it worse, integrating more specialized components within a heterogeneous system exacerbates physical design complexity, challenging even hardware design experts. For example, in Figure 2.2, the AMD Alveo U55C FPGA has three dies, each with some resources dedicated to the shell to communicate with the CPU host, and unprogrammable gap regions for built-in computation and communication IPs. The AMD Versal VP1552 FPGA consists of two dies, featuring networks-on-chip and an integrated ARM processor as heterogeneous components. The Intel Stratix 10 FPGA has I/O banks at the center of the programmable logic, with multi-die interconnect bridges and PCIe blocks on the sides [GCW21, Adv24a]. This architectural diversity exhibited in FPGA devices in component types, resource distribution, and wire latency, both across and within devices, requires careful design consideration to ensure proper physical layout optimization and poses significant challenges for adapting a design to another device.

While the FPGA-centric platforms are focused on in this thesis, many of these challenges are common across heterogeneous computing platforms. Other architectures, such as systems combining CPUs with GPUs or AIEs, face similar issues of code porting, hardware-specific optimization, and performance tuning. For instance, CPU+GPU systems require rewriting code for specialized programming models like CUDA or OpenCL, and embedded systems enforce coding style and resource constraints.

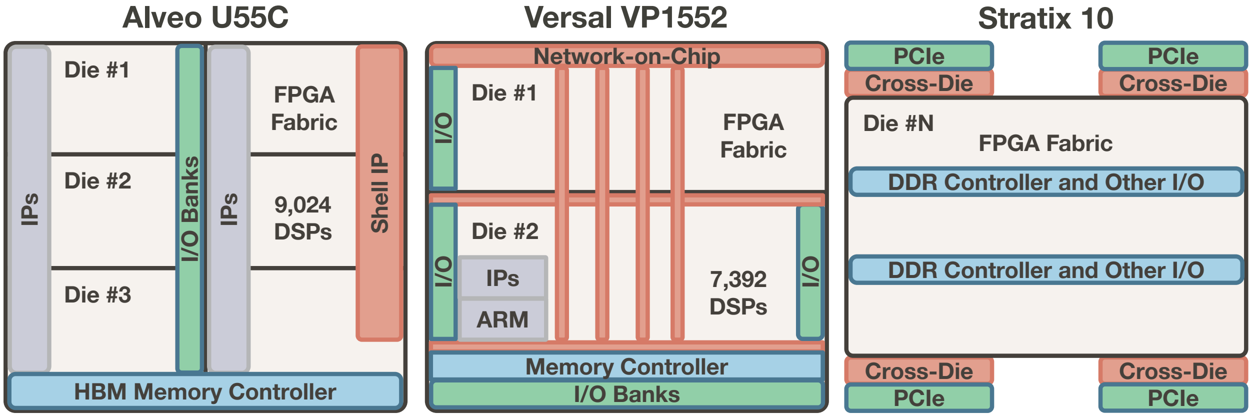


Figure 2.2: Modern FPGA devices integrate diverse heterogeneous computing and communication resources on multiple dies, complicating physical layout optimization.

2.2 Heterogeneous Compilers

As heterogeneous systems become more sophisticated, specialized compilers and electronic design automation (EDA) tools have been developed to manage complexity. These tools allow developers to describe designs at the algorithmic level and generate low-level design code, thereby reducing development effort [CLN11]. This section focuses on the existing tools and prior work for programming heterogeneous computing architectures, with a particular emphasis on FPGA High-Level Synthesis (HLS) compilers, while also briefly discussing compilers for other components in heterogeneous systems.

2.2.1 FPGA High-Level Synthesis Compilers

Traditionally, FPGAs were programmed using Hardware Description Languages (HDLs) like Verilog or VHDL at the Register Transfer Level (RTL). This approach required developers to specify register storage units, compute units, wires, and cycle-accurate behaviors, which were then mapped to the FPGA fabric through logical synthesis and physical placement and routing. However, as FPGAs have grown in size and complexity, it has gained traction to raise the abstraction level to enable the use of high-level languages like C++ or OpenCL for FPGA programming to reduce effort [CLN11, CLL22].

Modern FPGAs are highly complex devices, comprising millions of look-up tables (LUTs), thousands of embedded block memories (BRAMs), thousands of digital signal processing blocks (DSPs), and millions of flip-flop registers (FFs) [Xil19]. Each k-input LUT can implement any Boolean function with up to k inputs. To achieve the desired behavior, an FPGA must be programmed with a specific binary bitstream that specifies all the LUT, BRAM, DSP, and programmable switch configurations. Designing such a device in the traditional way using HDLs is a time-consuming and error-prone process.

As a solution, High-Level Synthesis (HLS) plays a central role in raising the abstraction level, translating untimed or partially timed specifications into low-level cycle-accurate RTL specifications [CLN11]. HLS tools perform three primary functions:

Scheduling. HLS analyzes the C/C++ code and assigns each operation from the source code to specific time slots (clock cycles). During this process, HLS estimates the delay of each operation using pre-characterized statistics for common hardware components such as adders, multipliers, registers, BRAMs (Block RAMs), and multiplexers. It transforms the untimed C/C++ code into a timed model by inserting appropriate clock boundaries, thereby breaking down the datapath into discrete clock cycles. It is important to note, however, that HLS tools typically estimate only the logic delay and base net delays and may not accurately account for additional delays caused by net fanout or congestion in more complex structures.

Binding. Operations and storage are mapped to specific resource types on the physical device, determining the number and type of hardware units (such as LUTs, FFs, BRAMs, DSPs) used for implementing functionality.

RTL Generation. From the scheduled and bound result, designs are generated in HDL format. It creates designs from a fixed RTL template, where each time slot is represented as a state in the finite state machine (FSM), controlling the hardware resource to perform corresponding operations. For fully-pipelined datapaths [ZPF16], enable or stall signals from the FSM are broadcast to every element of the pipeline for activation or flow control. Additionally, 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 completions is used as the next start signal and is broadcast to parallel modules in the subsequent stage.

Figure 2.3 depicts a typical FPGA HLS process. Initially, HLS code is parsed and compiled into an LLVM representation. Within the LLVM, data transfer and computational tasks are allocated to specific clock cycles (`T0`, `T1`, `T2`) based on an estimated delay model. During the subsequent binding phase, certain registers and operations are allocated for different values or operations in different timeslots; for instance, values `%a` and `%result` share a register in the given example. The RTL generation stage organizes the schedule for each component using fixed templates to construct FSMs. These FSMs enable or disable the resources during the respective clock cycles (e.g. `add` is enabled on `T1`, and `[%a / %result]` is used as `%a` on `T0` and `%result` on `T2`).

HLS has significantly simplified FPGA programming. However, it is still required for software developers delving into heterogeneous computing to understand both the limitations of HLS and the complexity of the synthesis process.

2.2.2 Challenges in FPGA HLS Compilers

Despite the promise of simplifying FPGA programming, HLS tools still present significant challenges for software developers. Taking AMD Vitis HLS [Xil21c] as an example, several limitations make it challenging for developers to write specifications, even though the HLS source code superficially resembles software programs:

Unsupported C/C++ Constructs. Many common software programming constructs are

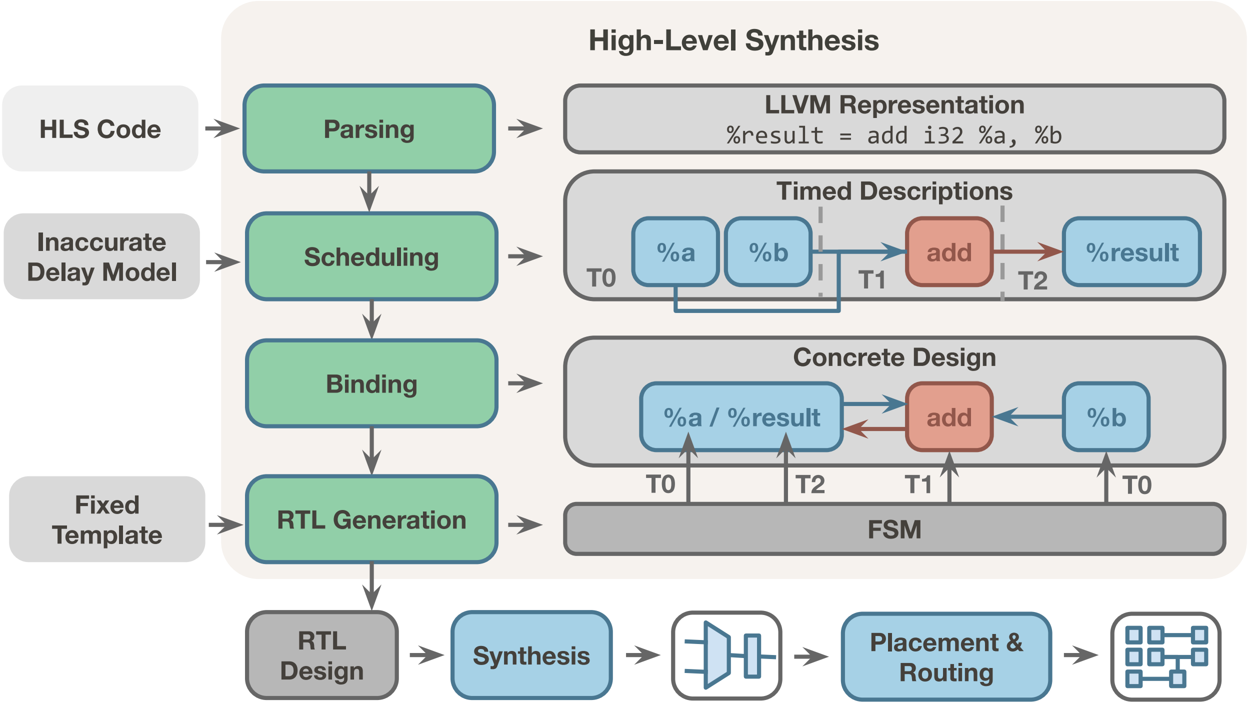


Figure 2.3: Overview of a typical FPGA HLS flow.

not supported in HLS due to FPGA hardware limitations. These include pointers, dynamic memory management, recursion, polymorphism, and exception handling. These restrictions significantly depart from what software programmers expect from C/C++ languages, creating a steep learning curve.

Inflexible Architecture Mapping. While HLS C/C++ allows the use of pragmas to specify the mapping of operations to physical units, not everything is customizable. Certain constructs map to fixed implementations that may not be suitable for all scenarios. For instance, pipelines are implemented as stalled datapaths, and operation controllers are compiled into FSMs. These fixed designs can lead to issues in hardware implementations and are challenging to work around.

Inaccurate Delay Estimation. HLS relies on delay estimation to schedule instructions into clock cycles. However, due to the lack of physical layout information, these estimations are often inaccurate. Underestimation leads to a degradation of design frequency, while overestimation wastes resources and increase latency.

Coupling of Logic and Physical Architecture. Many logical program constructs or coding styles in HLS directly impact the physical architecture of the compiled design. For example, a function call may result in a dedicated hardware module, with the number of implemented modules being non-trivial to control. Putting two function calls in the same hardware modules or implementing exclusive modules for each call often requires significant code restructuring.

Micro-Architecture Tuning Complexity. While software developers typically use standard data types like 32-bit or 64-bit integers, FPGA designs benefit from optimized bitwidths, especially when multiple operations can fit into one DSP arithmetic unit. However, this optimization requires in-depth knowledge of both the algorithm and hardware architecture, adding another layer of complexity.

Portability Issues. HLS programs often incorporate device-specific features, complicating their portability to other accelerators or different FPGAs. This limitation can hinder the adaptation of workloads across diverse components of a heterogeneous system or the deployment of a workload on a new device by reusing code.

2.2.3 Other Heterogeneous Compilers

While we focus primarily on AMD FPGAs, it's worth briefly noting that compilers for other heterogeneous components or from other FPGA vendors face similar challenges:

Other FPGA Compilers. FPGA compilers from various vendors generally face the same limitations as AMD Vitis HLS due to the inherent characteristics of FPGA hardware, with a few exceptions. Intel's HLS tools [Int24] adopt an approach that favors handshake signals over global signals, which can mitigate certain hardware implementation issues. However, this approach still suffers from inflexible architecture mapping, often resulting in excessive resource usage. The Merlin compiler [CHP16a, Sol20] takes a different approach by decoupling logic from architecture and automatically exploring physical design options. While this automation can simplify the development process, it heavily relies on automated analysis, potentially limiting opportunities for manual performance tuning by experienced developers. Despite these innovations, Merlin and other FPGA compilers continue to share the aforementioned fundamental limitations inherent to HLS tools, underscoring the persistent challenges in making FPGA programming accessible to developers while maintaining the flexibility needed for optimal performance.

AI Engine Compilers. AI Engines provide powerful instruction-based processing based on vector operations, yet programming them parallels the challenges faced in FPGA development [Xil21b]. The AMD AI Engine compilers offer separate paradigms for programming individual cores and their interconnections, using a combination of C++ with vector intrinsics (analogous to HLS) and the Adaptive Data Flow domain-specific language for core connections (analogous to RTL). This approach presents several limitations: unsupported C/C++ constructs due to 16 KB memory constraints and disabled language features, tight coupling between logical descriptions and physical architecture, such as the subtle difference between *stream* and *window*, and the necessity for complex low-level programming interfaces to achieve optimal performance [Xil21a], akin to physical layout optimization on FPGA. Though efforts like MLIR-AIE [Xil22b] aim to unify and simplify AI Engine programming, they demand considerable architectural expertise.

GPU Compilers (e.g., NVIDIA CUDA, AMD HIP). GPUs have become more accessible for experienced developers with a progressively easier learning curve. After decades of software and hardware evolution, most C/C++ constructs are now supported by compilers, although this was not the case initially. For instance, dynamic memory management on device global memory using `malloc` was only supported starting with CUDA 3.2 [Nvi11]. Nonetheless, GPU programming still demands a thorough understanding of the underlying architecture to achieve optimal performance. Portability across different GPU vendors remains challenging, and realizing peak theoretical performance often requires low-level optimizations.

DSP and Embedded Processor Compilers. Digital Signal Processors (DSPs) [KLT13] and other embedded processors, such as MicroBlaze [CCG13], often suffer from the same set of limitations as the AMD AI Engine compilers due to their similar processor architecture and resource constraints. They may require manual handling of hardware interfaces and interrupts, adding complexity for software developers.

The limitations of heterogeneous compilers highlight the persistent challenges in enabling software developers to effectively utilize diverse computing resources. While this dissertation focuses on FPGA development, many of the methodologies presented are applicable to other platforms. For instance, automated refactoring based on dynamic analysis could be applied to GPUs and embedded processors to rewrite unsupported programming constructs and optimize resource usage. Similarly, a tool akin to our physical layout optimization framework could be developed for AI Engines to accelerate the inferring of an optimized flow graph with layout-aware communication channels. These potential applications across various heterogeneous platforms form a key motivation for the research presented in this dissertation.

2.3 Challenges for Software Developers

This section examines the practical challenges faced by software developers when adapting applications for heterogeneous computing. Through a survey of 12 real-world kernels from diverse domains, we identify and categorize the primary obstacles that developers encounter. This analysis provides insight into the gaps between traditional software development practices and the requirements of heterogeneous computing, highlighting areas where improved tools and methodologies are needed.

Our survey includes a diverse range of applications, including:

3DR: 3D Image Rendering. Rosetta [ZGD18] is a set of realistic FPGA benchmarks created using HLS, where 3DR is a kernel for 3D image rendering.

CNN: Convolutional Neural Network. Cong *et al.* [CW18] proposed an automated compilation framework for generating high-performance systolic array architectures on FPGA, where a CNN accelerator is one of its applications [BSW23].

FDT: Face Detection. Srivastava *et al.* [SDM17] created a face detection accelerator based on the Viola-Jones algorithm, included in the Rosetta Benchmark [ZGD18].

GSQ: Genome Sequencing. Guo *et al.* [GLR19] designed an FPGA accelerator with co-optimization of the host program and the chaining algorithm [Li18] for accelerating long read pairwise overlapping in third-generation genome sequencing.

JAC: Jacobi. Chi *et al.* [CCW18] proposed an automated framework that takes in domain-specific language describing the stencil kernel [TYL23] and generates efficient HLS code. We use a 2D-Jacobi kernel from the framework's results to illustrate the challenges in designing a heterogeneous code generator.

KNN: K-Nearest Neighbor. Lu *et al.* [LFF20] creates a k-nearest neighbor accelerator for the Alveo U280 FPGA, using HLS kernels and a custom RTL interconnect.

LLM: Large Language Model. Chen *et al.* [CZD24] design a hybrid-source accelerator for large language model inference of the LLaMA2 model, manually optimized for a specific FPGA, writing code in a four-level nested pipeline.

MML: Matrix Multiplication. Cong *et al.* [CWY18a] design fast matrix multiplication on FPGA, serving as a fundamental component in various applications.

R2Y: RGB2YUV. Lau *et al.* [LSZ20] modify the RGB to YUV color scheme conversion program from OpenCV examples [BK08] for FPGA execution.

STB: Stream Buffer. Guo *et al.* [GLW20] design a memory to buffer stream that features a straightforward pipeline with large BRAM consumption.

STS: String Searching. Lau *et al.* [LSZ20] adapted the Aho-Corasick [AC75] algorithm, a string multi-pattern searching algorithm utilizing breadth-first search with a dynamic queue, a recursive dictionary tree [De 59], and a finite state machine.

VDC: Video Decoder. Liu *et al.* [LCN16] developed an H.264 decoder comprising over 6,000 lines of code and more than 100 functions. The design achieves real-time decoding at 34 fps with a resolution of 640×480.

From the comparison of the kernels above with their original programs, we observe that most of the developers have written many lines of code to address the challenges of current heterogeneous compilers. Additionally, some have unresolved problems in their implementations. We summarize these major challenges into three categories:

Limited Programmability Support. Many common software constructs are unsupported or restricted in heterogeneous computing environments. Examples include lack of support for pointers (**PTR**), which are fundamental in C/C++ for data structures and memory manipulation; restrictions on dynamic memory management (**DMM**), forcing developers to pre-allocate memory statically; and inability to use function recursion (**REC**), requiring manual conversion to iterative algorithms.

Difficulty in Achieving High Performance. Optimizing for heterogeneous platforms often requires hardware-specific knowledge unfamiliar to software developers. This includes optimization of integer bitwidth (**INT**) to reduce resource usage and increase parallelism; tuning floating point precision (**FPP**) to balance accuracy and performance; implementing workarounds for long hardware critical paths and high fanouts (**FAN**); and writing code that addresses the inflexible architecture mapping (**FIX**) imposed by compiler tools.

Time-Consuming Porting Across Architectures. Adapting code for different heterogeneous platforms or even different devices within the same family can be challenging. This includes rewriting code to adapt to a different programming model or constraints (**ADP**), which may involve new APIs, pragma systems, or even entirely different languages. Additionally, changing the physical layout design for a different device (**PHY**) often requires a deep understanding of the target hardware's overall structure and may require a complete rewrite of the code hierarchy.

We summarize the challenges found in each application in Table 2.1. The application name and the challenge name are abbreviated as in the brackets above. When a challenge exists in an application, either addressed by their proposed solutions or still remaining in the program, a checkmark is added to the corresponding column.

As can be seen from the table, these challenges are pervasive across the surveyed applications, with more complex applications like large language model (LLM) exhibiting a higher number of issues [ZLD24, YGB24, YMS24, NWL24, YRB22, YKW23]. The prevalence of these problems underscores the significant effort required to adapt software for heterogeneous platforms, often resulting in code that is less readable and maintainable due to the inclusion of architecture-specific details. In Chapter 6, we present the end-to-end frequency enhancement achieved through the optimization of the LLM application, within which all discussed challenges are present.

The root causes of these challenges can be attributed to four factors: (1) The level of abstraction provided by current HLS tools, while higher than RTL, still exposes many

Table 2.1: Programming challenges of heterogeneous kernels from diverse domains.

App	**Domain**	**Target**	**Programmability**			**Performance**				**Portability**	
			PTR	DMM	REC	INT	FPP	FAN	FIX	ADP	PHY
3DR	Media	Kintex-7		✓		✓	✓	✓	✓	✓	
CNN	AI	UltraScale+				✓	✓	✓	✓	✓	✓
FDT	AI	Kintex-7		✓		✓		✓	✓	✓	
GSQ	Genome	UltraScale+	✓	✓		✓		✓	✓	✓	✓
JAC	Scientific	UltraScale+				✓		✓	✓	✓	✓
KNN	AI	UltraScale+		✓		✓	✓	✓	✓	✓	✓
LLM	AI	Versal	✓	✓	✓	✓	✓	✓	✓	✓	✓
MML	Scientific	UltraScale+				✓	✓	✓	✓	✓	
R2Y	Media	UltraScale+				✓	✓		✓	✓	
STB	Network	UltraScale+						✓	✓		
STS	Network	UltraScale+	✓	✓	✓	✓		✓	✓	✓	
VDC	Media	Kintex-7	✓	✓		✓	✓	✓	✓	✓	

hardware-specific details to software developers; (2) achieving optimal performance often requires low-level optimizations that conflict with high-level programming paradigms; (3) the diversity of heterogeneous architectures makes it difficult to create portable code that performs well across different platforms, even different FPGA devices within the same family from the same vendor; and (4) current compilers and synthesis tools often make fixed architectural decisions that may not be optimal for all use cases, requiring developers to implement workarounds in their source code.

Addressing these challenges requires a multifaceted approach that includes:

Code Transformation. Developing code analysis, transformation and refactoring tools to automatically adapt software for heterogeneous platforms, abstracting hardware-specific details that are not performance-related.

Hardware-Aware Optimizations. Developing methods to bridge the gap between high-level expressions and low-level implementations, enabling architecture-aware and layout-aware performance optimizations without developer intervention.

Unified Optimization Framework. Proposing an intermediate representation to capture both high-level descriptions and low-level hardware intricacies, allowing abstract extension for multiple FPGA-centric heterogeneous devices.

The research presented in this dissertation aims to address these challenges by developing methodologies and tools that can decouple architectural details from logical programs. This approach seeks to improve programmability, automate performance optimizations, and facilitate easier porting across different heterogeneous platforms, ultimately making heterogeneous computing more accessible to software developers.

2.4 Case Study: Large Language Model Accelerator

To elaborate on the challenges software developers face when adapting applications for heterogeneous computing, we present a case study of a Large Language Model (LLM) accelerator. Since Chen et al. [CZD24] implement the algorithm from scratch with FPGA architecture details in mind, we compare their implementation with a potential original software implementation (based on llama2.c [Kar24]) to show what a software developer needs to modify if taking it as a starting point. This comparison highlights the significant modifications required to port a software application to an FPGA, aligning with the programming challenges identified in Table 2.1.

2.4.1 Pointer Usage and Memory Management

Dynamic memory allocation (**DMM**) is commonly used in software but is not supported in most FPGA compilers. In the original software implementation, memory is allocated dynamically during runtime using functions like `malloc()` and `free()`. This allows programs to request memory when needed and release it when no longer in use. This approach is flexible and efficient for handling varying memory requirements. For example, in llama2.c [Kar24], memory is allocated for an input prompt buffer:

```
char* str_buffer =
    malloc((t->token_length*2 +1 +2) * sizeof(char));
```

In contrast, FPGA implementations use static memory allocation, where the memory size and structure are determined at compile-time. Due to the lack of support for runtime memory management functions analogous to those in software, it requires designers to carefully plan and optimize memory usage ahead of time to avoid resource wastage and ensure performance, often leading to overly conservative memory usage.

```
#define INPUT_SIZE 4096
// io_pack_type is defined as a hardware optimized representation
// with reduced bitwidth (INT).
io_pack_type input_buffer[INPUT_SIZE];
```

Similarly, pointers (**PTR**) are used extensively for efficient memory access and manipulation. For example, to process text, a software programmer often uses pointers to traverse the text by incrementing the pointer to access subsequent characters and search the text by using the pointer to locate specific characters or substrings. This can enhance readability by using the same representation for a substring in the same way as the original text. For example, in llama2.c [Kar24], the text is traversed using a pointer:

```
for (char *sub = text; *sub != '\0'; sub++)
    if (/* condition... */)
        process_substring(sub);
```

However, in the FPGA implementation, pointers are not directly supported. Instead, the programmers need to iterate through the input buffer using indexed loops, and process the substring by passing the buffer and the index to the processing function:

```
io_pack_type input_buffer[INPUT_SIZE];
for (i = 0; i < input_size; i++)
    if (/* condition... */)
        process_substring(input_buffer, i);
```

This change illustrates the challenge of (**PTR**) the lack of pointer support and the lack of dynamic memory allocation support (**DMM**). Software developers must rewrite code to use fixed-size arrays and then access them using indices, leading to inefficient memory usage or limitations on input size, and requiring a restricted coding style.

2.4.2 Recursion and Algorithm Adaptation

Recursion (**REC**) is a powerful programming construct commonly used in software development for elegant and concise implementations of algorithms, particularly those with a naturally recursive structure. However, it presents significant challenges when implemented directly in hardware, such as FPGAs. This is due to the static nature of hardware designs and the potential for unbounded resource usage in recursive calls.

For example, in the original software implementation of llama2.c [Kar24], binary search is used for the efficient search of tokens. In llama2.c, the binary search uses the standard library function `bsearch()` to find a token in the vocabulary. Below, we show a simplified version of the binary search function for illustrative purposes:

```c
TokenIndex *binary_search(Token tok, TokenIndex *vocab, int size) {
    if (size == 0) return NULL;
    int mid = size / 2;
    if (tok == vocab[mid].token) return &vocab[mid];
    if (tok <  vocab[mid].token) return binary_search(tok, vocab, mid);
    return binary_search(tok, vocab + mid + 1, size - mid - 1);
}
```

This binary search algorithm is concise and intuitive, internally using recursion in its implementation. While Chen et al. [CZD24] do not implement the binary search algorithm in their FPGA accelerator, if this function is to be ported to an FPGA, this recursive algorithm would need to be rewritten as an iterative version, as shown below:

```c
int binary_search(Token tok, TokenIndex vocab[], int size) {
    int low = 0, high = size - 1;
    while (low <= high) {
        int mid = low + (high - low) / 2;
        if (tok == vocab[mid].token) return mid;
        if (tok <  vocab[mid].token) high = mid - 1;
        else                         low = mid + 1;
    }
    return -1;
}
```

This iterative version uses a while loop to implement the binary search, explicitly managing the search boundaries (equivalent to the function arguments). It is functionally equivalent but potentially less intuitive for developers accustomed to recursion.

The transformation from recursive to iterative algorithms often requires:

Explicit State Management. Developers must manually track the state that is implicitly handled by the call stack in recursive implementations. In the binary search example, the `low` and `high` indices, which were `vocab` and `size` in the recursion, are explicitly managed in the iterative version. These variables are not used after the function returns; otherwise, they would need to be stored in a separate stack.

Control Flow Adaptation. The control flow of the recursion must be translated into loop conditions, as shown in the `while` loop in iterative binary search, where the control flow continues looping until the search boundaries meet. If the function has multiple recursive calls, the iterative version needs to manage these cases explicitly.

This rewriting process can make code less intuitive and harder to maintain. It also requires developers to think in a more hardware-oriented manner, considering aspects like fixed resource utilization and deterministic execution paths.

Moreover, some algorithms that are naturally expressed recursively (e.g., tree traversals, merge sort [QGF23, QOG21], graph algorithms [CGC22]) become substantially more complex when rewritten iteratively. This can lead to increased development time and a higher likelihood of introducing bugs. We will present these more complex examples of algorithm adaptation in Chapter 3.

2.4.3 Data Representation and Precision

Optimizing data representation is crucial for efficient FPGA implementations, often involving reducing the bitwidth of integers (**INT**) and adjusting floating-point precision (**FPP**). These optimizations are essential for maximizing resource efficiency on FPGAs to improve performance but require careful consideration of the applicable range in the application domain, and trade-offs between result precision and design efficiency.

In the original software implementation of llama2.c [Kar24], standard integer and floating-point types are used without concern for bit-level optimization:

```
int *prompt_tokens = (int *)malloc((strlen(prompt)+3) * sizeof(int));
float *q, *k, *v, *wq, *wk, *wv;
matmul(q, xb, wq + ldimdim, dim, dim);  // ...
```

This approach is natural for software developers, who typically rely on the CPU's native support for these data types and don't need to consider the underlying bit-level representations. Even if the input data does not require the full range of the data type, reducing the bitwidth is not a common practice in software development due to the negligible performance improvements on CPUs. In fact, the performance can sometimes be even degraded due to the overhead of bit manipulation.

In contrast, the FPGA implementation by Chen et al. [CZD24] requires careful optimization of data types to reduce resource usage and fit more processing elements.

```
ap_uint <15>             prompt_tokens[INPUT_SIZE];
ap_fixed<8,  /*...*/>    k, v, q;
ap_fixed<24, /*...*/>    wk, wv, wq;
ap_fixed<32, /*...*/>    result;
result = k * wk + v * wv + q * wq;
```

This FPGA version uses custom bitwidths for integers and fixed-point arithmetic instead of `float`. The `ap_uint<15>` type for `prompt_tokens` indicates that 15 bits are sufficient to represent the token values, potentially saving significant FPGA resources compared to using 32-bit integers. Similarly, the matrix multiplication operations are implemented using fixed-point arithmetic with carefully chosen bitwidths.

These transformations require software developers to (1) analyze the range and precision requirements of each variable in their algorithm, (2) choose appropriate bitwidths that balance accuracy and resource usage, and (3) carefully track and manage the growth of bitwidths through arithmetic operations to prevent overflow or loss of precision.

This bit-level optimization is typically unfamiliar to software developers and requires a deep understanding of both the algorithm's numerical properties and the FPGA's resource constraints. It also introduces the risk of subtle numerical errors that may not be present in the original implementation, requiring careful monitoring of results.

2.4.4 Architecture-Specific Optimizations

FPGA implementations often require architecture-specific optimizations to achieve high performance. This includes addressing issues like fan-out (**FAN**) and inflexible architecture mapping (**FIX**). These optimizations often require a deep understanding of the underlying hardware architecture and can significantly impact the structure of the code.

In software implementations, developers typically do not need to consider low-level architectural details. For example, in a typical software implementation, the following code snippet in Chen et al.'s work [CZD24] might look innocent and high-performance:

```
for (int j = 0; j < block_size_b; j++) {
  #pragma HLS pipeline
  ap_int<64> acc_temp = block_C_drainer[j];
  for (int i = 0; i < block_size_a; i++) {
    ap_int<32> outp0_dp = acc_temp.range(31, 0);
    ap_int<32> outp1_dp = acc_temp.range(63, 32);
    ap_int<8> outp0 = outp0_dp * buf18[ps_offset + i];
    ap_int<8> outp1 = outp1_dp * buf18[ps_offset + i];
    outp_data_pack_0.range(i*8 + 7, i8) = outp0;
    outp_data_pack_1.range(i*8 + 7, i8) = outp1;
  }
  outp[j] = (outp_data_pack_1, outp_data_pack_0);
}
```

However, there are two pitfalls that are difficult to detect even for an FPGA HLS expert without conducting a thorough analysis of the generated RTL code.

First, the `acc_temp` variable is read in every iteration of the inner loop. In hardware, this generates a fan-out broadcast to each hardware entity of the loop iterations. Specifically, each iteration of the inner loop will generate a hardware subentity that occupies a physical resource on the FPGA, with the `acc_temp` variable being broadcast to all of them (**FAN**). Due to the limited physical routing resource on an FPGA, this can cause congestion and reduce the maximum achievable clock frequency.

Second, as the hardware entities inside the outer loop are created in parallel to work in a pipeline (`#pragma HLS pipeline`), they require a centralized control mechanism to ensure synchronization. This control mechanism is generated by the HLS tool and is typically implemented as a state machine. However, the state machine produced by the HLS tool may often not be flexible or optimal for the specific application (**FIX**). In this example, the pipeline depth of the outer loop is high, and when the input is not ready, a stall signal is generated and broadcast to each hardware entity. This control broadcast signal can also cause congestion and reduce the achievable clock frequency.

Figure 2.4 showcases these two potential challenges during the synthesis into hardware. The first challenge, depicted in the fan-out example, involves broadcasting the data stored in `acc_temp` to every `i = 0, 1, ..., A` iteration required for computing the result of `outp0_dp`. The second challenge, observed in the fixed architecture example, relates to the control signals of pipeline stages for the `j` loop. These stages are controlled by both the data source and the sink that receives the output. Hence, a stall in either the data source or sink results in broadcasting the stall signal across all pipeline stages.

To address the performance problems, an expert HLS developer will usually rewrite the code to reduce the fan-out or add pipeline registers to the broadcast signals and manually optimize the generated RTL to reduce the control signal broadcast. For instance, by adding a pipeline register to the `acc_temp` variable to mitigate the fan-out issue:

```
for (int j = 0; j < block_size_b; j++) {
  // ...
  for (int i = 0; i < block_size_a; i++) {
    ap_int<32> outp0_dp = HLS_REG(acc_temp).range(31, 0);
    // ...
```

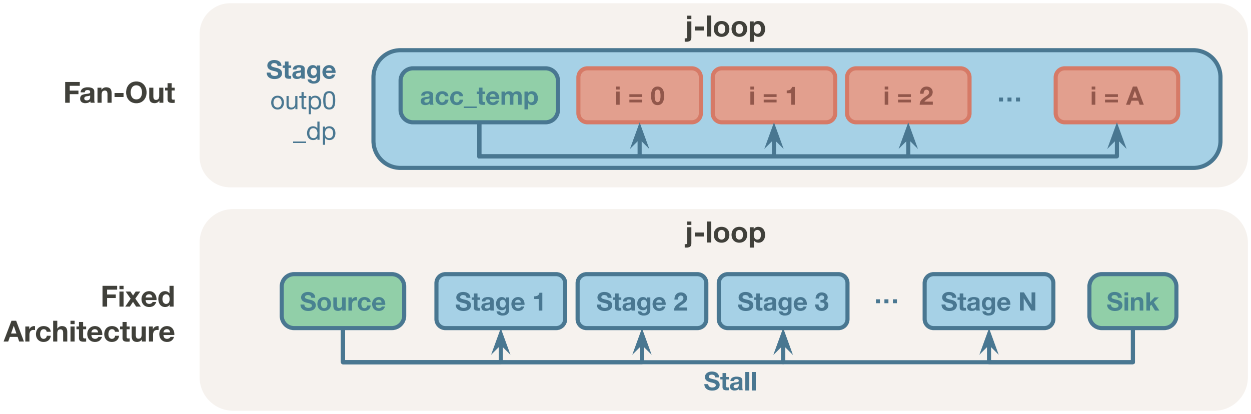


Figure 2.4: Fan-out data signals and fixed architecture templates of the pipeline stall signals in the LLM design [CZD24], potentially degrading operating frequency.

```
  }
}
```

These optimizations require a deep understanding of the HLS tool and the underlying FPGA architecture, which are foreign to typical software development practices.

2.4.5 Adaptation and Portability Challenges

Adapting code for different architectures (**ADP**) and ensuring portability across different FPGA devices (**PHY**) present significant challenges in heterogeneous computing.

In typical software, the code structure closely follows the logical flow of the algorithm. For example, in llama2.c [Kar24], the main processing loop might look like this:

```
encode(tokenizer, prompt, 1, 0, prompt_tokens, &num_prompt_tokens);
float* logits = forward(transformer, token, pos);
next = sample(sampler, logits);
char* piece = decode(tokenizer, token, next);
```

This code is straightforward, with clear separation between encoding, forward pass, sampling, and decoding steps. It's also highly portable across different CPUs with

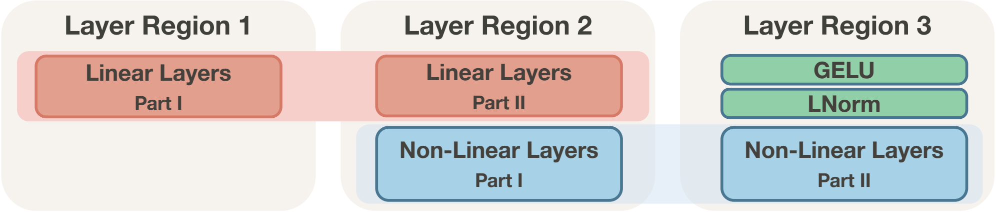


Figure 2.5: The physically-aware hierarchy of the LLM FPGA accelerator [CZD24].

minimal changes required. However, when adapting this code for FPGA implementation, as done by Chen et al. [CZD24], the structure changes dramatically:

```
layer_region_1(inp_addr_0, inp_addr_1, inp_addr_2,
    wk_addr, wv_addr, wq_addr, outp_k, outp_v, outp_q, outp_inp);
layer_region_2(w_ds0_addr, outp_k, outp_v, outp_q, outp_inp, outp_ln0);
layer_region_3(w_ds1_addr, w_ds2_addr, outp_ln0, outp_addr);
```

The logical flow of the algorithm is obscured by low-level hardware details in this modified version. In this version, the source code is structured into three functions, each in a different hardware region, with `layer_region_1` placed on the first FPGA chip die, `layer_region_2` placed on the second chip die, and so on. This code modification is illustrated in Figure 2.5. Here, the integral functionalities of Linear Layers are divided to fit into two distinct regions: Layer Region 1 and Layer Region 2.

With this methodology, originally integral functions are distributed across multiple physical regions, involving complex data dependencies and communication channels. Besides, this structure is highly optimized for the specific FPGA architecture but is difficult for software developers to understand and maintain (ADP). Additionally, porting to another FPGA platform requires extensive design restructuring (PHY).

2.4.6 Other Out-of-Scope Optimizations

There are some optimizations that are out of the scope of this work but are crucial for FPGA implementations. These optimizations include parallelism, pipelining, and memory hierarchy. While our research focuses on bridging the unaddressed gap between software development and heterogeneous computing, it is important to acknowledge these additional aspects where prior work has been extensively researched.

In FPGA programming, the interface between the host system and the FPGA device, or between the FPGA fabric and its peripheral IPs, must be explicitly defined. This interface is typically described using pragmas in the top-level function of HLS code. For instance, in Chen et al.'s work [CZD24], the top-level function is defined as follows:

```
void llm(io_pack_float *inp_0, io_pack_float *inp_1, /* ... */) {
  #pragma HLS interface m_axi port=inp_0 offset=slave bundle=gmem0
  #pragma HLS interface m_axi port=inp_1 offset=slave bundle=gmem1
  // ...
}
```

In this example, developers use specialized pragmas to specify that the inputs are passed to the FPGA design via memory-mapped AXI buses. In comparison, software developers are not accustomed to defining such interfaces explicitly, as the function parameters are passed through the stack or registers.

Another critical optimization is the use of parallelism and pipelining to maximize resource utilization and performance. In FPGA implementations, developers often use pragmas to specify parallelism and pipelining directives to the HLS tool. For example, in Chen et al.'s work [CZD24], the `#pragma HLS unroll` directive is used to specify that the loop's content `PE_int8_int16` function call should be executed in parallel.

```
for (int m = 0; m < block_size_a; m++) {
#pragma HLS UNROLL
  for (int n = 0; n < block_size_b; n++) {
  #pragma HLS UNROLL
    PE_int8_int16(A_fifo[m][n], A_fifo[m][n+1],
                  B_fifo[n][m], B_fifo[n][m+1],
                  C[m][n], inp_len);
  }
}
```

This is similar to the use of OpenMP pragmas in software development to specify parallelism. However, the FPGA pragmas are more low-level, as specifying excessive parallelism will oversubscribe the FPGA resources and cause compilation failures, while software can typically tolerate that using lightweight threads and context switching. Additionally, the parallelism specified in HLS must be supported by the concurrently designed memory hierarchy, such as array partitioning, to be described in the next paragraph, which is usually not a concern in software development.

Efficient management of the memory hierarchy is crucial for FPGA performance. This includes optimizing device memory access by packing data into the full width of DRAM accesses (typically 512 bits), splitting memory into multiple banks for parallel access, implementing techniques like double buffering to overlap communication with computation [CFH18], and explicitly managing on-chip caching through data tiling, batching, or reuse strategies [SMM03, PZS13, CCW18]. A simple example of memory hierarchy optimization is shown in Chen et al.'s work [CZD24], where the `C` matrix is partitioned into smaller blocks for parallel memory access:

```
ap_int<64> C[block_size_a][block_size_b] = {0};
#pragma HLS ARRAY_PARTITION variable = C complete dim = 1
#pragma HLS ARRAY_PARTITION variable = C complete dim = 2
```

Solutions to these challenges are outside the scope of this thesis, as they are well addressed in the literature, such as polyhedral model-based loop transformations [BBK08, CW18, PZS13, ZLC13], automatic parallelization exploration [CWY18a, KPZ16, YWG18, GLW20], data tiling, batching, or reusing for efficient memory hierarchy or movement [CHP16b, CWY18a, KPZ16, PSK15, PZS13, CCW18], and automatic device and host interface generation [CHP16b, YWG18, RHL18]. Instead, we focus on the challenges that have not been adequately addressed in prior work.

2.5 Related Work

As we discussed in Section 2.2, this dissertation focuses on three pressing challenges in development for heterogeneous computing platforms by software developers: (1) improving **programmability** by allowing more code constructs, (2) boosting **performance** by reducing resource usage and performing architecture-specific optimizations, and (3) providing a flexible and extensible optimization framework for **portable** physical optimizations. We present related work in these three directions in the following subsections.

2.5.1 Improving Programmability

This part discusses related work in two key areas enabling software developers to productively program for heterogeneous computing resources with their familiar paradigms: dynamic invariant analysis and support for recursive data structures.

2.5.1.1 Dynamic Invariant Analysis

Dynamic invariant analysis plays a central role in our approach to enabling refactoring from software programs to synthesizable counterparts for FPGAs, as FPGA mandates static resource allocations, distinct from software practices. This technique has been extensively explored in the software engineering community, with applications ranging from program understanding to automated debugging and refactoring.

One key piece of work in this area is Daikon [EPG07], which generates 22 types of invariants for C, C++, and Java programs. Daikon's approach to inferring likely invariants from program executions has been influential in the field. It uses dynamic analysis to understand program behavior. Building on this work, Kataoka et al. [KEG01] use Daikon to apply dynamic invariant detection to code refactoring, specifically for suggesting appropriate API changes based on observed program behavior.

While these approaches have proven effective in traditional software contexts, they typically require representative data sets to infer accurate invariants. Our proposed solution differs in this aspect, as it does not require representative data a priori. Instead, we leverage selective offloading to ensure correctness, allowing developers to use systematic test generation tools [GKS05, GLM08, SMA05] or test minimization techniques [HO09, TG05] to infer invariants without compromising on correctness.

2.5.1.2 Support for Recursive Data Structures

Enabling the use of recursive data structures in heterogeneous computing, particularly on FPGAs, has been a long-standing challenge. The root of this challenge lies in the fundamental difference in memory models between traditional CPUs and FPGAs. Unlike the unified memory model of CPUs, in HLS for FPGAs, each array has a separate address space, making it difficult to implement pointer-based data structures.

Several methodologies have been proposed to address this challenge. SynADT [XT16] offers an HLS library for representing linked lists, binary trees, hash tables, and vectors. It internally uses arrays and a shared system-wide memory allocator [XT15] to emulate these structures on FPGAs. While SynADT provides a valuable step towards supporting complex data structures in HLS, it is limited to a predefined set of structures and requires manual refactoring by developers. Thomas et al. [Tho16] took a different approach, using C++ templates to create a domain-specific language (DSL) that supports recursion in HLS. This method allows for more flexible recursive structures but comes at the cost of requiring extensive rewriting of control statements using lambda functions. Another approach, exemplified by CHiMPS [PEB09], uses FPGA distributed memory as caches, requiring a centralized main memory. While the approach of caching on FPGA can simplify the implementation of recursive structures, it may not be optimal for all FPGA designs and can introduce performance bottlenecks. Ahmad et al. [ADZ24] proposes fast Strassen's matrix multiplication on FPGAs with carefully designed architectures.

It's worth noting that similar challenges have been faced in other heterogeneous computing platforms. For instance, early versions of CUDA for GPUs [Nvi11] and OpenCL [SGS10] had limitations on dynamic memory management. CUDA, for example, did not support dynamic memory allocation on device global memory using `malloc` until version 3.2 [Nvi11]. To work around these limitations, researchers have developed custom memory allocators for GPUs [HRJ10, SKK12, AP14]. These allocators can work with arbitrary types and serve as replacements for `malloc` on various memory types. However, they still typically require manually specifying a heap size.

The key difference between GPU and FPGA approaches lies in the underlying memory model. GPUs provide a single address space with regular access widths, similar to CPUs, which simplifies the implementation of dynamic memory management. FPGAs, on the other hand, lack this unified address space, making the implementation of recursive structures and dynamic memory allocation significantly more challenging.

2.5.2 Improving Performance

In this part, we discuss related work in various areas of performance optimization, focusing on techniques that can be applied to improve the efficiency of FPGA-centric heterogeneous systems while maintaining a high-level programming model.

2.5.2.1 Integer Bitwidth Optimization

Integer bitwidth optimization is crucial for efficient FPGA implementations, as it directly impacts resource utilization and power consumption and allows for parallelization by duplicating processing elements. Klimovic et al. [KA13] propose an approach that optimizes FPGA accelerators for common-case inputs by reducing bitwidths using both bitmask analysis and program profiling [GA13]. Their method triggers a software fallback function when inputs exceed the common-case range. They estimate an average area reduction of 28% for accelerators with their bitwidth optimization method.

In contrast, static analysis methods like Bitwise [SBA00], which propagates bitwidth constraints based on bit flow graphs, and MiniBit [LGM05], which minimizes integer and fixed-point data signals using affine arithmetic, often result in over-approximation. Cong et al. [CGL09] employ a combination of affine arithmetic, general interval arithmetic, and symbolic arithmetic methods to optimize fixed-point data. While these static approaches provide valuable insights, they may be overly conservative for practical applications.

Our work differs from these approaches by focusing on dynamic analysis and automated refactoring, which can potentially achieve more aggressive optimizations while maintaining correctness through runtime checks and fallback mechanisms.

2.5.2.2 Floating-Point Precision Optimization

Optimizing floating-point precision can similarly reduce resource utilization and create spaces for parallelization. Several approaches offer HLS libraries for variable-width floating-point computation units but often need manual developer intervention. For example, Thomas [Tho19] introduces an HLS backend for generating customized floating-point accelerators using C++ template-based, parameterized types. However, users must manually specify bit widths for exponents and fractions, posing challenges for software developers lacking hardware expertise.

FPTuner [CBB17] uses static analysis for automatic precision-tuning of expressions, supporting single, double, or quadruple precision. Precimonious [RNN13] employs dynamic analysis and delta-debugging to identify lower-precision instructions that satisfy user-specified precision loss constraints. These tools provide automation but do not support arbitrary-width floating-point types or integration with HLS workflows.

Our proposed work aims to automate the process of floating-point precision optimization within the context of HLS, allowing software developers to benefit from these optimizations without requiring in-depth knowledge of floating-point representations or FPGA-specific implementation details.

2.5.2.3 High-Fanout Signal Optimization

High-fanout data dependencies can significantly impact the performance of FPGA designs, particularly in terms of achievable clock frequencies. While fanout optimization has been extensively studied in logic synthesis [PB91, HKP84, SS90] and physical design [OC97, BL03, WMP03, Wea08], these approaches are often limited by the cycle-accurate timing specifications of RTL inputs. For example, they cannot arbitrarily divide broadcast delays across multiple clock cycles, and techniques like retiming [Wea08, VHB06] are constrained by the availability of registers on critical paths.

Our work focuses on behavior-level optimizations that can change the schedule of broadcasts, offering more flexibility than traditional backend optimizations. This approach allows for significant frequency gains even in designs that have already undergone modern backend broadcast optimization. Cong et al. [CGH18] propose using a multi-level broadcast tree for control signals, but their approach requires iterative tuning to achieve a satisfactory tree topology.

2.5.2.4 Optimization of HLS Data Broadcast

Cong et al. [CWY18b] address a particular case of the data broadcast problem by attempting to alleviate critical paths when accessing large buffers that may be mapped to scattered BRAM units. However, their approach requires explicit user intervention and iterative tuning to explore the best topology. Moreover, it is limited to rearranging data interconnects between external ports and explicitly-defined processing elements, without addressing fine-grained datapaths.

2.5.2.5 Physically-Aware Optimization for HLS

Physically-aware optimization in HLS is crucial for bridging the gap between high-level descriptions and efficient hardware implementations. Zhang et al. [ZGR14] propose an iterative approach that runs placement and routing to calibrate delay information used by HLS. However, this method incurs significant compilation time overhead and may not address timing issues caused by auto-inferred control logic.

Zhao et al. [ZTD15] and Tan et al. [TDG15] propose considering technology mapping for logic operations to improve delay predictions. Fujiwara et al. [FKY15, FKY16] model clock skew at the behavior level. Cong et al. [CLL12] introduce metrics to assess the layout-friendliness of an RTL netlist, while Tatsuoka et al. [TWO15, TK18] identify source code lines leading to MUX and deMUX structures. TARO [CCL23] is an optimization technique applied to free-running dataflow kernels to minimize control signal overheads.

Our work builds on these efforts by developing a comprehensive framework for physically aware optimization, transparent to high-level software descriptions.

2.5.2.6 Other Automated Optimizations for HLS

Researchers have developed various methods to optimize hardware-software integration to streamline the development process. One focus is the automatic generation of device and host interfaces, which enables seamless integration between FPGA hardware accelerators and software components [CHP16b, YWG18, RHL18].

To optimize the performance of hardware accelerators, it is crucial to explore parallelization opportunities. Researchers have proposed various automated techniques to identify and exploit parallelism in algorithms [CWY18a, KPZ16, YWG18, GLW20]. These methodologies employ strategies such as loop unrolling, pipelining, and task-level parallelism. Notably, AutoSA [WGC21] and SODA [CCW18] generate hardware-efficient parallel architectures, including systolic arrays and stencil architectures, directly from software descriptions with constrained computation patterns. Furthermore, AutoDSE [SYG22] and HARP [SBS23] utilize automated exploration techniques to determine optimal pragma insertions for leveraging parallelism.

Efficient data movement between off-chip memory and on-chip resources is also crucial for reducing latency and power consumption. To address this challenge, researchers have developed various methods to optimize data transfer and minimize memory access latency [CHP16b, CWY18a, KPZ16, PSK15, PZS13, CCW18]. These techniques incorporate strategies such as data reuse, memory partitioning, and scheduling of memory operations to improve overall system performance.

2.5.3 Extensible Optimization Framework

Enabling software developers to effectively leverage heterogeneous computing resources requires not only high-level abstractions but also powerful, extensible optimization frameworks that can adapt to diverse hardware targets and design methodologies. This subsection discusses related work in the areas of High-Level Physical Synthesis (HLPS), intermediate representations (IRs), and dataflow design methodologies.

2.5.3.1 High-Level Physical Synthesis

High-level physical synthesis methodologies have been extensively explored in recent years [GCW21, LWK22, DLS23, DLZ24, GCL23, KTC23, NBN23, LLC23, MGC23, XHP22, XPN24], aiming to bridge the gap between high-level descriptions and efficient physical implementations. These approaches focus on integrating physical design considerations into the high-level synthesis process, enabling concurrent optimizations for maximum frequency.

A notable example is AutoBridge [GCW21], which improves timing by considering layout information during HLS stages, particularly for high-frequency multi-die FPGA designs. However, AutoBridge is limited to one-level pipelining with streaming ports at the top-level function in a dataflow manner, constraining software developers working on complex, hierarchical designs needing more flexible pipelining strategies.

Our work aims to address these limitations by providing a comprehensive framework that supports pipelining at arbitrary hierarchy levels and can accommodate hybrid-source designs. This approach allows for greater flexibility in optimizing complex systems, making it more accessible for software developers working with a mix of high-level software descriptions and optimized low-level hardware libraries.

High-level physical synthesis methodology for FPGAs has been significantly influenced by coarse-grained partitioning-based optimizations. Xiao et al. [PXM18, XPB19, XMB22, XAD20] propose accelerating compilation using partial reconfiguration (PR) with a pre-compiled network-on-chip. While reducing compile time, this approach necessitates application decomposition into fixed PR pages, potentially causing fragmentation and bandwidth limitations. Their subsequent work, HiPR [XHP22, XPN24], introduces an automated PR overlay generation system, adapting to application-specific requirements while maintaining efficiency.

Notably, recent advancements in high-level physical synthesis have significantly advanced co-optimization and design space exploration. In one of those work, Du et al. [DLS23] proposed an important framework for co-optimizing HLS directives and floorplans on multi-die FPGAs, employing latency-bottleneck-guided search and incremental floorplan legalization. This approach yields up to 8.78x performance improvement and 693x-4925x faster convergence. Their follow-up work [DLZ24] addressed limitations of synthesis-based methods by introducing an analytical QoR model-based directive search integrated with incremental floorplanning, achieving a 1.40x improvement over their previous results.

2.5.3.2 Intermediate Representations

Intermediate Representations (IRs) play a crucial role in enabling effective optimizations and transformations in compiler frameworks. Several IRs have been proposed to address various aspects of hardware design and synthesis, including MLIR [LAB21], CIRCT [LLV], Yosys IR [WGK13], ScaleHLS [YHC22], CIRRF [VPN10], HIR [MB24], and Xilinx IPI [Adv24b]. These IRs focus on different aspects of the compilation process, such as fine-grained logic, datapath descriptions, operation schedules, and IP integration.

However, existing IRs lack the necessary infrastructure to fully support the unique challenges posed by heterogeneous computing, particularly in handling current designs and enabling physical optimizations for FPGAs. Our research addresses this limitation by developing a reusable IR that can capture both high-level design intent and low-level physical constraints. This IR is designed to support a wide range of input formats and target devices, enabling more comprehensive and flexible optimizations that can benefit software developers working across diverse heterogeneous platforms.

2.5.3.3 Optimization for Other Targets

There exists a range of acceleration frameworks designed for platforms other than FPGA devices. To name a few, TensorFlow XLA [ABC16] is a domain-specific compiler for linear algebra that optimizes TensorFlow computations. NVIDIA's TensorRT [JKH22] is a deep learning inference optimizer and runtime that enhances the performance of deep learning applications on GPU platforms. Apache TVM [CMJ18] is an end-to-end deep learning compiler stack that optimizes deep learning workloads across diverse hardware backends. MLIR [LAB21] provides a unified infrastructure for high-performance machine learning compilers, enabling optimizations across different levels of abstraction. GMorph [YYX24] is a deep learning compiler that optimizes model execution on various hardware platforms by merging computation graphs and sharing similar operations across models to achieve significant speed-ups. Ruzhanskaia et al. [RXC24] uses programmed I/O to optimize communication between CPU and peripherals.

The extension of optimization frameworks to emerging applications and FPGA devices has the potential to be utilized in applications that demand real-time performance and energy efficiency. These applications include live streaming [DXX24], virtual reality and 3D data acquisition with surface reconstruction [LL20], the security and privacy protection of volumetric video streaming [TFX20, TPF23], wearable technology [JLD22], medical imaging [GMN23, GSF22], and the generation of protective perturbations for image recognition [YTP22, TYL24]. Evaluating geometric disparities between simulated and experimental structures through computer-vision-based methodologies also requires computational efficiency and real-time capabilities. These algorithms encompass feature extraction, pattern recognition, and symmetric difference analysis [XDC24]. FPGAs, with inherent parallelism and low latency, could facilitate defect detection and expedite feedback during design and manufacturing.

2.5.3.4 Dataflow Design Methodologies

Our research leverages dataflow to create an accessible and flexible optimization framework for software developers. By integrating dataflow principles into our IR and optimization passes, we enable high-level expression of parallel and pipelined computations.

Heterogeneous computing benefits from dataflow design methodologies. Various models, including Kahn Process Network (KPN) [Gil74] and Synchronous Data Flow (SDF) [LM87], offer different balances between expressiveness and analyzability. More constrained models like SDF allow for more precise throughput analysis [GGS06]. Du et al. [SDL24] propose an end-to-end solution for vertex-parallel graph processing that leverages dataflow methodologies. Their approach incorporates software-hardware co-optimization techniques to enhance scalability in graph processing systems.

Latency Insensitive Theory [CS00, LK03, LK06, CC07, AB18] enables analytical throughput calculations with SDF-like constraints. Research has focused on optimizing buffer placement in dataflow circuits and integrating IP blocks with varying latencies through delay insertion techniques. Our work builds on these principles, inserting delays to resolve timing issues and optimize throughput in heterogeneous computing designs.

CHAPTER 3

Dynamic Analysis and Automated Refactoring

The advancement of high-level synthesis (HLS) tools has raised the programming abstraction for heterogeneous computing, transitioning from hardware description languages (HDLs) to C/C++ environments. However, this progress has not fully bridged the gap between software development practices and customized hardware resources. Software developers still face challenges leveraging heterogeneous computing platforms, particularly with constructs like pointers, dynamic memory management, and recursion. These constructs, fundamental to many algorithms, are often incompatible with HLS tools and require manual refactoring to be synthesizable for FPGAs.

Achieving optimal performance on heterogeneous platforms requires extensive optimizations to improve resource efficiency and maximize frequency. These optimizations require deep knowledge of the hardware architecture, posing a significant barrier for developers lacking this expertise. To address these challenges, we introduce HeteroRefactor, a novel approach combining dynamic invariant analysis with automated refactoring and selective offloading techniques. HeteroRefactor operates in three stages:

Dynamic Invariant Analysis. HeteroRefactor monitors the dynamic invariants of the target program, including the bitwidth of integer and floating-point variables and the size of recursive data structures and stacks. This captures the actual runtime behavior, offering insights beyond static analysis. These invariants guide the refactoring process and enable the synthesis of traditionally incompatible programs.

Automated Refactoring. Using the knowledge of dynamic invariants, HeteroRefactor refactors the kernel to optimize resource usage and frequency. It transforms the program to make it synthesizable for customized resources while also optimizing resource usage and achievable operating frequency. This step alleviates the need for manual, error-prone refactoring by developers.

Selective Offloading. To ensure correctness and maintain software compatibility, HeteroRefactor implements a selective offloading mechanism. Computation is only transferred from the CPU to customized resources when the input falls within the observed dynamic invariants. This approach guarantees functional correctness while maximizing the use of heterogeneous resources.

HeteroRefactor is implemented using the ROSE compiler framework [QL11] for source-to-source transformations and is extended upon Daikon [EPG07] for dynamic analysis.

This chapter presents the HeteroRefactor approach. We evaluate HeteroRefactor on benchmarks, demonstrating its effectiveness in optimizing resource usage and frequency while maintaining functional correctness. Our results show that HeteroRefactor makes heterogeneous computing resources more accessible to software developers.

HeteroRefactore is the result of collaborative efforts. The integer refactoring techniques described in Section 3.2.2 were primarily developed by Aishwarya Sivaraman. Parts of the floating-point refactoring methods and parts of the experimental evaluation, covered in Sections 3.2.3 and 3.3 respectively, involved significant contributions from Qian Zhang. Professor Miryung Kim provided valuable insights and guidance throughout the research. This chapter is based on research originally published at the 2020 IEEE/ACM 42nd International Conference on Software Engineering [LSZ20]. The software distribution of HeteroRefactor could be found at `https://github.com/heterorefactor/heterorefactor`.

By addressing the challenges of code compatibility and optimization for heterogeneous platforms, HeteroRefactor represents a step towards enabling software developers to leverage the power of heterogeneous computing without requiring extensive hardware expertise. The following sections will delve into the details of our approach, its implementation, and its effectiveness in bridging the gap between software development practices and heterogeneous computing requirements.

3.1 Overview

3.1.1 Observations

The transition from traditional CPU-based computing to heterogeneous systems presents significant challenges for software developers. One of the key observations driving our research is that software kernels are often designed with a level of generality that, while having little impact on CPU-based execution, can lead to inefficiencies when implemented on customized accelerators. This over-provisioning, intended to handle a wide range of inputs, can significantly impact the efficiency of heterogeneous implementations, where resource utilization and performance are highly dependent on the actual ranges of values and data structure sizes encountered during execution.

For instance, a software developer might use a 32-bit integer to represent a human age, despite the fact that the practical range rarely exceeds 120. Similarly, an array might be conservatively allocated with a size of 16k elements, even though 99% of executions only require a size of 2k or less. While such design choices have minimal impact on CPU performance, they can lead to substantial inefficiencies in customized accelerators, affecting resource usage and maximum operating frequency.

3.1.2 Approaches

HeteroRefactor automatically transforms software kernels in high-level languages like C/C++ into forms that are compatible with HLS tools. This automation reduces the manual effort required to port existing software to heterogeneous platforms, addressing the "80-20 rule" often observed in FPGA design, where a disproportionate amount of effort is spent on code rewriting that constitutes a small portion of the runtime, and a large amount of storage is allocated for data range outliers that are rarely encountered.

By executing the kernel code on existing tests or a subset of input data, HeteroRefactor identifies target-specific dynamic invariants, capturing actual runtime behavior of the program, providing insights that static analysis alone cannot offer. Leveraging the knowledge obtained from dynamic invariant analysis, HeteroRefactor optimizes resource usage by reducing bitwidths for integers and floating-point variables, and by efficiently handling recursive data structures. These optimizations lead to significant improvements in resource utilization and maximum operating frequency on FPGAs.

HeteroRefactor uses selective offloading, ensuring behavior preservation by only offloading computation to customized accelerators when the input falls within the observed dynamic invariants. This approach ensures correctness while aggressively optimizing for common-case scenarios. It does not require a comprehensive dataset for identifying dynamic invariants. Instead, it focuses on optimizing accelerator efficiency for common-case inputs without sacrificing correctness or generality.

3.1.3 Contributions

We adapt and extend automated refactoring techniques, traditionally used for improving software maintainability, to lower the barriers for creating efficient customized circuits. This approach enables software developers to leverage heterogeneous computing resources without requiring extensive hardware design expertise.

By combining dynamic invariant analysis, automated kernel refactoring, and selective offloading, we create a unique approach for transparent customized accelerator compilation and optimization. This integration provides a powerful toolset for software developers to target heterogeneous platforms efficiently.

HeteroRefactor provides a vendor-agnostic solution for automating common code adaptation techniques. The synthetizability and resource reductions enabled by this framework allow developers to focus on performance bottlenecks and further parallelizing or pipelining their designs, rather than addressing low-level hardware specifics.

Results show that HeteroRefactor reduces code complexity in recursive programs, eliminating 185 lines of code for HLS compatibility, with up to 83% BRAM reduction and 42% frequency increase. For integer-intensive programs, bitwidth optimization saves 25% flip-flops, 21% LUTs, 41% BRAM, and 52% DSP. In floating-point computations, it reduces flip-flops by 61%, LUTs by 39%, and DSP by 50%.

3.2 Approach

HeteroRefactor is an end-to-end solution that combines dynamic invariant analysis, automated refactoring, and selective offloading. This approach addresses three key challenges in adapting software for heterogeneous platforms: rewriting recursive data structures, optimizing integer bitwidth, and tuning floating-point precision. Our methodology is founded on the insight that a priori dynamic analysis can significantly improve both the synthesizability of software for hardware platforms and the resource efficiency of the resulting implementations.

HeteroRefactor's workflow, as illustrated in Figure 3.1, consists of three main components: (A) Instrumentation for target-specific dynamic invariant analysis; (B) Source-to-source transformation guided by the identified dynamic invariants; (C) Selective offloading with runtime guard condition checks.

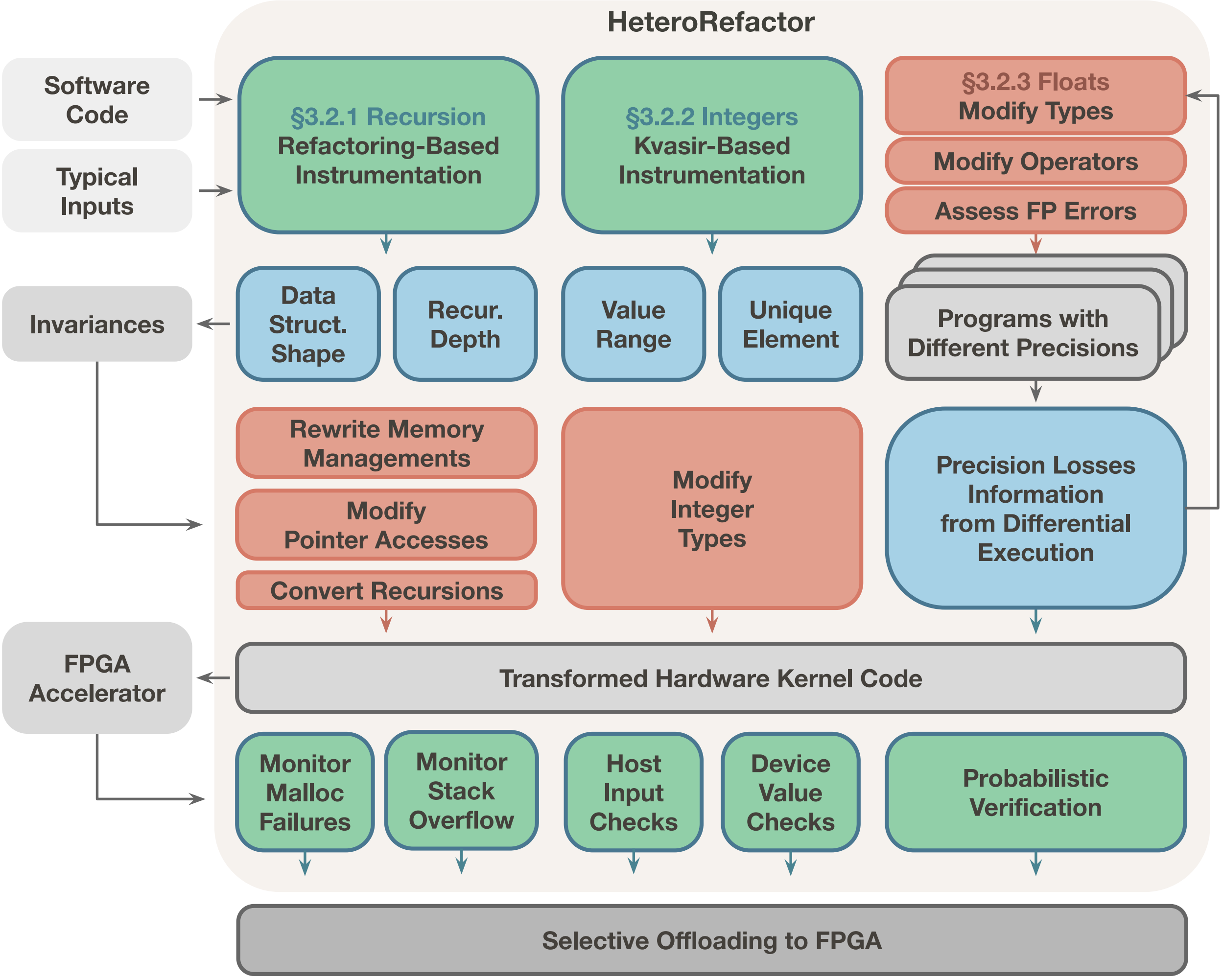


Figure 3.1: HeteroRefactor's overall framework.

3.2.1 Recursive Data Structure Transformation

Many software applications rely heavily on recursive data structures, dynamic memory allocation (`malloc` and `free`), and recursive function calls. However, as discussed in Section 2.3, these constructs are often unsupported or strictly limited in HLS tools for FPGA programming. For instance, Vitis HLS throws an error for unsynthesizable types when encountering pointer-based structures like linked lists (e.g., *an unsynthesizable type '[10 x struct.Node.0.1.2]*'*). In this situation, expert developers often rewrite recursive data structures to flattened arrays to comply with compiler constraints. This process involves declaring an array size that is larger than necessary for all input data, leading to over-provisioning, manually converting recursion into loop iterations, and over-provisioning the stack required to track the program state involved in recursive calls. This limitation significantly constrains the types of programs that can be automatically ported for heterogeneous computing. To address this challenge, HeteroRefactor employs a two-step process: dynamic analysis through refactoring-based instrumentation, followed by automated code transformation.

3.2.1.1 Refactoring-based Instrumentation

HeteroRefactor instruments the original code to collect crucial information about recursive data structures and function call depths. There are two main types of information that HeteroRefactor collects: (1) the number of elements allocated for each data structure and (2) the maximum recursion depth of each recursive function. To collect this information, we perform Memory Allocation Tracing and Recursion Depth Monitoring:

Memory Allocation Tracing. HeteroRefactor instruments memory allocation and deallocation function calls, such as `malloc` and `free` for linked list nodes. It monitors the count of elements allocated for each data structure. By analyzing the trace, HeteroRefactor obtains the typical peak element count within the data structure.

This data helps decide the appropriate size for the static array intended to replace the dynamic data structure.

Recursion Depth Monitoring. HeteroRefactor sets tracing points at both the entry and exit of recursive functions to monitor the recursion depth. From the trace, it manages a separate variable for each function, which is incremented every time the program enters a function call and decremented when it reaches a return statement. During execution, the typical highest value attained by this variable is reported and used as the bound for the corresponding stack.

For example, in a linked list implementation:

```c
Node* create_list(int n) {
    if (n == 0) return NULL;
    Node* node = (Node*)malloc(sizeof(Node));
    node->data = n;
    node->next = create_list(n - 1);
    return node;
}
```

HeteroRefactor would insert tracing points to record the number of allocated nodes and the maximum recursion depth:

```c
Node* create_list(int n) {
    trace_call("create_list");        // Record function entry
    if (n == 0) {
        trace_return("create_list");  // Record function exit
        return NULL;
    }
    Node* node = (Node*)malloc(sizeof(Node));
    trace_alloc("Node", 1);           // Record allocation
    node->data = n;
    node->next = create_list(n - 1);
    trace_return("create_list");      // Record function exit
    return node;
}
```

This instrumentation allows HeteroRefactor to determine the maximum size required for flattened arrays generated for each data structure and the corresponding stack depth for recursive functions required during typical program execution.

3.2.1.2 Refactoring

Using the data collected through dynamic analysis, HeteroRefactor applies a series of transformations with the ROSE compiler framework to make the code compatible with HLS tools while preserving its semantics:

Rule 1: Rewrite Memory Management. HeteroRefactor pre-allocates arrays for each data type, sized according to the dynamic analysis results. It implements a buddy memory system for efficient memory allocation and deallocation within these arrays, which requires less overhead and has little external fragmentation [WJN95]. With this transformation, all calls to `malloc` and `free` are replaced with calls to the memory system, which returns an available index from the pre-allocated array.

Rule 2: Convert Pointer Access to Array Access. HeteroRefactor transforms pointers to unsigned integers representing offsets in the pre-allocated arrays. It modifies pointer dereferences and structure accesses to use array indexing. The changes are propagated through the data flow graph to ensure that all pointer accesses are replaced with array accesses.

Rule 3: Convert Recursion to Iteration. HeteroRefactor creates a stack for each recursive function to store local variables and execution state. It transforms recursive functions into iterative ones by pushing the current context to the stack and updating the program counter to simulate function calls. Local variable accesses are replaced to reference the top of the stack.

For example, the function for creating a linked list can be adapted to non-recursive as illustrated below. We omit the implementation of the buddy memory system for brevity. Instead, we show an allocator from a free index without the capability of deallocating. We will explain the modifications in detail after the full code snippet for overview.

```
#define MAX_NODES        1000
#define MAX_STACK_DEPTH  100

struct Node {
  int data;
  unsigned int next;  // Index in the array instead of pointer
};

Node node_array[MAX_NODES];
unsigned int free_index = 0;

unsigned int create_list(int n) {
  struct {
    int n;
    unsigned int node;
    unsigned int result;
    int state;
  } stack[MAX_STACK_DEPTH];
  int sp = 0;  // Stack pointer

  stack[sp].n = n;
  stack[sp].state = 0;

  while (sp >= 0) {
    switch (stack[sp].state) {
    case 0:
      if (stack[sp].n == 0) {
        stack[--sp].result = -1;  // Equivalent to NULL
      } else {
        // Allocate node
        stack[sp].node = (unsigned int)free_index++;
        node_array[node].data = stack[sp].n;

        // Recursive call becomes stack push
        sp++;
        stack[sp].n = stack[sp-1].n - 1;
        stack[sp].state = 0;
```

```
Node* node =
  (Node*)malloc(sizeof(Node));
// allocated *node

Node* create_list(int n);
```

```
Node node_array[MAX_NODES];
unsigned int free_index = 0;
unsigned int node =
  (unsigned int)free_index++;
// allocated node_array[node]

unsigned int create_list(int n);
```

Figure 3.2: Original source code implementing the linked list memory management (left) and HeteroRefactor-transformed version (right).

```
        stack[sp-1].state = 1;
      }
      break;
    case 1:
      // Set next pointer and return
      node_array[stack[sp].node].next = stack[sp].result;
      stack[sp-1].result = stack[sp].node;
      sp--;
      break;
    }
  }
  return stack[0].result;
}
```

Rule 1: Rewrite Memory Management. HeteroRefactor modifies traditional dynamic memory allocation by utilizing a pre-allocated array, `node_array`, for storing all nodes. Instead of returning a pointer, the function `create_list` now provides an array index. The classic `malloc` function is substituted by incrementing the `free_index`, which allocates a new node. While `free` is typically employed for deallocating memory, it is excluded in this explanation for the sake of simplicity. In a complete implementation, HeteroRefactor employs a buddy system for efficient memory management, where the `node = free_index++` is replaced by an invocation to the system, though this detail is omitted here to maintain focus. The code is shown in Figure 3.2.

This integer `node` acts as an index referencing the target element in the pre-allocated

```
node->data = n;
node->next =
  create_list(n - 1);
```

```
node_array[node].data = n;
node_array[node].next =
  create_list(n - 1);
```

Figure 3.3: Original source code implementing the linked list pointer access (left) and HeteroRefactor-transformed version (right).

array `node_array`. Type transformations occur in three locations: (1) Rewriting variable declarations `Node* node` as `unsigned int node`; (2) Typecasting `(Node*)` to `(unsigned int)`; and (3) Modifying function parameters and the return value in both declarations and the definition, for example, `Node* create_list(int n)` to `unsigned int`. We perform a breadth-first search on the data flow graph to propagate the type changes.

Rule 2: Convert Pointer Access to Array Access. HeteroRefactor replaces pointer dereferences with array indexing. For instance, the `node->data` and `node->next` accesses are transformed into `node_array[node].data` and `node_array[node].next`, respectively. The code after this transformation rule is shown in Figure 3.3.

We convert all pointer dereferences to array accesses using the corresponding indices. The indirection operator (`*ptr`) and structure dereference operators (`ptr->, ptr->*`) are refactored into array accesses (`type_array_of_ptr[ptr]`), where the integer serves as the array index. Likewise, the subscript operators (`ptr[]`) are converted to array accesses, with the base integer added to the specified offset to determine the array index.

Rule 3: Convert Recursion to Iteration. HeteroRefactor modifies the recursive function `create_list` into its iterative counterpart by implementing a stack to maintain the local variables and the state of the computation. In the transformed `create_list` function, the stack is utilized to hold the following elements: the current value of `n`; the index of the current node, denoted as `node`; the return value, stored as `result`; the state of function execution, represented as `state`. The depth of the stack `MAX_STACK_DEPTH` is determined by the maximum recursion depth identified during dynamic analysis:

```
struct {
  int n;
  unsigned int node;
  unsigned int result;
  int state;
} stack[MAX_STACK_DEPTH];
int sp = 0;  // Stack pointer
```

For access to the local variables, HeteroRefactor replaces the original variable names with the corresponding stack elements:

```
node_array[stack[sp].node].next = stack[sp].result;
```

For each recursive call, the function pushes the current state to the stack and updates the state to reflect the next step `stack[sp-1].state = 1`. It also pushes new parameters `stack[sp].n` to the stack and sets the state to the initial value `stack[sp].state = 0`, so that the while loop can continue to the first line of the function:

```
sp++;
stack[sp-1].state = 1;
stack[sp].n = n;
stack[sp].state = 0;
continue;
```

The function then enters a loop that iterates until the stack is empty. At each iteration, a function `return` writes the return value `stack[sp-1].result` to the stack and decrements the stack pointer `sp-` to pop the top element so that the next iteration can continue from the previous state:

```
while (sp >= 0) {
  switch (stack[sp].state) {
  case 0:
    if (to_return) {
      // Return -1 and pop the stack
      stack[--sp].result = -1;
    } else // ...
```

```
      break;
    case 1:
      // Return node and pop the stack
      stack[sp-1].result = stack[sp].node;
      sp--;
      break;
    }
  }
```

In summary, for the linked list example, HeteroRefactor transforms the original recursive function into an iterative version that uses a stack to maintain the state of the computation. The transformed function iterates through the stack, processing each element according to its state. The code after this transformation rule is shown in Figure 3.4.

3.2.2 Integer Bitwidth Optimization

Integer bitwidth optimization is crucial for efficient FPGA implementations, as it directly impacts resource utilization and power consumption. Software developers often use standard integer types (e.g., 32-bit integers) by default, which can lead to over-provisioning in FPGA designs. HeteroRefactor addresses this challenge through dynamic analysis and automated refactoring.

3.2.2.1 Daikon-based Instrumentation

HeteroRefactor leverages Daikon [EPG07], a dynamic invariant detection tool, to identify likely program invariants during execution. Daikon consists of two main components:

Language-Specific Front-End: The front end instruments the program and extracts program state information by running the program.

Language-Independent Inference Engine: The inference engine analyzes the program

```
Node* create_list(int n) {
  // ... other skipped for brevity







  create_list(n - 1);





  return node;






}
```

```
unsigned int create_list(int n) {
  struct { /* stack */ } stack[DEPTH];
  int sp = 0;  // Stack pointer
  stack[sp].n = n;
  stack[sp].state = 0;

  while (sp >= 0) {
    switch (stack[sp].state) {
    case 0:
      sp++;
      stack[sp].n = stack[sp-1].n - 1;
      stack[sp].state = 0;
      stack[sp-1].state = 1;
      break;
    case 1:
      stack[sp-1].result =
        stack[sp].node;
      sp--;
      break;
    }
  }
  return stack[0].result;
}
```

Figure 3.4: Original source code implementing the recursive linked list function call (left) and HeteroRefactor-transformed version (right).

state traces to identify likely invariants.

Daikon is commonly employed for program comprehension and testing, producing invariants that describe program behavior, such as array sizes or binary comparisons, exemplified by `i>0`, `i<0`, `size(array)=0`, `size(array)>0`, etc. HeteroRefactor extends its capabilities to identify FPGA-specific invariants. For instance, reducing variable bitwidth or floating-point precision can directly decrease resource usage in FPGA designs [KA13].

3.2.2.2 FPGA-Specific Invariants

To optimize for FPGA, HeteroRefactor focuses on three types of target-specific invariants: (1) the minimum and maximum value of a variable based on range analysis, (2) the number and type of unique elements in an array, and (3) the size of an array.

These invariants are particularly relevant for FPGA designs, as they can be used to perform bitwidth reduction and efficient resource allocation. For example, consider this face detection kernel where 32-bit integers are used by default:

```
int cascadeClassifier(
    int SUM1_data[IMG_HEIGHT][IMG_WIDTH],
    int SQSUM1_data[IMG_HEIGHT][IMG_WIDTH],
    MyPoint pt) {
  int stddev = int_sqrt(dev);
  // ...
}
```

While 32-bit integers are standard in CPU architectures, FPGAs allow for arbitrary bitwidth specification, potentially leading to significant resource savings.

3.2.2.3 Refactoring

Based on the identified invariants, HeteroRefactor applies the following refactoring rule:

Rule: Modify Variable Type. HeteroRefactor converts standard integers to arbitrary precision integers using Vitis HLS's `ap_uint<k>` or `ap_int<k>` types, where `k` is the minimum number of bits required based on the observed value range. For example, if HeteroRefactor determines that a variable has a minimum value of 0 and a maximum value of 83, it can be represented using only 7 bits instead of 32.

In the refactoring process, HeteroRefactor parses the program's data flow, identifies the variable declarations and computation operations, and modifies the corresponding types based on the observed value range.

For the face detection kernel, the variable `stddev` in the method `cascadeClassifier` is declared as a 32-bit integer. However, HeteroRefactor determines that only 18 bits are required based on the observed value range. It modifies the type to `ap_uint<18>`:

```
int cascadeClassifier(/* ... */) {
  ap_uint<18> stddev = int_sqrt(dev);
}
```

In addition to modifying variable types, HeteroRefactor changes arithmetic operators to work with the new types and propagates type changes throughout the dataflow graph. It further implements a guard protection system to monitor invariant violations:

```
bool guard_error = false;
void guard_check(ap_int<65> value, int size, int sign) {
#pragma HLS inline off
  if (sign == 1) {
    if (value < 0) {
      if (value < -(1LL<<(size-1))) guard_error = true;
  } else { /.../ }
  } else { /.../ }
}

int cascadeClassifier(/* ... */) {
  ap_uint<18> stddev = int_sqrt(stddev);
  guard_check(ap_int<65>(int_sqrt(stddev)), 18, 0);
}
```

3.2.3 Floating-Point Precision Optimization

Optimizing floating-point (FP) precision presents unique challenges compared to integer bitwidth reduction. Unlike integer optimization, reducing FP precision can lead to accuracy loss, which must be carefully managed to maintain program correctness. Estimating the error resulting from reducing the bitwidth of an FP representation requires differential execution for reliable outcomes. Existing static analysis methods tend to excessively estimate FP errors, making them less effective for this purpose. HeteroRefactor addresses this challenge through a novel probabilistic, differential execution-based approach.

3.2.3.1 Probabilistic Verification Approach

HeteroRefactor's FP optimization strategy consists of four key steps: (1) Code transformation to generate program variants with different bitwidths; (2) Estimation of required input samples using Hoeffding's inequality [Hoe94]; (3) Test generation and differential execution; and (4) Probabilistic verification for FP errors.

The core insight of this approach is that we can empirically assess whether the relative error between a low-precision and high-precision program variant is within an acceptable range with a given probability. This method provides a probabilistic guarantee for unseen inputs, addressing the limitations of prior work that relied on fixed test sets [RNN13, RNM16].

To ensure statistical significance, HeteroRefactor uses Hoeffding's inequality [Hoe94] to determine the minimum number of samples required:

$$P[|\overline{c_i} - E[\overline{c_i}]| \geq \epsilon] \leq 2e^{-2n\epsilon^2} \tag{3.1}$$

$$n \geq ln(2/\alpha)/(2\epsilon^2) \tag{3.2}$$

Where: (1) $\overline{c_i}$ is the empirical measurement of the error distribution. (2) $E[\overline{c_i}]$ is the actual expectation. (3) ϵ is the acceptable deviation (error). (4) $(1 - \alpha)$ is the specified confidence level. n is the required number of samples.

This approach provides a conservative bound for expectations of any arbitrary distribution, making it suitable for situations where the FP loss distribution is unknown.

3.2.3.2 Refactoring Rules

HeteroRefactor applies the following refactoring rules to optimize FP precision:

Rule 1: Duplicate Method and Modify Type. HeteroRefactor creates low-precision variants of FP functions by redefining variable types using Thomas' templatized soft floating-point library [Tho19]. For example, the `l2norm` function from the KNN kernel is transformed from `float` to `thls::fp_flopoco<5,16>`, which has 22 bits in total (5 for the exponent, 16 for the fraction, and 1 for the sign bit), is shown in Figure 3.5.

Rule 2: Modify Arithmetic Operators. HeteroRefactor adapts arithmetic operations to work with the new low-precision types. For unsupported operations (e.g., subtraction in `thls::fp_flopoco`), it implements equivalent operations using supported primitives. For example, subtraction is converted to addition and negation using the `neg` function, as shown in Figure 3.5.

Rule 3: Assess FP Error for Differential Execution. HeteroRefactor generates code to compute the relative error between high and low-precision variants and verify if the error is within the acceptable range:

```
int main() {
    for (...) {
        // For each input sample args[]
        float highValue = l2norm(args[]);
        float lowValue = low_l2norm(args[]);
        float error = highValue - lowValue;

        if (fabs(error) > acceptableError) Failed++;
        else Passed++;
```

```
float l2norm(
    float query[],
    float data[],
    int dim
) {
  float dist = 0.0;
  for (int j = 0; j < dim; j++) {
    dist += ((query[j] - data[j])
          * (query[j] - data[j]));
  }
  return sqrt(dist);
}
```

```
using namespace thls;
typedef policy_flopoco<16,5>::value_t
    LOWBIT;
float low_l2norm(
    float query[],
    float data[],
    int dim
) {
  LOWBIT dist = 0.0;
  for (int j = 0; j < dim; j++) {
    LOWBIT fp_query_j =
        to<LOWBIT, policy>(query[j]);
    LOWBIT fp_data_j =
        to<LOWBIT, policy>(data[j]);
    LOWBIT fp_neg_1 =
        neg(fp_data_j);
    dist += (fp_query_j + fp_neg_1)
          * (fp_query_j + fp_neg_1);
  }
  return sqrt(to<float>(dist));
}
```

Figure 3.5: Original FP kernel code using `float`: `l2norm` from KNN (left) and HeteroRefactor-transformed version with low-precision (right).

```
        }
        if (double(Passed) / Samples > requiredProbability) {
            // Passed verification
        } else {
            // Failed verification
        }
    }
```

This approach allows software developers to specify their accuracy requirements (e.g., error less than 10^{-4} with 95% probability and 95% confidence) without needing to understand the intricacies of FP representation in hardware, tuning the precision to meet the desired accuracy, and manually maintaining program correctness.

3.2.4 Selective Offloading with Guard Check

Optimizations in software that utilize heterogeneous computing must maintain program correctness. HeteroRefactor applies a selective offloading strategy that incorporates guard checks. This method supports aggressive optimizations by utilizing dynamically observed invariants and ensures correctness across all potential inputs, even those not observed during the analysis phase.

The guard check system operates by inserting conditional checks in both the host program (which manages data transfer to the accelerator) and the kernel (the algorithm mapped to the accelerator). These checks verify that the current execution adheres to the assumptions made during optimization. If a violation is detected, the execution falls back to the CPU, ensuring correct results at the cost of performance loss for outlier cases.

HeteroRefactor implements guard checks differently depending on the optimization:

Recursive Data Structure Transformation. Guard conditions are inserted at memory allocation points. A global flag (`guard_error`) is set if the pre-allocated array is full or if the stack size exceeds the depth.

Integer Bitwidth Optimization. Guard conditions are added for each input, output, and intermediate value in the kernel. These guards proactively prevent overflow by checking if values exceed the reduced bitwidth range.

Floating-Point Precision Tuning. The guard check is implicit in the differential execution process, where results from low-precision and original variants are compared.

This guard check system provides a safety net that enables developers to confidently deploy heterogeneous computing without fear of silent errors or incorrect results.

3.3 Evaluation

To assess the effectiveness of HeteroRefactor in enabling software developers to leverage heterogeneous computing resources, particularly FPGAs, we conducted a comprehensive evaluation. Our analysis focuses on three key research questions:

RQ1. How effectively does HeteroRefactor expand the scope of HLS synthesizability for programs with recursive data structures?

RQ2. To what extent can HeteroRefactor reduce the manual effort required to create HLS-compatible programs?

RQ3. What level of resource reduction does HeteroRefactor achieve for recursive data structures, integer optimization, and floating-point optimization?

These questions evaluate HeteroRefactor's ability to bridge the gap between software development practices and the requirements of FPGA programming, addressing key challenges identified in our problem statement.

3.3.1 Benchmarks

We experiment with HeteroRefactor on programs from Section 2.3 and synthetic benchmarks that exhibit a variety of typical software constructs and algorithmic patterns. These programs present specific challenges associated with porting to heterogeneous computing resources, which include: Support for pointers (PTR); Restrictions on dynamic memory management (DMM); Inability to use function recursion (REC); Requiring optimization of integer bitwidth (INT); Tuning floating-point precision (FPP). These challenges align with the solutions proposed in this chapter. The chosen benchmarks are categorized into three groups based on their primary optimization target:

Recursive Data Structures (R1-R5). Aho-Corasick (R1) is a string pattern searching algorithm that employs breadth-first search with a dynamic queue, a recursive Trie tree, and a finite state machine [AC75]. It is the STS benchmark from §2.3. **Depth-First Search (R2)** is a graph traversal algorithm implemented using recursion. **Linked List (R3)** performs basic operations (insertion, removal, and sorting) on a linked list data structure. **Merge Sort (R4)** creates a recursive sorting algorithm implemented on a linked list. **Strassen's Algorithm (R5)** implements a recursive algorithm for matrix multiplication [HJJ96].

Integer Optimization (I6-I8). Face Detection (I6) is an image processing algorithm from the Rosetta benchmark suite [ZGD18, SDM17]. It is the FDT benchmark from §2.3. **3D Rendering (I7)** is a graphics processing algorithm, also from the Rosetta benchmark suite, corresponding to the 3DR benchmark. **Bubble Sort (I8)** is a synthetic benchmark that sorts an array of integers.

Floating-Point Optimization (F9-F10). KNN-l2norm (F9) uses the L2 norm calculation component of the K-Nearest Neighbors algorithm, adapted from OpenCV examples [BK08]. **RGB2YUV (F10)** is a color space conversion algorithm from OpenCV examples, corresponding to the R2Y benchmark in §2.3.

These benchmarks were chosen to match the challenges identified in our problem statement, which HeteroRefactor aims to address. For the recursive data structure benchmarks (R1-R5), the original programs were not synthesizable by HLS tools, allowing us to evaluate HeteroRefactor's ability to enable FPGA acceleration for previously incompatible software constructs. For the integer and floating-point benchmarks (I6-F10), we focus on resource utilization improvements, comparing HeteroRefactor's results against both unoptimized and manually optimized versions where available. For the comparison, hand-optimized programs of the synthetic benchmarks were developed by experienced graduate students from an FPGA research group at UCLA.

3.3.2 Experimental Setup

Our experiments were conducted on a machine with an Intel(R) Core(TM) i7-8750H 2.20GHz CPU and 16 GB of RAM, running Ubuntu 16.04. We used the following software tools and hardware targets:

Dynamic Invariant Analysis. Daikon version 5.7.2 with Kvasir as the front-end.

Automated Refactoring. ROSE compiler version 0.9.11.0.

HLS and RTL Synthesis. Vivado Design Suite 2018.03.

FPGA Target. AMD/Xilinx Virtex UltraScale+ XCVU9P on a VCU1525 Reconfigurable Acceleration Platform, with a target frequency of 300 MHz.

3.3.3 Results for Recursive Data Structures

To address **RQ1** and **RQ2**, we evaluated HeteroRefactor's ability to transform recursive data structure programs into HLS-synthesizable versions and quantified the reduction in manual effort. We further analyzed the impact of HeteroRefactor on resource utilization and operating frequency for these programs.

Table 3.1: Recursive data structure kernels: HeteroRefactor vs. manual refactoring effort.

ID/Program	Original LOC	Manual LOC	Δ LOC	Original Chars	Manual Chars	Δ Chars
R1/A.-C.	190	291	33%	5673	8776	35%
R2/DFS	86	198	57%	2236	5699	61%
R3/L. List	131	235	44%	3061	6686	54%
R4/M. Sort	128	342	63%	3267	9124	64%
R5/Strassen's	342	735	53%	10026	40971	76%
Geomean			49%			56%

3.3.3.1 Synthesizability and Manual Effort Reduction

Table 3.1 compares the original programs, manually refactored versions, and HeteroRefactor's output in terms of lines of code (LOC) and character count (Chars).

Synthesizability. Results show that all five recursive data structure programs, which were initially incompatible with HLS tools, were successfully transformed by HeteroRefactor into synthesizable versions.

Manual Effort Reduction. Manual refactoring resulted in significant code expansion, with an average increase of 49% in LOC and 56% in character count. HeteroRefactor eliminated the need for this manual refactoring, potentially saving developers substantial time and effort in adapting these algorithms for FPGA implementation.

3.3.3.2 Resource Utilization and Performance

To address RQ3, we compared the resource utilization and performance of HeteroRefactor-optimized designs against manually optimized versions with conservative array sizes. We used input data sizes of `1k`, `2k`, `4k`, or `8k` for profiling and compared them against

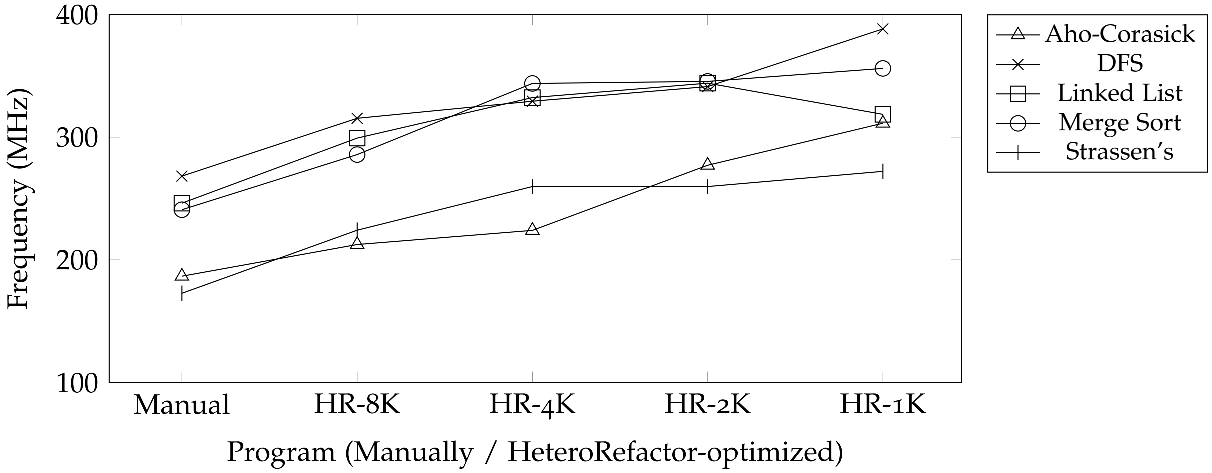


Figure 3.6: Operating frequency comparison between hand-optimized and HeteroRefactor-optimized versions on different typical input data sizes.

manual implementations with a fixed size of 16k, which corresponds to the common scenario where FPGA developers over-allocate resources to ensure correctness.

Table 3.2 summarizes the resource utilization results. Key findings include:

Memory Efficiency. HeteroRefactor achieved an average 83% reduction in BRAM usage for a typical input size of 2k, compared to the manually refactored versions.

LUT and FF Overhead. There was a small increase in LUT (302 units) and FF (494 units) usage on average, attributed to the fixed-size buddy memory system implemented by HeteroRefactor.

Frequency Improvement. As shown in Figure 3.6, HeteroRefactor-optimized designs achieved higher operating frequencies[1]. For a 2k input size, there was an average 42% increase in frequency compared to the hand-written code with a 16k size.

[1] The frequency is calculated statically by using the worst negative slack (WNS) in the report file: $F_{max} = 1/(1/300\text{MHz} + WNS)$ after placement and routing by Vivado.

Table 3.2: Resource utilization comparison of recursions transformed by HeteroRefactor.

ID/Program		LUT	FF	BRAM	DSP
R1/Aho-Corasick	Original	Not Synthesizable			
	Manual Refactoring	3287	4666	1939	7
	HeteroRefactor-8K	5492	5085	678	10
	HeteroRefactor-2K	5234	5006	206	10
R2/DFS	Original	Not Synthesizable			
	Manual Refactoring	1471	1961	221	0
	HeteroRefactor-8K	2634	2901	254	0
	HeteroRefactor-2K	2563	2881	69	0
R3/Linked List	Original	Not Synthesizable			
	Manual Refactoring	2993	3732	534	0
	HeteroRefactor-8K	3771	4044	318	0
	HeteroRefactor-2K	3655	3936	83	0
R4/Merge Sort	Original	Not Synthesizable			
	Manual Refactoring	2755	2878	519	0
	HeteroRefactor-8K	2751	2958	367	0
	HeteroRefactor-2K	2603	2951	105	0
R5/Strassen's	Original	Not Synthesizable			
	Manual Refactoring	21631	13722	919	12
	HeteroRefactor-8K	20303	14899	223	12
	HeteroRefactor-2K	19591	14654	68	12

If an FPGA programmer sets the buffer size to 32k, two algorithms, Merge Sort and Strassen's Matrix Multiplication, may not produce any bitstream. This is because their resource requirements exceed what is available, highlighting the importance of dynamic analysis in FPGA design.

These results highlight HeteroRefactor's ability to generate more efficient FPGA implementations by tailoring resource allocation to typical input sizes, while still maintaining the flexibility to handle larger inputs through CPU fallback mechanisms.

In summary, HeteroRefactor successfully addresses the challenges of implementing recursive data structures on FPGAs, enabling software developers to leverage these algorithms in heterogeneous computing environments with minimal manual intervention and improved resource efficiency. The accelerators, optimized for common-case inputs, exhibit an 83% increase in memory efficiency and a 42% improvement in frequency compared to hand-written code of a conservative size.

3.3.4 Results for Integer Optimization

To address RQ3 for integer-intensive programs, we evaluated HeteroRefactor's ability to reduce bitwidth based on dynamic invariants and its impact on resource utilization. We compared the results against both unoptimized original programs and manually optimized versions created by FPGA experts.

3.3.4.1 Dynamic Invariant Analysis

Table 3.3 presents the FPGA-specific invariants identified by HeteroRefactor for the integer optimization benchmarks.

For **Face Detection**, we utilize hexadecimal images from [SDM17], resizing them to a 16×16 format. The application employs an array of integers as pre-trained weights. HeteroRefactor determines that one of these arrays needs only 14-bit unsigned integers,

Table 3.3: FPGA's specific invariants for integer optimization.

Program	Variable	FPGA-specific Invariants			
		Min	Max	Unique	Size
I6/Face Detection	Weights Array 1	8192	12288	2	2913
I6/Face Detection	Stddev Variable	305	369	N/A	N/A
I6/Face Detection	Coord	0	6746969	21	12
I7/3D Rendering	Triangle 3D (x0)	38	255	49	100
I8/Bubble Sort	Input Array	0	10	11	400

given its maximum and minimum values and the presence of only two distinct values.

In the context of **3D Rendering**, test inputs from the dataset cited in [ZGD18] are divided into segments of 100 for each assessment run. HeteroRefactor establishes that the model's input range is `(38,150)` with a consistent size of 100.

For **Bubble Sort**, 400 integers are generated following a Chi-Square distribution [LS05]. HeteroRefactor confirms that the identified invariants align with the statistical parameters of the distribution and the unvarying size of the input set.

These invariants provide insights into the actual range and characteristics of integer variables in the programs, enabling HeteroRefactor to optimize bitwidths effectively.

3.3.4.2 Resource Utilization

Table 3.4 summarizes the resource utilization results for the integer optimization benchmarks. We compared three versions of each program:

Original. The unoptimized program with default integer types.

Manual Refactoring. A manually optimized version created by FPGA experts.

Table 3.4: Resource utilization comparison for integer optimization benchmarks.

ID/Program		LUT	FF	BRAM	DSP
I6/Face Detection	Original	11325	5784	49	39
	Manual Refactoring	10158	4800	49	37
	HeteroRefactor	10298	4770	47	28
I7/3D Rendering	Original	3828	2033	123	36
	Manual Refactoring	2239	1357	67	12
	HeteroRefactor	1907	878	39	9
I8/Bubble Sort	Original	313	125	2	0
	Manual Refactoring	306	125	1	0
	HeteroRefactor	302	125	1	0

HeteroRefactor. The version optimized by HeteroRefactor based on dynamic invariants.

From the experimental results, we observed a 25% reduction in Flip-Flops (FF), a 21% reduction in Look-Up Tables (LUT), a 41% reduction in Block RAM (BRAM), and a 52% reduction in Digital Signal Processors (DSP) for HeteroRefactor-optimized designs compared to the original programs. The reductions were 12% in FF, 5% in LUT, 15% in BRAM, and 16% in DSP compared to the manually optimized versions.

These results demonstrate that HeteroRefactor not only significantly improves resource utilization compared to unoptimized programs but also achieves better efficiency than carefully handcrafted optimizations by FPGA experts.

3.3.4.3 Performance Impact

We implemented the optimized designs on the target FPGA with a 300 MHz target frequency. All HeteroRefactor-optimized programs met this timing constraint, whereas the original version of 3D Rendering (I7) failed to meet timing and could only achieve 241 MHz. This demonstrates that HeteroRefactor's optimizations can lead to improved timing performance in addition to resource savings.

In summary, HeteroRefactor automates the bit width optimization for integers, significantly reducing manual refactoring effort. It achieves resource savings of 25% FF, 21% LUTs, 41% BRAM, and 52% DSP, outperforming expert hand-optimized kernels.

3.3.5 Results for Floating-Point Optimization

For floating-point programs, we evaluated HeteroRefactor's ability to reduce bitwidth while maintaining a probabilistic guarantee of accuracy. This addresses both RQ2 and RQ3 by demonstrating automated optimization with trade-offs between precision and resource utilization.

3.3.5.1 Probabilistic Verification

As detailed in Section 3.2.3, we utilize Hoeffding's inequality to determine the required number of input data samples, given a confidence interval $(1-\alpha)$. In our experimentation, ϵ is set at 0.03. We adjust the probability value, p, across three settings: 0.95, 0.99, and 0.999, aligning α such that $(1-\alpha) = p$. The corresponding minimum sample sizes required are 2049, 2943, and 4222, respectively.

Table 3.5 presents the results of HeteroRefactor's probabilistic floating-point verification for different configurations of acceptable loss (e) and probability (p). The 8 and 16 columns show the verification results for 8-bit and 16-bit floating-point types, respectively, where **Fail** indicates a verification failure. The **HR** column indicates the smallest bitwidth that passed verification identified by HeteroRefactor.

HeteroRefactor successfully reduced the bitwidth from the original 32 bits to as low as 21 bits while maintaining probabilistic guarantees of accuracy. Higher precision

Table 3.5: Probabilistic floating-point verification results.

Program	p	$e = 10^{-2}$			$e = 10^{-4}$			$e = 10^{-6}$		
		8-bits	16-bits	HR	8-bits	16-bits	HR	8-bits	16-bits	HR
F9/KNN	0.95	Fail	Fail	24	Fail	Fail	29	Fail	Fail	32
	0.99	Fail	Fail	25	Fail	Fail	29	Fail	Fail	32
	0.999	Fail	Fail	25	Fail	Fail	30	Fail	Fail	32
	p	$e = 10^{-4}$			$e = 10^{-5}$			$e = 10^{-6}$		
F10/R2Y	0.70	Fail	Fail	20	Fail	Fail	24	Fail	Fail	27
	0.80	Fail	Fail	21	Fail	Fail	24	Fail	Fail	28
	0.95	Fail	Fail	21	Fail	Fail	25	Fail	Fail	30
	0.99	Fail	Fail	22	Fail	Fail	25	Fail	Fail	32
	0.999	Fail	Fail	22	Fail	Fail	26	Fail	Fail	32

requirements (lower e) and higher confidence levels (higher p) generally resulted in larger bitwidths, demonstrating the tool's ability to balance accuracy and resource efficiency. We also observed that the achievable bitwidth reduction varied between programs, with RGB2YUV (F10) allowing for more aggressive optimization than KNN-l2norm (F9).

3.3.5.2 Resource Utilization

Table 3.6 summarizes the resource utilization results for the floating-point optimization benchmarks. The Original row indicates the original program with 32-bit float type and p represents the probability and the confidence level $(1 - \alpha)$. We compared the resource utilization of the original programs with 32-bit float types against HeteroRefactor-optimized versions with reduced bitwidths. The results show that HeteroRefactor can achieve up to 61% reduction in FF, 39% in LUT, and 50% in DSP. The resource savings varied with the chosen probability and acceptable loss, allowing developers to balance precision and resource utilization based on application requirements.

In summary, these results demonstrate HeteroRefactor's ability to automatically optimize floating-point computations for FPGA, providing software developers with a powerful tool to leverage the efficiency benefits of reduced precision while maintaining control over accuracy. HeteroRefactor reduces the floating-point bitwidth while providing a probabilistic guarantee for a user-specified quality loss, probability, and confidence level. It achieves up to 61% reduction in FF, 39% in LUT, and 50% in DSP.

3.3.6 Overhead and Performance Analysis

To provide a comprehensive evaluation of HeteroRefactor's practicality for software developers, we analyzed its runtime overhead and the performance impact of the optimized designs when leveraging this tool in the context of FPGA development.

3.3.6.1 Runtime Overhead

Tables 3.7 and 3.8 present the runtime overhead of HeteroRefactor for various benchmark categories, including recursive data structures, integers, and floating-point programs. The tables compare the time taken for instrumentation and refactoring against the synthesis time of the original programs.

Recursive Data Structures and Integers (R1-I8). The instrumentation overhead is generally less than 1% of synthesis time, with the exception of 3D Rendering (I7). The refactoring overhead is consistently less than 1% of synthesis time.

Floating-Point Programs (F9-F10). The differential execution overhead for floating-point programs is less than 2% compared to the synthesis time across all configurations. There is no instrumentation required for floating-point programs.

These results show that HeteroRefactor introduces minimal overhead to the develop-

Table 3.6: Resource utilization comparison for floating-point optimization benchmarks.

ID/Program		LUT	FF	BRAM	DSP
F9/KNN-l2norm	Original	88843	18591	30	32
	p	$e = 10^{-2}$			
	HeteroRefactor-0.95	80163	15257	30	16
	HeteroRefactor-0.99	82228	15626	30	16
	HeteroRefactor-0.999	82228	15626	30	16
	p	$e = 10^{-4}$			
	HeteroRefactor-0.95	88952	17102	30	32
	HeteroRefactor-0.99	88952	17102	30	32
	HeteroRefactor-0.999	88952	17855	30	32
	p	$e = 10^{-6}$			
	HeteroRefactor-0.95	88843	18591	30	32
	HeteroRefactor-0.99	88843	18591	30	32
	HeteroRefactor-0.999	88843	18591	30	32
F10/RGB2YUV	Original	398444	73437	30	288
	p	$e = 10^{-4}$			
	HeteroRefactor-0.95	243516	28379	30	144
	HeteroRefactor-0.99	250044	28827	30	144
	HeteroRefactor-0.999	250044	28827	30	144
	p	$e = 10^{-5}$			
	HeteroRefactor-0.95	304956	49468	30	144
	HeteroRefactor-0.99	304956	49468	30	144
	HeteroRefactor-0.999	311532	49964	30	144
	p	$e = 10^{-6}$			
	HeteroRefactor-0.95	372236	66381	30	288
	HeteroRefactor-0.99	398444	73437	30	288
	HeteroRefactor-0.999	398444	73437	30	288

Table 3.7: Runtime overhead for recursions and integers.

	Instrumentation		Refactoring	
Program	time (min)	ratio	time (sec)	ratio
R1/Aho-Corasick	0.10	0.26%	5.1	0.26%
R2/DFS	0.06	0.26%	4.7	0.34%
R3/Linked List	0.12	0.49%	4.5	0.31%
R4/Merge Sort	0.05	0.20%	4.5	0.29%
R5/Strassen's	0.09	0.20%	10	0.38%
I6/Face Detection	0.15	0.62%	10	0.69%
I7/3D Rendering	13.66	64.76%	10	0.79%
I8/Bubble Sort	10^{-3}	~ 0	10^{-3}	~ 0

Table 3.8: Differential execution overhead for FP (sec / %).

Program	p	$e = 10^{-2}$	10^{-4}	10^{-6}
F9/KNN	0.95	60.8 / 0.3%	29.4 / 0.2%	11.7 / 0.1%
	0.99	58.2 / 0.3%	30.4 / 0.2%	11.7 / 0.1%
	0.999	60.8 / 0.3%	25.9 / 0.1%	12.8 / 0.1%
	p	$e = 10^{-4}$	10^{-5}	10^{-6}
F10/R2Y	0.95	83.5 / 1.8%	59.3 / 1.3%	26.8 / 0.6%
	0.99	81.5 / 1.7%	58.5 / 1.2%	12.9 / 0.3%
	0.999	82.3 / 1.7%	54.5 / 1.2%	13.7 / 0.3%

ment process, making it a practical tool for software developers.

3.3.6.2 Performance Impact

We compared the execution performance of HeteroRefactor-optimized kernels against the original programs running on a CPU. For **floating-point programs**, our experiment shows a significant speedup of up to 7× and 19× in KNN-l2norm and RGB2YUV, respectively. These improvements are attributed to the inherent parallel computation capabilities of FPGAs. For **recursive programs**, the refactored kernels are slower than the CPU due to the use of sequential memory allocation in the refactored designs, the memory-bound nature of these algorithms, and the lower clock frequency of the FPGA compared to the CPU. They are designed for porting non-bottleneck parts of the program to the FPGA to achieve higher energy efficiency and less memory transfer instead of higher processing throughput. For **integer-intensive programs**, the end-to-end performance depending on the ease of exploiting data parallelism. The selected kernels (I6 and I7) from Rosetta were designed for energy efficiency rather than processing throughput, resulting in slightly slower execution than on CPU.

It's important to note that while some optimized kernels do not outperform CPU in terms of raw speed, they often offer other benefits such as energy efficiency or the ability to reduce memory movements between FPGAs and other heterogeneous components.

Additionally, HeteroRefactor's primary goal is to reduce resource usage and enable FPGA implementation of previously unsynthesizable code, rather than maximizing performance, which prior work [CWY18a, CHP16a] focused on. HeteroRefactor could be used jointly with these tools to produce fast and resource-efficient FPGA accelerators, increasing the operating frequency of generated hardware and the potential count of processing elements, which are important factors in performance.

3.4 Conclusion

This chapter has presented HeteroRefactor, a novel approach to enabling software developers to effectively leverage heterogeneous computing resources, particularly FPGAs, without requiring extensive hardware expertise. By adapting and expanding the scope of automated refactoring techniques traditionally used for software maintainability, HeteroRefactor addresses key challenges in porting software to heterogeneous platforms.

HeteroRefactor's end-to-end solution comprises three main components:

Dynamic Analysis. This component identifies common-case sizes and characteristics of data structures and variables, providing crucial insights for optimization without requiring developers to manually specify these details.

Automated Kernel Refactoring. Based on the dynamic analysis results, HeteroRefactor automatically transforms software to enhance synthesizability for FPGAs and reduce on-chip resource usage, bridging the gap between software development practices and hardware implementation requirements.

Selective Offloading with Guard Checking. This mechanism ensures correctness by dynamically deciding whether to execute on the FPGA or fall back to CPU execution based on runtime input conditions, allowing for aggressive optimizations while maintaining program correctness.

We evaluate the effectiveness of HeteroRefactor across a range of program types:

Recursive Data Structures. Recursions are traditionally challenging to implement on FPGAs. HeteroRefactor automatically refactored them to be synthesizable, achieving an 83% reduction in BRAM usage and a 42% increase in operating frequency compared to conservative manual implementations.

Integer-Intensive Programs. HeteroRefactor reduced the bitwidth of integers by 76% on average, leading to a 41% decrease in BRAM usage and in other resource types.

Floating-Point Computations. HeteroRefactor achieved a 50% reduction in DSP usage while maintaining user-specified precision guarantees, demonstrating its ability to balance accuracy and resource efficiency.

These results highlight HeteroRefactor's potential to significantly lower the barriers to entry for software developers in the field of heterogeneous computing. By automating complex transformations and optimizations, HeteroRefactor enables developers to:

- Port a wider range of software constructs to FPGAs.
- Achieve resource efficient hardware implementations without requiring in-depth knowledge of FPGA architecture or design principles.
- Explore trade-offs between precision, performance, and resource utilization.
- Maintain a single, high-level codebase that targets both CPU and FPGA execution.

While HeteroRefactor represents a step forward in enabling heterogeneous computing for software developers, it also opens up several avenues for future research. For example, it can be extended to support other types of accelerators, such as GPUs or domain-specific processors, to provide a more comprehensive heterogeneous computing solution. By integrating HeteroRefactor with existing HLS tools, a more seamless development experience may be created. HeteroRefactor provides insights into dynamic characteristics, allowing other performance-oriented optimizations to achieve both efficient resource utilization and high performance.

In conclusion, HeteroRefactor demonstrates the potential of applying software engineering techniques to the challenge of heterogeneous computing. By automating the process of adapting software for efficient execution on FPGAs, HeteroRefactor empowers a broader range of developers to leverage the benefits of heterogeneous computing platforms. This work contributes to the broader goal of making heterogeneous computing more accessible and practical for software developers.

CHAPTER 4

Architecture-Driven Optimization for Implicit Broadcasts

As we continue our exploration of enabling heterogeneous computing for software developers, we turn our attention to a critical aspect of FPGA acceleration: achieving high clock frequencies in High-Level Synthesis (HLS) generated designs. This chapter introduces Adroit (**A**rchitecture-**Dr**iven **O**ptimization for **I**mplicit Broadcas**t**s), a novel approach to addressing timing issues in HLS-generated FPGA designs.

FPGAs play a pivotal role in heterogeneous computing systems, often serving as a communication hub for various specialized resources. The operating frequency of these FPGA-based hubs can significantly impact overall system efficiency. While HLS tools have greatly simplified the process of implementing new applications on FPGAs, allowing software developers to work at a higher level of abstraction, they often fall short in producing designs that achieve optimal timing performance.

Our investigation into this challenge revealed a common issue across a diverse set of realistic FPGA designs: the primary cause of frequency degradation in HLS-generated designs is often related to implicit broadcast structures. These broadcasts, automatically inferred or created by HLS compilers in both datapath and control logic, are typically not explicitly present in the source code and are therefore often overlooked by developers.

Based on our observations, we have identified three major types of broadcasts in HLS designs: (1) high-fanout data signals, (2) pipeline flow control signals, and (3) synchronization signals for concurrent modules. These broadcast structures pose significant challenges for downstream physical design flows in achieving timing closure, leading to suboptimal clock frequencies that can bottleneck the entire heterogeneous system.

Adroit addresses these challenges through a set of architecture-driven techniques:

Broadcast-Aware Scheduling. This technique considers the impact of data broadcasts during the HLS scheduling process, minimizing high-fanout signals.

Redundant Synchronization Pruning. Adroit analyzes the control flow to eliminate unnecessary synchronization signals, reducing control signal broadcasts.

Skid-Buffer-Based Flow Control. Adroit introduces a more flexible pipeline control mechanism that reduces the impact of control signal broadcasts.

The architecture-driven approach sets Adroit apart by focusing on the impact of FPGA architecture on timing performance, contrasting with software optimizations tailored to fixed hardware data paths in CPU processors. Additionally, Adroit improves the inherent quality of dataflow dependencies in HLS designs, rather than depending on optimizations specific to vendor-provided FPGA architectures. This strategy allows the optimized RTL to be effectively processed by downstream tools, *independent* of the specific FPGA architecture targeted. In the subsequent chapter, RapidIR will focus on the challenges *specific* to diverse FPGAs, which make it challenging to port a design to another FPGA device, filling the gap left by Adroit's FPGA-architecture-neutral approach.

By incorporating these methodologies into the end-to-end optimization framework, Heterosys, Adroit narrows the gap between abstract software models and optimized FPGA deployments. This solution is crucial for software developers using heterogeneous computing platforms, enabling high-performance FPGA configurations without detailed hardware design knowledge or manual RTL adjustments. Adroit allows developers to fully utilize FPGAs in heterogeneous settings. By resolving complex timing issues from implicit broadcasts, Adroit lets developers focus on algorithm design while ensuring hardware implementations meet strict timing requirements.

In the following sections, we dive into the details of our broadcast classification, the limitations of current HLS tools in handling these broadcasts, and the techniques employed by Adroit to overcome these challenges. We also present case studies demonstrating the effectiveness of our approach across a range of real-world FPGA designs.

This chapter builds upon collaborative research that was originally published at the 2020 Design Automation Conference, where it received recognition as a best paper nominee. The work presented here is the result of joint efforts with Licheng Guo, whose contributions were significant in the problem categorization detailed in Section 4.2.

The software distribution of Adroit could be found at `https://github.com/Licheng-Guo/vivado-hls-broadcast-optimization`.

4.1 Overview

4.1.1 Observations

Our investigation into HLS-generated designs revealed a surprising commonality in frequency issues across a diverse set of real-world FPGA implementations:

Implicit Broadcasts as Performance Bottlenecks. The primary cause of frequency degradation in HLS-generated designs is often related to implicit broadcast structures. These broadcasts, automatically inferred or created by HLS compilers in both datapath and control logic, are typically not explicitly present in the source code and are therefore often overlooked by developers.

Classification of Broadcast Structures. From the FPGA design benchmarks, we identified two major categories of broadcasts in HLS-generated designs:

Data Broadcasts. High-fanout signals in the datapath often result from shared access in unrolled loops or a single point of access to partitioned arrays.

Control Broadcasts. High-fanout control signals that start and stop processing elements, including synchronization signals and pipeline control signals.

Difficulty in Debugging. These broadcast-related timing issues are extremely challenging for software developers and even expert FPGA developers to identify and debug, as they are not apparent in the high-level source code and often result from HLS tool optimizations rather than explicit programming decisions.

Compounding Effects. Data and control broadcasts often interact, exacerbating timing issues. Even seemingly simple designs can suffer from both types of broadcasts, requiring comprehensive solutions to achieve significant frequency improvements.

4.1.2 Approaches

To address these challenges and enable software developers to achieve high-performance FPGA designs through HLS, Adroit employs several key approaches:

Broadcast-Aware Scheduling. This technique enhances the HLS scheduler's delay model to account for the additional delay introduced by broadcast structures. By calibrating the scheduler with more accurate delay estimates, Adroit improves the quality of the generated schedule, leading to better timing performance.

Synchronization Logic Pruning. Adroit analyzes the parallelism patterns inferred by HLS tools and eliminates unnecessary synchronization logic. This reduces the complexity of the generated design and minimizes control broadcasts.

Skid-Buffer-Based Pipeline Control. To address the issues with high-fanout pipeline control signals, Adroit transforms the pipeline controller into a more hardware-friendly form. This approach uses skid buffers to manage pipeline stalls, reducing the broadcast factor of control signals.

These approaches are designed to be integrated into existing HLS flows, providing automatic optimizations that software developers can benefit from without requiring in-depth knowledge of FPGA architecture or manual RTL optimization.

4.1.3 Contributions

The key contributions of this work in the context of enabling heterogeneous computing for software developers include:

Identification and Classification of Implicit Broadcasts. We provide the first comprehensive analysis and classification of implicit broadcast structures in HLS-generated designs. This insight is crucial for understanding and addressing a major source of performance degradation in FPGA accelerators.

Optimization Techniques. Adroit introduces a set of architecture-driven optimization techniques that address the timing issues caused by implicit broadcasts. These techniques are designed to be integrated into HLS tools, allowing software developers to benefit from improved timing performance without manual intervention.

Performance Improvements. Our evaluation demonstrates an average frequency improvement of 53% across a set of nine real-world HLS benchmarks, with some cases showing gains of over 100 MHz. These improvements are achieved with minimal area overhead, demonstrating the efficiency of our approach.

Bridging the Gap for Software Developers. By addressing the limitations of current HLS tools in handling broadcast structures, Adroit enhances the ability of these tools to generate high-frequency FPGA designs from high-level descriptions, making a significant step towards more accessible FPGA acceleration for software developers.

By addressing these critical aspects of HLS-generated FPGA designs, Adroit contributes to the broader goal of enabling software developers to effectively leverage heterogeneous computing resources. It allows developers to focus on algorithmic design in high-level languages while still achieving implementations that meet timing requirements, which is a crucial factor in the overall performance of heterogeneous systems.

4.2 Problem Categorization

To enable software developers to effectively leverage heterogeneous computing resources, particularly FPGAs, it is important to understand the challenges that arise when high-level code is translated into hardware implementations. Our analysis of HLS-generated designs reveals that a significant source of performance degradation stems from implicit broadcast structures. These broadcasts, which are not present in the source code, can lead to timing issues that are difficult for software developers to identify and address. In this section, we categorize these implicit broadcasts into three main types: data signal broadcasts, synchronization control broadcasts, and pipeline control broadcasts.

4.2.1 Data Signal Broadcast

Data signal broadcasts occur in the datapath of HLS-synthesized designs and are a result of common software programming patterns that, when translated to hardware, lead to high-fanout signals. We identify two primary scenarios where these broadcasts occur:

4.2.1.1 Loop Unrolling

Consider the following code snippet, which is common in software development:

```
data_t source = ...;       // loop-invariant variable
for (size_t i = 0; i < 1024; i++) {
    #pragma HLS unroll
    foo = foo_func(i);
```

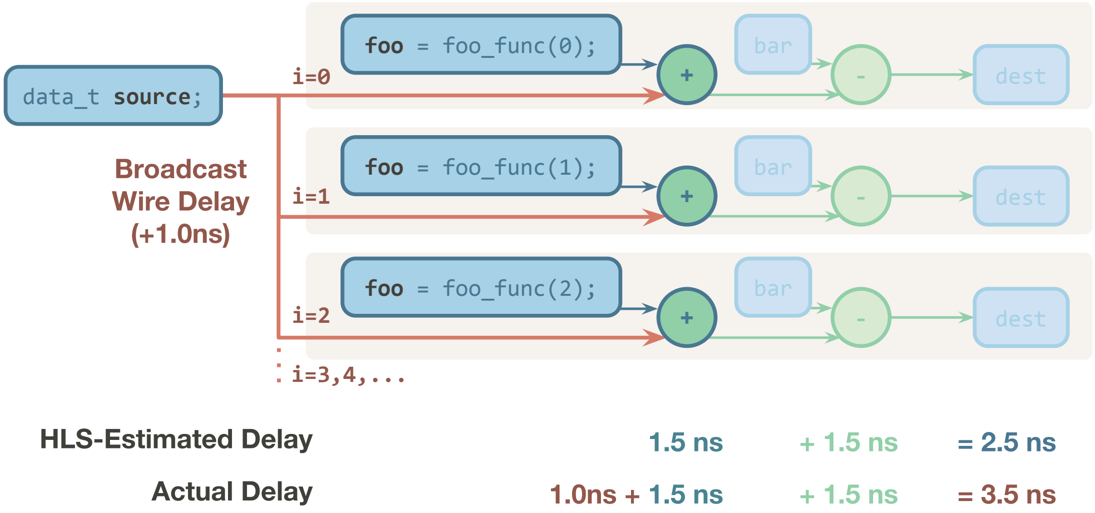


Figure 4.1: HLS-generated architecture showing data broadcast in unrolled loop.

```
        bar = bar_func(i);    // loop-dependent
        dest[i] = source + foo - bar;
    }
```

The #pragma HLS unroll directive is analogous to the #pragma omp parallel for used in OpenMP for parallelizing loops in software development. A software developer might anticipate that this pragma would similarly lead to efficient parallelization of the code because, in CPU architectures, elements such as the source variable can be cached to enhance performance. Nevertheless, the practical impact on FPGAs is contrary to that seen in CPUs. Unlike CPUs, the data reuse can, in fact, degrade FPGA performance.

When this code is synthesized for FPGA implementation with loop unrolling, it results in a hardware structure where the source variable is broadcast to 1024 instances of the loop body, as illustrated in Figure 4.1.

The challenge arises because current HLS tools do not accurately account for the additional delay introduced by this broadcast. For example, if a simple addition operation typically has a delay of 1.5ns, the actual delay for the 1024-way broadcast addition might

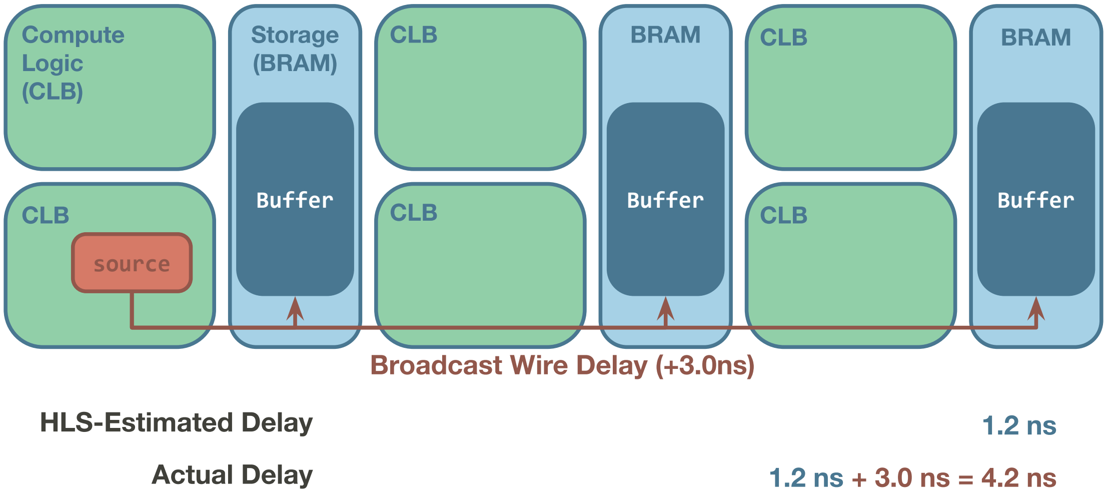


Figure 4.2: HLS-generated architecture showing data broadcast to distributed memory.

be 2.5ns. However, the HLS scheduler, unaware of this increased delay, might incorrectly schedule operations, leading to timing violations in the final implementation.

4.2.1.2 Large Buffer and Memory Arrays

Another common scenario involves large on-chip buffers, as shown in this example:

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

Similarly, a software developer will expect that the source variable is written to a single memory location in the buffer and therefore assumes that, no matter the buffer size, it should not cause any performance issues. However, the reality is different.

In FPGA implementations, large buffers are typically distributed across multiple Block RAM (BRAM) units. This distribution leads to a scenario where the source signal fans out to many physically scattered memory units, as illustrated in Figure 4.2.

Current HLS tools fail to account for the increased delay in load and store operations as buffer sizes grow. This oversight can result in inadequate pipelining between memory units and data sources/sinks, leading to timing issues in the final implementation.

4.2.2 Synchronization Control Signal

Synchronization control broadcasts arise from the HLS tool's approach to parallelizing sequential code. While this parallelization is crucial for performance improvements, the generated synchronization logic can introduce critical paths, especially as the degree of parallelism increases. Consider this streaming design example:

```
#pragma HLS dataflow
while (1) {
    // Part #A
    inFifoA.read(&a);
    outFifoA1.write(a.foo);
    outFifoA2.write(a.bar);

    // Part #B
    inFifoB.read(&b);
    outFifoB1.write(b.foo);
    outFifoB2.write(b.bar);
}
```

In this case, HLS tools often generate excessive synchronization logic, treating independent streams as if they were tightly coupled, as shown in Figure 4.3. However, in fact, Part #A and Part #B are independent of each other.

Another scenario involves multiple independent function calls:

```
data_t kernel(...) {
```

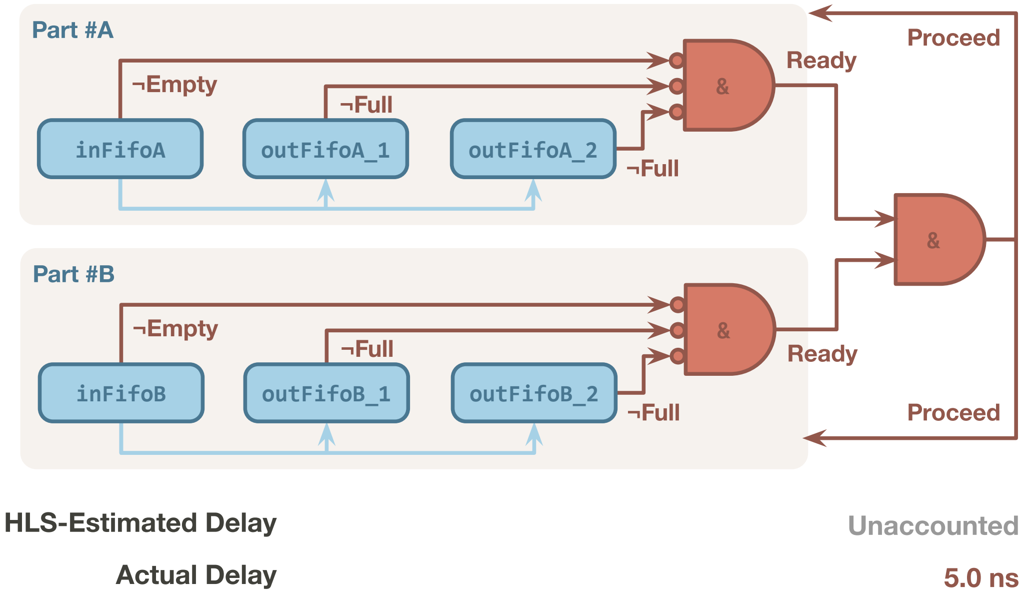


Figure 4.3: HLS-generated architectures showing synchronization broadcasts in a single logical function coupling two sets of unrelated FIFOs.

```
      aOut = PE_1(aIn);
      bOut = PE_2(bIn);
      cOut = PE_3(cIn);
      return aOut + bOut + cOut;
  }
```

Here, HLS tools typically generate synchronization logic that waits for all parallel executions to complete and asserts the `done` signal before proceeding, as illustrated in Figure 4.4. This "reduce-broadcast" pattern of synchronization can become a critical path as the design scales, significantly impacting performance.

4.2.3 Pipeline Control Signal

Pipeline control broadcasts occur in fully-pipelined datapaths, particularly when interacting with flow-controlled interfaces like FIFOs. Consider this example:

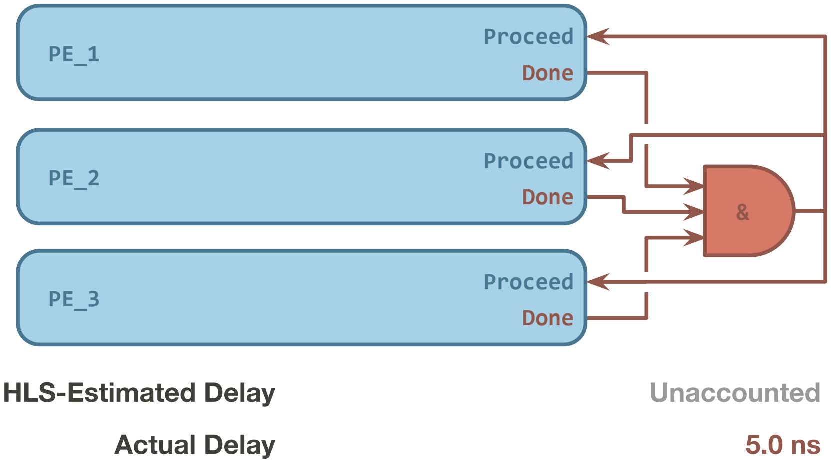


Figure 4.4: HLS-generated architectures showing synchronization broadcasts between multiple logical functions each proceeds only when all functions are completed.

```
for (int i = 0; i < ITER; i++) {
    #pragma HLS pipeline
    input_fifo.read(&a);
    b = inlined_datapath_foo(a);
    output_fifo.write(b);
}
```

In the resulting hardware, shown in Figure 4.5, back-pressure signals (e.g., if the input is ready, and if the downstream process is ready for taking output) are broadcast to control the entire pipeline to control whether they proceed or not.

While this pipeline control method is effective for small designs, it can become a timing-critical path as pipeline depth increases, leading to performance bottlenecks.

These implicit broadcasts pose significant challenges for software developers targeting heterogeneous computing platforms. They are not apparent in the high-level source code and are difficult to identify and optimize without deep hardware knowledge. In the following sections, we will present our approach to automatically addressing these

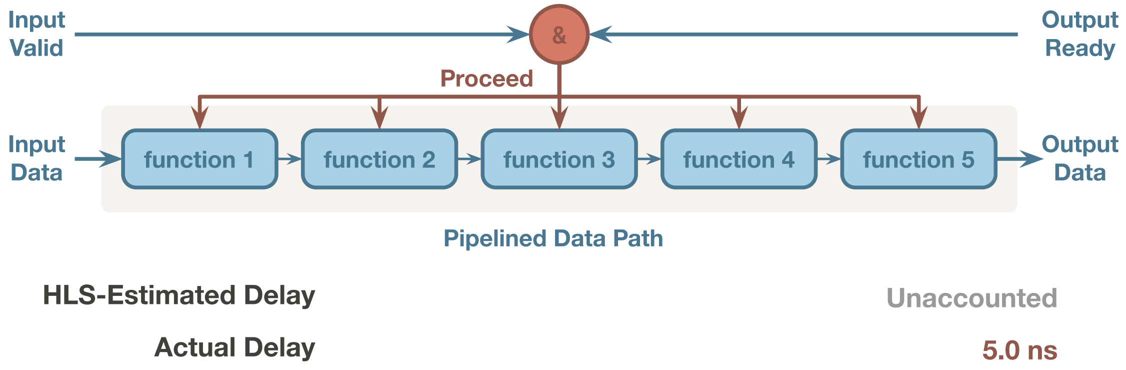


Figure 4.5: HLS-generated architecture showing pipeline control broadcast.

issues, enabling software developers to achieve high-performance FPGA implementations without requiring extensive hardware expertise.

4.3 Approach

Having identified the key challenges posed by implicit broadcasts in HLS-generated designs, we present Adroit's approach to addressing these issues. Adroit introduces a set of automated techniques that target the three main categories of implicit broadcasts: data signal broadcasts, synchronization control broadcasts, and pipeline control broadcasts.

Our approach is designed to be integrated into existing HLS flows, providing automatic optimizations that software developers can benefit from without manual intervention. By addressing these broadcast-related issues, Adroit aims to bridge the gap between high-level software descriptions and efficient FPGA implementations, making heterogeneous computing more accessible to a broader range of developers.

In this section, we detail three components of Adroit: (1) Broadcast-Aware Scheduling; (2) Synchronization Logic Pruning; And (3) Skid-Buffer-Based Pipeline Control.

Each of these components targets a specific aspect of the implicit broadcast problem, working together to improve the overall performance of HLS-generated designs.

4.3.1 Broadcast-Aware Scheduling

The first component of Adroit addresses the challenge of data signal broadcasts by introducing a broadcast-aware scheduling technique. This approach aims to provide more accurate delay estimations for operations involving broadcasts, enabling the HLS scheduler to make better decisions about cycle boundaries and resource allocation.

4.3.1.1 Delay Calibration Methodology

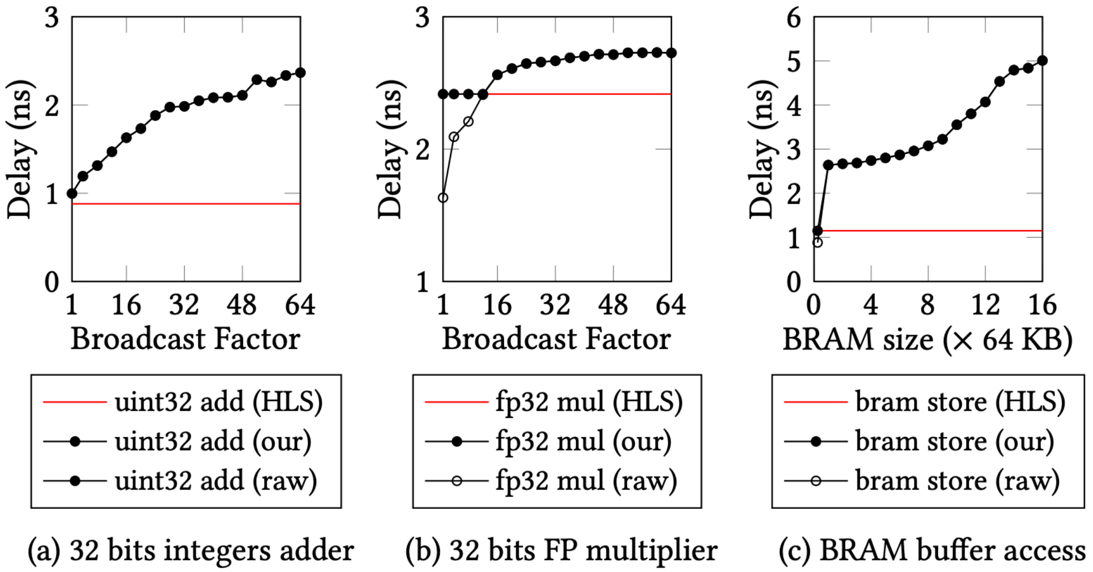
To overcome the limitations of current HLS delay models, which do not account for the additional wire delay caused by broadcasts, we propose a simple yet effective method for approximating this extra delay.

We implement skeleton broadcast structures on an empty FPGA to capture post-routing delay data. For each permutation of operator, data type, and broadcast factor, we collect reusable statistics on calibrated delays. These delays are then utilized to optimize the HLS scheduling process. Although the placement results on an empty FPGA may differ from the real situation, it is an effective lower bound on the delay penalty.

For example, to calibrate the delay for an addition operation with a broadcast factor of 64, we instantiate 64 adders on an empty FPGA, with one of the two input ports of each adder connected to a common source register. For buffer access operations (`load`, `store`), we record the actual delays of different buffer sizes using a similar methodology.

Figure 4.6 illustrates the results of our delay calibration for different operations. Key observations from our calibration results are:

- For addition and buffer access operations, our calibrated delays closely match the HLS-predicted values for small broadcast factors. However, for large broadcast factors, our measurements significantly exceed the HLS-predicted values, revealing the inaccuracy of current delay estimations in these scenarios.



(a) 32 bits integers adder (b) 32 bits FP multiplier (c) BRAM buffer access

Figure 4.6: Comparison of Vitis HLS estimated delay, our calibrated delay, and raw experimental delay for different operators.

- For floating-point operations, HLS-predicted delays are sometimes more conservative than our measurements, possibly a deliberate overestimation.

4.3.1.2 Integration with HLS Flow

To integrate our calibrated delay model into the HLS process, we first parse HLS scheduling reports to identify broadcast structures and their factors. We then apply our calibrated delays to these structures, inserting register modules in the source code for violations of the target frequency, effectively forcing the scheduler to split operations across multiple cycles. Certain operations, particularly floating-point multiplications, inherently incur delays that exceed the target. To address this, additional pipelining is inserted after the operations to facilitate downstream retiming. This retiming strategy involves relocating the registers within the operations.

This approach, when incorporated into HLS tools, allows software developers to benefit from more accurate timing estimates without requiring them to manually analyze and optimize broadcast structures. By adjusting the scheduling based on more realistic delay estimates, Adroit enables the generation of more efficient FPGA implementations from high-level software descriptions.

In the following subsections, we will discuss how Adroit addresses the challenges of synchronization control broadcasts and pipeline control broadcasts, further enhancing its ability to generate high-performance FPGA designs from software-oriented code.

4.3.2 Synchronization Logic Pruning

The second component of Adroit addresses the challenge of synchronization control broadcasts. These broadcasts, which arise from the HLS tool's approach to parallelizing sequential code, can significantly impact the maximum achievable frequency of the generated design. While downstream logic synthesis tools cannot optimize away these synchronization structures due to a lack of high-level information, Adroit, working on high-level descriptions, leverages its understanding of the original software code to identify and eliminate redundant synchronization logic.

4.3.2.1 Dataflow Synchronization Optimization

For dataflow synchronization, such as the scenario illustrated in Figure 4.8, Adroit employs the following approach:

1. Reconstruct the dataflow graph at the granularity of elementary flow control units, rather than relying on the HLS tool's inferred synchronization.
2. Identify isolated sub-graphs within user-defined streaming kernels.
3. Split independent flows into separate loops, avoiding unwanted synchronization generated by the HLS compiler.

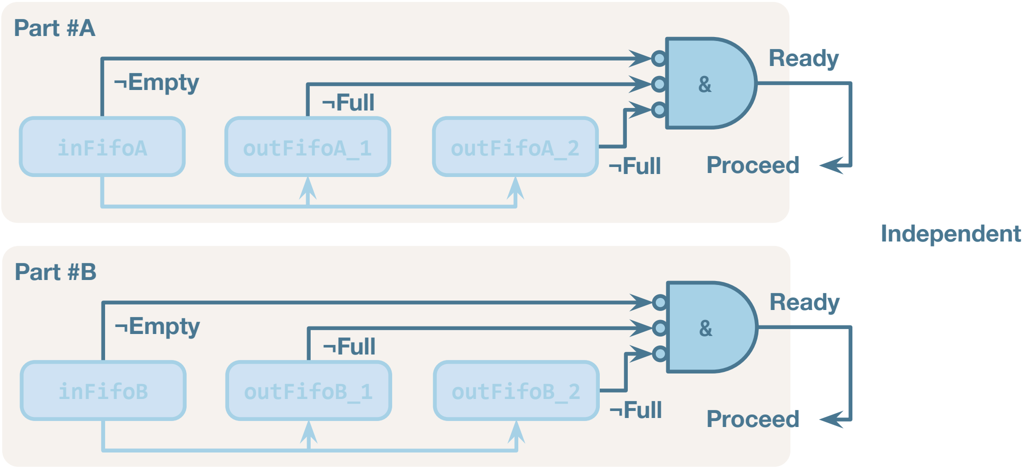


Figure 4.7: Optimized control-pruned architecture corresponding to Figure 4.3.

Figure 4.7 illustrates the optimized logic structure resulting from this approach. By isolating independent flow paths, Adroit reduces the complexity of synchronization logic and minimizes the impact of control broadcasts on timing.

4.3.2.2 Parallel Module Synchronization Optimization

For scenarios involving synchronization of parallel modules, as shown in Figure 4.10, Adroit adopts the following strategy:

1. Analyze the HLS schedule report to identify modules with deterministic latency.
2. For modules with known latency, implement a synchronization scheme that only waits for the slowest part to finish.
3. Generate optimized control logic that reduces the broadcast factor.

Figure 4.9 shows the resulting optimized logic structure for this scenario, where PE_1 is the slowest processing element to finish. Its Done signal is used to control whether all three modules PE_1, PE_2, and PE_3 can proceed.

```
#pragma HLS dataflow
while (1) {
    /* --- inferred parallelization --- */
    // Part #A
    inFifoA.read(&a);
    outFifoA1.write(a.foo);
    outFifoA2.write(a.bar); // #A

    // Part #B
    inFifoB.read(&b);
    outFifoB1.write(b.foo);
    outFifoB2.write(b.bar);
    /* --- HLS infers excessive synchronization --- */
}
```

Figure 4.8: Example code showing dataflow synchronization.

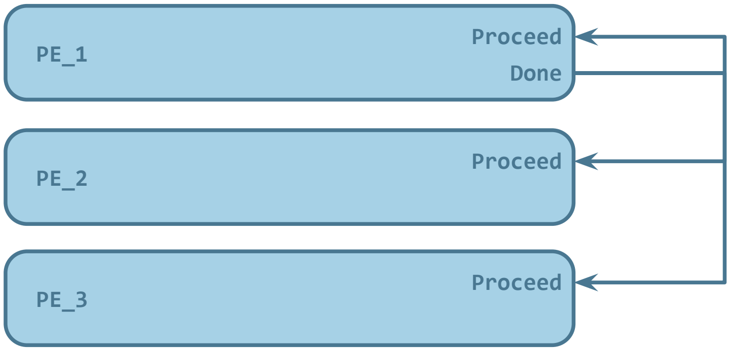


Figure 4.9: Optimized control-pruned architecture corresponding to Figure 4.4.

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

Figure 4.10: Example code showing parallel module synchronization.

This approach significantly reduces the complexity of synchronization logic for parallel modules with deterministic latency. For modules with dynamic latency, Adroit currently maintains the original synchronization scheme to ensure correctness. Future work may explore the use of symbolic execution techniques to handle more complex scenarios with variable latency.

4.3.3 Skid-Buffer-Based Pipeline Control

The third component of Adroit addresses the challenge of pipeline control broadcasts. These broadcasts occur in fully-pipelined datapaths, particularly when interacting with flow-controlled interfaces like FIFOs. To mitigate the performance impact of these broadcasts, Adroit implements a skid-buffer-based pipeline control strategy.

4.3.3.1 Skid Buffer Concept

The key idea behind the skid-buffer approach is to keep the pipeline always flowing and use additional buffering to handle backpressure. This strategy replaces the traditional stall-based control, which requires broadcasting control signals to all pipeline stages.

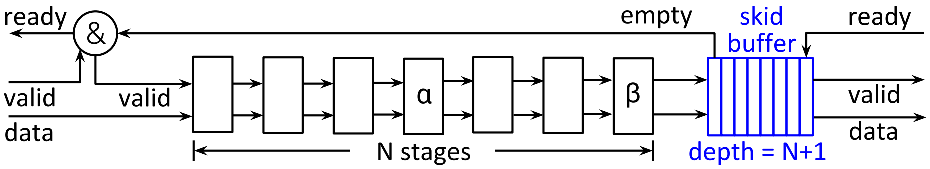


Figure 4.11: Skid-buffer-based pipeline control architecture.

Figure 4.11 illustrates the basic concept of skid-buffer-based pipeline control:

In the proposed method, the pipeline maintains continuity of operations even when downstream components are unable to process incoming data. To address this, a skid buffer [Int19] is integrated at the pipeline's terminal stage. This buffer serves to collect data during periods when downstream is not ready to avoid overflow due to the continued flow of the pipeline. The state of the skid buffer influences pipeline behavior: upon detecting data in the buffer (buffer non-empty), the reading of upstream data is halted, resulting in the transmission of invalid bubbles in subsequent pipeline stages.

The skid buffer's capacity is determined by setting its depth to $N + 1$, where N represents the number of stages in the pipeline. This configuration ensures the prevention of any overflow, with the additional unit $(+1)$ accounted for by the latency of the `empty` signal de-assertion, which occurs one cycle post the entry of the initial data element.

This method eliminates the need for global stall signals broadcast to the whole pipeline. Instead, the skid buffer's state is only passed to the immediate upstream stage, significantly reducing the impact of control broadcasts on timing.

4.3.3.2 Area Optimization

While effective at improving timing, the skid-buffer approach introduces additional area overhead. To minimize this overhead, Adroit implements a dynamic programming algorithm that optimizes the placement and sizing of skid buffers throughout the pipeline.

The primary insight is that the skid buffer's management can be decentralized within the datapath, as depicted in Figure 4.12. Rather than employing a single skid buffer with an N-depth and width w_{fi} at the end of the pipeline, Adroit places an $(M+1)$-depth buffer with width w_{ff}, corresponding to the data width at the intermediate M-th stage, and an $(N-M+1)$-depth buffer with width w_{fi}, corresponding to the data width of output after the final stage. This distributed buffering offers equivalent functionality to the singular, centralized buffer configuration illustrated in Figure 4.11.

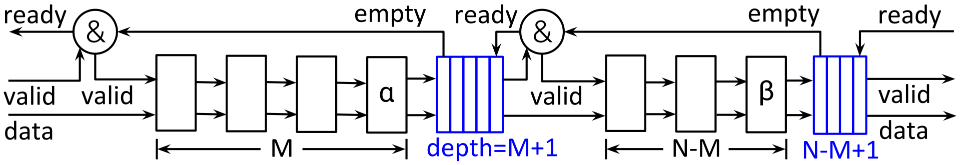


Figure 4.12: Multi-level skid-buffer-based pipeline control.

The new area overhead will be:

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

Let N be the number of stages of the pipeline; w_i be the width of data passed from stage #i to stage #$\{i+1\}$. S_i^j represents the subset of the pipeline which includes stages #$\{i, ..., j\}$. We aim to identify the optimal partition set $\mathbf{p}$ that effectively the entire pipeline into a sequence of sub-pipelines $S_1^{p_1}, S_{p_1+1}^{p_2}, \ldots, S_{p_t+1}^{N}$. Here, p_i represents the division points between stages, and t denotes the total number of such points, i.e., $|\mathbf{p}| = t$.

$$\begin{aligned} \min_{\mathbf{p}} \quad & TotalBufferArea_{\mathbf{p}} = \sum_{1 \le i \le t+1} (p_i - p_{i-1} + 1) \cdot w_{p_i} \\ \text{s.t.} \quad & p_i \in \mathbb{Z},\ p_i < p_{i+1},\ p_0 = 0,\ p_{t+1} = N \end{aligned} \tag{4.1}$$

We solve this optimization problem through dynamic programming algorithm. Specifically, for the initial N' stages of the pipeline, we determine the minimal total buffer area, represented by $f_{N'}$. Assuming that f_i is known for all $i < N'$, $f_{N'}$ is computed as follows in Equation 4.2:

$$f_{N'} = \min \begin{cases} (N'+1) \cdot w_{N'} \\ \min\{f_i + (N'-i+1) \cdot w_{N'} \mid 0 < i < N'\} \end{cases} \tag{4.2}$$

From the functions $f_1, \ldots, f_{N'-1}$, we derive $f_{N'}$. Ultimately, this series allows us to compute f_N, which represents the minimal area overhead required to incorporate skid buffers into the designated pipeline.

To determine w_i, we examine the operations within each stage. Let us define v_i as the set of all values produced at stage i; u_i as the set of all values consumed at stage i; and b_r as the bitwidth of any arbitrary value r. Values generated at stage i and used at stage j are propagated through all intermediate stages from i to j. Hence:

$$w_i = \sum_{j \le i} \sum_{v' \in v_j} b_{v'} \cdot c_{i+1,v'} \tag{4.3}$$

where $c_{i,v'} = 1$ if $\sum_{j \ge i} |u_j \cap \{v'\}| > 0$, otherwise $c_{i,v'} = 0$.

To determine the data width b_r transmitted between pipeline stages, we analyze the schedule report. This involves extracting both the definition and usage locations of each variable within the pipeline stages. By aggregating these data, we calculate the total data width transferred across stages.

By implementing and optimizing skid-buffer-based pipeline control, Adroit enables software developers to achieve high-performance pipelined designs without requiring them to manually manage complex flow control mechanisms. This approach is particularly valuable for long pipelines or designs with complex flow control requirements, where manual optimization would be challenging and time-consuming.

4.4 Evaluation

To assess the effectiveness of Adroit in enabling software developers to leverage FPGAs, we conducted a comprehensive evaluation using a diverse set of real-world applications. Our evaluation aims to demonstrate how Adroit's automated optimizations can improve the performance of HLS-generated designs without requiring manual intervention or hardware expertise from developers.

In this section, we present the results of our experiments, focusing on how Adroit addresses the challenges posed by implicit broadcasts in HLS-generated designs. We evaluate the impact of our optimizations on both the achievable clock frequency and resource utilization, demonstrating the potential for software developers to achieve high-performance FPGA implementations from high-level code descriptions.

Our evaluation seeks to answer the following key questions:

RQ1: How effective is Adroit in improving the clock frequency of HLS-generated designs across a range of applications?

RQ2: What is the impact of Adroit's optimizations on resource utilization?

4.4.1 Benchmarks

To ensure a comprehensive evaluation of Adroit's capabilities, we selected a diverse set of benchmarks that represent a range of application domains and computational patterns. These benchmarks were chosen from Section 2.3 to reflect real-world scenarios where software developers might leverage FPGA acceleration in heterogeneous computing environments, with fanout (FAN) and fixed architecture (FIX) issues as defined in Section 2.3 as the primary concerns. Our benchmark suite includes:

Genome Sequencing (GSQ). An accelerator for long-read pairwise overlapping in third-generation genome sequencing from Guo et al. [GLR19]. This benchmark allows us to evaluate Adroit's performance on data-intensive bioinformatics applications.

LSTM Inference Network. A benchmark of a Long Short-Term Memory (LSTM) network implementation for inference tasks, based on the work of Chen et al. [CHB18]. We focus on the N-Node component, using floating-point data types with N set to 256. This benchmark evaluates Adroit's performance on deep learning inference workloads, in addition to those examined in the Background chapter.

Face Detection (FDT). An implementation of the Viola-Jones face detection algorithm, sourced from the Rosetta benchmark suite [ZGD18]. This computer vision application demonstrates Adroit's effectiveness on image processing tasks.

Matrix Multiplication (MML). An optimized matrix multiplication kernel adapted from Cong et al. [CWY18a]. We further increased the parallelism to stress-test Adroit's ability to handle highly parallel designs.

String Search (STS). A string pattern matching accelerator, also adapted from Cong et al. [CWY18a]. This benchmark represents text processing applications.

Jacobi (JAC). A Jacobi stencil computation kernel generated by SODA [CCW18]. We include both a standard version and an HBM (High Bandwidth Memory) version to evaluate Adroit's performance with different memory architectures.

Streaming Buffer (STB). A custom design consisting of two loops that write to and read from a very large buffer. This benchmark tests Adroit's ability to optimize designs with complex memory access patterns.

These benchmarks were chosen to cover a wide range of application characteristics, including compute-intensive, memory-intensive, and control-intensive workloads. In brackets, we provide the abbreviated names used in Section 2.3 for each benchmark.

For each benchmark, we implemented both the original design from the benchmark and an Adroit-optimized version using Vitis HLS and Vivado 2018.2 with default optimization settings. We enabled the fan-out optimization included in the Vivado tool, demonstrating the impact of Adroit's additional optimizations thanks to the high-level information. The target FPGA models were selected based on the choices made in the original designs, ensuring a fair comparison. We summarize the results in Table 4.1, comparing the resources and frequency of these two versions.

In the following subsections, we will present detailed results for a subset of these benchmarks, highlighting how Adroit addresses specific broadcast-related challenges in each case. We will then provide a comprehensive summary of results across all benchmarks, demonstrating the overall impact of Adroit in enabling software developers to achieve high-performance FPGA implementations.

4.4.2 Broadcast-Aware Scheduling in Genome Sequencing Acceleration

To demonstrate the effectiveness of Adroit's broadcast-aware scheduling in enabling software developers to achieve high-performance FPGA implementations, we present a case study of a genome sequencing accelerator. This accelerator, originally designed by the authors in previous work [GLR19], targets the acceleration of Minimap2 [Li18], a state-of-the-art genomics tool renowned for its speed and accuracy.

The accelerator focuses on the chaining step of Minimap2 [Li18], which constitutes 70% of the tool's execution time. This step performs dynamic programming to identify sequences of matches with consistent distances between read pairs, indicating potential shared sub-sequences within the genome.

Accelerating this algorithm on FPGAs presents several challenges that are representative of the difficulties software developers face when targeting heterogeneous computing platforms: (1) Poor inherent parallelism in the original algorithm. (2) Large and variable

Table 4.1: Timing improvements and resources on HLS designs using Adroit.

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%

input data sizes, complicating task-level parallelism. (3) Complex data dependencies that can lead to broadcast-related performance bottlenecks.

The accelerator employs a fully pipelined streaming architecture. A key component of this design is a loop that performs distance calculations and score computations, unrolled to process multiple elements in parallel:

```
#pragma HLS pipeline
#define UNROLL_FACTOR 64
for (int j = 0; j < UNROLL_FACTOR; j++) {
#pragma HLS unroll
    dist_x = prev[j].x - curr.x;
    dist_y = prev[j].y - curr.y;

    dd = dist_x > dist_y ? dist_x - dist_y : dist_y - dist_x;
    min_d = dist_y < dist_x ? dist_y : dist_x;
    log_dd = log2(dd); // a series of if-else
    temp = min_d > prev[j].w ? prev[j].w : min_d;

    dp_score[j]= temp - dd * avg_qspan - (log_dd >> 1)
    if ((dist_x == 0 || dist_x > max_dist_x ) ||
        (dist_y > max_dist_y || dist_y <= 0) ||
        (dd > bw) || (curr.tag != prev[j].tag)) {
        dp_score[j] = NEG_INF_SCORE;
    }
} // ...
```

This code structure, while natural for software developers, leads to significant broadcast-related challenges when synthesized for FPGA implementation. For example,

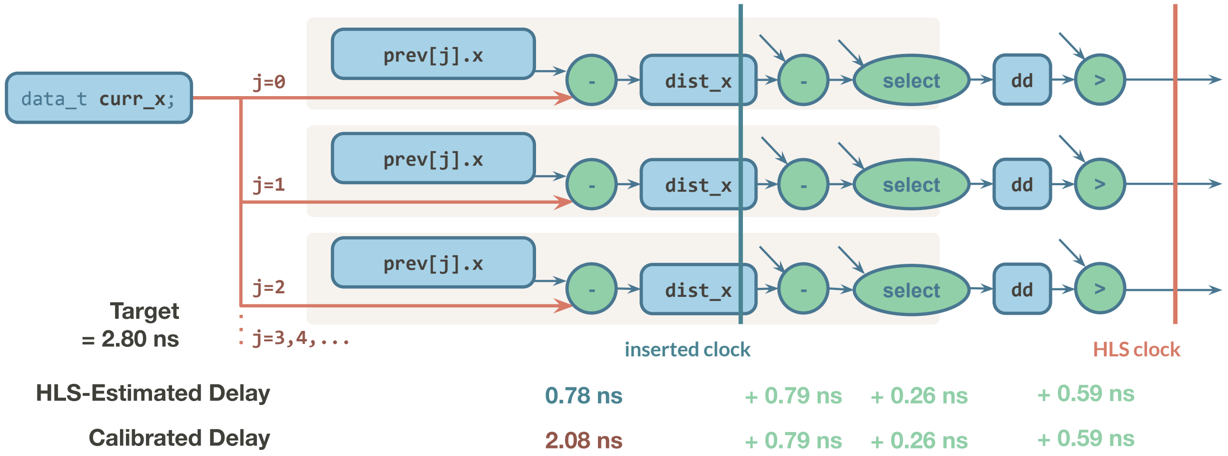


Figure 4.13: An operation chain with broadcast operators identified by Adroit.

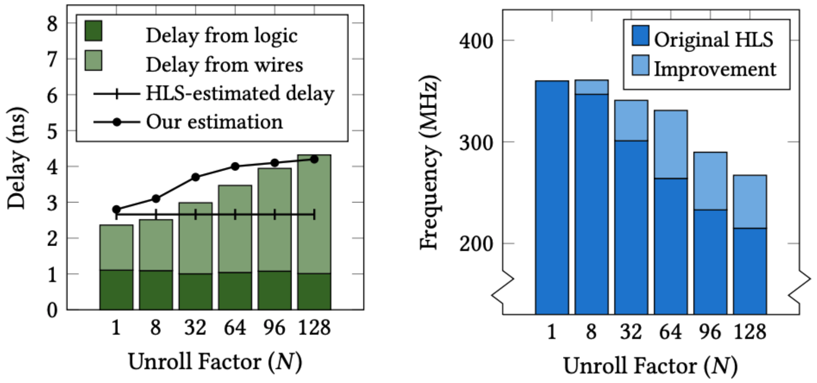
the variables like curr.x, curr.y, and avg_qspan are broadcast to multiple operations within the unrolled loop, creating high-fanout data signals that limit the achievable clock frequency.

4.4.2.1 Adroit's Broadcast-Aware Scheduling

Adroit's broadcast-aware scheduling technique identified that variables like curr.x are broadcast to multiple operations within the unrolled loop. The HLS tool, unaware of the additional delay introduced by these broadcasts, generates a suboptimal schedule that packs operations too tightly, even though the design cannot meet the desired clock frequency due to the broadcast-related delays.

Figure 4.13 illustrates an operation chain with broadcast operators as identified by Adroit, where curr.x is consumed by 64 sub operators.

Analysis revealed that the HLS tool predicted a delay of 0.78ns for each subtraction operation involving the broadcast variables. However, Adroit's calibrated delay model, accounting for the broadcast factor of 64, adjusted this prediction to 2.08ns. Based on this more accurate estimate, Adroit inserts a register module to split the operation chain,



(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 4.14: Calibrated delay estimation of Adroit and frequency improvement achieved by Adroit's architecture-aware scheduling with different broadcast factors.

preventing the accumulation of broadcast-related delays within a single clock cycle.

4.4.2.2 Results and Impact

The impact of Adroit's broadcast-aware scheduling on the achievable clock frequency is illustrated in Figure 4.14. In this optimization, Adroit's calibrated delay model provides a more accurate approximation of the actual delay, especially as the broadcast factor increases. Compared to the HLS-estimated delay, which remains constant, failing to account for the impact of broadcasts.

Adroit's optimization results in a significant frequency improvement, particularly for larger broadcast factors (unroll factor of the loop in the design). The optimization increased the pipeline depth by one, maintaining the same initiation interval of one. This minor increase in pipeline depth resulted in a negligible overhead in flip-flop while enabling a substantial improvement in clock frequency.

This case study demonstrates how Adroit enables software developers to achieve high-performance FPGA implementations without requiring manual identification and optimization of broadcast-related issues. By applying broadcast-aware scheduling, Adroit allows developers to write natural, high-level code while still benefiting from optimizations that traditionally required deep hardware expertise.

4.4.3 Synchronization Logic Pruning and Pipeline Control Optimization

To further demonstrate how Adroit enables software developers to write FPGA implementations that achieve high clock frequencies, we present two case studies focusing on synchronization logic pruning and pipeline control optimization. These studies illustrate how Adroit addresses complex hardware-specific issues that typically require a deep understanding of FPGA architecture, as well as HLS tool behavior, and typically involve manual interventions in the non-human-readable generated RTL code.

4.4.3.1 Synchronization Logic Pruning in HBM-based Jacobi Stencil Acceleration

Our first case study examines an HBM-based (High-Bandwidth Memory) Jacobi stencil acceleration kernel generated by the SODA compiler [CCW18]. This kernel represents a common scenario in high-performance computing where software developers aim to leverage advanced memory architectures for improved performance.

The kernel utilizes 28 independent memory ports of the HBM, with 512-bit data from each port scattered into eight 64-bit FIFOs for processing by different streaming kernels. While this approach allows for high memory bandwidth utilization, it introduces synchronization challenges that can impact performance.

Challenges. The SODA compiler, designed to simplify the development of stencil accelerators, expresses the 28 independent data flows together in a single loop. This results in a synchronization broadcast pattern similar to that shown in Figure 4.3, where unnecessary synchronization is introduced among all HBM ports and destination FIFOs.

Adroit's Optimization. Adroit's synchronization logic pruning identified the independent data flows and split them into separate loops, eliminating unnecessary synchronization. This optimization, which typically requires a deep understanding of the HLS tool's behavior, was performed by Adroit.

Results and Impact. The optimization resulted in a significant frequency improvement from 191 MHz to 324 MHz, a 69.6% increase. This substantial performance gain was achieved without requiring the software developer to manually analyze and modify the HLS-generated design, demonstrating Adroit's ability to enable high-performance FPGA implementations from high-level code.

4.4.3.2 Skid-Buffer-Based Pipeline Control Optimization

Our second case study focuses on pipeline control optimization, another area where achieving high performance typically requires hardware-specific expertise. We examine two scenarios: a 2D Jacobi kernel pipeline and a synthetic vector product example.

2D Jacobi Kernel Pipeline. We utilized the SODA compiler [CCW18] to generate a 2D Jacobi kernel as a pipeline. To evaluate Adroit's effectiveness on pipelines of varying complexity, we concatenated iterations of the kernel to create pipelines of increasing size. As pipeline depth increases, traditional stall-based control mechanisms can lead to timing closure issues due to the broadcast of control signals across the entire pipeline.

Adroit's Solution. Adroit analyzes the HLS schedule report to calculate the width of transferred data between pipeline stages. It then determines the optimal configuration of skid buffers and inserts them into the RTL design.

Result. Figure 4.15 illustrates the frequency improvements achieved by Adroit's skid-buffer-based pipeline control. For a super pipeline of eight Jacobi iterations (370 datapath stages), Adroit's optimization improved the achievable frequency by over 50% while requiring only about 23KB of additional BRAM resources.

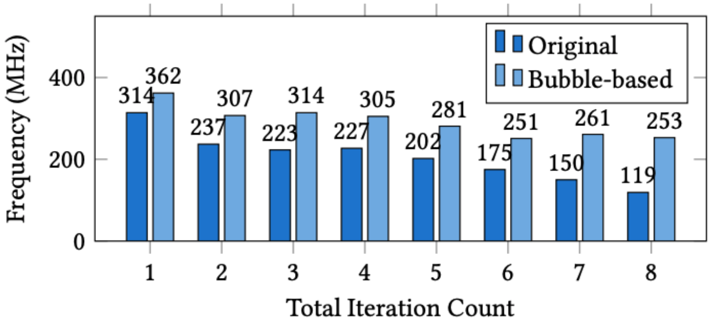


Figure 4.15: Achieved frequency of Jacobi kernels in different iteration counts with different pipeline control strategies, original design vs. optimized by Adroit.

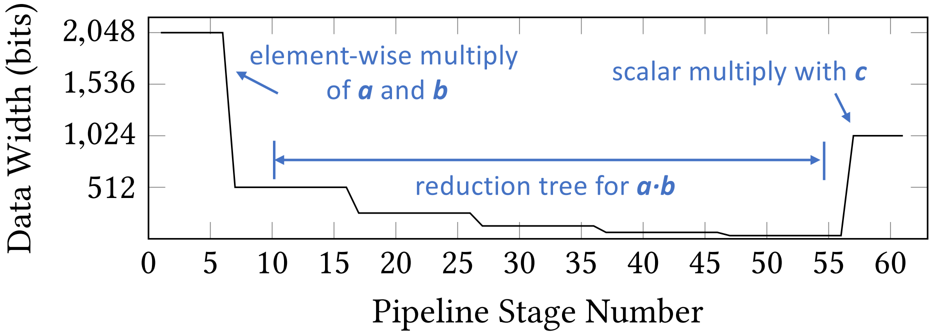


Figure 4.16: Bitwidth of the passed data between stages.

Synthetic Vector Product Pipeline. To further demonstrate the resource efficiency of Adroit's skid buffer optimization, we examined a synthetic example computing $(\mathbf{a} \cdot \mathbf{b})\mathbf{c}$, where the dot product of vectors **a** and **b** is scalar-multiplied with vector **c**. This computation pattern results in a pipeline with varying data widths between stages, as illustrated in Figure 4.16. In Stage #56 only one number (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, i.e., 8× smaller.

Adroit's Solution. Adroit's dynamic programming algorithm identified the aforementioned optimal placement of skid buffers, taking into account the varying data widths

Table 4.2: 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%

throughout the pipeline, resulting in a more resource-efficient design.

Result. Table 4.2 shows the implementation results for a 512-wide floating-point vector on a Xilinx UltraScale+ FPGA. Adroit's optimized skid buffer implementation achieved a 54% frequency improvement over the stall-based approach while using significantly fewer resources than a naive skid buffer implementation. The optimized version required only 0.02% BRAM usage compared to 12% for the naive approach, demonstrating Adroit's ability to balance performance improvements with resource efficiency.

4.4.4 Combined Results

While individual optimization techniques can yield significant improvements, many real-world applications require a combination of approaches to fully address timing degradation issues. This section demonstrates how Adroit's integrated approach, combining data broadcast optimization, synchronization logic pruning, and pipeline control optimization, enables software developers to achieve optimal performance without requiring deep hardware expertise.

4.4.4.1 Case Study: Stream Buffer

To illustrate the combined effect of Adroit's optimizations, we examine a stream buffer implementation that exhibits both data and control broadcasts. This example is represen-

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

loop2: for (int i = 0; i < BIG_SIZE; i++) {
#pragma HLS pipeline II=1
    out_fifo.write(buffer[i]);
} // data out of buffer
```

Figure 4.17: High-level code for the stream buffer example.

tative of common patterns in data processing applications when targeting FPGAs. Figure 4.17 shows the high-level code for this stream buffer. This seemingly simple code leads to two broadcast-related challenges when synthesized for FPGA implementation:

Data Broadcast. The source data register `in_fifo.read()` is connected to each BRAM unit `buffer[i]`, forming an implicit data broadcast.

Control Broadcast. The `enable` back-pressure signal is broadcast to all BRAM units implemented for the `buffer` array for flow control.

To address these issues, Adroit applies a combination of optimizations:

Data Broadcast Optimization. Based on the array size and pipeline depth, Adroit adds additional latency between `in_fifo` and `buffer` to optimize the data broadcast. The same optimization is applied to the `out_fifo.write()` operation.

Pipeline Control Optimization. Adroit implements skid-buffer-based pipeline control to eliminate the broadcast of the `enable` signal to the `buffer` BRAMs.

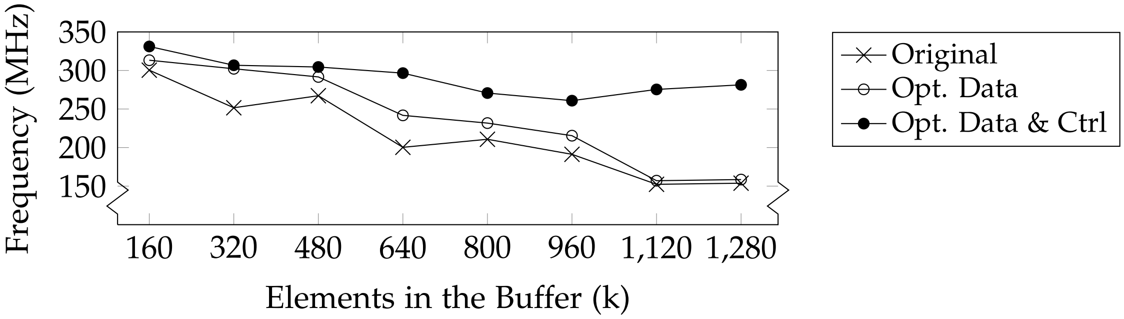


Figure 4.18: Achieved frequencies of the stream buffer design with different combinations of Adroit's optimizations compared to the original design.

Figure 4.18 presents the achieved frequency for varying buffer sizes under three scenarios: the original one (Original); the version which only has the data broadcast optimized (Opt. Data); the version with both optimizations (Opt. Data & Ctrl).

The results demonstrate that optimizing both data and control broadcasts is necessary to achieve scalable performance, especially as buffer sizes increase. This combined approach allows the design to maintain high frequency even for large buffer sizes.

4.4.4.2 Case Study: Pattern Matching

To further illustrate the combined effect of optimizations, we examine a pattern matching accelerator adapted from Cong et al. [CWY18a]. This design exhibits both data and synchronization control broadcasts, similar to the pattern shown in Figure 4.4.

Adroit identified and addressed both the data broadcast and synchronization control issues in this design by applying a combination of broadcast-aware scheduling techniques to optimize data-intensive operations and synchronization logic pruning to remove unnecessary synchronization, reducing control signal broadcasts.

Table 4.3 presents the implementation results for the pattern matching accelerator.

From the results, we observe that data broadcast optimization alone yielded an 11.2%

Table 4.3: Experiment results on pattern matching.

Implementation	Frequency	LUT	FF	BRAM	DSP
Original	187 MHz	17%	5%	9%	0%
Data Optimized	208 MHz	18%	7%	9%	0%
Adroit Fully Optimized	278 MHz	17%	7%	9%	0%

frequency improvement, while combining data and control optimizations resulted in a 48.7% frequency increase over the original design. The fully optimized version achieved this significant performance improvement with minimal impact on resource utilization.

4.4.4.3 Implications for Software Developers

The combined effect of Adroit's optimization techniques demonstrates its ability to address complex, interrelated performance issues in FPGA designs. This capability is particularly valuable for software developers, as it allows them to:

Write Platform-Neutral High-Level Code. Developers can express their algorithms in familiar, high-level constructs without worrying about low-level hardware optimizations, such as adding additional registers manually.

Achieve Scalable Performance. As demonstrated by the stream buffer example, Adroit's optimizations enable designs to maintain high performance even as problem sizes increase, exhibiting a similar behavior on traditional platforms.

Leverage Complex Optimizations. The pattern matching case study shows how Adroit can apply sophisticated, combined optimizations that would typically require extensive hardware expertise.

Maintain Code Readability. By handling optimizations automatically, Adroit allows developers to maintain clean, readable code without sacrificing performance.

By providing these capabilities, Adroit significantly lowers the barriers to entry for heterogeneous computing and aligns with our goal of making heterogeneous computing more accessible and practical for software developers across various domains.

4.5 Conclusion

This chapter has presented Adroit, a novel approach to addressing critical performance bottlenecks in HLS-generated FPGA designs. By focusing on the challenges posed by implicit broadcasts, Adroit significantly advances the goal of enabling software developers to effectively leverage heterogeneous computing resources, particularly FPGAs, without requiring extensive hardware expertise. Adroit's key contributions include:

Broadcast-Aware Scheduling. By implementing a more accurate delay model that accounts for broadcast-related delays, Adroit enables HLS tools to generate more efficient schedules, leading to improved timing performance.

Synchronization Logic Pruning. Adroit's ability to identify and eliminate unnecessary synchronization logic addresses a common source of performance degradation in parallel designs.

Skid-Buffer-Based Pipeline Control. The introduction of an area-efficient skid buffer implementation allows for high-performance pipelined designs without the timing penalties associated with traditional stall-based control mechanisms.

Integrated Optimization Approach. By combining these techniques, Adroit demonstrates the ability to address complex, interrelated performance issues that arise in real-world applications.

Our evaluation on a diverse set of benchmarks including genome sequencing, stencil computation, and pattern matching, demonstrates the effectiveness of Adroit's approach:

Frequency Improvements. Across various synthetic examples and real-world benchmarks, Adroit achieved an average frequency improvement of 53%, with some cases showing gains of over 100 MHz.

Minimal Overhead. These performance improvements were realized with minimal area overhead, ensuring efficient use of FPGA resources.

Scalability. Adroit's optimizations proved effective across a range of design sizes and complexities, from simple stream buffers to complex, multi-stage pipelines.

The significance of Adroit in enabling heterogeneous computing for software developers is multifaceted:

Abstraction of Hardware Complexity. By addressing complex broadcast-related issues, Adroit allows developers to write high-level code without concern for low-level hardware optimizations.

Performance Accessibility. Adroit enables software developers to achieve performance levels that would typically require deep FPGA expertise, democratizing access to high-performance heterogeneous computing.

Code Maintainability. The easy-integration nature of Adroit's optimizations allows developers to maintain clean, readable code while still benefiting from sophisticated hardware-specific optimizations.

Adroit signifies an advancement in democratizing FPGA acceleration for software developers; however, there are multiple directions for further exploration. First, although Vitis HLS has incorporated Adroit's skid-buffer and synchronization optimizations, it lacks data broadcast analysis. Integration of the data broadcast optimization approach into commercial HLS tools could streamline the optimization process for developers. Second, investigating methods for dynamic adaptation of optimizations, particularly synchronization pruning, could prove beneficial in settings where static analysis falls short. Third, extending Adroit's architecture-aware optimizations to broader programming models and other heterogeneous platforms, where data and control communication affinity plays a crucial performance role, such as AI engines, could further reduce the learning curve for software developers new to heterogeneous computing.

In conclusion, Adroit shows the potential for automated, architecture-aware optimizations to bridge the gap between software development practices and efficient FPGA implementation. By addressing implicit broadcasts in HLS-generated designs, Adroit enables software developers to leverage heterogeneous computing more effectively.

CHAPTER 5

Infrastructure for High-Level Physical Synthesis Optimizations

The evolution of Field-Programmable Gate Arrays (FPGAs) into larger, multi-die devices has significantly enhanced their capability to accelerate complex computations, offering improved performance and energy efficiency for applications such as large language models. This advancement, however, has introduced new challenges in design complexity and physical optimization, particularly for software developers leveraging High-Level Synthesis (HLS) tools to target these sophisticated FPGA architectures.

While HLS has made significant progress in simplifying FPGA programming by allowing developers to describe designs at an algorithmic level, it often falls short in addressing critical physical optimization challenges, especially when targeting specific FPGA boards. The absence of cycle-accurate and physical layout information in high-level specifications can lead to mismatches between frontend HLS and backend physical implementation, hindering timing closure and overall design quality.

To bridge this gap, we introduce RapidIR, a novel framework designed to enable software developers to compose high-performance FPGA systems without requiring deep hardware expertise. RapidIR addresses key limitations of existing High-Level Physical Synthesis (HLPS) approaches, offering support for hierarchical design optimizations, integration of diverse design sources (including HLS-generated modules, handcrafted RTL, and vendor IPs), and portability across various FPGA platforms.

RapidIR represents a significant step forward in making heterogeneous computing more accessible to software developers. By providing an extensible infrastructure for exploring physical optimizations in complex FPGA designs, RapidIR aims to democratize access to large-scale high-performance FPGA acceleration, enabling developers to leverage the full potential of modern multi-die FPGA architectures without sacrificing productivity or requiring extensive hardware design expertise.

In this chapter, we explore the key features of RapidIR, its underlying architecture, and its potential impact on software developers to effectively utilize heterogeneous computing resources. Through case studies and experimental results, we demonstrate how RapidIR can significantly improve design efficiency and portability, making FPGA acceleration more accessible and practical for a wider range of applications and developers.

RapidIR is presented at the 43rd IEEE/ACM International Conference on Computer-Aided Design as RapidStream IR [LXX24]. The software distribution of RapidIR could be found at `https://docs.rapidstream-da.com`.

5.1 Overview

5.1.1 Observations

Our investigation into the challenges faced by software developers in heterogeneous computing, particularly when targeting modern FPGA architectures, revealed several key observations:

Mismatch between HLS and Physical Implementation. The absence of cycle-accurate and physical layout information in high-level specifications often leads to suboptimal implementations when translated to hardware.

Scalability Issues. As FPGA designs grow in complexity, current EDA tools struggle to efficiently handle large designs, particularly when distributing processing logic across multiple dies.

Limited Optimization Scope. Existing HLPS solutions often focus on a narrow set of designs and FPGA devices, limiting their applicability to real-world scenarios, especially those applications written by software developers.

Integration Challenges. Many real-world accelerator designs incorporate a mix of HLS-generated modules, handcrafted RTL, and vendor-specific IPs, which existing tools struggle to optimize holistically.

To illustrate these challenges, consider the large language model (LLM) FPGA accelerator from Section 2.3, developed by Chen et al. [CZD24], as shown in Figure 5.1.

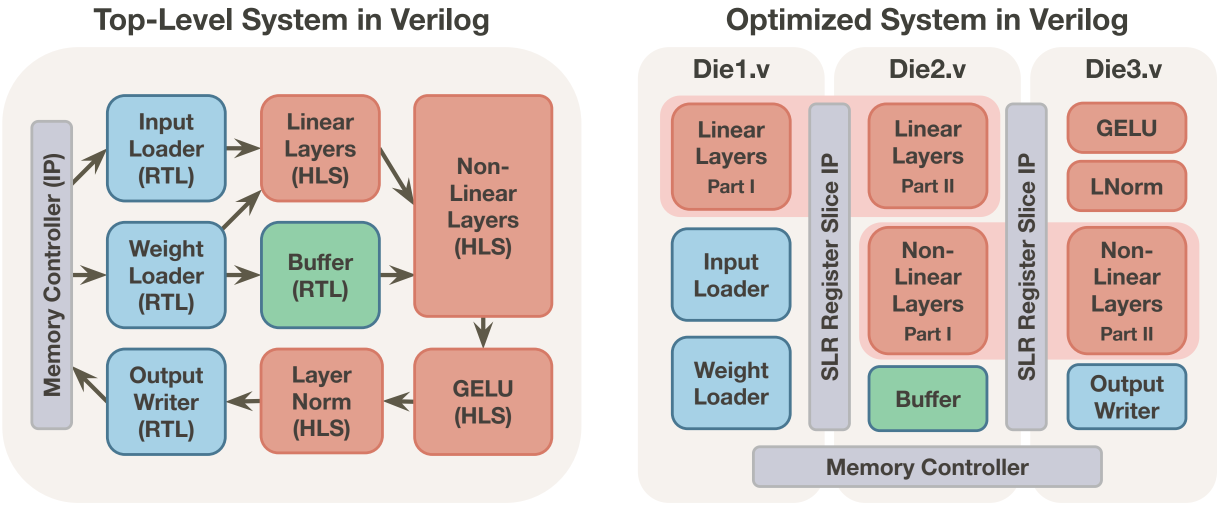


Figure 5.1: FPGA HLS accelerator design for large language models (LLM) before and after physical optimizations.

This design incorporates various source formats, including HLS-generated designs, such as the Linear Layer kernels and the GELU layer; reusable RTL components, such as the input loaders; and Intellectual Property (IP) components, such as the memory controller and pipeline stages between FPGA dies. This design requires manual optimization to achieve high performance. The initial implementation achieved only 150 MHz on a Xilinx Alveo U280 FPGA. Through extensive U280-specific optimizations, such as manual distribution of modules across FPGA dies and strategic register insertion, the frequency was improved to 245 MHz. However, this manual process is time-consuming, error-prone, difficult to port to new hardware platforms such as the AMD Versal ACAP platforms, and completely foreign to software developers.

While existing HLPS approaches have automated frequency optimizations for limited FPGA design styles, they do not support (1) integration of hybrid-source designs, akin to using libraries in software development, and (2) pipelining at multiple hierarchy levels, compelling developers to write program interactions in a single, flat fashion.

5.1.2 Approaches

To address these challenges and enable software developers to effectively leverage heterogeneous computing resources, RapidIR introduces several key innovations:

Flexible Intermediate Representation (IR). RapidIR offers an extensible IR that captures module connectivity, hierarchical structure, and physical spatial information of the design. This IR can be manipulated using any programming language, providing a powerful abstraction layer for optimization.

Reusable Optimization Passes. RapidIR provides a set of reusable passes for design transformation, including hierarchical rebuilding, module partitioning, and module insertion, allowing for exploration of different optimization strategies.

Support for Diverse Design Formats. RapidIR implements analyzers for various design formats, including Verilog, Xilinx Compiled IPs (XCI), Xilinx Vitis Object files (XO), and Vitis HLS-generated designs. This flexibility allows for the integration of diverse components common in real-world designs.

Cross-Platform Portability. RapidIR gives users a programming interface for defining the physical information of new FPGA devices, enabling portability across different platforms without modifying core optimization passes.

5.1.3 Contributions

The key contributions of RapidIR in enabling heterogeneous computing for software developers include:

Unified HLPS Infrastructure. RapidIR is the first HLPS infrastructure to support hierarchical composition of FPGA designs from diverse sources, enabling exploration of physical optimizations in complex designs while maintaining productivity.

Extensible IR for HLPS. The flexible and extensible IR allows for the creation of reusable passes that cater to various design formats and device targets, significantly reducing the effort required to support new design types or FPGA architectures.

Automated Optimization. RapidIR automates the physical layout optimization process for complex FPGA designs, achieving comparable or even better performance than manual optimization without requiring any code modifications.

Enhanced Portability. The framework enables seamless porting of designs across different FPGA platforms, with evaluations showing frequency improvements ranging from 30% to 62% across six FPGA devices.

Research Facilitation. Through case studies in floorplan exploration, parallel synthesis, and design debugging, RapidIR demonstrates its ability to facilitate research and exploration in FPGA design optimization.

By addressing these critical aspects of FPGA design and optimization, RapidIR enables developers to achieve high-performance FPGA system implementations from high-level descriptions, automates complex physical optimizations, and provides a flexible framework for exploring new optimization strategies.

The following sections will delve into the technical details of RapidIR, its implementation, and its impact on enabling software developers to effectively utilize heterogeneous computing resources, particularly FPGAs, in their large-scale, multi-kernel applications.

5.2 Problem Statement

As software developers increasingly turn to heterogeneous computing to meet the growing demands of complex applications, they face significant challenges when targeting modern FPGA architectures. This section outlines the key problems that RapidIR aims to address, focusing on the barriers that prevent software developers from fully leveraging the potential of FPGAs in heterogeneous computing environments.

5.2.1 Architectural Complexity of Modern FPGAs

The evolution of FPGAs into larger, multi-die devices has introduced new layers of complexity that are often opaque to software developers. Figure 5.2 illustrates this complexity using three representative FPGA architectures:

AMD Alveo U55C. This three-die FPGA features dedicated resources for the Vitis shell and 32 High-Bandwidth Memory (HBM) channels. The unprogrammable gap regions and the complex memory architecture pose significant challenges for efficient resource utilization and timing optimization. Additionally, within the high-level description of Vitis HLS, the HBM channels are represented as global memory, and developers utilize array arguments to access the channels without direct control over the location of the accessor module or the pipeline levels to the HBM controller. Communication between modules and the accessor is typically through stream

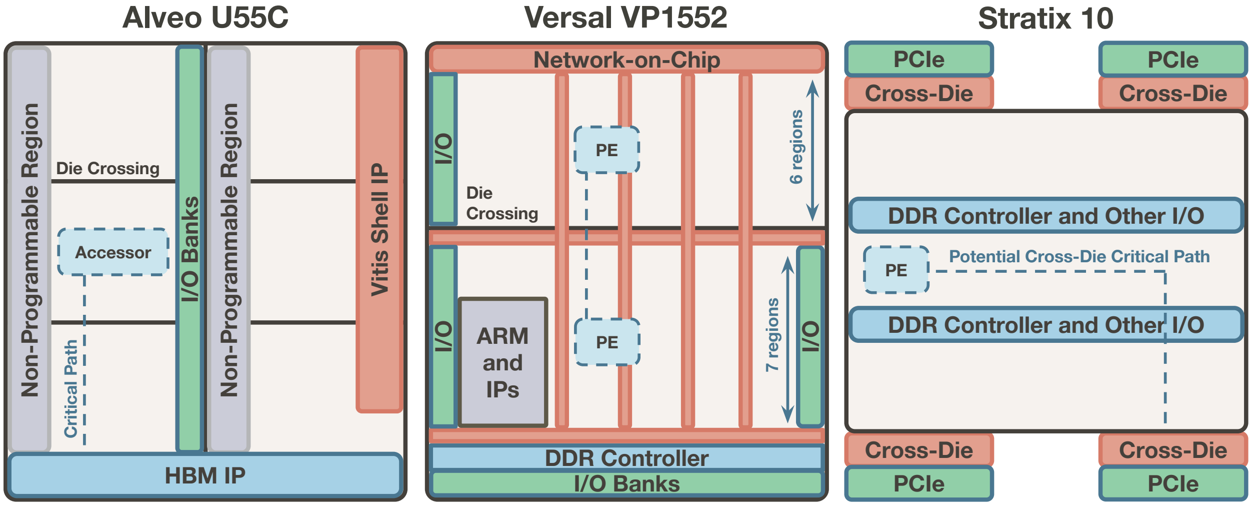


Figure 5.2: Layout of modern FPGA devices, highlighting architectural complexities not considered in HLS.

channels, which are implemented as straightforward FIFO modules, irrespective of the physical distance separating the interacting modules.

AMD Versal VP1552. Comprising two dies with different resource configurations, this FPGA includes network-on-chip interconnects and an integrated ARM processor. The discontinuities caused by IP blocks and the high latency of die crossings are not reflected in high-level synthesis, leading to potential performance bottlenecks.

Intel Stratix 10. This FPGA's unique layout, with I/O banks at the center and multi-die interconnects on the sides, introduces complexities not modeled in HLS tools. The current HLS tools do not consider the physical locations of the I/Os, potentially leading to inefficient architectural decisions and timing closure challenges.

These coarse-grained architectural layout intricacies are typically not exposed to software developers working with high-level synthesis tools, leading to a significant mismatch between the high-level description and the optimal hardware implementation.

5.2.2 Limitations of Current Approaches

While HLS has made significant strides in simplifying FPGA programming for software developers, it falls short in several critical areas when targeting modern, complex FPGA architectures:

Lack of Physical Awareness. HLS tools operate without knowledge of the underlying FPGA's physical layout, leading to suboptimal coarse-grained layout decisions, hindering downstream placement and routing. This is particularly problematic for multi-die FPGAs, where crossing die boundaries incurs substantial latency.

Limited Scope of Optimization. Current HLS tools and research focus primarily on local optimizations within individual modules, failing to capture global optimization opportunities across the entire design hierarchy.

Inflexibility in Design Composition. Many real-world FPGA designs incorporate a mix of HLS-generated modules, hand-crafted RTL libraries, and vendor-specific IPs. Current HLS flows struggle to optimize these heterogeneous designs holistically.

Poor Adaptability to New Architectures. As FPGA architectures evolve, HLS optimization tools often lag in supporting novel architectural elements, limiting the ability of software developers to leverage cutting-edge hardware.

5.2.3 The Need for High-Level Physical Synthesis

To bridge the gap between high-level software descriptions and efficient FPGA implementations, High-Level Physical Synthesis (HLPS) has emerged as a promising approach. HLPS aims to provide HLS with physical layout information, enabling more informed decisions about module partitioning, floorplanning, and pipeline insertion. Figure 5.3 illustrates the HLPS flow using the first few stages of the LLM accelerator. The process can be summarized in the following stages:

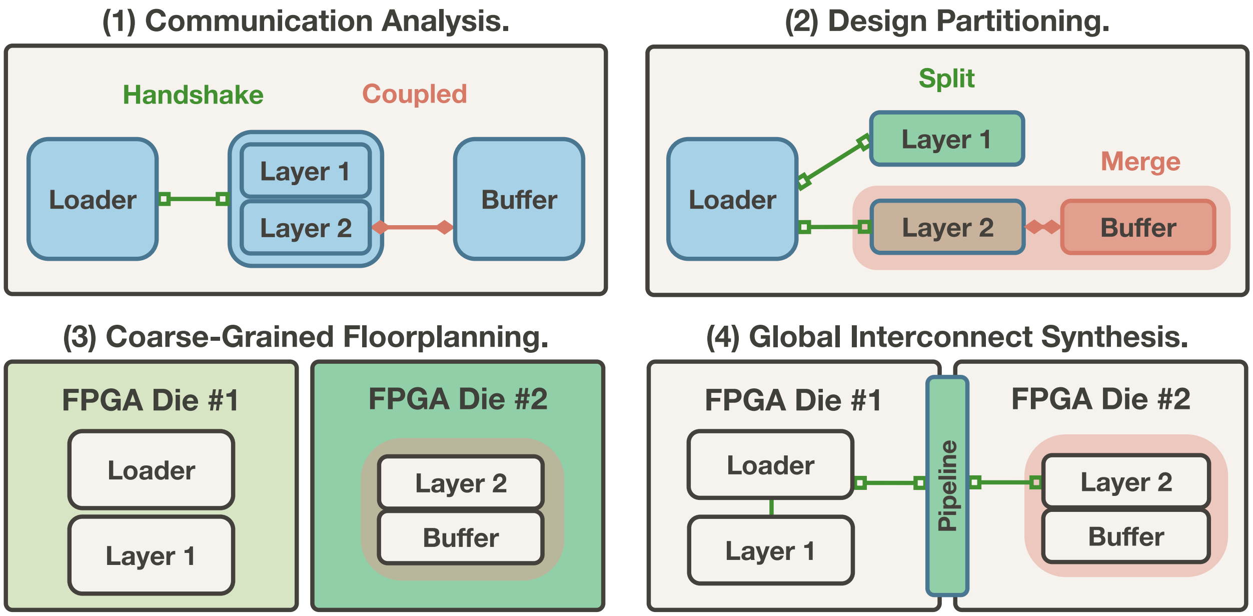


Figure 5.3: HLPS for the first three stages of the LLM design [CZD24].

Communication Analysis. The high-level specification, such as C++ code, is analyzed to identify connections between module units that can tolerate latency, such as handshakes and interconnect buses. These communications are typically represented as streams, data flow regions, and function arguments in C++.

Design Partitioning. The design is divided into partitions based on communication patterns, allowing only latency-tolerant connections between groups. These partitions can be distributed across distant regions or different dies, and the connections between them can be pipelined, breaking global critical paths.

Coarse-Grained Floorplanning. Partitions are allocated to coarse-grained regions on the FPGA, optimizing multiple objectives such as minimizing inter-region wire crossings, managing regions with limited available resources, and balancing resource distribution to prevent local routing congestion.

Global Interconnect Synthesis. Once the location of each partition is determined, the partitions are interconnected based on estimated delay to break critical paths.

A number of studies [GCW21, DLS23, DLZ24, GCL23, KTC23, NBN23, LLC23, MGC23] have investigated methodologies for HLPS and demonstrated their effectiveness in automatically optimizing the frequency of HLS designs. However, the application of HLPS is limited by several shortcomings in (1) global optimization across hierarchical levels, (2) integration of diverse source formats, and (3) adaptation to new devices:

Limited Hierarchical Support. Existing HLPS approaches often fail to optimize across different hierarchical levels of a design, adding pipeline stages only at the module boundaries in the top-level module, leading to suboptimal global solutions.

Lack of Integration for Diverse Design Sources. Many HLPS tools focus solely on HLS modules, neglecting the reality of mixed-source designs common in real-world applications, such as the use of vendor-specific IPs and reusable RTL libraries.

Device-Specific Implementations. Current HLPS solutions are often tied to specific FPGA devices or families, limiting their applicability across different platforms.

Insufficient Infrastructure for Research and Exploration. The lack of a flexible, extensible framework for HLPS hinders research into new optimization strategies and adaptation to emerging FPGA architectures.

5.2.4 Challenges in Real-World Physical Layout Optimization

To illustrate these challenges, consider the Large Language Model (LLM) FPGA accelerator design shown in Figure 5.4. This design exemplifies the complexities faced by software developers when targeting heterogeneous computing platforms:

Mixed-Source Integration. The design incorporates library modules implemented in Verilog RTL (Input Loader, FIFO) alongside HLS-generated components (Linear

Layers), connected by top-level Verilog logic. Current HLS approaches struggle to optimize such heterogeneous designs comprehensively. However, software developers, without deep hardware expertise, often rely on vendor-provided, highly optimized RTL libraries for critical components.

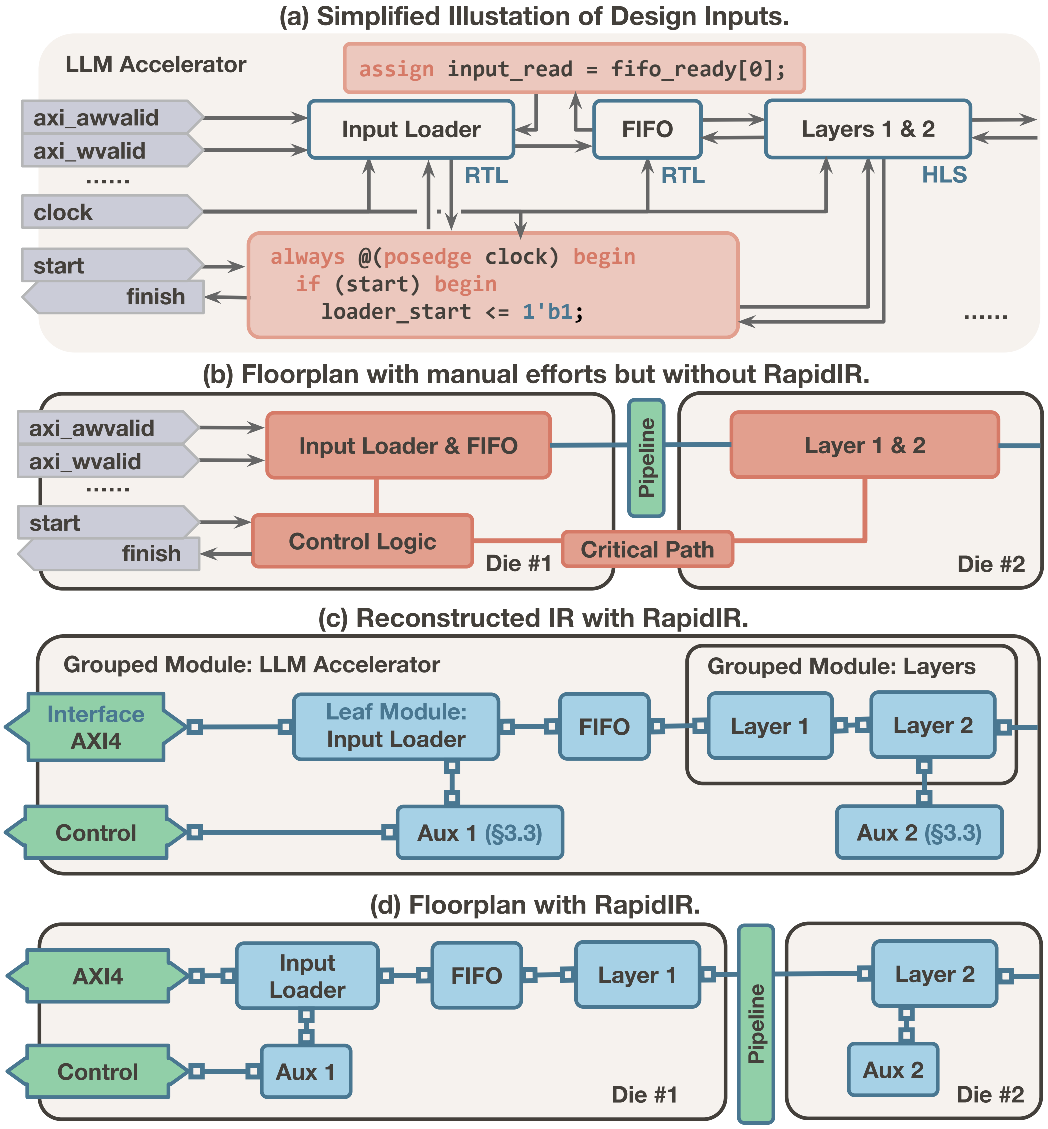
Hierarchical Optimization. The two Linear Layers, while logically part of a function, would benefit from independent placement and pipelining to balance resource utilization across the FPGA. Existing tools lack this level of hierarchical flexibility. Software programmers are forced to flatten their designs to achieve optimal performance, leading to complex monolithic programs that are difficult to maintain.

Control Logic Partitioning. The top-level Verilog control logic, if treated as a monolithic unit, can lead to non-pipelined connections and global critical paths. Intelligent partitioning of this logic is crucial for optimal performance but is beyond the capabilities of current HLPS solutions.

Portability Concerns. Adapting this design to new FPGA architectures or exploring different optimization strategies requires substantial manual effort. Even worse, without a deep understanding of the underlying FPGA hardware layout, software developers may inadvertently introduce inefficiencies that hinder performance.

These challenges underscore the need for a more comprehensive, flexible approach to HLPS that can empower software developers to effectively leverage heterogeneous computing resources, particularly FPGAs, without requiring deep hardware expertise.

In the following sections, we introduce RapidIR, a novel optimization infrastructure designed to address these challenges and enable software developers to achieve high-performance FPGA implementations from high-level descriptions across a wide range of applications and target architectures.



(a) Simplified Illustation of Design Inputs.

```
assign input_read = fifo_ready[0];
```

```
always @(posedge clock) begin
  if (start) begin
    loader_start <= 1'b1;
```

(b) Floorplan with manual efforts but without RapidIR.

(c) Reconstructed IR with RapidIR.

(d) Floorplan with RapidIR.

Figure 5.4: LLM accelerator design optimized with and without RapidIR, highlighting challenges in optimizing complex mixed-source FPGA implementations.

5.3 Approach

RapidIR is a comprehensive framework designed to bridge the gap between high-level software descriptions and FPGA implementations with efficient coarse-grained layout. RapidIR addresses the challenges outlined in the previous section by providing a flexible, extensible infrastructure for HLPS that empowers software developers to leverage the full potential of modern multi-die FPGA architectures without requiring deep hardware expertise. It consists of three key components:

Intermediate Representation (IR). A progressively refined representation that remains agnostic to specific HLS frameworks, EDA tools, or coding styles.

Utility Plugins. Tools for inputting design specifications and outputting to EDA tools, bridging the gap between high-level descriptions and low-level implementations.

Transformation Passes. A set of reusable operations for composing design optimizations, allowing for flexible and extensible optimization strategies.

Figure 5.5 illustrates the overall architecture of RapidIR. RapidIR takes three types of inputs: (1) FPGA designs (e.g., Verilog, netlists); (2) high-level interface information (e.g., HLS reports, pragmas); and (3) Python directives for device information and EDA tool interaction. These inputs are processed by plugins into the *IR*, which is then modified by transformation *passes* to perform the HLPS flow. The final IR is processed back by the *plugins* into optimized design code and layout hints and constraints for EDA tool implementation.

5.3.1 Design Principles

In developing RapidIR, we adhered to several key design principles aimed at making heterogeneous computing more accessible to software developers:

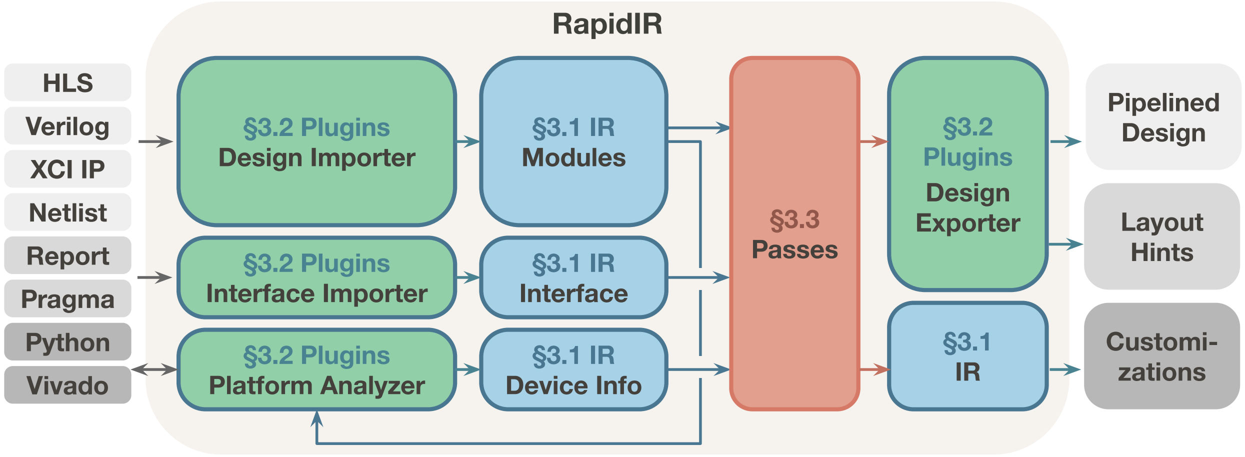


Figure 5.5: RapidIR's overall architecture, consisting of the intermediate representation (IR, blue), utility plugins (green), and transformation passes (red).

Enabling Incremental Analysis and Transformation. We intentionally designed the IR to be lightweight and robust, allowing for incremental refinement of the design. This approach makes passes and plugins simpler and more modular, facilitating easier optimization development and maintenance.

Scoping Flexibility. Rather than attempting to create an all-in-one solution like MLIR [LAB21], we focused on the practical requirements for HLPS methodologies. This approach allows us to prioritize coarse-grained module interactions while maintaining support for fine-grained logic that cannot be easily translated into IR.

Language Agnosticism. Recognizing that not all computations warrant the development overhead of C++, we made the IR as simple as possible, using a subset of the JSON schema [JSO20]. This design choice supports all major programming languages and provides automated language binding generators, allowing developers to work in their preferred language.

Consistency and Debuggability. We provide "Design Rule Checking (DRC)" passes to ensure consistency in design information and maintain a mapping between original design components and their transformed counterparts. This feature enhances human readability and debuggability, crucial for complex FPGA designs.

In the following sections, we will delve into the details of each component of RapidIR, demonstrating how they work together to enable efficient FPGA design for software developers without requiring extensive hardware expertise.

5.3.2 Progressively Refined Intermediate Representation

At the heart of RapidIR is a progressively refined Intermediate Representation (IR) that serves as a bridge between high-level software descriptions and low-level FPGA implementations. This IR is designed to incrementally incorporate a design's coarse-grained information, keeping the original fine-grained logic intact if it is unused in the passes. This make it particularly suitable for developers who may not be familiar with the intricacies of fine-grained hardware design.

5.3.2.1 Design Elements

The RapidIR IR captures the following key elements of a design:

Module. A design entity classified into *grouped module* and *leaf module*. Each module is identified by a name and consists of multiple ports, each having direction and width attributes, interconnecting with other modules. They can incorporate *interface* that identifies the potential pipeline methods of the ports.

Leaf Module. A basic design unit treated atomically by HLPS, which keeps it intact. Leaf modules can be in any format, such as RTL or IPs, provided they are supported by subsequent EDA tools. RapidIR provides various utility plugins to obtain the required attributes of a leaf module. A leaf module may be progressively reconstructed into a grouped module or partitioned into multiple leaf modules using RapidIR's transformation passes.

Grouped Module. A reconstructed hierarchy from a leaf module, organizing submodules. Grouped modules act only as containers without adding logic, which implies that each submodule connection must be via a single identifier. RapidIR progressively partitions its submodules while adhering to this rule.

Interface. A pipeline strategy that can be applied to a set of ports. The *type* of the interface guides the pipelining strategy, such as handshake or feedforward. When a port is included in an interface, it allows for pipelining by introducing additional pipeline stages. For instance, a *feedforward* interface, carrying only scalar signals, can be pipelined by inserting a flip-flop to break critical paths. A *handshake* interface, involving valid, ready, and data ports, can be pipelined by adding a relay station [BCd09] or an almost-full FIFO [GLC20]. Figure 5.6 illustrates these two most common interfaces and their pipelining methods.

Additional Metadata. The IR can include extra data such as floorplan constraints, resource utilization, and timing characteristics, appended to any IR node as additional fields and progressively inferred and updated by analysis passes as needed.

5.3.2.2 Invariant Assumptions

A valid IR must adhere to several invariant assumptions at all times:

1. Each wire in a grouped module must connect precisely two modules, prohibiting broadcast or fan-out in the intermediate representation.
2. Each submodule port in a grouped module must connect to only one identifier or a constant, without operations such as concatenation or bit selection.

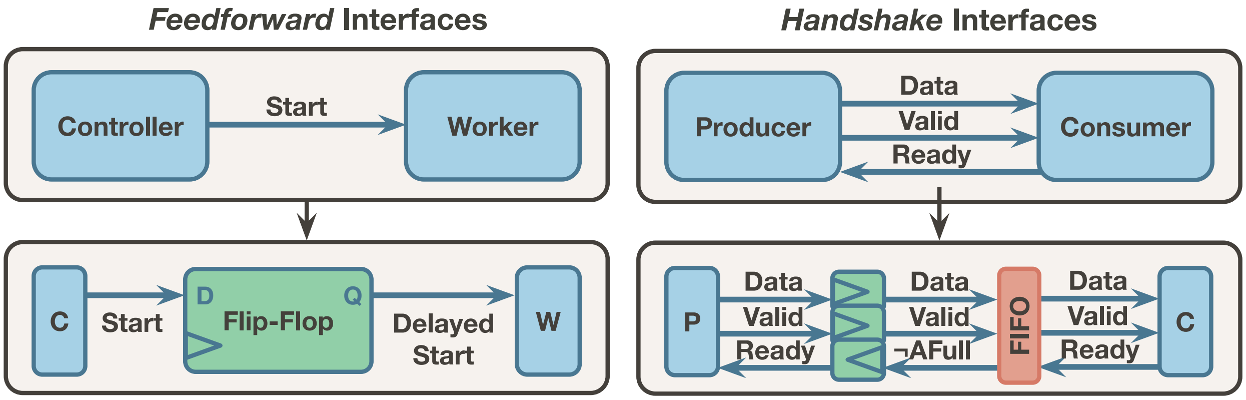


Figure 5.6: *Feedforward* interfaces are pipelined using flip-flop registers, and *handshake* interfaces are pipelined with an almost-full FIFO and registers. AFull indicates that the FIFO is almost full, preventing overflow due to flip-flop latency.

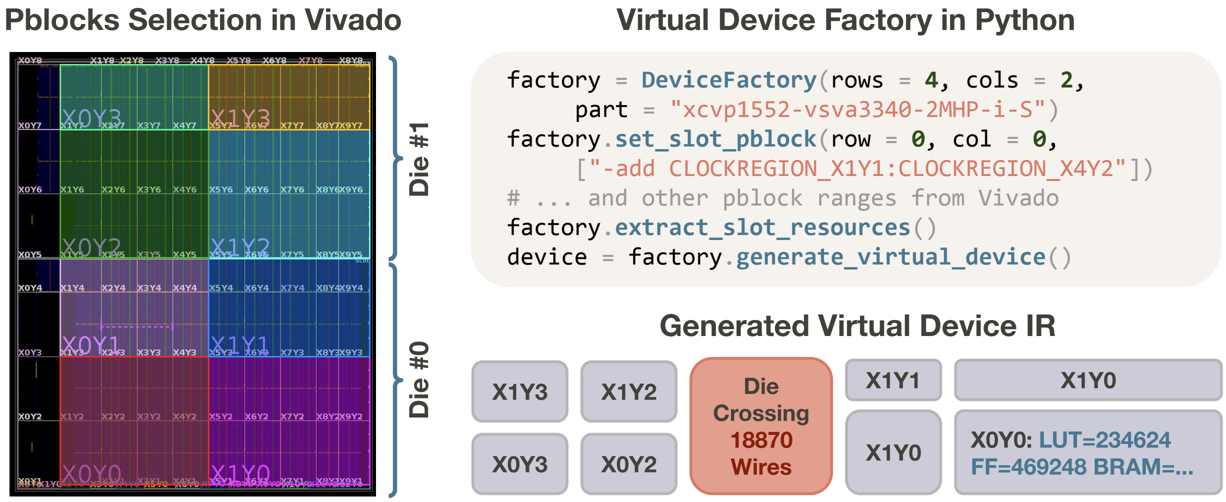
3. All non-constant ports on an interface should be fully connected to another module, disallowing the splitting or omission of signals.

These restrictions maintain the simplicity and ease of manipulation of the IR. Despite being restricted, our core passes enable the handling of complex designs in this form.

5.3.2.3 Virtual Device Definition

To support a wide range of FPGA devices, RapidIR introduces the concept of virtual device descriptions stored in the IR. These descriptions contain the resource distribution within the device and the number of inter-die wires. The virtual device description divides the physical FPGA device into slots.

During floorplanning, design modules are mapped to these slots. RapidIR includes predefined virtual devices for UltraScale+ and Versal, based on empirical data. Users can also customize the virtual device by specifying parameters such as the FPGA device part number and the slot shapes. RapidIR then uses vendor tools to extract the necessary


Pblocks Selection in Vivado
Virtual Device Factory in Python
X0Y3
X1Y3
X0Y2
X1Y2
X0Y1
X1Y1
X0Y0
X1Y0
Die #1
Die #0
Generated Virtual Device IR
X1Y3
X1Y2
X0Y3
X0Y2
Die Crossing 18870 Wires
X1Y1
X1Y0
X1Y0
X0Y0: LUT=234624 FF=469248 BRAM=...

```
factory = DeviceFactory(rows = 4, cols = 2,
     part = "xcvp1552-vsva3340-2MHP-i-S")
factory.set_slot_pblock(row = 0, col = 0,
     ["-add CLOCKREGION_X1Y1:CLOCKREGION_X4Y2"])
# ... and other pblock ranges from Vivado
factory.extract_slot_resources()
device = factory.generate_virtual_device()
```

Figure 5.7: Pblocks for VP1552, RapidIR virtual device description, and inferred resource and die-crossing wire capacity by RapidIR plugins.

resource information and automatically generates the virtual device description.

Figure 5.7 shows an example of a virtual device description for the Versal VP1552.

By using description files from hardware experts or built-in settings in RapidIR, this approach allows software developers to target different FPGA architectures without needing to understand the low-level details of each device.

5.3.2.4 Sample IR Format

RapidIR uses a structured format that can be validated using JSON Schema, making it accessible to a wide range of programming languages and tools. The choice of storage and exchange format for the IR, such as YAML [BEI09], JSON [BRS17], or XML [SW03], can optionally vary depending on the programming languages utilized. Figure 5.8 illustrates an example segment of the IR for an LLM accelerator, presented in YAML format for clarity, alongside its corresponding block graph.

```
- module_name:      LLM
  module_ports:
  - { name: ap_clk, direction:   in,  width:      1 } # ..
  module_wires:     [{ name: I_wire,  width:     64 },   ..]
  module_submodules:
  - instance_name:  InputLoader_inst
    module_name:    InputLoader
    connections:    [{ port:      I,  value: I_wire },   ..]
  - instance_name:  FIFO_inst
    module_name:    FIFO
    connections:    [{ port:      I,  value: I_wire },   ..]
  - instance_name:  Layers_inst                       # ..

- module_name:      FIFO
  module_ports:
  - { name: I,      direction:    in, width:     64 }
  - { name: I_rdy,  direction:   out, width:      1 }
  - { name: I_vld,  direction:    in, width:      1 }
  - { name: ap_clk, direction:    in, width:      1 } # ..
  module_verilog:   "module FIFO (I, ..); ..; endmodule"
  module_interfaces:
  - iface_type:     handshake
    iface_ports:  { data:      [ I ], clk:   ap_clk ,
                    ready:     I_rdy, valid:  I_vld }
  module_metadata:
    resource:     { FF: 10, LUT: 39, DSP: 0, BRAM: 0,   ..}
    floorplan:      "SLOT_X1Y1"

- virtual_device:   VP1552
  floorplan_slots:
  - name:           "SLOT_X0Y0"
    pblocks:      [ "CLOCKREGION_X1Y1:CLOCKREGION_X4Y2" ]
    resource:     { LUT: 234624, FF: 469248, ... }   # ..
```

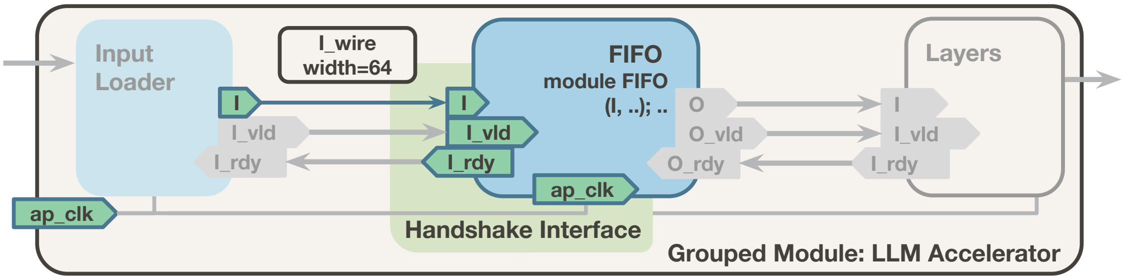


Figure 5.8: Part of the LLM's IR and corresponding block graph.

The top-level grouped module, LLM (Lines 1-12), instantiates three submodules under it: InputLoader, FIFO, and Layers (Lines 5-12), which are interconnected via handshake interfaces. InputLoader retrieves text input from memory, FIFO buffers this data, and Layers executes linear layer computations on the buffered input. The IR captures coarse-grained information such as module names (Lines 1, 14), ports (Lines 2-3, 15-19), and wires (Line 4). The instantiation of the FIFO module is denoted as FIFO_inst (Lines 9-11), connecting I_wire to its I port (Line 11). Within the leaf module FIFO, the IR preserves its native form, such as Verilog source code (Line 20). Details regarding the pipeline are specified in the interface section (Lines 21-24), which defines the handshake interface and its associated ports. Each object can optionally contain additional metadata specific to different transformation passes, such as resource utilization for floorplanning solver passes and floorplan constraints for pipeline insertion (Lines 25-27).

The device information can be embedded in the IR to facilitate transformation passes that optimize the design for a specific FPGA target or generate constraints for EDA tools. For instance, Lines 29-33 present the virtual device description for the VP1552 FPGA, as illustrated in Figure 5.7.

This representation allows developers and pass designers to work with a high-level, abstract view of the design while still capturing the necessary details for efficient FPGA implementation.

5.3.2.5 Type System

RapidIR incorporates a validation-based type system to ensure the robustness of its IR. This type system checks design consistency and enables optimizations across diverse input formats and target architectures, enabling software developers to work with heterogeneous components without extensive hardware knowledge.

The rationale behind employing checking over syntax-based typing, which prohibits typing mismatches in representation, lies in the fact that RapidIR's input originates from various sources, rather than being solely defined by end-users. It is an overly stringent requirement to mandate that users refactor all design components to conform to the typing system's specifications.

Type Definitions. The type system in RapidIR is embedded in the representation as attributes and checked statically during optimization runtime. It operates at three levels:

1. **High-level typing:** Despite not being part of the RapidIR representation, when operating on HLS designs, high-level languages such as Vitis HLS C/C++ verify the communication protocol, including streams and shared memory, and data types, such as integers and floating points, within the program representation. This verification ensures that data transfer between modules is accurately matched. This information is subsequently passed to the lower level of interface-level typing using HLS reports.

2. **Interface-level typing:** Not all design modules are programmed using a high-level programming language. Therefore, with the high-level typing, the checking is not performed for the libraries, analogous to the syscalls used by software. Interface-level typing assigns higher-level types (e.g., '`handshake`', '`feedforward`', '`AXI`') to interfaces, encapsulating the semantic meaning of port groups. Although one might assume that the direction of interfaces and widths should be recorded in the type system for consistency checking, this information is actually embedded in the ports constituting the interface and is checked at the prior level. RapidIR provides an array of pipeline templates to be inserted in the pipeline insertion pass so that the types after transformation remain matched.

3. **Port-level typing:** Assigns basic types (e.g., '`reg`', '`wire`'), directions (e.g., '`input`', '`output`', '`inout`'), and widths to each port, extracted from original design sources.

The type system guides optimization processes, such as pipeline insertion, ensuring that control signals in handshake interfaces maintain protocol correctness with proper pipeline modules. Users are able to define additional types by specifying the ports to be included in an interface and providing a pipeline stage template.

Type Inference. RapidIR implements type inference algorithms to deduce types from modules with known types, such as modules generated by HLS or libraries annotated by their designers, and automatically deduce rule-based definitions from input designs when explicit type information is not provided. The type inference is particularly useful when working with legacy RTL designs that may not have explicit type annotations.

Type Checking. To ensure interface compatibility between modules, RapidIR implements a matching algorithm that checks for type consistency when connecting modules. This algorithm verifies: (1) Port width compatibility; (2) Protocol compatibility (e.g., ensuring handshake interfaces are connected to other handshake interfaces); and (3) Direction compatibility (ensuring inputs connect to outputs and vice versa). When mismatches are detected, RapidIR raises an error to the developer with detailed diagnostic information.

This type system enables software developers to confidently compose complex FPGA designs from heterogeneous components, automatically handling many low-level details of interface compatibility that typically require hardware design expertise.

5.3.2.6 Comparisons with Other IRs

RapidIR focuses on HLPS for *existing* designs in various formats, differing from other representations in several key ways:

- Unlike Xilinx IP Integrator (IPI), RapidIR focuses on HLPS for existing designs in various formats. Xilinx IP Integrator (IPI) assembles IPs into systems and treats large

HLS-generated modules monolithically. In contrast, large HLS-generated modules can be partitioned in RapidIR thanks to the flexible representation that supports incremental analysis through passes, where all modules are initially treated as indivisible leaf modules and partitioned as needed.

- Compared to accelerator description languages like Chisel [BVR12] and Calyx [NTL21], RapidIR is tailored for capturing coarse-grained information pertinent to HLPS. These accelerator description languages enable the specification of fine-grained hardware designs and are orthogonal to RapidIR. Given the language-agnostic design of RapidIR, it can directly incorporate these representations as leaf modules, allowing the transformation of these modules using reusable passes.
- In contrast to MLIR [LAB21], RapidIR is specifically designed for HLPS with a coarse-grained focus and accommodates arbitrary formats in leaf modules. MLIR serves as a general-purpose IR across abstraction levels, mandates the use of C++ for transformation passes, and does not have a native representation to keep fine-grained information intact. Although RapidIR can be forced into MLIR with a custom dialect to represent the JSON schema, such a representation would require C++ for the passes, which RapidIR intentionally avoids. Moreover, MLIR lacks reusable passes for HLPS, negating the motivation for its use in this domain.

These distinctions make RapidIR well-suited for software developers to work with complex FPGA designs without requiring extensive hardware design expertise. By providing a flexible, incrementally refined representation, RapidIR allows developers to focus on high-level design concepts while the framework handles the intricacies of mapping to efficient FPGA layout implementations.

5.3.3 Practical Utility Plugins

RapidIR's utility plugins serve as bridges between the abstract IR and concrete implementations. These plugins are designed to be modular and extensible, supporting a wide range of source formats and EDA tools to accommodate diverse development workflows.

The utility plugins in RapidIR are categorized into three main types: (1) *importers* parsing the user design input and generating RapidIR, (2) *analyzers* obtaining target-dependent information of the design from backend tools, and (3) *exporters* producing optimized designs for the user to continue in their flow.

5.3.3.1 Leaf Module Importer

The leaf module importer is responsible for extracting metadata from a module's source format and constructing a corresponding leaf module in the IR. This process involves:

1. Parsing module names, ports, and other relevant data.
2. Embedding the original source code or binary in the IR to maintain design integrity.

RapidIR supports a variety of formats, including Verilog, VHDL, netlists, and Xilinx Compiled IP (XCI). For Verilog, RapidIR utilizes the Slang tool [Pop24] to extract module information from the syntax tree. Other formats are handled using appropriate parsers or by transforming module signatures into Verilog stub files, which are then processed by the Slang-based Verilog importer.

This flexibility allows software developers to work with their preferred languages, vendor tools, IP blocks, or reusable RTL libraries without needing to manually translate or refactor between different source code formats.

```
module InputLoader (
  output wire m_axi_AWVALID,  input wire m_axi_AWREADY,
  output wire m_axi_WVALID,   input wire m_axi_WREADY,
  // ... 33 other AXI ports
);
  // pragma handshake pattern=m_axi_{bundle}{role} \
     role.valid=VALID role.ready=READY role.data=.*
endmodule
```

Figure 5.9: Interface `pragmas` in Verilog mapping ports with the `m_axi_` prefix to handshake interfaces and bundle ports with the same prefix (e.g., `m_axi_AW`). Suffixes `VALID` and `READY` indicate port roles, while any other suffixes denote data.

5.3.3.2 Interface Importer

The interface importer extracts high-level interface information essential for HLPS from various sources, including interface information from Vitis HLS report files and Xilinx IPs' XCI files. If interface data is missing, users can provide it using *pragmas in source-code comments* or *interface rules* specified in our Python API with regular expressions.

Figure 5.9 demonstrates how a single-line pragma can be used to set the handshake interface for all 37 AXI ports of the handcrafted memory input loader RTL library:

5.3.3.3 Platform Analyzer

The platform analyzer interfaces with downstream vendor tools to collect essential information for design optimizations, such as resource utilization per module and timing information. This data is used for balancing resource allocation across device regions and making informed decisions about module placement and pipelining. With platform analyzers, RapidIR abstracts the interaction with low-level information from the FPGA development flow, simplifying design decisions, especially for software developers.

5.3.3.4 Design Exporter

The design exporter generates the final design output from the IR, ensuring compatibility with downstream EDA tools. Its key functions include:

1. Outputting unchanged leaf modules in their original source format.
2. Generating corresponding Verilog files for modified modules.
3. Creating constraint files for floorplanning guidance and other metadata.

This component enables transition from the high-level representation used in RapidIR to the concrete implementations required by FPGA synthesis and implementation tools.

The utility plugins abstract away many of the complexities associated with working directly with hardware description languages and vendor-specific tools, allowing developers to focus on their application logic while still benefiting from efficient FPGA implementations. The modular nature of these plugins also ensures that RapidIR can be easily extended to support new source formats, analysis techniques, or EDA tools as they emerge, future-proofing the framework and making it adaptable to evolving heterogeneous computing landscapes.

5.3.4 Composable Transformation Passes

RapidIR's composable transformation passes form the core of its optimization capabilities, enabling automated and sophisticated FPGA optimizations. These passes progressively gather data and incrementally refine the IR to optimize the design. Each pass is designed to focus on a specific aspect of the optimization process, ensuring robustness, maintainability, and extensibility.

To illustrate the functionality of these passes, we'll use a subset of the LLM accelerator example [CZD24] as shown in Figure 5.10 with the core passes of RapidIR applied.

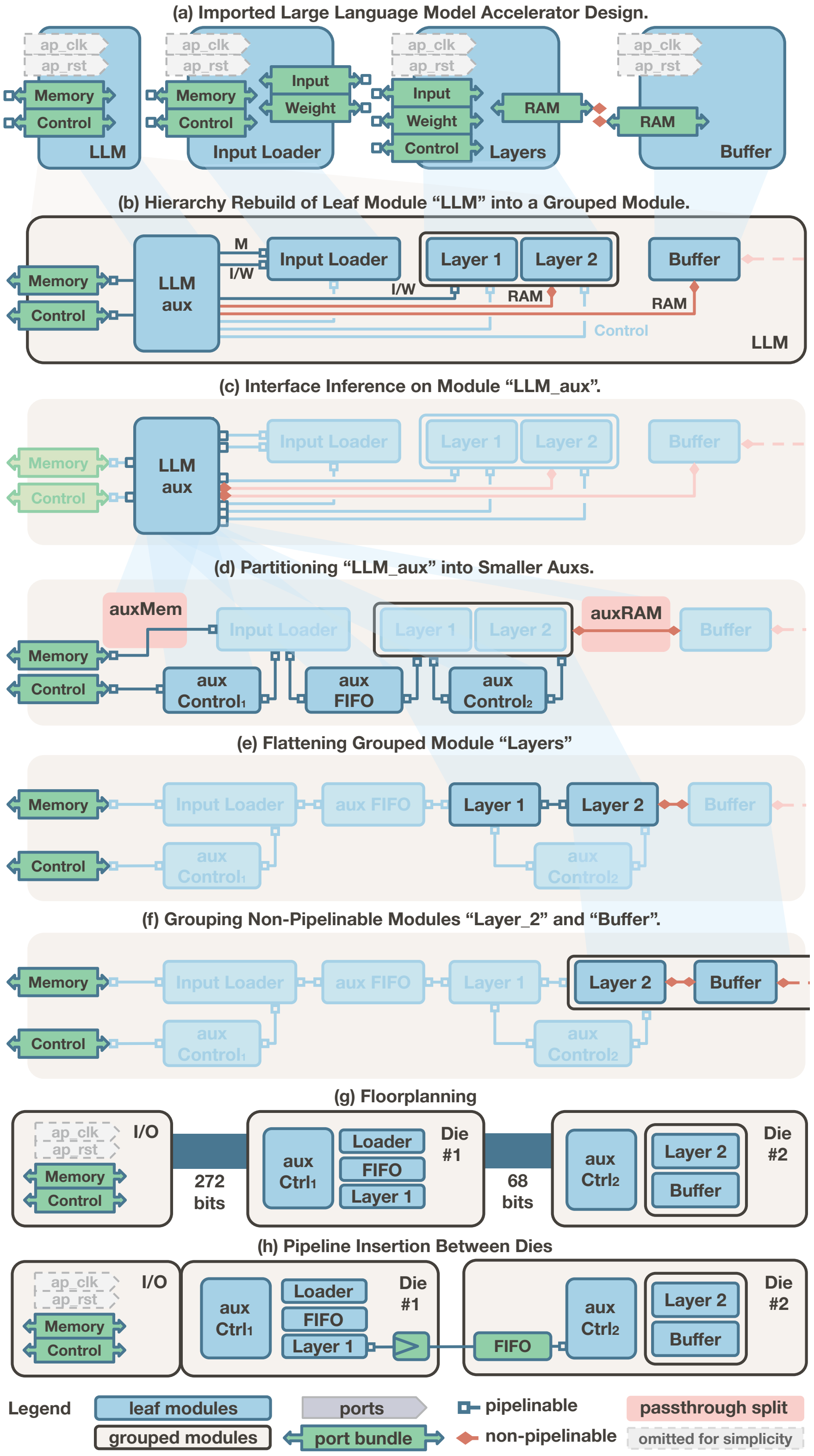


Figure 5.10: RapidIR's passes applied to the LLM accelerator example.

5.3.4.1 Hierarchy Rebuild Pass

The hierarchy rebuild pass converts imported leaf modules into grouped modules, reconstructing the design hierarchy. This pass is crucial for handling complex designs with multiple levels of hierarchy, including a mix of module containers and logic descriptions.

It creates a grouped module comprising extracted submodules and residual logic, which is defined as an **aux (auxiliary) module**. The reconstructed grouped module maintains the same module ports as the original leaf module. At this point in the process, this pass does *not* analyze the interconnection among submodules. Rather, it introduces corresponding ports on the aux module for every port of the original submodules. Each submodule has its ports connected to the aux module. Similarly, the newly formed grouped module's ports are fully connected to the aux module.

As shown in Figure 5.10b, the rebuild pass restructures the `LLM` module into a grouped module containing its submodules and an aux module, `LLM_Aux`. This aux module contains the control logic and interconnects of `LLM`. Note that directly analyzing `LLM`'s interconnect is challenging due to the complexity of its source format, which includes Verilog language syntax such as `always` and `generate`, requiring a full elaborator. Maintaining and updating such an elaborator for various design formats would be labor-intensive.

This transformation pass preserves the IR assumptions (§5.3.2.2):

1. In the newly formed grouped module, each wire is connected to exactly two modules: the aux module and a submodule.

2. (a) Ports of an extracted submodule are connected to a wire identifier that interconnects with the aux module.

   (b) Each port of the aux module is directly connected by a wire to an extracted submodule or to a port on the restructured grouped module; in either case, it is an identifier.

3. Every port on a submodule is wholly connected to the aux module, ensuring that there is no splitting of the interface.

The implementation of the hierarchy rebuild pass is straightforward for *any design source format* if there exists a syntax rewriter providing three functionalities: (1) extraction of submodule names and port connections; (2) addition of new ports to a module; and (3) connection of expressions to these new ports.

For example, the Slang [Pop24] tool allows for the extraction of Verilog submodule information; new ports are added by modifying the syntax tree; connections are rerouted by appending new `assign` statements. This approach is adaptable to other source formats by implementing a similar syntax rewriter. Note that even without a dedicated rewriter for a particular source format, RapidIR can still manage modules in this format by treating them as leaf modules; thus, it is still possible to insert pipeline stages between these modules or to partition them as needed.

5.3.4.2 Interface Inference Pass

When design modules lack explicit interface information necessary for pipeline insertion, the interface inference pass deduces interfaces from other modules. This pass is particularly useful for developers who may not have specified detailed interface information in their high-level descriptions due to their lack of hardware-specific knowledge. For instance, a user instantiating a grouped module for a reusable RTL library may have all ports directly connected to submodules, yet the module itself lacks interface data. By leveraging the interface details from these submodules, the interface inference pass can deduce the interface for the parent module.

Interface information propagates not only between parent and child modules but also among siblings. Specifically, for aux modules created during the hierarchy rebuild pass, the interface inferencer defines their interfaces by transferring information from the aux's sibling modules, the extracted submodules, completing the aux module's interface.

Figure 5.10c illustrates how this pass completes the interface information for the aux module by transferring data from its sibling modules. As this pass does not modify the design structure in the IR, the IR assumptions (§5.3.2.2) remain intact.

5.3.4.3 Partitioning Pass

The partitioning pass divides a leaf module into components for separate floorplanning, effectively serving as a communication analysis pass. This pass is crucial for dividing and optimizing the placement of design components across the FPGA fabric.

It partitions the aux module created by the hierarchy rebuild pass to decentralize submodule communications. Figure 5.10d shows the partitioning of the `LLM_aux` module, which initially connects all submodules. After partitioning, it is divided into five components, including memory connections (`auxMem` and `auxRAM`) and control logic ($\texttt{auxControl}_1$ and $\texttt{auxControl}_2$). In this example, the `LLM` module implements FIFO logic in the Verilog body, connecting the input loader and the first layer, which is partitioned into `auxFIFO`.

This pass is implemented by converting modules in *arbitrary formats* to netlists using EDA flows and applying the union-find algorithm [GF64] to merge logically connected nets, excluding clock and reset signals due to their shared use in submodules. From the unioned results, RapidIR analyzes port connectivity. Disjoint ports are separated into new modules. The partitioned modules are created by *wrapping* the original aux module, exposing only the necessary ports and preserving the internal logic. Unconnected logic remains undriven, which will be eliminated by subsequent EDA flows. The new components replace the original aux in the IR, and clock and reset signals are distributed to all submodules through dedicated broadcasting aux modules. Instead of modifying the logic content, the wrapping method makes it less prone to errors and remains language-agnostic.

It maintains IR assumptions (§5.3.2.2) by merging ports in a common interface into a union set, preventing the interface from spanning multiple splits. It introduces no new connections except for clock and reset signals, which are managed by broadcasting modules. This ensures that, post-transformation, all wires still connect exactly two modules, and all ports connect to either an identifier or a constant.

5.3.4.4 Passthrough Pass

The passthrough pass identifies and optimizes direct connections between interfaces, bypassing unnecessary intermediate modules. This pass can significantly simplify the design structure and reduce the number of modules, making the optimized design more readable and easier to optimize with other passes.

In order to implement this pass, if netlist analysis in the partitioning pass shows that an interface on a module connects solely and directly to another interface on the same module, the module can be bypassed by rerouting connections between the interfaces, eliminating the need for the intermediate partitioned component.

In Figure 5.10d, the `auxRAM` component is bypassed, allowing direct connections between the `Layer_2` and `Buffer` modules. This simplifies the IR and reduces the number of modules, making the design more readable and easier to optimize.

The passthrough pass maintains IR assumptions (§5.3.2.2) by detaching a wire from one module before connecting it to another.

5.3.4.5 Flattening Pass

The flattening pass transforms a hierarchical design into a flat one, which is often required for certain HLPS optimization formulations, such as integer linear programming (ILP) used in AutoBridge [GCW21]. This pass is essential for enabling global optimizations that may not otherwise be possible within a hierarchical structure.

The flattening pass performs the recursive merging of all grouped modules into a single one. During this process, wires are consolidated and renamed to avoid conflicts, and submodules and their connections are re-established in the new module.

Figure 5.10e shows the flattening pass incorporating the `Layer_1` and `Layer_2` submodules into the `LLM` module, allowing for more flexible partitioning of these resource-intensive components. Without this pass, these two modules would have to be grouped into a single partition, resulting in suboptimal partitioning.

This pass adheres to the IR assumptions (§5.3.2.2) by not introducing new interconnections between modules. It solely consolidates existing wires and submodules, thus preserving the original properties.

5.3.4.6 Wrapping Pass

The wrapping pass encapsulates a module within a template, adding helper submodules and connecting them to the wrapped module. This pass is particularly useful for implementing partitioning and adding pipeline stages.

It adds helper submodules and connects them to the wrapped module, further allowing the wrapper grouped module's ports to connect to either helpers or the wrapped module. This pass implements partitioning by exposing specific ports. It can also add pipeline stages as helper submodules. Typically, a flattening pass follows to elevate the helpers to the parent level, effectively *inserting* the helper modules.

5.3.4.7 Grouping Pass

The grouping pass restructures a flat design into a hierarchy, allowing for the creation of physical groupings that can be used to specify floorplanning constraints.

In Figure 5.10f, non-pipelinable submodules `Layer_2` and `Buffer` are grouped into a new module, so that it can have floorplanning constraints specified.

By providing these composable transformation passes, RapidIR enables software developers to call the passes to apply sophisticated FPGA optimization techniques without requiring in-depth knowledge of hardware design principles. The passes work together to progressively refine the design representation, automatically handling many of the complex transformations that would traditionally require hardware expertise.

5.3.5 Framework Integration

To demonstrate the practical application of RapidIR, we have developed a complete HLPS system that integrates our utility plugins and transformation passes. This integrated framework follows the HLPS methodology described in Section 5.2.3, showcasing RapidIR's applicability to real-world design scenarios.

Figure 5.10 illustrates the workflow of this integrated HLPS system, which consists of four main stages: communication analysis, design partitioning, coarse-grained floorplanning, and global interconnect synthesis. The numbered items below correspond to the subfigure numbers in Figure 5.10.

5.3.5.1 Communication Analysis

This stage captures coarse-grained communication patterns between modules, laying the foundation for subsequent optimizations. The process involves:

(a) Design and Interface Importers. Importing design and interface data into RapidIR IR using the leaf module importers of each source format and interface importers for HLS reports, XCI files, and user-specified pragmas.

(b) Hierarchy Rebuild Pass. Restructuring large modules into grouped modules by extracting their submodules and creating aux (auxiliary) modules.

(c) Interface Inference Pass. Inferring interfaces from the parent and siblings for aux modules (`LLM_aux`) and for modules lacking interface information.

(d) Partitioning Pass. Partitioning centralized broadcasting modules and applying pass-through to components with only wire assignments, especially for aux modules.

These steps allow the framework to automatically extract and analyze the underlying communication structures.

5.3.5.2 Design Partitioning

This stage focuses on partitioning the design into pipelinable sections based on the identified communication patterns. The key steps are:

(e) Flattening Pass. Converting the design into a flat representation for subsequent optimization formulations, such as integer linear programming (ILP), which requires a flat graph view of the design flow.

(f) Grouping Pass. Grouping non-pipelined modules with adjacent ones.

This partitioning process enables efficient mapping of the design onto the FPGA fabric using optimization techniques, a task that would typically require significant hardware design expertise to reconstruct in the source code format.

5.3.5.3 Coarse-Grained Floorplanning

Leveraging AutoBridge's integer linear programming (ILP) formulation [GCW21], this stage optimizes the placement of modules into predefined slots on a virtual device and designs a pipeline insertion scheme. The process:

(g) ILP Floorplanning. Optimizing module placement, aiming to minimize cross-region wiring and adhere to constraints such as DSP count and the number of boundary-crossing wires. It designs a pipeline insertion scheme based on the floorplan.

This stage abstracts away the complexities of FPGA floorplanning, allowing software developers to benefit from optimized physical layouts without needing to understand the physical structure of FPGAs.

5.3.5.4 Global Interconnect Synthesis

The final stage generates inter-partition connections based on estimated delay to break critical paths and aid in timing closure. The steps include:

(f) Grouping Pass. Clustering modules in the same region.

(h) Wrapping Pass. Inserting pipeline stages between regions using the reusable wrapping pass. This stage generates inter-partition connections based on estimated delays to break critical paths and aid in timing closure.

After this stage, the optimized design is exported for implementation using the design exporter utility plugin.

By integrating these stages, RapidIR provides a comprehensive HLPS system that automates many of the complex tasks involved in optimizing FPGA designs. This integration allows software developers to focus on their application logic while the framework handles the intricacies of mapping that logic to an efficient FPGA implementation.

The framework's modular design, centered around the progressively refined IR and composable passes, offers several key advantages for software developers:

Abstraction of Hardware Complexity. Developers can work with high-level descriptions while the framework automatically handles low-level optimizations.

Flexibility. The modular nature of the passes allows for easy customization and extension of the optimization process. Even without knowledge of hardware description languages or architecture, software developers can call and even modify optimization passes to explore different strategies.

Portability. By using a virtual device description and abstract IR, the framework facilitates porting designs across different FPGA architectures.

Incremental Optimization. The progressive refinement approach allows for step-by-step optimization, making the process more manageable and easier to debug.

In summary, the integration of RapidIR's components with AutoBridge's formulation [GCW21] into a complete HLPS system demonstrates its potential to significantly lower the barriers to entry for software developers in the field of heterogeneous computing. By automating complex FPGA optimization tasks and providing a flexible, extensible framework, RapidIR enables developers to leverage the power of FPGAs without requiring extensive hardware design expertise.

5.4 Evaluation

To assess RapidIR's effectiveness in enabling software developers to leverage heterogeneous computing resources, we conducted a comprehensive evaluation. Our experiments were designed to answer three key research questions, corresponding to the portable physical layout design challenges (PHY) faced by software developers targeting modern FPGA architectures, identified in Section 2.3.

RQ1 Can RapidIR effectively handle FPGA designs in various input formats, including handcrafted Verilog and HLS designs generated by different vendor tools?

RQ2 Does RapidIR reduce the effort required for developers to implement new research exploration tasks in FPGA design optimization?

RQ3 Can RapidIR provide significant frequency improvements for complex FPGA designs across different target devices?

Our evaluation was conducted using AMD FPGAs with the Vivado 2023.2 flow, on a server with an AMD EPYC 7282 CPU, 128 GB of RAM, and Ubuntu 22.04. For optimization tasks requiring integer linear programming (ILP), we used the COIN-OR solver [Sal02] with a 400-second limit. To test RapidIR's ability to handle diverse design formats, we also utilized Dynamatic 2.0, Catapult HLS 2021.1, and Intel FPGA HLS 19.4.0 to produce RTL inputs for RapidIR and checked the functionality of output design.

5.4.1 Support for Diverse High-Level Synthesis Inputs

To demonstrate RapidIR's flexibility in handling various input formats (**RQ1**), we extended its support to include RTL designs generated by different HLS tools. This capability is crucial for software developers who may be working with a variety of high-level synthesis tools or integrating components from different sources.

We focused on three HLS tools: Dynamatic [JGI18, JGI20], Catapult HLS [Sie24], and Intel HLS [Int24]. Prior HLPS work lacks the infrastructure to manipulate the RTL designs generated by these tools. To develop a new frontend in RapidIR that accepts the generated designs from these tools, three new components are needed: (1) a metadata parser, (2) an interface analyzer, and (3) a code rewriter.

The metadata parser and code rewriter are common to most input that use standard hardware description languages such as Verilog or VHDL. The interface analyzer, however, is specific to each HLS framework. We focus on the interface analyzer in this section, as the parser implementation was discussed in Section 5.3.3.1.

The total additional lines of Python or Verilog code required for RapidIR to handle inputs from these HLS tools are shown in Table 5.1.

Table 5.1: Code in Python or Verilog required to support different HLS tools in RapidIR.

Software	**Dynamatic**	**Catapult HLS**	**Intel HLS**
Lines of code	146	158	204

For Dynamatic, we used 20 Python-based interface rules for RapidIR interface importer to specify all its handshake interfaces. Figure 5.11 shows two of them: one specifies reset signals using the regular expression `".*"` to match and apply to all modules, and the other defines the handshakes of the top-level module.

```
add_reset(module=".*", port="rst|reset", active="high")
add_handshake(module=top_level, pattern="{bundle}_{role}",
      role={ready:"ready", valid:"valid", data:"in|out"})
```

Figure 5.11: Snippet of the interface rules for Dynamatic in RapidIR.

For Catapult HLS and Intel HLS, we used similar approaches, leveraging their consistent naming conventions and custom design libraries to infer handshake interfaces. Catapult HLS synthesizes handshakes using customizable design libraries such as `ccs_out_wait` and `ccs_in_wait`; with simple pragmas in these modules' Verilog code, the interface can be automatically propagated during the interface inference pass to neighboring modules. Intel HLS creates handshakes mostly with consistent port naming, making them also compatible with the Python-based interface rules method.

To validate our approach, we tested RapidIR with:

- All 29 examples from the Dynamatic repository [EPF24].

- A sparse linear algebra accelerator for Catapult HLS [Du24].
- All 12 benchmarks from the CHStone suite for Intel HLS [DLZ18].

RapidIR successfully extracted interface information, imported designs into its IR, transformed their hierarchy, inserted pipelines, and exported functionally equivalent RTL designs for all benchmarks.

This evaluation demonstrates RapidIR's ability to handle a wide range of input formats, enabling software developers to work with their preferred HLS tools while still benefiting from advanced FPGA optimizations.

5.4.2 Multi-Floorplan Exploration

To address **RQ2**, we evaluated RapidIR's ability to simplify complex FPGA design tasks, such as floorplan exploration. Floorplanning is a critical step in FPGA design optimization that typically requires deep hardware expertise. We used the LLM design [CZD24] as a case study to demonstrate how RapidIR can automate this process.

Figure 5.12 illustrates the relationship between resource distribution, wirelength, and frequency for ten different floorplans of the LLM design. This figure highlights the complex trade-offs involved in floorplanning. The line chart in the figure shows that decreasing the amount of logic in the most congested area of the floorplan reduces local congestion but potentially leads to longer wire lengths, which adversely affect global routing results, and vice versa. Additionally, the bar chart in the figure highlights the complexity of these tradeoffs, indicating a variation in the operating frequency of up to 20 MHz depending on the chosen tradeoff point between local and global optimization.

Without RapidIR, designers would need to manually explore the design space by refactoring the code to partition the design, restructuring the hierarchy as previously shown in Figure 5.1, modifying the floorplan constraints for the backend EDA tools, and

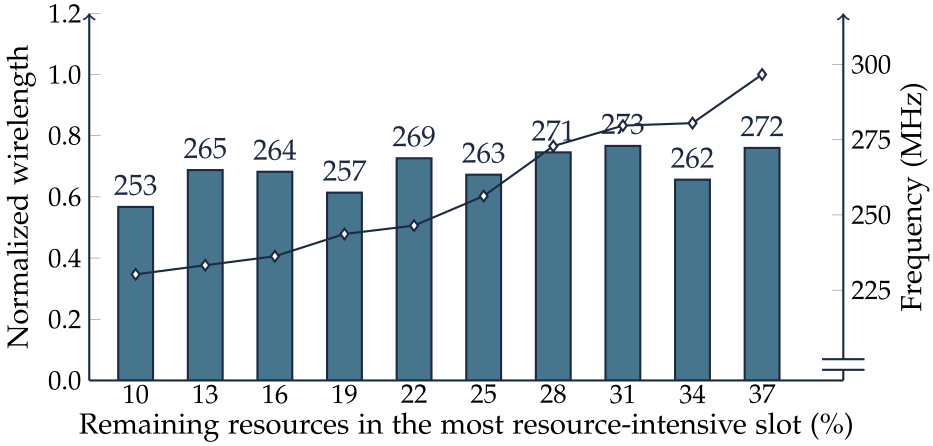


Figure 5.12: Relationship between resource distribution, wirelength, and frequency for the LLM design on VHK158.

re-executing the synthesis and place-and-route processes for multiple iterations.

As an evaluation of RapidIR's applicability, we applied our methodology to this floorplan exploration task. By adjusting the maximum allowable resource utilization for each slot using the virtual device model described in Section 5.3.2.2, RapidIR optimizes the wire length in placements given the constraints. In this way, RapidIR automatically explores the design space of tradeoffs and approximates Pareto optimality. This approach creates a variety of floorplans, as we have presented in Figure 5.12, allowing designers to evaluate the balance between wire length and resource distribution. This automation is implemented as a standalone RapidIR plugin, written in 207 lines of Python code, that can be reused across different designs. In contrast, manual exploration of this design alone would require a significant rewrite of the RTL code, consisting of hundreds of lines, potentially introducing errors and requiring numerous iterations.

This case study demonstrates how RapidIR can simplify the extension of high-level physical optimizations, such as the exploration of different floorplan schemes, thus significantly reducing the effort required for complex FPGA design tasks.

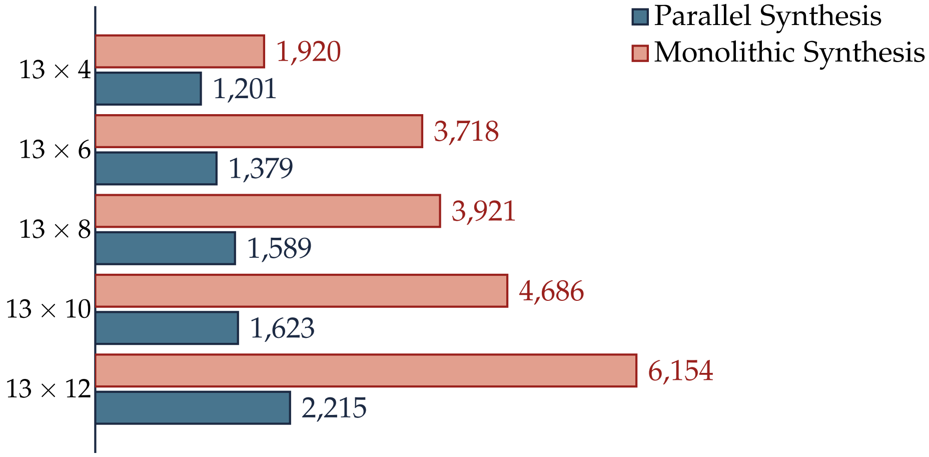


Figure 5.13: Synthesis wall time in seconds.

5.4.3 Parallel Synthesis

Another example of RapidIR's extensibility (**RQ2**) is its ability to enable parallel synthesis. We implemented a parallel synthesis plugin in 299 lines of Python code, leveraging RapidIR's design partitioning capabilities.

In RapidIR, we divide the design into several coarse-grained groups, each corresponding to a device slot. This approach intrinsically spawns the potential to perform parallel synthesis, where slots can be synthesized in parallel. The top-level module can be synthesized along with these slots by marking the slots as black boxes. Finally, we assemble these post-synthesis netlists to obtain the complete design.

We evaluate the RapidIR parallel synthesis plugin using HLS benchmarks of systolic array architectures for convolutional neural networks, which are generated using AutoSA [WGC21]. The evaluation is performed on the Alveo U250 FPGA by synthesizing the device slots in parallel, as shown in Figure 5.13. For systolic processing element arrays with sizes ranging from 13 × 4 to 13 × 12, the plugin achieves an average synthesis wall time acceleration of 2.49×.

This demonstrates RapidIR's potential to not only optimize FPGA designs but also to accelerate the development process itself, further enhancing productivity for software developers working with heterogeneous computing platforms.

5.4.4 Benchmarking

To comprehensively evaluate RapidIR's effectiveness in enabling software developers to achieve high-performance FPGA implementations (addressing **RQ3**), we conducted extensive benchmarking using a diverse set of real-world FPGA designs. These designs were selected from Section 2.3 and represent scenarios where physical layout optimizations for specific target FPGA devices are challenging in meeting timing requirements (PHY). The chosen designs encompass a range of applications and complexities that software developers might encounter when targeting heterogeneous computing platforms. In brackets, we provide the abbreviated names used in Section 2.3 for each benchmark.

We compared the frequency results obtained using RapidIR against those from AutoBridge [GCW21] and the standard EDA tool supplied by the FPGA vendor, AMD Vivado. This comparison allows us to assess RapidIR's performance relative to both state-of-the-art research tools and industry-standard solutions.

The benchmark designs used in our evaluation are as follows:

1. **Convolutional Neural Network (CNN)** refers to a neural network accelerator designed with AutoSA [WGC21] into a systolic array architecture in AMD Vitis HLS. It features a flat hierarchy, which is supported by AutoBridge. We use this benchmark to compare the frequency results between RapidIR and AutoBridge.

2. **LLaMA2 Language Model (LLM)** refers to a hybrid-source accelerator designed for large language model inference of the LLaMA2 model, initially optimized for the AMD Alveo U280 FPGA with a four-level nested pipeline using HLS, Xilinx IPs, and manual RTL [CZD24]. AutoBridge does not support it due to its complex hierarchical design. We further ported it to AMD Versal boards using RapidIR and compared the frequency performance with that of Vivado, demonstrating RapidIR's

adaptability for multi-source and multi-target designs.

3. **Minimap2 (GSQ)** refers to an accelerator for long-read genome sequencing with multiple hierarchical levels of pipelines, initially developed for AMD UltraScale+ VU9P using Vitis HLS [GLR19]. In the benchmarking, we retained the original hierarchical structure of Minimap2 and ported it to the AMD Versal VP1552 device, showcasing RapidIR's ability to automatically port designs to new architectures.

4. **K-Nearest Neighbor (KNN)** refers to a k-nearest neighbor accelerator for the Alveo U280 FPGA, using HLS kernels and a custom RTL interconnect, implemented on the Vitis platform [LFF20, LLS23]. RapidIR directly ingests the Vitis-packed Xilinx Object (XO) files for optimization and outputs the optimized design in the same format, acting as a transparent plugin to the Vitis framework.

RapidIR successfully ingests all designs and applies transformations using the HLPS methodology, as summarized in Table 5.2, where the design features, such as multiple levels of pipelined hierarchy, a mixture of different source formats, including RTL, HLS, and IP, and implementation targets for new FPGA devices, are listed in the "Benchmark Features" columns. These features are unsupported by existing HLPS frameworks. In the "Freq (MHz)" columns, the "Original" column shows the frequency of the original design before optimizations implemented using Vivado; the "RapidIR" column presents the frequency of designs optimized by RapidIR; and the "Other" column includes results from existing literature [GCW21, CZD24]. We indicate the frequency results for benchmarks that fail routing as "-". Regarding resource utilization, we report the percentages of LUTs, FFs, BRAMs, DSPs, and URAMs used in the original design. The change in resource utilization post-optimization is minimal, within 1%, across all benchmarks.

Key observations from our benchmarking results:

Consistent Performance Improvements. RapidIR achieved frequency improvements ranging from 30% to 62% across various designs and target devices. For the Convolu-

Table 5.2: Frequency improvements automated with RapidIR for various design formats on different FPGAs.

Application	Target	Benchmark Features			LUT	FF	BRAM	DSP	URAM		Freq (MHz)	
		Hierarchy	Mixed-Source	New FPGAs	(%)	(%)	(%)	(%)	(%)	Original	RapidIR	Others
CNN 13×4	U250				13	11	10	17	0	233	335 (+44%)	325 [GCW21]
CNN 13×6	U250				15	16	13	26	0	234	327 (+40%)	324 [GCW21]
CNN 13×8	U250				26	22	16	24	0	245	332 (+36%)	320 [GCW21]
CNN 13×10	U250				30	27	28	43	0	-	320 (+∞%)	322 [GCW21]
CNN 13×12	U250				27	33	30	51	0	-	305 (+∞%)	295 [GCW21]
LLM	VP1552	✓	✓	✓	32	16	13	22	18	198	258 (+30%)	N/A
LLM	VHK158	✓	✓	✓	32	16	13	22	18	206	273 (+33%)	N/A
LLM	U55C	✓	✓	✓	49	25	24	18	24	165	247 (+50%)	N/A
LLM	VU9P	✓	✓		59	32	23	24	24	141	212 (+50%)	N/A
LLM	U250	✓	✓		42	23	20	14	19	159	228 (+43%)	N/A
LLM	U280	✓	✓		49	25	24	18	25	150	243 (+62%)	245 [CZD24]
LLM (opt)	U280	✓	✓		35	19	15	18	25	201	306 (+52%)	245 [CZD24]
GSQ	VP1552	✓		✓	39	15	10	31	0	265	285 (+8%)	N/A
KNN	U280		✓		56	28	10	14	0	-	292 (+∞%)	N/A
Average Treating Unroutable Designs as Zeros					36	22	18	24	11	157	283 (+80%)	
Average Excluding Originally Unroutable Designs					36	20	16	21	14	200	277 (+39%)	

tional Neural Network (CNN) design on U250, RapidIR matched the performance of AutoBridge, demonstrating its competitiveness with state-of-the-art HLPS tools.

Handling Complex Designs. RapidIR successfully optimized the LLaMA2 Language Model (LLM) design, which AutoBridge couldn't support due to its complex hierarchy. On the U280, RapidIR achieved a frequency of 243 MHz for LLM, comparable to the 245 MHz achieved through manual optimization.

Portability Across Devices. RapidIR effectively ported the LLM design to different FPGA architectures (U280, VHK158, VP1552), demonstrating significant frequency improvements in each case. Additionally, The Minimap2 (GSQ) design was successfully ported from UltraScale+ VU9P to Versal VP1552, with a 8% improvement.

Resource Utilization. The change in resource utilization post-optimization was minimal across all benchmarks, indicating that RapidIR's improvements come primarily from better design physical layout rather than increased resource consumption.

Potential for Further Optimizations. For the LLaMA2 Language Model design, further refactoring the design into smaller components enabled by RapidIR ("LLM (opt)") boosted the frequency to 306 MHz on U280, showcasing the framework's potential for advanced optimizations when the developer has design-specific knowledge.

Flexibility in Design Inputs. RapidIR successfully handled designs from various sources (HLS, RTL, IP cores) and in different formats (including Vitis-packed XO files), demonstrating its versatility in real-world scenarios.

These benchmarking results highlight RapidIR's effectiveness in enabling software developers to achieve high-performance FPGA implementations across a range of designs and target devices. By automating complex optimization tasks and supporting diverse input formats, RapidIR significantly lowers the barrier to entry for software developers looking to leverage FPGAs in heterogeneous computing environments.

The framework's ability to match or exceed the performance of specialized tools like AutoBridge, while offering broader design support and eliminating the need for complete refactoring of the design, demonstrates its potential as a comprehensive solution for software developers targeting heterogeneous platforms. Furthermore, RapidIR's success in porting designs to new FPGA architectures showcases its value in maintaining code portability and performance across evolving hardware landscapes.

5.5 Conclusion

This chapter presented RapidIR, a comprehensive framework designed to bridge the gap between high-level software development and efficient FPGA physical implementation in heterogeneous computing environments.

RapidIR's main contributions to enabling heterogeneous computing include:

Flexible Intermediate Representation. RapidIR introduces a progressively refined IR that captures design information at various levels of abstraction, allowing for incremental optimization and analysis.

Support for Diverse Input Formats. The framework successfully handles designs from various sources, including handcrafted RTL, HLS-generated code from different vendors, and IP cores, providing flexibility for real-world development scenarios.

Automated Optimization. RapidIR implements a suite of composable transformation passes that automate complex FPGA optimization tasks, including hierarchy rebuilding, interface inference, and floorplanning.

Performance Improvements. Our evaluation demonstrated significant frequency improvements ranging from 30% to 62% across various designs and FPGA platforms, with some initially unroutable designs achieving around 300 MHz after optimization.

Extensibility. RapidIR's modular design allows for easy extension to support new optimization techniques, input formats, and target devices, as demonstrated by our case studies in floorplan exploration and parallel synthesis.

RapidIR's approach to FPGA optimization has several important implications for software developers targeting heterogeneous computing platforms:

Abstraction of Hardware Complexity. By automating many low-level physical optimization tasks, RapidIR allows developers to focus on high-level application logic while still benefiting from efficient FPGA implementations.

Improved Portability. The framework's ability to handle diverse input formats and target different FPGA architectures enhances code portability, a crucial factor in the rapidly evolving landscape of heterogeneous computing.

Enhanced Productivity. Automated optimization and exploration tools, such as the floorplan exploration case study, significantly reduce the time and effort required to achieve high-performance FPGA implementations.

While RapidIR represents a significant step forward in enabling heterogeneous computing for software developers, several promising research directions could further enhance its capabilities:

Automated NoC Synthesis. Extending RapidIR to support Network-on-Chip (NoC) synthesis for FPGA HLS designs could enable more efficient communication between modules in complex designs. This would involve automatically integrating routers between handshake modules, analyzing intra-node patterns for bandwidth requirements, and optimizing the network to minimize the impact on throughput. It would further enable software developers to utilize new interconnect components by automatically refactoring HLS-generated code or RTL library to use NoC.

Parallel Placement and Routing. Building on RapidIR's hierarchy flattening and reorganization capabilities, future work could explore parallel placement and routing techniques for FPGA designs. This could potentially leverage existing tools like RapidStream [GMZ22, GMZ23] while extending support to a broader range of HLS tools and design hierarchies. Challenges include interfacing with vendor tools or developing a custom placer and router using RapidWright [LK18].

Automated Design Instrumentation. Enhancing RapidIR to automatically insert performance counters and monitoring IPs could greatly aid in onboard design profiling and debugging. This capability would help developers identify performance bottlenecks and analyze runtime behaviors, further bridging the gap between software development practices and FPGA optimization.

Machine Learning-Driven Optimization. Incorporating machine learning techniques into RapidIR's optimization process could potentially lead to more efficient design space exploration and better-optimized FPGA implementations.

Integration with High-Level Programming Models. Further research could explore ways to integrate RapidIR with emerging high-level programming models for heterogeneous computing, such as OpenCL or SYCL, to provide a more seamless development experience for software developers.

In conclusion, RapidIR makes heterogeneous computing, particularly FPGA acceleration, more accessible to software developers. By automating complex optimization tasks, supporting diverse input formats, and achieving substantial performance improvements across multiple FPGA platforms, RapidIR empowers software developers to leverage the full potential of heterogeneous computing platforms without requiring extensive hardware-design expertise on each FPGA architecture.

CHAPTER 6

Discussion and Evaluation

The integration of HeteroRefactor (Chapter 3), Adroit (Chapter 4), and RapidIR (Chapter 5) results in an end-to-end optimization framework, Heterosys, that transforms software programs into hardware-efficient accelerator systems, thereby simplifying heterogeneous hardware development for software engineers. HeteroRefactor, serving as the frontend of Heterosys, converts dynamic software program functions into synthesizable hardware kernel designs with optimized resource utilization, alleviating the need for software developers to engage in complex program analysis and refactoring. Adroit further refines these kernel designs, optimizing them into architecture-aware equivalents with enhanced broadcast patterns, automatically addressing implicit performance issues that may elude even experienced FPGA experts. RapidIR completes Heterosys by composing multiple kernel designs into a cohesive system with an optimized floorplan, achieving high operating frequencies without manual intervention or code restructuring. Collectively, Heterosys bridges the gap between software development practices and hardware design, addressing challenges overlooked by previous research efforts.

This chapter examines a deceptively simple synthetic design that, despite its apparent simplicity, requires extensive hardware optimizations for efficient FPGA implementation. This minimal example design serves as a case study to discuss the collective functionality of Heterosys and to analyze the mechanisms underlying the observed performance improvements. Additionally, we present a case-study of the end-to-end performance when these approaches in Heterosys are applied in concert to a large language model

(LLM) accelerator, a complex real-world application identified in Chapter 2 as presenting numerous challenges to software engineers. Finally, we demonstrate the practical benefits of our Heterosys framework by applying it to accelerate a real-world genome sequencing application. This case study showcases how Heterosys empowers software developers to create highly efficient FPGA accelerators, even for complex bioinformatics tasks. Our results highlight the framework's ability to significantly improve performance in genomic data processing, illustrating its potential impact on computational biology and other data-intensive fields.

All evaluation of Heterosys in this chapter was conducted using AMD Alveo U50 FPGAs and Vivado 2023.2. The experiments were performed on a system equipped with an AMD EPYC 7282 CPU and 128 GB of RAM, running Ubuntu 22.04. For Integer Linear Programming (ILP) optimization tasks, we employed the COIN-OR solver [Sal02], setting a maximum runtime limit of 400 seconds per optimization instance.

6.1 Case Study 1: Synthetic Design

To delve deeper into the mechanisms underlying the observed resource efficiency and to elucidate the inner workings of Heterosys, we have devised a deceptively simple synthetic example. This case study serves to demonstrate how Heterosys transforms software code to achieve optimized results. In this section, we present a detailed analysis of the code transformations and their corresponding impacts, offering insights into the optimization process and its effects on hardware implementation.

6.1.1 Software Code

The following software code implements a simple synthetic data processing algorithm that performs a series of "folding" operations on an input stream. This "folding" process involves reading all data from the input stream and storing it locally, then writing half of the original data size to the output stream. The output values are computed by subtracting corresponding elements from the end of the buffer from those at the beginning. This process is applied to the data four times in succession, with each iteration's output serving as the input for the next, before the final result is written to the output stream.

```
void kernel1(hls::stream<long long> &input,
             hls::stream<long long> &output) {
  int size_of_input = input.read();

  long long *buffer =
    (long long *)malloc(size_of_input * sizeof(long long));
  for (int i = 0; i < size_of_input; i++)
    buffer[i] = input.read();

  output.write(size_of_input / 2);
  for (int i = 0; i < size_of_input / 2; i++)
    output.write(abs(buffer[i] - buffer[size_of_input - 1 - i]));

  free(buffer);
}

// ... same for kernel2, kernel3, kernel4

void system_top(hls::stream<long long> &input,
                hls::stream<long long> &output) {
  hls::stream<long long> a, b, c;
  kernel1(input, a);
  kernel2(a, b);
  kernel3(b, c);
  kernel4(c, output);
}
```

The presented software code, while seemingly straightforward, embodies several characteristics that make it challenging to implement efficiently in hardware.

Firstly, the utilization of `malloc` for dynamic memory allocation is incompatible with HLS, necessitating a refactoring into statically allocated arrays. Secondly, the indiscriminate use of standard types may lead to inefficiencies when the actual data range is more constrained. For instance, if all values are below 2^{40}, a more compact data type of ap_uint<40> could be employed instead of long long, reducing 37.5% resource needs. Furthermore, the buffer array with a large size is fragmented across multiple BRAM units dispersed throughout the FPGA. This fragmentation can result in extensive data broadcasting, which is inherently inefficient in hardware designs. Lastly, the overall system lacks proper floorplanning, with kernel placements on the FPGA device left unspecified. This absence of strategic component placement can lead to suboptimal pipelining between FPGA dies, ultimately compromising the system's performance and efficiency.

These factors collectively underscore the need for Heterosys to bridge the gap between software algorithms and their optimal hardware implementations.

6.1.2 Optimizations with HeteroRefactor

HeteroRefactor extracts dynamic invariants from software functions, such as kernel1 by executing the code with typical inputs. When most data values are below 2^{40} and the input size is less than 200,000, it automatically transforms the data types and memory allocation from the original code. In the original code, the input data and output data, as well as local variables, are all of type int or long long:

```
void kernel1(hls::stream<long long> &input,
             hls::stream<long long> &output) {
  int size_of_input = input.read();
  // ...
}
```

These data types are refactored based on their actual observed value ranges:

```
void kernel1(hls::stream<ap_uint<40>> &input,
             hls::stream<ap_uint<40>> &output) {
  ap_uint<18> size_of_input = input.read();
  // ...
}
```

The original code utilizes dynamic allocation malloc for the buffer array:

```
void kernel1(...) {
  buffer = (long long *)malloc(size_of_input * sizeof(long long));
  buffer[i] = input.read();
}
```

This will fail with the error "Undefined function malloc" when synthesized using Vitis HLS 2023.2. To make it synthesizable, HeteroRefactor translates this into the following code using a buddy memory allocation system (reformatted for readability):

```
union __alloc_list_ap_uint_40 {
  struct {
    ap_uint<18> prev;
    ap_uint<18> next;
  } _link;
  ap_uint<40> _data;
} __ap_uint_40[200001U];

ap_uint<18> __malloc_ap_uint_40(ap_uint<18> request) {
  // ... omitted for simplicity ...
}

void kernel1(...) {
  buffer = __malloc_ap_uint_40(size_of_input);
  __ap_uint_40[buffer + i]._data = input.read();
}
```

Here, the new function __malloc_ap_uint_40 implements the buddy memory allocation from the __ap_uint_40 array. Each element in this array is a union of either a linked list for the buddy system (_link), or the actual stored data (_data). With the allocated index buffer, memory access is translated to access on the __ap_uint_40 array.

HeteroRefactor's refactoring process optimizes data types and memory allocation, transforming the code into a form suitable for efficient hardware implementation while preserving its original functionality. Through these optimizations, HeteroRefactor translates the initially unsynthesizable program into a hardware kernel, with resource utilization optimized for efficient hardware deployment.

Without HeteroRefactor's automated refactoring, developers would need to manually translate memory allocation into appropriate memory systems and convert all pointer usages into array accesses. This process is not only tedious but also prone to errors. HeteroRefactor automates this transformation, ensuring the program becomes synthesizable with minimal human intervention. The dynamic analysis performed by HeteroRefactor is crucial for efficient resource usage. In the absence of such analysis, programmers might opt for overly conservative estimates. For example, without proper analysis, a developer might allocate an excessive buffer size of 1,000,000 for the `kernel1` buffer array and employ standard `long long` data types for all variables. Such an approach could result in a 2.8× BRAM resource requirements compared to what is available on the U50 FPGA, causing implementation failure due to resource constraints. In contrast, HeteroRefactor's optimizations reduce BRAM resource consumption to 64% while allowing the design to be successfully implemented and achieving a clock frequency of 191 MHz.

6.1.3 Optimizations with Adroit

While HeteroRefactor focuses on utilizing information gathered from dynamic analysis to refactor software code into a hardware-friendly form, Adroit further optimizes the hardware kernels using architecture-aware information specific to the FPGA fabric.

In the original code, access to the buffer becomes a data broadcast operation, as a single BRAM cannot hold all data in the array. Accessing an element in the array requires consulting multiple BRAMs distributed across the FPGA device. For example, in the code already refactored by HeteroRefactor:

```
__ap_uint_40[buffer + i]._data = input.read();
```

This access to `__ap_uint_40` becomes a broadcast operation. The data obtained from `input` is distributed to all BRAM units of `__ap_uint_40` in the same clock cycle. As this broadcast involves long wire connections, the clock latency must be increased to accommodate this operation, thus degrading the operating frequency.

Adroit refactors this code to insert an additional clock cycle with HLS_REG between the input data read operation and the distributed BRAM access, allowing the long latency broadcast operation to be broken into two clock cycles:

```
template<class T> T HLS_REG(T in){
#pragma HLS pipeline
#pragma HLS inline off
#pragma HLS interface port=return register
    return in;
}

__ap_uint_40[buffer + i]._data = HLS_REG(input.read());
```

Furthermore, Adroit addresses redundant synchronization signals in system_top from all four parallel kernels. Since kernel4 always completes last, the synchronization of four kernels in system_top can be simplified. In the original code:

```
void system_top(hls::stream<long long> &input,
                hls::stream<long long> &output) {
  hls::stream<long long> a, b, c;
  kernel1(input, a);
  kernel2(a, b);
  kernel3(b, c);
  kernel4(c, output);
}
```

The generated RTL code waits for all four kernels to complete before concluding the process of system_top. This generates a global control signal broadcast that requires signals from all four kernels, residing in multiple FPGA regions, to arrive at the control logic of system_top in one clock cycle:

```
assign ap_done = (
    kernel4_U0_ap_done & kernel3_U0_ap_done &
    kernel2_U0_ap_done & kernel1_U0_ap_done);
```

Adroit rewrites the control signal in the generated Verilog code so that the ap_done signal only depends on the ap_done signal of kernel4, avoiding the signal fanout:

```
assign ap_done = kernel4_U0_ap_done;
```

With a minimal area overhead of 787 LUTs and 934 FFs, which accounts for less than 0.1% of the available resources on the U50 FPGA, Adroit improves the operating frequency from 191 MHz to 242 MHz.

These optimizations demonstrate Adroit's capability to fine-tune hardware implementations beyond what is achievable through software-level refactoring alone. By addressing architecture-specific challenges such as data broadcasting and control signal optimization, Adroit complements HeteroRefactor's efforts, resulting in more efficient and higher-performing FPGA designs.

6.1.4 Optimizations with RapidIR

RapidIR enhances optimization for FPGA heterogeneous systems by partitioning and rearranging kernels to compose an efficient overall system. It takes the transformed RTL from Adroit and restructures the global hierarchy and interconnect using high-level physical synthesis methodologies to produce an optimized system.

In the RTL from Adroit, the system_top top-level module instantiates four child modules, with kernel1 occupying 77% BRAM on an AMD Alveo U50 FPGA die:

```
module system_top (
    input_r_dout, input_r_empty_n, input_r_read,
    output_r_din, output_r_full_n, output_r_write,
    ap_clk, ap_rst, ap_start, ap_done, ap_ready, ap_idle
);
// ... port and wire definitions omitted ...

// kernel1 occupies 38% BRAM of the device, 77% BRAM of an FPGA die
system_top_kernel1 kernel1_U0(
    // ... other ports omitted ...
    .input_r_dout(input_r_dout),
    .input_r_empty_n(input_r_empty_n),
    .input_r_read(kernel1_U0_input_r_read),
    .a_din(kernel1_U0_a_din),
    .a_full_n(a_full_n),
    .a_write(kernel1_U0_a_write)
);

system_top_kernel2 kernel2_U0(/* ... */);
system_top_kernel3 kernel3_U0(/* ... */);
system_top_kernel4 kernel4_U0(/* ... */);

endmodule // system_top
```

The intuitive approach of an expert RTL designer might involve manually assigning kernel1 to a dedicated FPGA die and distributing other kernels across the remaining die, with appropriate pipelining implemented between them. However, such specialized knowledge is often beyond the expertise of software engineers. Interestingly, this intuitive allocation proves suboptimal in this case. RapidIR's analysis reveals that a more efficient configuration involves co-locating kernel1 and kernel4 on the same die.

RapidIR automates this complex, device-specific optimization and exploration process. It begins by importing the RTL design into an intermediate representation, which serves as a foundation for subsequent analysis and optimizations:

```
- module_name: system_top
  module_ports:
  - { name: input_r_dout,       direction: in,      width: 40   }
  - { name: input_r_empty_n,    direction: in,      width: 1    }
  - { name: input_r_read,       direction: out,     width: 1    }
  - { name: output_r_din,       direction: out,     width: 40   }
  - { name: output_r_full_n,    direction: in,      width: 1    }
  - { name: output_r_write,     direction: out,     width: 1    }
                                                        # ...

  module_wires:
  - { name: kernel1_U0_input_r_read,                width: 1    }
  - { name: kernel1_U0_a_din,                       width: 40   }
  - { name: a_full_n,                               width: 1    }
  - { name: kernel1_U0_a_write,                     width: 1    }
                                                        # ...

  module_submodules:
  - instance_name: kernel1_U0
    module_name:   system_top_kernel1
    connections:
    - { port: input_r_dout,     value: input_r_dout             }
    - { port: input_r_empty_n,  value: input_r_empty_n          }
    - { port: input_r_read,     value: kernel1_U0_input_r_read  }
    - { port: a_din,            value: kernel1_U0_a_din         }
    - { port: a_full_n,         value: a_full_n                 }
    - { port: a_write,          value: kernel1_U0_a_write       }

  - instance_name: kernel2_U0                           # ...
  - instance_name: kernel3_U0                           # ...
  - instance_name: kernel4_U0                           # ...
```

Through a series of analysis and optimization steps, creates a dedicated floorplan for this specific design and device combination. It leverages information about the U50 FPGA architecture and the resource needs of each module to create an optimized layout. In this arrangement, RapidIR assigns `kernel1` and `kernel4` to the lower die of the FPGA, `SLOT_X0Y0`, while placing the other two kernels on the top die, `SLOT_X0Y1`:

```
- instance_name: kernel1_U0                             # ...
  instance_metadata:
    floorplan: "SLOT_X0Y0"

- instance_name: kernel2_U0                             # ...
  instance_metadata:
    floorplan: "SLOT_X0Y1"

- instance_name: kernel3_U0                             # ...
  instance_metadata:
    floorplan: "SLOT_X0Y1"

- instance_name: kernel4_U0                             # ...
```

```
  instance_metadata:
    floorplan: "SLOT_X0Y0"
```

RapidIR continues the optimization process by reorganizing the module instances into two distinct slot modules: SLOT_X0Y0 and SLOT_X0Y1. To address global critical paths, it introduces an additional module, Pipeline_SLOT_X0Y0_SLOT_X0Y1, which adds necessary pipeline stages between these design slots:

```
- module_name: system_top                               # ...
  module_submodules:
  - instance_name: SLOT_X0Y0_inst
    module_name:   SLOT_X0Y0                            # ...
  - instance_name: SLOT_X0Y1_inst
    module_name:   SLOT_X0Y1                            # ...
  - instance_name: Pipeline_SLOT_X0Y0_SLOT_X0Y1_inst
    module_name:   Pipeline_SLOT_X0Y0_SLOT_X0Y1         # ...

- module_name: SLOT_X0Y0                                # ...
  module_submodules:
  - instance_name: kernel1_U0
    module_name:   system_top_kernel1                   # ...
  - instance_name: kernel4_U0
    module_name:   system_top_kernel4                   # ...

- module_name: SLOT_X0Y1                                # ...
  module_submodules:
  - instance_name: kernel2_U0
    module_name:   system_top_kernel2                   # ...
  - instance_name: kernel3_U0
    module_name:   system_top_kernel3                   # ...
```

It generates constraint files and rebuilds RTL hierarchy as the final output of Heterosys:

```
module system_top (
    input_r_dout, input_r_empty_n, input_r_read,
    output_r_din, output_r_full_n, output_r_write,
    ap_clk, ap_rst, ap_start, ap_done, ap_ready, ap_idle
);
// ... port and wire definitions omitted ...
```

```
SLOT_X0Y0 SLOT_X0Y0_inst(
    // ... other ports omitted ...
    .input_r_dout(input_r_dout),        // ... `input`'s ports
    .output_r_din(output_r_din),        // ... `output`'s ports
    .kernel_1_a_din(kernel_1_a_din),    // ... `a`'s ports
    .kernel_4_c_dout(kernel_4_c_dout),  // ... `c`'s ports
);

SLOT_X0Y1 SLOT_X0Y1_inst(/* ... */);

Pipeline_SLOT_X0Y0_SLOT_X0Y1 Pipeline_SLOT_X0Y0_SLOT_X0Y1_inst(
    // ... other ports omitted ...
    .kernel_1_a_din(kernel_1_a_din),    // pipeline source of `a`
    .kernel_2_a_dout(kernel_2_a_dout),  // => pipeline sink of `a`
    .kernel_3_c_din(kernel_3_c_din),    // pipeline source of `c`
    .kernel_4_c_dout(kernel_4_c_dout),  // => pipeline sink of `c`
);

endmodule // system_top
```

RapidIR demonstrates substantial performance enhancements over the post-Adroit stage. With a negligible area overhead of less than 0.1% of the FPGA resources, it boosts the operating frequency from 248 MHz to 278 MHz, yielding a 15% performance improvement. This optimization represents the culmination of the Heterosys toolchain's efforts, achieving a remarkable 46% overall performance gain compared to the initial implementable version after HeteroRefactor's optimizations.

6.1.5 Results and Performance Analysis

Table 6.1 summarizes the improvements achieved at each stage of Heterosys.

Given that the original baseline design unsynthesizable, we developed a manually optimized baseline for comparative analysis. This modified baseline incorporates an expansive buffer size of 1,000,000 elements and utilizes standard 64-bit `long long` data types for input data, while employing 32-bit `int` types for all other variables. However, this approach's overly conservative estimation of input data size, coupled with the absence of data type representation optimizations, results in resource requirements that exceed the available capacity of the Alveo U50 FPGA. Consequently, the implementation fails during the placement phase of the synthesis process, highlighting the challenges associated with naive implementations on resource-constrained hardware platforms.

HeteroRefactor significantly improves resource utilization and allows the design to be synthesized. Table 6.1 shows that after applying HeteroRefactor, BRAM usage drops from 281.0% to 63.8% of available resources. This reduction allows the design to fit within the FPGA's resources, solving the placement failure seen in the hand-transformed baseline. LUT and FF usage remain low and stable, as this design doesn't involve much computation using these resources. The optimized design runs at a maximum frequency of 191 MHz, showing that HeteroRefactor enables successful implementation. These results demonstrate that HeteroRefactor effectively refactors code for FPGA synthesis automatically, greatly reducing resource needs while maintaining functionality.

Adroit further enhances the performance of the design with minimal resource overhead. As shown in Table 6.1, the post-Adroit implementation maintains similar resource utilization compared to the post-HeteroRefactor stage, with only a slight increase in LUT usage from 1.1% to 1.2%. The BRAM usage remains constant at 63.8%, preserving the significant reduction achieved by HeteroRefactor. Notably, Adroit substantially improves the maximum operating frequency from 191 MHz to 242 MHz, a 27% increase. This improvement is achieved with a negligible area overhead of 787 LUTs and 934 FFs, accounting for less than 0.1% of the available resources on the U50 FPGA. The minimal resource impact coupled with the significant frequency boost allows Adroit to contribute to the overall effectiveness of the Heterosys compilation framework.

The final stage of the Heterosys toolchain, RapidIR, further optimizes the design's performance while maintaining resource efficiency. The post-RapidIR implementation maintains the same LUT usage at 1.2% and slightly increases FF usage from 0.5% to

Table 6.1: Resource utilization and maximum operating frequency of the synthetic design across Heterosys optimization stages.

Heterosys Stage	LUT (%)	FF (%)	BRAM (%)	DSP (%)	URAM (%)	Frequency (MHz)
Software Baseline				Unsynthesizable		
Hand-Transformed Baseline	1.2	0.2	281.0	0.0	0.0	Failed to Place
Post-HeteroRefactor	1.1	0.5	63.8	0.0	0.0	191
Post-Adroit	1.2	0.5	63.8	0.0	0.0	242
Post-RapidIR	1.2	0.6	63.8	0.0	0.0	278

0.6% compared to the post-Adroit stage. In absolute terms, this translates to a modest increase of 76 LUTs (from 10,598 to 10,674) and 522 FFs (from 9,300 to 9,822). The BRAM utilization remains constant at 63.8%,. Most notably, RapidIR pushes the maximum operating frequency from 242 MHz to 278 MHz, representing a 14.9% improvement over the post-Adroit stage and a 45.5% increase from the post-HeteroRefactor frequency.

These results highlight the synergistic effects of the Heterosys toolchain, where each successive stage builds upon the previous optimizations, resulting in a design that balances resource efficiency with significantly enhanced performance.

6.2 Case Study 2: Large Language Model

This section presents an end-to-end evaluation of Heterosys optimizing large language models (LLMs) for FPGA acceleration [CZD24], demonstrating the transformation of a software-oriented baseline into an efficient hardware implementation through a series of automated optimization stages.

6.2.1 Benchmark Setup

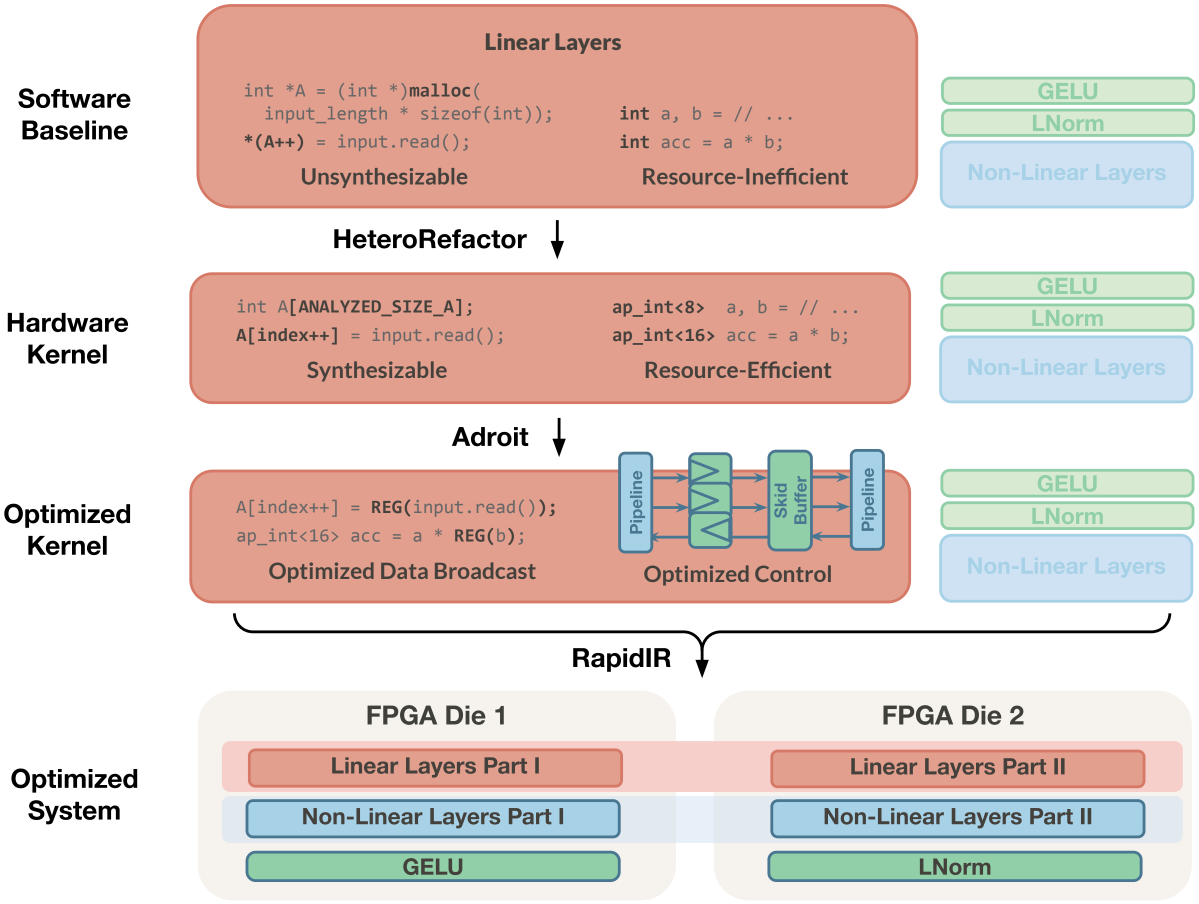
Chen et al. [CZD24] propose a complex hybrid-source accelerator for the inference phase of the LLaMA2 large language model, originally manually optimized for the AMD Alveo U280 FPGA, implementing a four-level nested pipeline. In this section, we revert their manual optimizations to a software-only version as a baseline that corresponds to what a software programmer would typically write and optimize for this specific task. Specifically, we revert the optimization of the input buffer with a static size to dynamic allocation, transform array access to the buffer back to pointer access, employ the original standard 32-bit int and float data types, disable broadcast-aware optimizations, and revert the FPGA-die based function partitioning to its logically natural form.

For the purpose of demonstration, we retain other optimizations that are outside the scope of this dissertation. For instance, the interface pragmas of the top-level function are preserved, and the unrolling and array partitioning directives remain intact. These optimizations could be achieved by other works in the literature, such as AutoDSE [SYG22], Merlin [Sol20], and AutoSA [WGC21]. In this evaluation, we assume that the input to Heterosys has already been optimized by these tools. In fact, the kernel code of the LLM accelerator partially reflects the methodology of AutoSA [WGC21].

6.2.2 Optimization Flow

Starting with the baseline software code, we applied a series of automated optimizations using Heterosys. First, we used HeteroRefactor to perform automated refactoring on the kernel code based on observed dynamic invariants. Next, we optimized broadcast patterns using architecture-aware approaches in Adroit. Finally, we utilized RapidIR for partitioning, floorplanning, and pipelining to integrate the kernels into an optimized system. Figure 6.1 illustrates the overall optimization flow.

In the initial stage, HeteroRefactor serves as the frontend of Heterosys, analyzing



```
int *A = (int *)malloc(
  input_length * sizeof(int));
*(A++) = input.read();
```

```
int a, b = // ...
int acc = a * b;
```

```
int A[ANALYZED_SIZE_A];
A[index++] = input.read();
```

```
ap_int<8>  a, b = // ...
ap_int<16> acc = a * b;
```

```
A[index++] = REG(input.read());
ap_int<16> acc = a * REG(b);
```

Figure 6.1: End-to-end Heterosys optimization flow for the LLM accelerator.

software function behavior to determine typical sizes of dynamically allocated arrays (e.g., A) and translating allocations (malloc) into static arrays (A[ANALYZED_SIZE_A]). Leveraging dynamic invariants obtained during the instrumentation phase, it further optimizes data representation, converting standard data types like int into bit-width optimized versions (e.g., ap_int<8>). This process renders the kernel both synthesizable and resource-efficient, making it suitable for hardware implementation.

Adroit subsequently processes the hardware kernel, analyzing broadcast patterns within the code. It improves large array access by introducing additional pipelines (e.g., inserting REG between read and A) and optimizes the broadcast of invariant variables to all loop iterations (e.g., REG(b)). Furthermore, it refines the control flow by transforming stall-based pipeline control broadcasts into skid-buffer-based free-flowing pipelines, thereby improving overall efficiency with a higher operating frequency.

In the final stage, RapidIR integrates the optimized kernels, partitioning them into smaller, more manageable units (such as multiple components of the Linear Layers) and strategically floorplanning these units across different FPGA regions. To mitigate global critical paths, it introduces pipeline stages between regions. As the culmination of Heterosys' optimization process, the original software baseline is transformed into a highly optimized FPGA heterogeneous computing system, demonstrating significant improvements in performance and resource utilization.

6.2.3 End-to-End Results

Table 6.2 presents a summary of the LLM design's resource utilization and maximum operating frequency at each stage of optimization. The resource requirements are expressed as percentages of the available resources on the AMD Alveo U50 FPGA. All reported results were obtained using Vitis' default target frequency of 300 MHz and standard settings. This data provides an overview of the impact of each stage on resource

Table 6.2: Resource utilization percentage and maximum operating frequency of the LLM FPGA accelerator after each Heterosys optimization stage.

	LUT	FF	BRAM	DSP	URAM	Frequency
Heterosys Stage	(%)	(%)	(%)	(%)	(%)	(MHz)
Software Baseline				Unsynthesizable		
Hand-Transformed Baseline	91.9	45.0	89.8	59.8	60.6	Failed to Route
Post-HeteroRefactor	76.8	39.7	45.2	28.1	37.2	157
Post-Adroit	76.9	39.6	45.2	28.1	37.2	198
Post-RapidIR	75.8	39.8	40.4	28.1	37.2	232

consumption and performance throughout the Heterosys optimization process.

The software baseline implementation, representing the initial software-oriented design, is not synthesizable due to unsupported program constructs such as dynamic memory allocation and pointer usage. To compare it with Heterosys-transformed versions, we manually transform these structures into synthesizable versions with known data bounds from Chen et al.'s optimized design. This hand-transformed baseline version exhibits high resource utilization across all categories, particularly in LUTs (92%) and BRAMs (90%). Notably, this excessive resource utilization, which nearly saturates the entire FPGA device, results in implementation failure due to routing congestion–a hardware-specific compilation error that falls outside the expertise of software developers. This outcome underscores the challenges inherent in transitioning from software to hardware implementations without specialized tools or knowledge.

Following the application of HeteroRefactor, a significant reduction in resource utilization is observed across multiple categories. Notably, LUT usage decreases from 92% to 77%, RAM consumption is reduced from 90% to 45%, and DSP utilization diminishes from 60% to 28%. This optimization phase not only alleviates resource constraints but also enables successful compilation of the design, achieving a maximum operational frequency of 157 MHz. These improvements can be attributed to HeteroRefactor's ability to optimize data representations and transform dynamic allocations into more resource-efficient static structures, thereby reducing the FPGA resource requirements.

The application of Adroit following HeteroRefactor leads to a slight increase in LUT usage and a minor decrease in FF utilization (to 76.8% and 39.6%, respectively). This modest change can be attributed to the addition of pipeline registers and control logic for optimizing broadcast patterns, as well as potential variations in the backend Vivado EDA flow. Notably, Adroit significantly improves the frequency to 198 MHz, an improvement that underscores the effectiveness of its architecture-aware optimizations.

In the final phase, RapidIR enhances the design's performance, reaching the target frequency of 232 MHz. This significant frequency improvement comes with only a small increase in FF resource usage. The minor rise in resource utilization is due to the extra register stages needed for inter-region pipelining. Interestingly, there is a decrease in BRAM usage after RapidIR, which can be explained by the replacement of HLS-generated FIFO with our optimized pipeline stages.

These results collectively demonstrate Heterosys' effectiveness in converting software-oriented designs into efficient hardware implementations. The framework successfully balances resource utilization and operating frequency, ultimately achieving an optimized frequency of 232 MHz while reducing overall resource consumption.

6.3 Case Study 3: Genome Sequencing

To further demonstrate the effectiveness of Heterosys in accelerating complex real-world applications, we present a case study of optimizing a genomic sequencing algorithm. Specifically, we focus on the chaining step of Minimap2 [Li18], a state-of-the-art tool for pairwise sequence alignment in genome sequencing. In this section, we explore hardware acceleration of long read pairwise sequence overlapping using FPGA, and optimize the hardware accelerator with our Heterosys framework.

Together with co-first author Licheng Guo, we proposed an algorithmic optimization approach at the 2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines [GLR19]. This corresponds to the optimized version presented in Figure 6.4. The same optimization methodology was subsequently applied to GPUs, resulting in significant accelerations. Heterosys further optimized it for FPGA-specific features, thereby reducing resource requirements and increasing operating frequency from the high-level optimizations.

6.3.1 Background and Challenges

Minimap2 [Li18] is a genomics tool that excels in speed and quality of results, targeting versatile tasks including pairwise overlapping of long reads in genome sequencing. We focus on the bottleneck in the tool, chaining, which performs dynamic programming to find series of feature matches that have consistent distances on the two reads. When two reads have a long series, i.e., a long chain, it implies that they have continuous matches and may share the same sub-sequence from the whole genome.

We summarize the chaining algorithm detailed in Li's paper [Li18] as follows: a match between read a and b can be represented by a 3-tuple (x, y, l), describing an exact match of length l: "$a_{x-l+1}..a_x$" = "$b_{y-l+1}..b_y$". All matches are assumed to be sorted first by x and then by y. The chaining score $f(t)$ of match t is calculated by evaluating its 64 previous matches s using the following formula:

$$f(t) = \max\{\max_{s \prec t}\{f(s) - \beta(s,t) + \alpha(s,t)\}, 0 + l(t)\}$$

In this formula, $l(t)$ represents the length of match t, $\beta(s,t)$ quantifies the inconsistency cost in location between matches s and t, and $\alpha(s,t)$ is the contribution of match t to the

Input: *match*[]: ordered list of matches between a pair of reads, represented as a 3-tuple (x, y, l), describing an exact match of length l: “$a_{x-l+1}..a_x$” $=$ “$b_{y-l+1}..b_y$” in the genome sequence read results. α and β: weight functions defined in [Li18], used in computing the transition function weight $w[j][i]$ from match j to match i.

Output: π[]: the predecessors and f[]: the chaining scores.

```
for $i = 1$ to $n$ do
    for $j \in [\max(0, i-64), i-1]$ do
        $w[j][i] \leftarrow (-\beta + \alpha)(match[i], match[j])$
    end for
    $f[i] \leftarrow \max_j\{f[j] + w[i][j]\}$
    $\pi[i] \leftarrow \arg\max_j\{f[j] + w[i][j]\}$
    if $f[i] < match[i].l$ then
        $f[i] = match[i].l; \quad \pi[i] = \emptyset$
    end if
end for
```

Figure 6.2: Original chaining algorithm for the long read pairwise sequence overlapping.

total matching length if it is added to a chain ending at match s. The skeleton of the original chaining algorithm is presented in Figure 6.2.

An illustration of the 3-tuple *match* is provided in Figure 6.3, which contains seven matches. The tuple (x, y, l) corresponding to $match_5$ is highlighted in red.

This step is a one-dimensional dynamic programming algorithm that presents several challenges for hardware acceleration:

1. **Poor parallelizability:** The original computation pattern involves dependencies that limit parallelism, which is essential for hardware acceleration.

2. **Data reuse:** The algorithm exhibits a high degree of data reuse, requiring careful

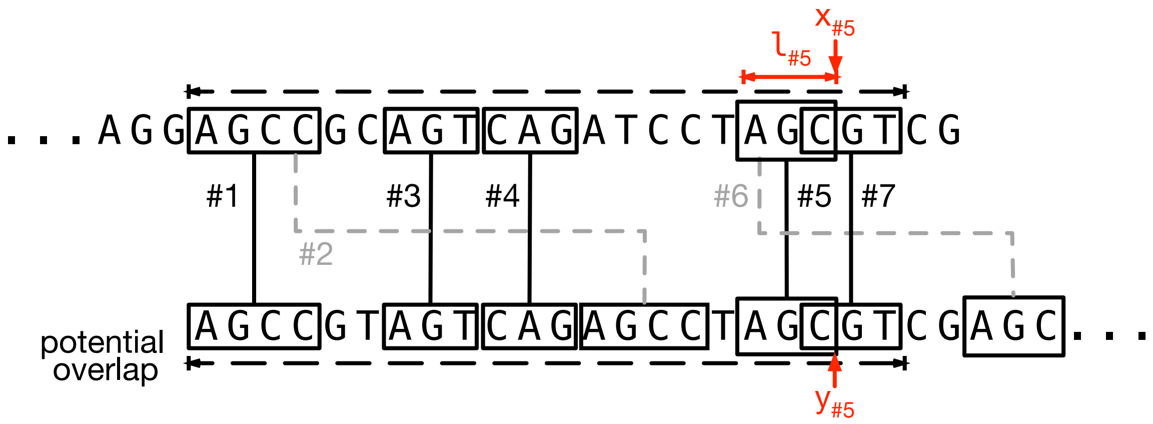

x#5
l#5
...AGGAGCCGCAGTCAGATCCTAGCGTCG
#1
#2
#3
#4
#6
#5
#7
potential overlap
AGCCGTAGTCAGAGCCTAGCGTCGAGC...
y#5

Figure 6.3: Example of the *match* array in a pair of genome reads.

consideration of data placement and movement in the hardware design.

3. **Variable input sizes:** The large and irregular sizes of input data make it difficult to leverage task-level parallelism effectively.

6.3.2 Parallelization Approach

In [GLR19], we propose several key optimizations to address the challenges in implementing hardware acceleration for the genome sequencing task:

1. **Intra-task parallelism.** The algorithm was transformed to a hardware-friendly form that enables full parallelization of the inner updating loop and reduces pipeline initiation interval. By distributing the reduction operations across iterations, the dependency chain was shortened, allowing for more efficient pipelining. This transformation reduced the initiation interval from the latency of 7 comparison operations to just one comparison.

2. **Data reuse.** Metadata, scores, and predecessors of matches $i + 1$ to $i + 64$ are stored in local memory to dramatically reduce memory bandwidth requirements. This

```
Input: $match[]$: ordered list of matches between a pair of reads as in Algorithm 6.2.
Output: $\pi[]$: the predecessors and $f[]$: the chaining scores.
Initialize all $f[i] \leftarrow match[i].l$, and $\pi[i] \leftarrow \emptyset$ in parallel.
for (pipeline) $i = 1$ to $n$ do
    for (parallel) $k \in [i+1, \min(n, i+64)]$ do
        $w[k][i] \leftarrow (-\beta + \alpha)(match[i], match[k])$
        if $f[k] < f[i] + w[k][i]$ then
            $f[k] = f[i] + w[k][i]; \pi[k] = i$
        end if
    end for
end for
```

Figure 6.4: Optimized chaining for the long read pairwise sequence overlapping [GLR19].

optimization reduced the theoretical bandwidth requirement from 7345 Gbps to just 112 Gbps, making it feasible for implementation on current hardware. The local storage forms a sliding window that enables efficient parallel access to the data.

3. **Task-level parallelism.** Multiple levels of parallelism are leveraged by concatenating tasks from different read pairs and separating them with runtime tags. This approach addresses the challenges of limited fast storage and the inefficiency of processing very small tasks. By combining multiple tasks, the design can better utilize hardware resources and amortize overhead across multiple read pairs.

The optimized pseudocode is shown in Figure 6.4. With parallelization approaches familiar to high-performance software developers and existing HLS tools, the code is implemented into FPGA accelerators. The implementation of Figure 6.4 could be found at `https://github.com/UCLA-VAST/minimap2-acceleration`.

6.3.3 Application of Heterosys Framework

The Heterosys framework was applied to the optimized Minimap2 chaining FPGA kernel, Algorithm 6.4, implemented in Vitis HLS, to enhance performance and resource utilization of the FPGA implementation. The process comprised the following stages:

1. **HeteroRefactor:** This tool was employed to analyze the optimized algorithm implementation and execute automated refactoring. The focus was on optimizing data representations through integer bitwidth reduction and floating-point precision adjustment. For example, the length of genome sequence reads typically does not exceed 30,000 base pairs, so the location information x and y of *match* could be shorten from 32 bits into 15-bit integers.

2. **Adroit:** This component was utilized to further refine the hardware-friendly code generated by HeteroRefactor. The emphasis was on optimizing broadcast patterns and implementing control signal optimizations tailored to the FPGA architecture.

3. **RapidIR:** Finally, this tool was used to optimize the overall system architecture, encompassing partitioning, floorplanning, and inter-region pipelining strategies.

6.3.4 Results and Analysis

The performance and resource utilization of the parallelized Minimap2 chaining algorithm on FPGA across various Heterosys optimization stages are presented in Table 6.3.

Resource Utilization. HeteroRefactor significantly reduced resource utilization across all categories. LUT usage decreased from 90.2% to 42.1%, FF usage from 46.4% to 24.6%, and DSP usage from 26.6% to 9.3%. This substantial reduction in resource usage enables potential scaling of the design to process multiple set of data in parallel, or integration of additional functionalities, such as other parts in the overall genome sequencing pipeline.

Table 6.3: Resource utilization and frequency of the parallelized Minimap2 chaining algorithm on FPGA across Heterosys optimization stages.

Heterosys Stage	LUT (%)	FF (%)	BRAM (%)	DSP (%)	URAM (%)	Frequency (MHz)
Optimized Baseline (Algo. 6.4)	90.2	46.4	16.3	26.6	0.6	169.9
Post-HeteroRefactor	42.1	24.6	15.0	9.3	0.6	243.7
Post-Adroit	42.2	24.6	15.0	9.3	0.6	262.0
Post-RapidIR	42.3	24.8	15.0	9.3	0.6	279.4

Frequency Improvement. The baseline implementation of the optimized algorithm achieved a frequency of only 169.9 MHz due to its excessive resource usages. After applying HeteroRefactor, the frequency increased to 243.7 MHz, a 43.4% improvement. Adroit further increased the frequency to 262.0 MHz, resulting in a 54.2% improvement compared to the baseline. The final RapidIR stage achieved a frequency of 279.4 MHz, representing a total improvement of 64.4% over the optimized baseline.

Optimization Overhead. The post-Adroit and post-RapidIR stages show only slight increases in resource usage compared to the post-HeteroRefactor stage. LUT usage increased marginally from 42.1% to 42.2% (Adroit) and 42.3% (RapidIR), while FF usage remained stable at 24.6% for Adroit and increased slightly to 24.8% for RapidIR. This demonstrates that Adroit and RapidIR optimizations introduce minimal resource overhead while providing significant frequency improvements.

6.3.5 Discussion

The case study results demonstrate Heterosys' effectiveness in optimizing complex genomic algorithms for FPGA acceleration. Key findings include:

- **Automated Optimization:** Heterosys successfully automated many optimizations previously implemented manually, potentially reducing development time and expertise requirements for efficient FPGA accelerators.
- **Performance and Efficiency:** The significant increase in operating frequency and dramatic reduction in resource utilization suggest superior performance compared to the original optimized designs. This efficiency allows for potential scaling to larger datasets or implementation of additional functionalities.

These findings indicate that Heterosys can significantly enhance the development process and performance of FPGA accelerators for genomic algorithms, potentially lowering barriers to entry for software developers to implement FPGA accelerators for data processing and other applications.

6.4 Conclusion

This chapter has presented an end-to-end evaluation of Heterosys, demonstrating its effectiveness in bridging the gap between software development practices and efficient hardware design for FPGA acceleration. Through a synthetic design example, a real-world case study of a Large Language Model (LLM) accelerator, and optimization over a highly optimized genome sequencing accelerator, we have illustrated the synergistic effects of HeteroRefactor, Adroit, and RapidIR in transforming software-oriented code into optimized hardware implementations.

The synthetic design case study highlighted the ability of Heterosys to address challenges that are often overlooked in software-to-hardware translations. HeteroRefactor's automated refactoring significantly reduced resource utilization, enabling successful synthesis where a naive implementation failed. Adroit's architecture-aware optimizations further improved performance with minimal resource overhead, increasing the maximum operating frequency by 27%. RapidIR's system-level optimizations pushed the performance even further, achieving a 46% overall frequency improvement compared to the post-HeteroRefactor stage.

In the more complex LLM accelerator case study, Heterosys demonstrated its capability to handle real-world applications. Starting from an unsynthesizable software baseline, each stage of Heterosys contributed to substantial improvements. HeteroRefactor's optimizations led to significant reductions in resource utilization across all categories, enabling successful compilation. Adroit's optimizations further enhanced the operating frequency, while RapidIR's system-wide strategy brought the design to 232 MHz.

The case study on genome sequencing further validates Heterosys' effectiveness in optimizing complex real-world applications for FPGA acceleration. By focusing on the chaining step of long read genome sequencing, we demonstrated how Heterosys can significantly reduced resource utilization while improving operating frequency. HeteroRefactor's data representation optimizations led to substantial reductions in LUT, FF, and DSP usage, while Adroit and RapidIR further increased the operating frequency with minimal resource overhead. The final implementation achieved a 64.4% frequency improvement over the baseline, with dramatically reduced resource utilization.

These results underscore the power of Heterosys in automating the optimization process for FPGA acceleration. By addressing challenges at multiple levels – from code refactoring to architecture-aware optimizations and system-level integration – Heterosys enables software engineers to leverage FPGA acceleration without requiring deep hardware design expertise. The framework's ability to balance resource utilization and performance optimization demonstrates its potential to significantly streamline the development of efficient FPGA-based accelerators for complex applications.

In conclusion, Heterosys represents a significant step forward in democratizing FPGA acceleration, making it more accessible to software developers and potentially accelerating the adoption of FPGA technology across a broader range of applications.

CHAPTER 7

Conclusion and Future Work

7.1 Dissertation Summary

This dissertation has presented a set of methodologies and tools aimed at bridging the gap between high-level software development and efficient utilization of heterogeneous computing resources, particularly FPGA acceleration. The primary goal of this work was to make heterogeneous computing more accessible to a broader range of software developers by abstracting away much of the hardware complexity while still enabling developers to leverage the performance benefits of specialized hardware effectively.

The key contributions of this dissertation can be summarized as follows:

HeteroRefactor: Dynamic Analysis and Automated Refactoring. HeteroRefactor is an automated refactoring tool that combines dynamic invariant analysis and intelligent code transformations to make software code FPGA-compatible and hardware-friendly. HeteroRefactor significantly reduces the manual effort required to port software to FPGAs, achieving up to 83% reduction in resource usage and 42% increase in operating frequency compared to manual implementations.

Adroit: Architecture-Driven Optimization for Implicit Broadcasts. Adroit is a novel approach to identifying and optimizing implicit broadcast structures in HLS-generated designs. Adroit's broadcast-aware scheduling, synchronization logic pruning, and skid-buffer-based pipeline control techniques achieve an average frequency improvement of 53% across a range of real-world benchmarks, while incurring minimal area overhead.

RapidIR: Infrastructure for High-Level Physical Synthesis Optimizations. RapidIR is a comprehensive infrastructure for high-level physical synthesis optimizations. It integrates coarse-grained floorplanning with behavior-level pipelining, supporting hierarchical composition of heterogeneous designs from diverse sources. RapidIR automates the exploration of various physical optimization strategies, freeing programmers from designing device-specific hardware layouts for each target device. It achieved frequency improvements ranging from 30% to over 62% compared to state-of-the-art EDA tools and enabled some previously unimplementable designs to reach a frequency of around 300 MHz.

These contributions collectively form Heterosys, an end-to-end optimization framework that simplifies heterogeneous hardware development by decoupling algorithmic descriptions from underlying fabrics and offering layout-driven and architecture-driven design generation. The research presented in this dissertation has demonstrated substantial performance improvements across diverse applications and benchmarks, including genomic sequencing and large language model acceleration.

The methodologies and tools presented in this dissertation have significant implications for enabling software developers to leverage heterogeneous computing resources:

Improved Programmability. HeteroRefactor and RapidIR significantly enhance the accessibility of FPGA technology for software developers by abstracting complex hardware details. HeteroRefactor supports the inclusion of recursive and dynamic programming patterns in regions that are not performance-limiting, while RapidIR allows developers to maintain their program's hierarchical structure without the need to convert the code into a single-level representation or translate it into a specialized HLS language prior to optimization for physical synthesis. This abstraction facilitates the implementation of efficient FPGA designs without necessitating deep hardware knowledge, thereby broadening the pool of developers who can effectively utilize FPGAs for various applications.

Automated Performance Optimization. HeteroRefactor, Adroit, and RapidIR automate complex performance optimization tasks, such as resource optimization, broadcast optimization, and physical synthesis, enabling software developers to achieve high-performance FPGA implementations without manual intervention. HeteroRefactor leverages dynamic analysis to identify performance issues and automatically refactors the code to improve performance, while Adroit statically optimizes implicit broadcast structures in kernel codes to enhance the frequency of HLS-generated designs. RapidIR automates the exploration of physical synthesis optimizations, such as floorplanning and pipelining, to compose heterogeneous design components from diverse sources, enabling software developers to achieve high-performance FPGA implementations with minimal effort. These tools collectively reduce the time and effort required to achieve high-performance FPGA implementations, making it easier for software developers to leverage FPGA acceleration.

Enhanced Portability. All three tools enhance the portability of FPGA designs by enabling developers to maintain code that is independent of the target device and fabric. HeteroRefactor automatically refactors software code to make it FPGA-compatible, enabling developers to write code that can be easily ported to FPGA devices. Adroit optimizes implicit broadcast structures in HLS-generated designs, reducing the gap between software expectations and hardware realities. RapidIR automates the exploration of physical synthesis optimizations, enabling developers to write device-independent designs without physical layout information that can be easily ported to different target devices. These tools collectively enhance the portability of FPGA designs, enabling developers to write code that can be easily deployed on different FPGA devices without manual intervention.

By addressing these key challenges, our work contributes to the broader goal of democratizing heterogeneous computing and making FPGA acceleration more accessible and practical for software developers.

7.2 Future Research Directions

This dissertation has made significant contributions to enabling heterogeneous computing for software developers. However, several promising directions for future research could further enhance the accessibility, performance, and applicability of the proposed methodologies and tools:

Integration with High-Level Programming Models. One potential direction is to explore ways to integrate our tools with emerging high-level programming models for heterogeneous computing, such as PyTorch [PGM19], SYCL [Khr21] or MLIR [LAB21], which could provide a more seamless development experience for software developers. Integration with MLIR-based solutions, such as ScaleHLS [YHC21], Allo [CZX24], and MLIR-AIE [Xil22b], will enable users to seamlessly enhance the optimized outputs from these frameworks using Heterosys efficiently.

Automated Design Space Exploration. Incorporating machine learning techniques into the optimization process could potentially lead to more efficient design space exploration and better-optimized FPGA implementations. This could involve learning from previous optimization results and adapting to new design patterns and FPGA architectures. When applied to HeteroRefactor and RapidIR, the remaining manual or heuristic optimization strategies could be replaced. For instance, ActiveCEM [DSL24] enables the application of existing optimization databases to emerging techniques, while AutoDSE [SYG22] could be extended to optimize architecture- and physical-aware features of FPGA designs.

Extending to Other Heterogeneous Platforms. While this dissertation primarily focused on FPGA acceleration, many of the techniques and principles could be adapted to other heterogeneous platforms, such as GPUs or domain-specific accelerators like AI Engines. An example extension to the AI Engine includes CHARM [ZLY23, ZLY24], which leverages FPGA acceleration techniques and integrates FPGA components into AI engine-centric accelerators. Investigating the applicability of our methodologies to these platforms could further broaden the impact of our work.

Automated Design Instrumentation and Debugging. Enhancing our tools to automatically insert performance counters and monitoring IPs could greatly aid in onboard design profiling and debugging. This capability would help developers identify performance bottlenecks and analyze runtime behaviors, further bridging the gap between software development practices and hardware optimization. Examples include HeteroFuzz [ZWK21], which tests high-level descriptions of FPGA designs to detect synthesis divergence, and HeteroGen [ZWX22], which automatically generates tests and repairs programs. These tools can be enhanced with RapidIR to incorporate production-environment components into the testing process, providing additional insights.

Collaborative Optimization Framework. Developing a collaborative optimization framework, akin to MLIR [LAB21] but extended to incorporate more hardware-specific features, or HeteroCL [LCH19] with enhanced component integration, would facilitate the sharing and reuse of optimization strategies, thereby accelerating the adoption of heterogeneous computing. Such a framework could incorporate version control, performance tracking, and a library of reusable optimization passes, fostering a community-driven approach to heterogeneous computing.

These future research directions aim to further lower the barriers to entry for heterogeneous computing, making it more accessible, efficient, and user-friendly for software developers. By continuing to narrow the gap between software development and hardware optimization, we can unlock the full potential of heterogeneous computing and drive innovation across a wide range of applications and domains.

7.3 Concluding Remarks

The rise of heterogeneous computing presents both opportunities and challenges for software developers. While the potential for performance and efficiency gains is significant, the complexity of leveraging these specialized hardware resources has hindered widespread adoption. This dissertation has presented a comprehensive set of methodologies and tools designed to address these challenges and enable software developers to effectively utilize heterogeneous computing resources, particularly FPGA acceleration.

Through HeteroRefactor, Adroit, and RapidIR, we have demonstrated the potential for automated analysis, refactoring, and optimization techniques to bridge the gap between high-level software development and efficient hardware implementation. By abstracting away much of the hardware complexity and automating key optimization tasks, our work empowers software developers to leverage the power of FPGAs without requiring extensive hardware expertise.

As we look to the future, it is clear that heterogeneous computing will continue to play an increasingly important role in meeting the computational demands of the 21st century. By continuing to develop and refine methodologies and tools that enable software developers to effectively leverage these resources, we can unlock the full potential of heterogeneous computing and drive the next generation of technological advancements across a wide range of applications and domains, from scientific computing and machine learning to data analytics and beyond.

In conclusion, this dissertation represents a step forward in enabling heterogeneous computing for software developers. By providing a set of methodologies and tools that bridge the gap between software development and hardware optimization, our work paves the way for more widespread adoption of heterogeneous acceleration in software development practices. As we continue to build upon these foundations and explore new research directions, we move closer to a future where the power of heterogeneous computing is accessible to all developers, regardless of their hardware expertise.

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.

[ABC16] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. "TensorFlow: a system for large-scale machine learning." In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, pp. 265–283, 2016.

[AC75] Alfred V Aho and Margaret J Corasick. "Efficient string matching: an aid to bibliographic search." *Communications of the ACM*, **18**(6):333–340, 1975.

[Adv24a] Advanced Micro Devices, Inc. *Versal Adaptive SoC Technical Reference Manual*, 4 2024.

[Adv24b] Advanced Micro Devices, Inc. *Vivado Design Suite User Guide: Designing with IP*, 5 2024.

[ADZ24] Afzal Ahmad, Linfeng Du, and Wei Zhang. " Fast and Practical Strassen's Matrix Multiplication using FPGAs ." In *2024 34th International Conference on Field-Programmable Logic and Applications (FPL)*, pp. 311–317, Los Alamitos, CA, USA, September 2024. IEEE Computer Society.

[AP14] Andrew V Adinetz and Dirk Pleiter. "Halloc: a high-throughput dynamic memory allocator for GPGPU architectures." In *GPU Technology Conference (GTC)*, volume 152, 2014.

[ASB19] Sagheer Ahmad, Sridhar Subramanian, Vamsi Boppana, Shankar Lakka, Fu-Hing Ho, Tomai Knopp, Juanjo Noguera, Gaurav Singh, and Ralph Wittig. "Xilinx First 7nm Device: Versal AI Core (VC1902)." In *2019 IEEE Hot Chips 31 Symposium (HCS)*, pp. 1–28, 2019.

[BBK08] Uday Bondhugula, Muthu Baskaran, Sriram Krishnamoorthy, Jagannathan Ramanujam, Atanas Rountev, and Ponnuswamy Sadayappan. "Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model." In *International Conference on Compiler Construction*, pp. 132–146. Springer, 2008.

[BCd09] Julien Boucaron, Anthony Coadou, and Robert de Simone. "Latency-Insensitive Design: Retry Relay-Station and Fusion Shell." In *Proceedings of the 4th International Workshop on the Application of Formal Methods for Globally*

Asynchronous and Locally Synchronous Design (FMGALS '09), volume 245, pp. 23–33, 2009.

[BEI09] Oren Ben-Kiki, Clark Evans, and Brian Ingerson. "Yaml ain't markup language (yaml™) version 1.1." *Working Draft 2008*, **5**(11), 2009.

[BK08] Gary Bradski and Adrian Kaehler. *Learning OpenCV: Computer vision with the OpenCV library*. O'Reilly Media, Inc., 2008.

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

[BRS17] Pierre Bourhis, Juan L. Reutter, Fernando Suárez, and Domagoj Vrgoč. "JSON: Data model, Query languages and Schema specification." In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, PODS '17, p. 123–135, New York, NY, USA, 2017. Association for Computing Machinery.

[BSW23] Suhail Basalama, Atefeh Sohrabizadeh, Jie Wang, Licheng Guo, and Jason Cong. "FlexCNN: An End-to-end Framework for Composing CNN Accelerators on FPGA." *ACM Trans. Reconfigurable Technol. Syst.*, **16**(2), March 2023.

[BVR12] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. "Chisel: constructing hardware in a Scala embedded language." In *Proceedings of the 49th Annual Design Automation Conference*, DAC '12, p. 1216–1225, New York, NY, USA, 2012. Association for Computing Machinery.

[Car20] Luca P Carloni. "Scalable Open-Source System-on-Chip Design." In *Proceedings of the IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*, 2020.

[CBB17] Wei-Fan Chiang, Mark Baranowski, Ian Briggs, Alexey Solovyev, Ganesh Gopalakrishnan, and Zvonimir Rakamariundefined. "Rigorous Floating-Point Mixed-Precision Tuning." In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, p. 300–315, New York, NY, USA, 2017. Association for Computing Machinery.

[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.

[CCG13] Yu-Ting Chen, Jason Cong, Mohammad Ali Ghodrat, Muhuan Huang, Chunyue Liu, Bingjun Xiao, and Yi Zou. "Accelerator-rich CMPs: From concept to real hardware.” In *2013 IEEE 31st International Conference on Computer Design (ICCD)*, pp. 169–176, 2013.

[CCL23] Young-Kyu Choi, Yuze Chi, Jason Lau, and Jason Cong. “TARO: Automatic Optimization for Free-Running Kernels in FPGA High-Level Synthesis.” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, **42**(7):2423–2427, 2023.

[CCP16] Adrian M Caulfield, Eric S Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, et al. “A cloud-scale acceleration architecture.” In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, p. 7. IEEE Press, 2016.

[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.

[CFH18] Jason Cong, Zhenman Fang, Yuchen Hao, Peng Wei, Cody Hao Yu, Chen Zhang, and Peipei Zhou. “Best-Effort FPGA Programming: A Few Steps Can Go a Long Way.” *arXiv preprint arXiv:1807.01340*, 2018.

[CGC22] Yuze Chi, Licheng Guo, and Jason Cong. “Accelerating SSSP for Power-Law Graphs.” In *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA ’22, p. 190–200, New York, NY, USA, 2022. Association for Computing Machinery.

[CGG14] Jason Cong, Mohammad Ali Ghodrat, Michael Gill, Beayna Grigorian, Karthik Gururaj, and Glenn Reinman. “Accelerator-rich architectures: Opportunities and progresses.” In *Proceedings of the 51st Annual Design Automation Conference*, pp. 1–6. ACM, 2014.

[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 *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 210–2104. IEEE, 2018.

[CGL09] Jason Cong, Karthik Gururaj, Bin Liu, Chunyue Liu, Zhiru Zhang, Sheng Zhou, and Yi Zou. “Evaluation of static analysis techniques for fixed-point precision optimization.” In *2009 17th IEEE Symposium on Field Programmable Custom Computing Machines*, pp. 231–234. IEEE, 2009.

[CGL21] Yuze Chi, Licheng Guo, Jason Lau, Young-kyu Choi, Jie Wang, and Jason Cong. “Extending High-Level Synthesis for Task-Parallel Programs.” In *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 204–213, 2021.

[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.

[CHK21] Anthony Cabrera, Seth Hitefield, Jungwon Kim, Seyong Lee, Narasinga Rao Miniskar, and Jeffrey S Vetter. "Toward Performance Portable Programming for Heterogeneous Systems on a Chip: A Case Study with Qualcomm Snapdragon SoC." In *2021 IEEE High Performance Extreme Computing Conference (HPEC)*, pp. 1–7. IEEE, 2021.

[CHP16a] Jason Cong, Muhuan Huang, Peichen Pan, Yuxin Wang, and Peng Zhang. "Source-to-source optimization for HLS." In *FPGAs for Software Programmers*, pp. 137–163. Springer, 2016.

[CHP16b] Jason Cong, Muhuan Huang, Peichen Pan, Di Wu, and Peng Zhang. "Software infrastructure for enabling FPGA-based accelerations in data centers." In *Proceedings of the 2016 International Symposium on Low Power Electronics and Design*, pp. 154–155. ACM, 2016.

[CLL12] Jason Cong, Bin Liu, Guojie Luo, and Raghu Prabhakar. "Towards layout-friendly high-level synthesis." In *ISPD '12*, 2012.

[CLL22] Jason Cong, Jason Lau, Gai Liu, Stephen Neuendorffer, Peichen Pan, Kees Vissers, and Zhiru Zhang. "FPGA HLS Today: Successes, Challenges, and Opportunities." *ACM Transactions on Reconfigurable Technology and Systems*, **15**(4), Aug 2022.

[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.

[CMH10] Eric S Chung, Peter A Milder, James C Hoe, and Ken Mai. "Single-chip heterogeneous computing: Does the future include custom logic, FPGAs, and GPGPUs?" In *2010 43rd annual IEEE/ACM international symposium on microarchitecture*, pp. 225–236. IEEE, 2010.

[CMJ18] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. "TVM: An automated end-to-end optimizing compiler for deep learning." In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pp. 578–594, 2018.

[Con01] J. Cong. "An interconnect-centric design flow for nanometer technologies." *Proceedings of the IEEE*, **89**(4):505–528, 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.

[CSG11] Andrew A Chien, Allan Snavely, and Mark Gahagan. "10x10: A general-purpose architectural approach to heterogeneity and energy efficiency." *Procedia Computer Science*, **4**:1987–1996, 2011.

[CSR10] Jason Cong, Vivek Sarkar, Glenn Reinman, and Alex Bui. "Customizable domain-specific computing." *IEEE Design & Test of Computers*, **28**(2):6–15, 2010.

[CW18] Jason Cong and Jie Wang. "PolySA: polyhedral-based systolic array auto-compilation." In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 1–8. IEEE, 2018.

[CWY18a] Jason Cong, Peng Wei, Cody Hao Yu, and Peng Zhang. "Automated accelerator generation and optimization with composable, parallel and pipeline architecture." In *Proceedings of the 55th Annual Design Automation Conference (DAC)*, pp. 1–6. IEEE, 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.

[CZD24] Hongzheng Chen, Jiahao Zhang, Yixiao Du, Shaojie Xiang, Zichao Yue, Niansong Zhang, Yaohui Cai, and Zhiru Zhang. "Understanding the Potential of FPGA-Based Spatial Acceleration for Large Language Model Inference." *ACM Transactions on Reconfigurable Technology and Systems*, Apr 2024.

[CZX24] Hongzheng Chen, Niansong Zhang, Shaojie Xiang, Zhichen Zeng, Mengjia Dai, and Zhiru Zhang. "Allo: A Programming Model for Composable Accelerator Design." *Proc. ACM Program. Lang.*, **8**(PLDI), June 2024.

[De 59] Rene De La Briandais. "File searching using variable length keys." In *Papers presented at the the March 3-5, 1959, Western Joint Computer Conference*, pp. 295–298. ACM, 1959.

[DKR18] Michael Ditty, Ashish Karandikar, and David Reed. "NVIDIA's Xavier SoC." In *Hot chips: a symposium on high performance chips*, 2018.

[DLS23] Linfeng Du, Tingyuan Liang, Sharad Sinha, Zhiyao Xie, and Wei Zhang. "FADO: Floorplan-Aware Directive Optimization for High-Level Synthesis Designs on Multi-Die FPGAs." In *Proceedings of the 2023 ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '23)*, 2023.

[DLZ18] Steve Dai, Gai Liu, and Zhiru Zhang. "A Scalable Approach to Exact Resource-Constrained Scheduling Based on a Joint SDC and SAT Formulation." In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '18)*, 2018.

[DLZ24] Linfeng Du, Tingyuan Liang, Xiaofeng Zhou, Jinming Ge, Shangkun Li, Sharad Sinha, Jieru Zhao, Zhiyao Xie, and Wei Zhang. "FADO: Floorplan-Aware Directive Optimization Based on Synthesis and Analytical Models for High-Level Synthesis Designs on Multi-Die FPGAs." *ACM Trans. Reconfigurable Technol. Syst.*, **17**(3), September 2024.

[DSL24] Zijian Ding, Atefeh Sohrabizadeh, Weikai Li, Zongyue Qin, Yizhou Sun, and Jason Cong. "Efficient Task Transfer for HLS DSE.", 2024.

[Du24] Yixiao Du. "Cornell University: Building Sparse Linear Algebra Accelerators with HLS." Webinar, April 2024.

[DXX24] Fan Dang, Yifan Xu, Rongwu Xu, Xinlei Chen, and Yunhao Liu. "LSync: A Universal Timeline-Synchronizing Solution for Live Streaming." *IEEE/ACM Transactions on Networking*, **32**(5):4144–4159, 2024.

[EPF24] EPFL Processor Architecture Laboratory. "DHLS (Dynamic High-Level Synthesis) Compiler Based on MLIR.", 04 2024.

[EPG07] Michael D Ernst, Jeff H Perkins, Philip J Guo, Stephen McCamant, Carlos Pacheco, Matthew S Tschantz, and Chen Xiao. "The Daikon system for dynamic detection of likely invariants." *Science of computer programming*, **69**(1-3):35–45, 2007.

[Fei12] Tom Feist. "Vivado design suite." *White Paper*, **5**:30, 2012.

[FKY15] Koichi Fujiwara, Kazushi Kawamura, Masao Yanagisawa, and Nozomu Togawa. "Clock skew estimate modeling for FPGA high-level synthesis and its application." In *ASICON '15*, 2015.

[FKY16] Koichi Fujiwara, Kazushi Kawamura, Masao Yanagisawa, and Nozomu Togawa. "A high-level synthesis algorithm for FPGA designs optimizing critical path with interconnection-delay and clock-skew consideration." In *VLSI-DAT '16*, 2016.

[GA13] Marcel Gort and Jason H Anderson. "Range and bitmask analysis for hardware optimization in high-level synthesis." In *2013 18th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 773–779. IEEE, 2013.

[GCL23] Licheng Guo, Yuze Chi, Jason Lau, Linghao Song, Xingyu Tian, Moazin Khatti, Weikang Qiao, Jie Wang, Ecenur Ustun, Zhenman Fang, Zhiru Zhang, and Jason Cong. "TAPA: A Scalable Task-parallel Dataflow Programming Framework for Modern FPGAs with Co-optimization of HLS and Physical Design." *ACM Transactions on Reconfigurable Technology and Systems*, **16**(4), Dec 2023.

[GCW21] 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 *Proceedings of the 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '21)*, 2021.

[GF64] Bernard A. Galler and Michael J. Fisher. "An improved equivalence algorithm." *Communications of the ACM*, **7**(5):301–303, May 1964.

[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.

[GKS05] Patrice Godefroid, Nils Klarlund, and Koushik Sen. "DART: Directed Automated Random Testing." In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pp. 213–223, New York, NY, USA, 2005. ACM.

[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 *Proceedings of the 2020 57th ACM/IEEE Design Automation Conference (DAC '20)*, 2020.

[GLM08] Patrice Godefroid, Michael Y. Levin, and David A Molnar. "Automated Whitebox Fuzz Testing." In *Network Distributed Security Symposium (NDSS)*. Internet Society, 2008.

[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 *Proceedings of the 2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM '19)*, 2019.

[GLW20] Licheng Guo, Jason Lau, Jie Wang, Cody Hao Yu, Yuze Chi, Zhe Chen, Zhiru Zhang, and Jason Cong. "Analysis and Optimization of the Implicit Broadcasts in FPGA HLS to Improve Maximum Frequency." In *Proceedings of the 28th ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*. ACM, 2020.

[GMN23] Chang Gao, Zhengyang Ming, Kim-Lien Nguyen, Xiaodong Zhong, and John Paul Finn. "Undersampling reconstruction of ferumoxytol-enhanced cardiac cine MRI using a spatiotemporal neural network." In *Proeedings of the 31st Annual Meeting of ISMRM*, p. 391, 2023.

[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 (FPGA '22)*, 2022.

[GMZ23] Licheng Guo, Pongstorn Maidee, Yun Zhou, Chris Lavin, Eddie Hung, Wuxi Li, Jason Lau, Weikang Qiao, Yuze Chi, Linghao Song, Yuanlong Xiao, Alireza Kaviani, Zhiru Zhang, and Jason Cong. "RapidStream 2.0: Automated Parallel Implementation of Latency–Insensitive FPGA Designs Through Partial Reconfiguration." *ACM Trans. Reconfigurable Technol. Syst.*, **16**(4), September 2023.

[GSF22] Chang Gao, Shu-Fu Shih, J Paul Finn, and Xiaodong Zhong. "A projection-based k-space transformer network for undersampled radial mri reconstruction with limited training subjects." In *International Conference on Medical Image Computing and Computer-Assisted Intervention*, pp. 726–736. Springer, 2022.

[GZK21] Abraham Gonzalez, Jerry Zhao, Ben Korpan, Hasan Genc, Colin Schmidt, John Wright, Ayan Biswas, Alon Amid, Farhana Sheikh, Anton Sorokin, et al. "A 16mm 2 106.1 GOPS/W Heterogeneous RISC-V Multi-Core Multi-Accelerator SoC in Low-Power 22nm FinFET." In *ESSCIRC 2021-IEEE 47th European Solid State Circuits Conference (ESSCIRC)*, pp. 259–262. IEEE, 2021.

[HJJ96] Steven Huss-Lederman, Elaine M Jacobson, Jeremy R Johnson, Anna Tsao, and Thomas Turnbull. "Implementation of Strassen's algorithm for matrix multiplication." In *Supercomputing'96: Proceedings of the 1996 ACM/IEEE Conference on Supercomputing*, pp. 32–32. IEEE, 1996.

[HKP84] H James Hoover, Maria M Klawe, and Nicholas Pippenger. "Bounding fan-out in logical networks." *Univ. of Toronto*, 1984.

[HO09] Hwa-You Hsu and Alessandro Orso. "MINTS: A General Framework and Tool for Supporting Test-suite Minimization." In *Proceedings of the 31st International*

Conference on Software Engineering, ICSE '09, pp. 419–429, Washington, DC, USA, 2009. IEEE Computer Society.

[Hoe94] Wassily Hoeffding. "Probability inequalities for sums of bounded random variables." In *The Collected Works of Wassily Hoeffding*, pp. 409–426. Springer, 1994.

[HRJ10] Xiaohuang Huang, Christopher I Rodrigues, Stephen Jones, Ian Buck, and Wen-mei Hwu. "Xmalloc: A scalable lock-free dynamic memory allocator for many-core machines." In *2010 10th IEEE International Conference on Computer and Information Technology*, pp. 1134–1139. IEEE, 2010.

[Int19] Intel. *Intel Hyperflex Architecture High-Performance Design Handbook*. Intel, 2019.

[Int24] Intel. "Intel High Level Synthesis Compiler." `https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/hls-compiler.html`, 2024.

[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 (FPGA '18)*, 2018.

[JGI20] Lana Josipović, Andrea Guerrieri, and Paolo Ienne. "Invited Tutorial: Dynamatic: From C/C++ to Dynamically Scheduled Circuits." In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '20)*, 2020.

[JKH22] EunJin Jeong, Jangryul Kim, and Soonhoi Ha. "TensorRT-based framework and optimization methodology for deep learning inference on jetson boards." *ACM Transactions on Embedded Computing Systems (TECS)*, **21**(5):1–26, 2022.

[JLD22] Xinqi Jin, Lingkun Li, Fan Dang, Xinlei Chen, and Yunhao Liu. "A survey on edge computing for wearable technology." *Digital Signal Processing*, **125**:103146, 2022. Sensing, Signal Processing and Computing for the Era of Wearables.

[JSO20] JSON Schema. "JSON Schema: A Media Type for Describing JSON Documents." `https://json-schema.org/specification.html`, 2020. Draft 2020-12.

[KA13] Ana Klimovic and Jason H Anderson. "Bitwidth-optimized hardware accelerators with software fallback." In *2013 International Conference on Field-Programmable Technology (FPT)*, pp. 136–143. IEEE, 2013.

[Kar24] Andrej Karpathy. "llama2.c: Inference LLaMA 2 in one file of pure C.", 2024.

[KEG01] Yoshio Kataoka, Michael D. Ernst, William G. Griswold, and David Notkin. "Automated support for program refactoring using invariants." In *ICSM 2001, Proceedings of the International Conference on Software Maintenance*, pp. 736–743, Florence, Italy, November 6–10, 2001.

[Khr21] Khronos. "SYCL 2020 Specification revision 3." `https://www.khronos.org/registry/SYCL/specs/sycl-2020/pdf/sycl-2020.pdf`, 2021.

[KLT13] Sen M Kuo, Bob H Lee, and Wenshun Tian. *Real-time digital signal processing: fundamentals, implementations and applications*. John Wiley & Sons, 2013.

[KPZ16] David Koeplinger, Raghu Prabhakar, Yaqi Zhang, Christina Delimitrou, Christos Kozyrakis, and Kunle Olukotun. "Automatic generation of efficient accelerators for reconfigurable hardware." In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pp. 115–127. Ieee, 2016.

[KSR18] Didier Keymeulen, Simon Shin, Jason Riddley, Matthew Klimesh, Aaron Kiely, Elliott Liggett, Peter Sullivan, Michael Bernas, Hamid Ghossemi, Greg Flesch, et al. "High performance space computing with system-on-chip instrument avionics for space-based next generation imaging spectrometers (NGIS)." In *2018 NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, pp. 33–36. IEEE, 2018.

[KTC23] Moazin Khatti, Xingyu Tian, Yuze Chi, Licheng Guo, Jason Cong, and Zhenman Fang. "PASTA: Programming and Automation Support for Scalable Task-Parallel HLS Programs on Modern Multi-Die FPGAs." In *Proceedings of the 2023 IEEE 31st Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM '23)*, 2023.

[LAB21] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. "MLIR: Scaling Compiler Infrastructure for Domain Specific Computation." In *Proceedings of the 2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO '21)*, 2021.

[LCH19] Yi-Hsiang Lai, Yuze Chi, Yuwei Hu, Jie Wang, Cody Hao Yu, Yuan Zhou, Jason Cong, and Zhiru Zhang. "HeteroCL: A multi-paradigm programming infrastructure for software-defined reconfigurable computing." In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 242–251, 2019.

[LCN16] Xinheng Liu, Yao Chen, Tan Nguyen, Swathi Gurumani, Kyle Rupnow, and Deming Chen. "High level synthesis of complex applications: An H. 264

video decoder." In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 224–233. ACM, 2016.

[LFF20] Alec Lu, Zhenman Fang, Nazanin Farahpour, and Lesley Shannon. "CHIP-KNN: A Configurable and High-Performance K-Nearest Neighbors Accelerator on Cloud FPGAs." In *Proceedings of the 2020 International Conference on Field-Programmable Technology (ICFPT '20)*, 2020.

[LGM05] Dong-U Lee, Altaf Abdul Gaffar, Oskar Mencer, and Wayne Luk. "MiniBit: bit-width optimization via affine arithmetic." In *Proceedings of the 42nd annual Design Automation Conference*, pp. 837–840. ACM, 2005.

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

[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 *Proceedings of the 2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM '18)*, 2018.

[LL20] Wuyue Lu and Ligang Liu. "Surface reconstruction via cooperative evolutions." *Computer Aided Geometric Design*, **77**:101831, 2020.

[LLC23] Jianwen Luo, Xinzhe Liu, Fupeng Chen, and Yajun Ha. "HRFF: Hierarchical and Recursive Floorplanning Framework for NoC-Based Scalable Multidie FPGAs." *IEEE Transactions on Circuits and Systems I: Regular Papers*, **70**(11):4295–4308, 2023.

[LLS23] Kenneth Liu, Alec Lu, Kartik Samtani, Zhenman Fang, and Licheng Guo. "CHIP-KNNv2: A Configurable and High-Performance K-Nearest Neighbors Accelerator on HBM-based FPGAs." *ACM Trans. Reconfigurable Technol. Syst.*, **16**(4), December 2023.

[LLV] LLVM. "CIRCT: Circuit IR Compilers and Tools." `https://circt.llvm.org/`.

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

[LS05] Henry Oliver Lancaster and Eugene Seneta. "Chi-square distribution." *Encyclopedia of biostatistics*, **2**, 2005.

[LSZ20] Jason Lau, Aishwarya Sivaraman, Qian Zhang, Muhammad Ali Gulzar, Jason Cong, and Miryung Kim. "HeteroRefactor: refactoring for heterogeneous computing with FPGA." In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, p. 493–505, New York, NY, USA, 2020. Association for Computing Machinery.

[LWK22] Sihao Liu, Jian Weng, Dylan Kupsh, Atefeh Sohrabizadeh, Zhengrong Wang, Licheng Guo, Jiuyang Liu, Maxim Zhulin, Rishabh Mani, Lucheng Zhang, Jason Cong, and Tony Nowatzki. "OverGen: Improving FPGA Usability through Domain-specific Overlay Generation." In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 35–56, 2022.

[LXX24] Jason Lau, Yuanlong Xiao, Yutong Xie, Yuze Chi, Linghao Song, Shaojie Xiang, Michael Lo, Zhiru Zhang, Jason Cong, and Licheng Guo. "RapidStream IR: Infrastructure for FPGA High-Level Physical Synthesis." *arXiv preprint arXiv:2410.13079*, 2024.

[MB24] Kingshuk Majumder and Uday Bondhugula. "HIR: An MLIR-based Intermediate Representation for Hardware Accelerator Description." In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '23. Association for Computing Machinery, 2024.

[MGC23] Mohammadmahdi Mazraeli, Yu Gao, and Paul Chow. "Partitioning Large-Scale, Multi-FPGA Applications for the Data Center." In *Proceedings of the 2023 33rd International Conference on Field-Programmable Logic and Applications (FPL '23)*, 2023.

[NBN23] Tan Nguyen, Zachary Blair, Stephen Neuendorffer, and John Wawrzynek. "SPADES: A Productive Design Flow for Versal Programmable Logic." In *Proceedings of the 2023 33rd International Conference on Field-Programmable Logic and Applications (FPL '23)*, 2023.

[NTL21] Rachit Nigam, Samuel Thomas, Zhijing Li, and Adrian Sampson. "A compiler infrastructure for accelerator generators." In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, p. 804–817, New York, NY, USA, 2021. Association for Computing Machinery.

[Nvi11] Nvidia. "Nvidia CUDA C programming guide." *Nvidia Corporation*, **120**(18):8, 2011.

[NWL24] Xuefei Ning, Zifu Wang, Shiyao Li, Zinan Lin, Peiran Yao, Tianyu Fu, Matthew B. Blaschko, Guohao Dai, Huazhong Yang, and Yu Wang. "Can LLMs Learn by Teaching for Better Reasoning? A Preliminary Study.", 2024.

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

[PB91] Massoud Pedram and Narasimha B Bhat. "Layout Driven Logic Restructuring/Decomposition." In *ICCAD'91*, 1991.

[PCC14] Andrew Putnam, Adrian M Caulfield, Eric S Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, et al. "A reconfigurable fabric for accelerating large-scale datacenter services." *ACM SIGARCH Computer Architecture News*, **42**(3):13–24, 2014.

[PEB09] Andrew Putnam, Susan Eggers, Dave Bennett, Eric Dellinger, Jeff Mason, Henry Styles, Prasanna Sundararajan, and Ralph Wittig. "Performance and power of cache-based reconfigurable computing." *ACM SIGARCH Computer Architecture News*, **37**(3):395–405, 2009.

[PGM19] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. "PyTorch: An Imperative Style, High-Performance Deep Learning Library." In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.

[Pop24] Michael Popoloski. "Slang: SystemVerilog compiler and language services." `https://github.com/MikePopoloski/slang`, 2024.

[PSK15] Nam Khanh Pham, Amit Kumar Singh, Akash Kumar, and Mi Mi Aung Khin. "Exploiting loop-array dependencies to accelerate the design space exploration with high level synthesis." In *2015 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 157–162. IEEE, 2015.

[PSS18] Yu Pu, Chunlei Shi, Giby Samson, Dongkyu Park, Ken Easton, Rudy Beraha, Adam Newham, Mark Lin, Venkat Rangan, Karam Chatha, et al. "A 9-mm 2 ultra-low-power highly integrated 28-nm CMOS SoC for Internet of Things." *IEEE Journal of Solid-State Circuits*, **53**(3):936–948, 2018.

[PXM18] Dongjoon Park, Yuanlong Xiao, Nevo Magnezi, and André DeHon. "Case for Fast FPGA Compilation using Partial Reconfiguration." In *FPL*, pp. 235–2353, Dublin, Ireland, 2018. IEEE.

[PZS13] Louis-Noel Pouchet, Peng Zhang, Ponnuswamy Sadayappan, and Jason Cong. "Polyhedral-based data reuse optimization for configurable computing." In *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays*, pp. 29–38. ACM, 2013.

[QGF23] 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." *IEEE Transactions on Emerging Topics in Computing*, **11**(2):404–419, 2023.

[QL11] Dan Quinlan and Chunhua Liao. "The ROSE source-to-source compiler infrastructure." In *Cetus users and compiler infrastructure workshop, in conjunction with PACT*, volume 2011, p. 1. Citeseer, 2011.

[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, 2021.

[RHL18] Zhenyuan Ruan, Tong He, Bojie Li, Peipei Zhou, and Jason Cong. "ST-Accel: A high-level programming platform for streaming applications on FPGA." In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 9–16. IEEE, 2018.

[RLL11] Kyle Rupnow, Yun Liang, Yinan Li, and Deming Chen. "A study of high-level synthesis: Promises and challenges." In *2011 9th IEEE International Conference on ASIC*, pp. 1102–1105. IEEE, 2011.

[RNM16] Cindy Rubio-González, Cuong Nguyen, Benjamin Mehne, Koushik Sen, James Demmel, William Kahan, Costin Iancu, Wim Lavrijsen, David H Bailey, and David Hough. "Floating-point precision tuning using blame analysis." In *Proceedings of the 38th International Conference on Software Engineering*, pp. 1074–1085. ACM, 2016.

[RNN13] Cindy Rubio-González, Cuong Nguyen, Hong Diep Nguyen, James Demmel, William Kahan, Koushik Sen, David H Bailey, Costin Iancu, and David Hough. "Precimonious: Tuning assistant for floating-point precision." In *SC'13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pp. 1–12. IEEE, 2013.

[RXC24] Anastasiia Ruzhanskaia, Pengcheng Xu, David Cock, and Timothy Roscoe. "Rethinking Programmed I/O for Fast Devices, Cheap Cores, and Coherent Interconnects.", 2024.

[Sal02] Matthew J. Saltzman. *Coin-OR: An Open-Source Library for Optimization*, pp. 3–32. Springer US, Boston, MA, 2002.

[SBA00] Mark Stephenson, Jonathan Babb, and Saman Amarasinghe. "Bidwidth Analysis with Application to Silicon Compilation." In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, pp. 108–120, New York, NY, USA, 2000. ACM.

[SBS23] Atefeh Sohrabizadeh, Yunsheng Bai, Yizhou Sun, and Jason Cong. "Robust GNN-Based Representation Learning for HLS." In *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, pp. 1–9, 2023.

[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*, FPGA '22, p. 65–77, New York, NY, USA, 2022. Association for Computing Machinery.

[SDL24] Chunyou Su, Linfeng Du, Tingyuan Liang, Zhe Lin, Maolin Wang, Sharad Sinha, and Wei Zhang. "GraFlex: Flexible Graph Processing on FPGAs through Customized Scalable Interconnection Network." In *Proceedings of the 2024 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '24, p. 143–153, New York, NY, USA, 2024. Association for Computing Machinery.

[SDM17] Nitish Srivastava, Steve Dai, Rajit Manohar, and Zhiru Zhang. "Accelerating Face Detection on Programmable SoC Using C-Based Synthesis." In *25th ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, Feb 2017.

[SGS10] John E Stone, David Gohara, and Guochun Shi. "OpenCL: A parallel programming standard for heterogeneous computing systems." *Computing in science & engineering*, **12**(3):66–73, 2010.

[Sie24] Siemens. *Catapult High-Level Synthesis and Verification*, 2024.

[SKK12] Markus Steinberger, Michael Kenzel, Bernhard Kainz, and Dieter Schmalstieg. "ScatterAlloc: Massively parallel dynamic memory allocation for the GPU." In *2012 Innovative Parallel Computing (InPar)*, pp. 1–10. IEEE, 2012.

[SMA05] Koushik Sen, Darko Marinov, and Gul Agha. "CUTE: A Concolic Unit Testing Engine for C." In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, pp. 263–272, New York, NY, USA, 2005. ACM.

[SMM03] Giacinto Paolo Saggese, Antonino Mazzeo, Nicola Mazzocca, and Antonio GM Strollo. "An FPGA-based performance analysis of the unrolling, tiling, and pipelining of the AES algorithm." In *International Conference on Field Programmable Logic and Applications*, pp. 292–302. Springer, 2003.

[Sol20] Xilinx/Falcon Computing Solutions. "Merlin Compiler." `https://www.mentor.com/hls-lp/catapult-high-level-synthesis`, 2020.

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

[SW03] Jérôme Siméon and Philip Wadler. "The essence of XML." *SIGPLAN Not.*, **38**(1):1–13, jan 2003.

[SYG22] Atefeh Sohrabizadeh, Cody Hao Yu, Min Gao, and Jason Cong. "AutoDSE: Enabling Software Programmers to Design Efficient FPGA Accelerators." *ACM Trans. Des. Autom. Electron. Syst.*, **27**(4), feb 2022.

[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. ACM, 2015.

[TFX20] Zhongze Tang, Xianglong Feng, Yi Xie, Huy Phan, Tian Guo, Bo Yuan, and Sheng Wei. "VVSec: Securing Volumetric Video Streaming via Benign Use of Adversarial Perturbation." In *Proceedings of the 28th ACM International Conference on Multimedia*, MM '20, p. 3614–3623, New York, NY, USA, 2020. Association for Computing Machinery.

[TG05] Sriraman Tallam and Neelam Gupta. "A Concept Analysis Inspired Greedy Algorithm for Test Suite Minimization." In *Proceedings of the 6th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '05, pp. 35–42, New York, NY, USA, 2005. ACM.

[Tho16] David B Thomas. "Synthesisable recursion for C++ HLS tools." In *2016 IEEE 27th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pp. 91–98. IEEE, 2016.

[Tho19] David B Thomas. "Templatised Soft Floating-Point for High-Level Synthesis." In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2019.

[TK18] Masato Tatsuoka and Mineo Kaneko. "Wire congestion aware high level synthesis flow with source code compiler." In *ICICDT'18*, 2018.

[TPF23] Zhongze Tang, Huy Phan, Xianglong Feng, Bo Yuan, Yao Liu, and Sheng Wei. “Security-Preserving Live 3D Video Surveillance.” In *Proceedings of the 14th ACM Multimedia Systems Conference*, MMSys '23, p. 266–277, New York, NY, USA, 2023. Association for Computing Machinery.

[TWO15] Masato Tatsuoka, Ryosuke Watanabe, Tatsushi Otsuka, Takashi Hasegawa, Qiang Zhu, Ryosuke Okamura, Xingri Li, and Tsuyoshi Takabatake. “Physically aware high level synthesis design flow.” In *DAC '15*, 2015.

[TYL23] 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.” *ACM Trans. Reconfigurable Technol. Syst.*, **16**(2), April 2023.

[TYL24] Zhongze Tang, Mengmei Ye, Yao Liu, and Sheng Wei. “Privacy-Preserving Multimedia Mobile Cloud Computing Using Protective Perturbation.”, 2024.

[VHB06] Babette Van Antwerpen, Michael D Hutton, Gregg Baeckler, and Richard Yuan. “Register retiming technique.”, October 10 2006. US Patent 7,120,883.

[VPN10] Jason Villarreal, Adrian Park, Walid Najjar, and Robert Halstead. “Designing Modular Hardware Accelerators in C with ROCCC 2.0.” In *Proceedings of the 2010 18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM '10)*, 2010.

[Wea08] Nicholas Weaver. “Retiming, repipelining and c-slow retiming.” In *Reconfigurable Computing*, pp. 383–399. Elsevier, 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 (FPGA '21)*, 2021.

[WGK13] Clifford Wolf, Johann Glaser, and Johannes Kepler. “Yosys: A free Verilog synthesis suite.” In *Proceedings of the 21st Austrian Workshop on Microelectronics (Austrochip)*, volume 97, 2013.

[WJN95] Paul R Wilson, Mark S Johnstone, Michael Neely, and David Boles. “Dynamic storage allocation: A survey and critical review.” In *International Workshop on Memory Management*, pp. 1–116. Springer, 1995.

[WMP03] Nicholas Weaver, Yury Markovskiy, Yatish Patel, and John Wawrzynek. “Post-placement C-slow retiming for the Xilinx Virtex FPGA.” In *Proceedings of the 2003 ACM/SIGDA eleventh international symposium on Field programmable gate arrays*, pp. 185–194. ACM, 2003.

[XAD20] Yuanlong Xiao, Syed Ahmed, and André DeHon. "Fast Linking of Separately Compiled FPGA Blocks without a NoC." In *ICFPT*, pp. 196–205, Maui, HI, USA, 2020. IEEE.

[XDC24] Yutong Xie, Benyamin Davaji, Ivan Chakarov, Sandy Wen, Michael Hargrove, David Fried, Peter C. Doerschuk, and Amit Lal. "Quantitative Comparison of Simulation and Experiment Enabling a Lithography Digital Twin." *IEEE Transactions on Semiconductor Manufacturing*, **37**(4):546–552, 2024.

[XHP22] Yuanlong Xiao, Aditya Hota, Dongjoon Park, and André DeHon. "HiPR: High-level Partial Reconfiguration for Fast Incremental FPGA Compilation." In *Proceedings of the 2022 32nd International Conference on Field-Programmable Logic and Applications (FPL '22)*, 2022.

[Xil19] Xilinx. "UltraScale Architecture and Product Data Sheet: Overview." `https://www.xilinx.com/support/documentation/data_sheets/ds890-ultrascale-overview.pdf`, 2019.

[Xil20] Xilinx. "Xilinx Vitis Unified Platform.", 2020.

[Xil21a] Xilinx. *AI Engine Intrinsics User Guide (UG1078): (v2021.2)*. Xilinx, 2021.

[Xil21b] Xilinx. *AI Engine Kernel Coding Best Practices Guide: UG1079 (v2021.1)*. Xilinx, 2021.

[Xil21c] Xilinx. *Vitis High-Level Synthesis User Guide: UG1399 (v2021.2)*. Xilinx, 2021.

[Xil22a] Xilinx. "Adaptive Compute Acceleration Platform.", 2022.

[Xil22b] Xilinx. "MLIR-AIE: MLIR-based AIEngine toolchain.", 2022.

[Xil22c] Xilinx. "VCK5000 Versal Development Card.", 2022.

[Xil22d] Xilinx. "Versal AI Core Series VCK190 Evaluation Kit.", 2022.

[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*, ASPLOS 2022, pp. 933–945, New York, NY, USA, 2022. Association for Computing Machinery.

[XPB19] Yuanlong Xiao, Dongjoon Park, Andrew Butt, Hans Giesen, Zhaoyang Han, Rui Ding, Nevo Magnezi, and André DeHon. "Reducing FPGA Compile Time with Separate Compilation for FPGA Building Blocks." In *ICFPT*, pp. 153–161, TianJin, China, 2019. IEEE.

[XPN24] Yuanlong Xiao, Dongjoon Park, Zeyu Jason Niu, Aditya Hota, and André Dehon. "ExHiPR: Extended High-Level Partial Reconfiguration for Fast Incremental FPGA Compilation." *ACM Transactions on Reconfigurable Technology and Systems*, **17**(2), Mar 2024.

[XT15] Zeping Xue and David B Thomas. "SysAlloc: A hardware manager for dynamic memory allocation in heterogeneous systems." In *2015 25th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 1–7. IEEE, 2015.

[XT16] Zeping Xue and David B Thomas. "SynADT: Dynamic Data Structures in High Level Synthesis." In *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 64–71. IEEE, 2016.

[YGB24] Peiran Yao, Kostyantyn Guzhva, and Denilson Barbosa. "Semantic Graphs for Syntactic Simplification: A Revisit from the Age of LLM.", 2024.

[YHC21] Hanchen Ye, Cong Hao, Jianyi Cheng, Hyunmin Jeong, Jack Huang, Stephen Neuendorffer, and Deming Chen. "ScaleHLS: A New Scalable High-Level Synthesis Framework on Multi-Level Intermediate Representation." *arXiv preprint arXiv:2107.11673*, 2021.

[YHC22] Hanchen Ye, Cong Hao, Jianyi Cheng, Hyunmin Jeong, Jack Huang, Stephen Neuendorffer, and Deming Chen. "ScaleHLS: A New Scalable High-Level Synthesis Framework on Multi-Level Intermediate Representation." In *Proceedings of the 2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA '22)*, 2022.

[YKW23] Peiran Yao, Matej Kosmajac, Abeer Waheed, Kostyantyn Guzhva, Natalie Hervieux, and Denilson Barbosa. "NLP Workbench: Efficient and Extensible Integration of State-of-the-art Text Mining Tools." In *Proceedings of the 17th Conference of the European Chapter of the Association for Computational Linguistics: System Demonstrations*, p. 18–26. Association for Computational Linguistics, 2023.

[YMS24] Peiran Yao, Jerin George Mathew, Shehraj Singh, Donatella Firmani, and Denilson Barbosa. "A Bayesian Approach Towards Crowdsourcing the Truths from LLMs." In *NeurIPS 2024 Workshop on Bayesian Decision-making and Uncertainty*, 2024.

[YRB22] Peiran Yao, Tobias Renwick, and Denilson Barbosa. "WordTies: Measuring Word Associations in Language Models via Constrained Sampling." In Yoav Goldberg, Zornitsa Kozareva, and Yue Zhang, editors, *Findings of the Association for Computational Linguistics: EMNLP 2022*, pp. 5959–5970, Abu

Dhabi, United Arab Emirates, December 2022. Association for Computational Linguistics.

[YTP22] Mengmei Ye, Zhongze Tang, Huy Phan, Yi Xie, Bo Yuan, and Sheng Wei. "Visual privacy protection in mobile image recognition using protective perturbation." In *Proceedings of the 13th ACM Multimedia Systems Conference*, MMSys '22, p. 164–176, New York, NY, USA, 2022. Association for Computing Machinery.

[YWG18] Cody Hao Yu, Peng Wei, Max Grossman, Peng Zhang, Vivek Sarker, and Jason Cong. "S2FA: an accelerator automation framework for heterogeneous computing in datacenters." In *Proceedings of the 55th Annual Design Automation Conference (DAC)*, p. 153. ACM, 2018.

[YYX24] Qizheng Yang, Tianyi Yang, Mingcan Xiang, Lijun Zhang, Haoliang Wang, Marco Serafini, and Hui Guan. "GMorph: Accelerating Multi-DNN Inference via Model Fusion." In *Proceedings of the Nineteenth European Conference on Computer Systems*, EuroSys '24, p. 505–523, New York, NY, USA, 2024. Association for Computing Machinery.

[ZGD18] Yuan Zhou, Udit Gupta, Steve Dai, Ritchie Zhao, Nitish Srivastava, Hanchen Jin, Joseph Featherston, Yi-Hsiang Lai, Gai Liu, Gustavo Angarita Velasquez, Wenping Wang, and Zhiru Zhang. "Rosetta: A Realistic High-Level Synthesis Benchmark Suite for Software-Programmable FPGAs." *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, Feb 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. ACM, 2014.

[ZLC13] Wei Zuo, Peng Li, Deming Chen, Louis-Noël Pouchet, Shunan Zhong, and Jason Cong. "Improving polyhedral code generation for high-level synthesis." In *2013 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ ISSS)*, pp. 1–10. IEEE, 2013.

[ZLD24] Shulin Zeng, Jun Liu, Guohao Dai, Xinhao Yang, Tianyu Fu, Hongyi Wang, Wenheng Ma, Hanbo Sun, Shiyao Li, Zixiao Huang, Yadong Dai, Jintao Li, Zehao Wang, Ruoyu Zhang, Kairui Wen, Xuefei Ning, and Yu Wang. "FlightLLM: Efficient Large Language Model Inference with a Complete Mapping Flow on FPGAs." In *Proceedings of the 2024 ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '24)*, 2024.

[ZLY23] Jinming Zhuang, Jason Lau, Hanchen Ye, Zhuoping Yang, Yubo Du, Jack Lo, Kristof Denolf, Stephen Neuendorffer, Alex Jones, Jingtong Hu, Deming Chen, Jason Cong, and Peipei Zhou. "CHARM: Composing Heterogeneous AcceleRators for Matrix Multiply on Versal ACAP Architecture." In *Proceedings of the 2023 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '23, p. 153–164, New York, NY, USA, 2023. Association for Computing Machinery.

[ZLY24] Jinming Zhuang, Jason Lau, Hanchen Ye, Zhuoping Yang, Shixin Ji, Jack Lo, Kristof Denolf, Stephen Neuendorffer, Alex Jones, Jingtong Hu, Yiyu Shi, Deming Chen, Jason Cong, and Peipei Zhou. "CHARM 2.0: Composing Heterogeneous Accelerators for Deep Learning on Versal ACAP Architecture." *ACM Trans. Reconfigurable Technol. Syst.*, **17**(3), September 2024.

[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*, p. 157. ACM, 2015.

[ZWK21] Qian Zhang, Jiyuan Wang, and Miryung Kim. "HeteroFuzz: fuzz testing to detect platform dependent divergence for heterogeneous applications." In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2021, p. 242–254, New York, NY, USA, 2021. Association for Computing Machinery.

[ZWX22] Qian Zhang, Jiyuan Wang, Guoqing Harry Xu, and Miryung Kim. "HeteroGen: transpiling C to heterogeneous HLS code with automated test generation and program repair." In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, p. 1017–1029, New York, NY, USA, 2022. Association for Computing Machinery.