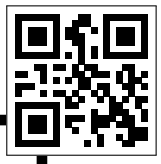


RV College of Engineering®
Mysore Road, RV Vidyaniketan Post, Bengaluru - 560059, Karnataka, India

Go, change the world

Towards Efficient ML Hardware : FPGA Architectural Exploration and PCA Accelerator Design

A Major Project Report (21EC81P)

Submitted by,

Srivaths Ramasubramanian **1RV21EC168**

Under the guidance of

Dr. K.S Geetha
Professor & Vice Principal
Dept. of ECE
RV College of Engineering

In partial fulfillment of the requirements for the degree of Bachelor of Engineering in

Electronics and Communication Engineering

2024-25

RV College of Engineering®, Bengaluru

(*Autonomous institution affiliated to VTU, Belagavi*)

Department of Electronics and Communication Engineering

CERTIFICATE

Certified that the major project (21EC81P)work titled ***Towards Efficient ML Hardware : FPGA Architectural Exploration and PCA Accelerator Design*** is carried out by **Srivaths Ramasubramanian** (**1RV21EC168**) who is bonafide student of RV College of Engineering, Bengaluru, in partial fulfillment of the requirements for the degree of **Bachelor of Engineering** in **Electronics and Communication Engineering** of the Visvesvaraya Technological University, Belagavi during the year 2024-25. It is certified that all corrections/suggestions indicated for the Internal Assessment have been incorporated in the major project report deposited in the departmental library. The major project report has been approved as it satisfies the academic requirements in respect of major project work prescribed by the institution for the said degree.

Guide	Head of the Department	Principal
Dr. K.S Geetha	Dr. H V Ravish Aradhya	Dr. K. N. Subramanya

External Viva

Name of Examiners	Signature with Date
1.	
2.	

DECLARATION

I, **Srivaths Ramasubramanian** students of eighth semester B.E., Department of Electronics and Communication Engineering, RV College of Engineering, Bengaluru, hereby declare that the major project titled '**Towards Efficient ML Hardware : FPGA Architectural Exploration and PCA Accelerator Design**' has been carried out by me and submitted in partial fulfilment for the award of degree of **Bachelor of Engineering** in **Electronics and Communication Engineering** during the year 2024-25.

Further I declare that the content of the dissertation has not been submitted previously by anybody for the award of any degree or diploma to any other university.

I also declare that any Intellectual Property Rights generated out of this project carried out at RVCE will be the property of RV College of Engineering, Bengaluru and we will be one of the authors of the same.

Place: Bengaluru

Date:

Name	Signature
1. Srivaths Ramasubramanian(1RV21EC168)	

ACKNOWLEDGEMENTS

I am indebted to my guide, **Dr. K.S Geetha**, Professor & Vice Principal, RV College of Engineering . for the wholehearted support, suggestions and invaluable advice throughout my project work and also helped in the preparation of this thesis.

I also express our gratitude to my panel members **Dr. Roopa J**, Assistant Professor and **Prof. Ravishankar Holla**, Assistant Professor, Department of Electronics and Communication Engineering for their valuable comments and suggestions during the phase evaluations.

My sincere thanks to the project coordinators **Dr. Veena Devi S V** and **Prof. Sindhu Rajendran** for their timely instructions and support in coordinating the project.

My gratitude to **Prof. Narashimaraja P** for the organized latex template which made report writing easy and interesting.

My sincere thanks to **Dr. H V Ravish Aradhya**, Professor and Head, Department of Electronics and Communication Engineering, RVCE for the support and encouragement.

I express sincere gratitude to our beloved Principal, **Dr. K. N. Subramanya** for the appreciation towards this project work.

I thank all the teaching staff and technical staff of Electronics and Communication Engineering department, RVCE for their help.

Lastly, I take this opportunity to thank my family members and friends who provided all the backup support throughout the project work.

ABSTRACT

Machine learning (ML) workloads on edge devices demand high performance with energy efficiency—needs poorly met by general-purpose processors and power-hungry GPUs. FPGAs, with their parallelism and configurability, offer a promising alternative, yet current architectures and accelerator designs remain suboptimal for ML. This report addresses these challenges through a two-part solution: (1) architectural space exploration of FPGA fabrics tailored for ML benchmarks, and (2) the design and realization of Manojavam, a domain-specific PCA accelerator. Together, they aim to enhance ML performance using fine-grained architectural choices and application-driven hardware specialization.

The objectives include evaluating carry chain-based FPGA architectures (single and double chain) against DSP-heavy designs, particularly for low-precision and approximate computing. In parallel, we indigenously designed and implemented the Manojavam accelerator entirely in RTL, with a fully original architecture tailored for matrix-centric ML workloads. It features 8× 4×4 TPU-style systolic arrays for matrix multiplication, a novel Jacobian Unit for eigendecomposition via the Jacobi method, and a dual-tier cache hierarchy designed to support block streaming and operand reuse. A key novelty of Manojavam lies in its ability to execute both matrix multiplication and SVD within a single unified hardware pipeline, enabling complete PCA acceleration in a standalone and streaming-optimized design.

Simulation and synthesis were conducted using the VTR toolchain (for architectural benchmarks), Vivado (for FPGA flow), and OpenLane (for ASIC flow). Benchmarks included FIR filters and bit-serial MACs, and test matrices from 1000×4 to 1000×1024. Results showed Manojavam running at 200 MHz with 1.2W power use. Carry chain architectures consistently outperformed hard block designs in ML tasks, and Manojavam showed significant latency and energy improvements over CPU/GPU baselines.

Post-simulation, the design was validated on Xilinx Artix-7, with successful RTL integration, floorplanning, timing closure, and GDSII generation via OpenLane, confirming hardware viability.

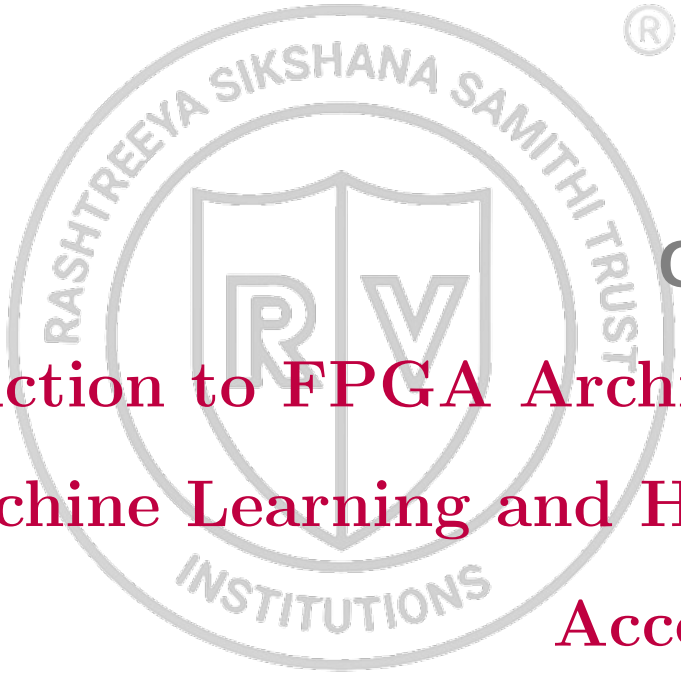
CONTENTS

LIST OF FIGURES

LIST OF TABLES


RASHTREEYA SIKSHANA SAMITHI TRUST
RV
INSTITUTIONS


RASHTREEYA SIKSHANA SAMITHI TRUST
RV
INSTITUTIONS



Chapter 1

Introduction to FPGA Architectures for Machine Learning and Hardware Acceleration

CHAPTER 1

INTRODUCTION TO FPGA ARCHITECTURES FOR MACHINE LEARNING AND HARDWARE ACCELERATION

1.1 Introduction

The exponential growth of machine learning (ML) across domains such as healthcare, finance, autonomous systems, and natural language processing has driven a parallel need for hardware platforms capable of executing ML models with high throughput, low latency, and energy efficiency. While data centers have traditionally handled most of this computation using CPUs and GPUs, there is now a critical push to bring ML closer to the source of data — on edge and embedded devices — where constraints on power, size, and compute capability are far more stringent.

Conventional general-purpose processors (CPUs) are not well-suited for ML inference tasks due to limited parallelism and sequential instruction execution. Graphics processing units (GPUs), though highly parallel, consume considerable power and rely heavily on memory bandwidth, which can become a bottleneck for real-time or streaming applications. As a response to these limitations, field-programmable gate arrays (FPGAs) have garnered attention due to their reconfigurability, high-performance-per-watt ratio, and ability to implement custom parallel datapaths for specific ML operations.

Despite these advantages, FPGA design for ML is far from straightforward. Several challenges exist, such as selecting the most suitable internal architecture (e.g., use of carry chains vs. DSP blocks), managing memory hierarchies, balancing performance with resource usage, and designing application-specific accelerators that leverage the unique strengths of the FPGA fabric.

This project, titled "Design of Efficient ML Hardware: FPGA Architectural Optimization and Domain-Specific Hardware", is a comprehensive attempt to tackle these challenges from two complementary perspectives:

1. **Architectural Exploration Using the VTR Toolchain:** This component investigates and compares different FPGA fabric architectures — particularly carry chain-based versus DSP block-centric designs — for their efficiency in handling ML-

centric benchmarks. The study focuses on low-precision arithmetic, approximate computing, and non-standard number representations commonly used in neural networks and digital signal processing. Benchmark circuits such as adder trees and FIR filters are refactored to utilize carry chains effectively, and performance metrics such as delay, area, and power are evaluated.

2. **Design of a Domain-Specific Accelerator – Manojavam:** This custom-built hardware accelerator targets Principal Component Analysis (PCA), a widely used dimensionality reduction technique in ML. The design includes an array of TPU-style 4×4 systolic arrays for matrix multiplication and a novel Jacobian Unit that uses the Jacobi method to perform eigendecomposition. The architecture incorporates efficient memory access patterns through block streaming and cache hierarchies and is implemented entirely in RTL. Manojavam was synthesized and tested on an FPGA (Xilinx Artix-7), and also realized through ASIC flow using OpenLane.

This dual-focus effort is especially relevant in the current era where ML model optimization must go hand-in-hand with hardware optimization. The ability to fine-tune the architecture — either by modifying the FPGA fabric or by building domain-specific RTL pipelines — is critical for improving performance-per-watt and ensuring scalability in edge ML applications.

In essence, this project bridges two essential layers of ML acceleration — architectural suitability and domain-specific hardware design — making it not only a technically enriching exploration but also a contribution toward the future of ML compute platforms.

1.2 Motivation

The motivation behind choosing this project stems from two persistent and interrelated challenges in the field of machine learning hardware: the need for energy-efficient, real-time ML execution platforms suitable for edge devices, and the growing importance of performing architectural exploration of FPGA platforms using established frameworks such as the widely adopted VTR toolchain, particularly in the context of ML-centric workloads.

As ML becomes increasingly embedded into devices like drones, sensors, wearables, and autonomous systems, the demand for compact, low-power hardware accelerators has intensified. General-purpose processors fall short due to their serial execution model,

while GPUs — though powerful — are often unsuitable for power-sensitive environments. This made the case for exploring FPGAs, which offer flexibility and energy efficiency. However, the choice of FPGA internal architecture significantly impacts ML performance. The first part of the project involved evaluating existing FPGA architectures using the VTR toolchain. Architectures were assessed using adder tree and FIR filter benchmarks, which are representative of machine learning workloads due to their reliance on fundamental operations like addition and multiplication.

Simultaneously, in academic and applied research, Principal Component Analysis (PCA) remains a cornerstone of dimensionality reduction in ML and signal processing. Yet, very few hardware implementations efficiently support PCA operations like covariance matrix computation and eigendecomposition. This motivated the second part of the project — designing Manojavam, a domain-specific accelerator that performs PCA using custom RTL logic.

The idea of combining architectural benchmarking with the design of a complete hardware accelerator emerged as a powerful way to bridge theory and application. It allowed for deep exploration of both low-level FPGA features and high-level ML operations, and aligned well with current trends in edge AI and domain-specific hardware design.

1.3 Problem statement

General-purpose processors are inherently inefficient for machine learning (ML) workloads due to their limited parallelism, sequential instruction execution, and high power consumption, while GPUs, though better suited for compute-intensive tasks, suffer from excessive energy usage and memory bandwidth constraints. Although FPGAs offer a promising balance of configurability, parallelism, and energy efficiency, current FPGA architectures are not optimized for ML tasks, particularly those involving low-precision arithmetic or approximate computing. At the same time, domain-specific accelerators capable of performing foundational ML algorithms like Principal Component Analysis (PCA) in an efficient and scalable manner are largely absent in RTL-level research and development. This project aims to address these challenges by conducting an architectural exploration of FPGA fabrics for ML workloads and designing a custom PCA accelerator, Manojavam, capable of efficient covariance matrix computation and eigendecomposition using systolic arrays and RTL-based optimization.

1.4 Objectives

The objectives of the project are-

1. To explore and evaluate FPGA architectural features, specifically carry chain-based designs versus DSP block-based architectures, for their suitability in machine learning workloads using the VTR toolchain.

2. To design and implement a domain-specific hardware accelerator named Manojavam for Principal Component Analysis (PCA), featuring matrix multiplication using systolic arrays and eigendecomposition using a custom Jacobian Unit.

3. To synthesize, simulate, and validate the proposed accelerator on both FPGA and ASIC flows, and compare its performance with conventional CPU/GPU implementations as well as existing state-of-the-art PCA accelerators.

1.5 Literature Review

1.5.1 Related Work on FPGA Architectural Exploration for ML Workloads

Over the past three decades, the architecture of Field-Programmable Gate Arrays (FPGAs) has undergone extensive evolution, driven by the increasing complexity of target applications and advances in process technologies. A core objective of FPGA architecture research has been to balance flexibility with performance, area efficiency, and power consumption.

Initial FPGA designs employed basic logic elements (BLEs) consisting of small look-up tables (LUTs) and flip-flops. These BLEs were clustered into larger logic blocks with local interconnect to enhance packing efficiency and reduce inter-block routing delay. Ahmed and Rose[1] empirically studied the area-delay trade-offs of different LUT sizes and BLE clusterings, concluding that 4–6 input LUTs and cluster sizes of 3–10 BLEs provide an optimal area-delay product. Fracturable LUTs were introduced in Altera's Stratix II[2] and Xilinx's Virtex-5[3] to mitigate underutilization in large LUTs. These allowed a single 6-LUT to be decomposed into two 5-LUTs, improving logic density and enabling more effective packing of smaller functions.

Arithmetic operations—especially additions and multiplications—are central to many FPGA applications, from DSP to machine learning. Initially, such operations were im-

plemented using LUTs, but this approach suffered from excessive logic utilization and long critical paths. To address this, modern FPGAs integrate hardened carry chains within their logic elements. Murray et al[4] found that 22% of the logic elements in typical FPGA workloads implement arithmetic, justifying the need for specialized support. Their study shows that hardening carry chains improves performance by 75% for arithmetic microbenchmarks and 15% for general workloads.

The latest Intel Agilex devices[5] and Xilinx Versal architecture[6] further advance this trend by implementing carry-skip and carry-lookahead adders, respectively. The Agilex architecture hardens 2-bit carry-skip adders per logic element, enabling dense and high-speed arithmetic, while Versal enables 1-bit per logic element with advanced carry logic. Stratix devices also use fracturable 6-LUTs that support 2-bit arithmetic per adaptive logic module (ALM), achieving both compact area and high arithmetic throughput.

Proposals to enhance arithmetic density include logic elements with 4-bit carry chains arranged in single or dual configurations[4][7]. These studies demonstrated that dual-chain configurations improve multiply-accumulate (MAC) operation density by up to 17% and reduce area by 8%, showing promise for arithmetic-intensive ML and DSP workloads.

A critical aspect of FPGA architecture research is evaluating trade-offs across area, timing, and power. The typical methodology involves defining an architecture model, selecting benchmark circuits, and utilizing a CAD toolchain to simulate implementation results[7]. The Verilog-to-Routing (VTR) tool[8] is a widely used open-source CAD framework for this purpose. It allows researchers to model custom FPGA architectures through descriptive XML files, map benchmark designs, perform placement and routing, and obtain post-routing metrics. This methodology underpins much of the academic work in architecture exploration and was adopted in the present study for fair evaluation of different carry chain designs.

Benchmarking traditionally relied on the MCNC suite[9], which includes small logic-centric designs. However, these are no longer representative of modern FPGA applications. The Titan benchmark suite[8], derived from industrial designs, provides larger and more realistic circuits and has become a standard in architecture evaluation. Other works have emphasized the need for emerging benchmark suites that better reflect ML and signal-processing-heavy workloads.

Modern commercial FPGAs have embraced increasing heterogeneity, incorporating DSP blocks, embedded memories, and even hardened interconnect pipelining. For example, Intel's Stratix 10 integrates optional pipeline registers within routing drivers using configurable pulse latches[10], while Xilinx Versal includes input-side block registers to reduce long interconnect delay[6]. These additions aim to mitigate timing bottlenecks associated with deeper pipelining in high-performance workloads.

Additionally, attempts to repurpose the logic fabric for low-precision ML tasks, such as implementing 8-bit MACs using LUTs and carry chains[7], have driven research into logic elements that better support partial product reduction and carry-propagation efficiency. Carry chains have proven useful in implementing tree-based multipliers and accumulators, underscoring their utility beyond addition.

1.5.2 Related Work on PCA and SVD Accelerators

There have been attempts to overcome computational challenges associated with Principal Component Analysis (PCA) and Singular Value Decomposition (SVD). As PCA and SVD are inherently related to each other, the below sections highlight the state of art approaches to accelerate both PCA and SVD workloads on high performance computing hardware.

In [11], the authors implement a full fledged accelerator for Principal Component Analysis, where the architecture solves both the learning and mapping phases. They implement eigenvalue decomposition using QR factorization with Givens CORDIC rotation on the computed covariance matrix. The accelerator works at an operational frequency of 183 MHz, and consumes 56% of LUTs, 19% of memory and 88% of the DSP blocks, on a Xilinx Virtex-7 FPGA. However, the design does not scale well for large matrices and operates better than an Nvidia GPU and Intel processors for small matrices, with a maximum tested dimension of 16×30. The authors employ a parallel vector based matrix multiplication approach.

A reconfigurable PCA accelerator was proposed in [12], where the accelerator was employed to preprocess data from satellites to the ground in low bandwidth scenarios, saving them cost. The authors implemented an EVD engine, and used a DMA module with prefetching to hide latencies associated with memory bottlenecks. The architecture employs 67% of the available FPGA resources, and shows a performance improvement of 10x and 12x on the AVIRIS Cuprite and Jasper datasets respectively. However, the

chip does not accelerate matrix multiplication, and the authors claim that it consumes too much overhead, and works faster on software. Moreover, the key microarchitectural details about the Jacobi algorithm are not revealed.

A full-fledged PCA accelerator was proposed in [13], implementing both learning and mapping phases, designed entirely using High-Level Synthesis (HLS). The primary application of the accelerator is hyperspectral imaging, where it computes the covariance matrix of the input dataset using a Covariance Unit and performs Eigenanalysis through an SVD unit. However, the paper does not disclose specific microarchitectural details of these units. The design was implemented on two FPGA platforms: the Zynq-7000 XC7Z020CLG484-1 and the Virtex-7 XC7VX690TFFG1761-2, with reported power consumption of 2.376W and 9.786W, respectively. A notable drawback of the design is its reliance on HLS, which, while simplifying hardware development, abstracts away low-level optimizations that could improve efficiency and performance.

An outlier detection scheme for Network Intrusion Detection Systems (NIDS) is proposed in [14], leveraging Principal Component Analysis for anomaly detection after an initial Feature Extraction Module (FEM) processes the network traffic. The PCA framework follows a two phase approach : an offline PCA phase, where the principal components are obtained from a training dataset, and the online PCA phase, where the incoming data is projected onto these principal components to detect outliers in real time. The authors note that while PCA significantly reduces the computational complexity, eigenvector calculations and sorting remain sequential and difficult to parallelize, making them less suited for FPGA based acceleration. However, the principal component score calculations, which involve matrix to vector computations, are efficiently parallelized in hardware.

A dynamic on-the-fly reconfigurable PCA accelerator was proposed in [15], where the accelerator dynamically reconfigures itself during different stages of the PCA algorithm. The implementation was developed on the ML605 FPGA board, which is based on 40 nm CMOS technology and features a Xilinx Virtex-6 (XC6VLX240T-FF1156) FPGA, MicroBlaze soft processors, 2MB of on-chip BRAM, and 512MB of DDR3-SDRAM for large-scale data storage. The design was implemented using a mix of Verilog and VHDL and operates at a maximum clock frequency of 100 MHz. The authors employed a Covariance Unit to compute the covariance matrix efficiently and used QR factorization for

Eigenvalue Decomposition (EVD). However, the paper does not provide microarchitectural details of these units. While dynamic reconfiguration led to a 71% area savings, it introduced significant reconfiguration overheads in the order of milliseconds, which could degrade overall performance. Additionally, the authors note that while the cyclic Jacobi method is effective for small matrices, alternative methods such as QR or Householder decomposition may be better suited for larger matrices.

The work in [16] proposes an FPGA accelerator for Singular Value Decomposition employing fixed point arithmetic, using the Hestenes-Jacobi algorithm and CORDIC. The accelerator works at an operational frequency of 250 MHz on a Xilinx Kintex-7 XC7K325T FPGA. It achieves a speedup between 8.5× and 15.3× over MATLAB and between 2.1× and 6.3× over GPU implementations. However, the design is not scalable to very large matrices due to block RAM limitations, and the authors suggest that an array of FPGAs may be needed for handling larger matrices beyond 128×128.

An FPGA based accelerator for both Singular Value Decomposition and Eigenvalue Decomposition is proposed in [17], where the accelerator is optimized for real time performance. The accelerator is based on the Jacobi algorithm, employing the Brent-Luk systolic array to efficiently compute SVD, and using CORDIC based algorithm for EVD. The design, when dumped on the Xilinx Virtex-5 FPGA, operates at a frequency of 130 MHz, and shows a performance improvement of 4x compared to a 2.53 GHz Intel processor, and an improvement of 2x for EVD when compared to the previous FPGA based designs. The hardware utilization for an 8×8 matrix reaches 98%, but while this demonstrates high efficiency, the increasing resource demand for larger matrices may impact scalability.

The work in [18] proposes an SVD engine, designed on 90nm CMOS technology, wth an operational frequency of 101.2 MHz and a power of 125 mW, for very high throughput MIMO-OFDM systems. The authors implemented an adaptive SVD engine, which would dynamically modify the output eigenvectors and values with changes in inputs. An Orthogonal Reconstruction (OR) scheme is also implemented to correct errors due to fixed point arithmetic, ensuring very high accuracy. However, the SVD engine primarily relies on a sequential execution model, particularly in Jacobi-based updates and CORDIC-based rotations, which may limit scalability for larger matrices beyond 4×4.

The work in [19] compares and contrasts the execution of the Jacobi algorithm on the CPU, GPU, and FPGA, revealing that the GPU suffers from performance degradation due to unstructured global memory access patterns. To address this, the paper proposes three enhanced memory access patterns—Modified Access (MA), Symmetric Access (SA), and Maximum Coalesced Access (MCA)—which significantly improve GPU performance. The study highlights the critical role of memory access optimization in accelerating the Jacobi algorithm, particularly for GPU implementations, while also confirming that FPGA outperforms both GPU and CPU in overall efficiency for eigenanalysis.

The work in [20] proposes a parallel fast-converging Jacobi algorithm designed for larger matrices, improving upon traditional Jacobi-based methods. Instead of processing elements in a fixed order, the algorithm selects N/2 of the largest off-diagonal elements in each iteration while maintaining row-column exclusivity. This reduces the number of Jacobi sweeps required and accelerates convergence. The design employs CORDIC-based rotations for efficient computation of trigonometric functions and is implemented on the Xilinx Virtex-6 XCVLX365T FPGA. Although the engine introduces an optimized Jacobi approach, it does not include a dedicated matrix multiplication accelerator to enhance intermediate transformations within the algorithm. As a result, row and column updates rely solely on DSP-based multipliers, which may introduce computational bottlenecks for larger matrices. Integrating a matrix multiplication accelerator (such as a systolic array or TPU-like cores) could further improve the throughput and scalability of the SVD core, especially for matrices beyond 32×32.

A Hestenes-Jacobi processor for SVD was developed in [21], which computed SVD for arbitrary size matrices in IEEE-754 double precision floating point format. The architecture comprised three dedicated units : the Hestenes Preprocessor, Jacobi Rotation unit and the Update unit. The authors do not employ a systolic array for matrix multiplication, noting that traditional systolic architectures are optimized for square matrices. However, the paper neither explores nor considers alternative approaches such as block-streaming systolic architectures, which could potentially improve efficiency in rectangular matrix multiplications. The design was dumped on a Xilinx Virtex-5 XC5VLX330 FPGA, operating on 150 MHz, consuming 89% LUTs, 91% BRAMs and 53% DSPs. Matrices of dimension in the order of 1024 were tested successfully. However, they employ Xilinx Coregen IPs in their design, consuming 9, 14, 57 and 57 clock cycles for multiplier,

adder-subtractor, divider and square root blocks respectively.

1.6 Brief Methodology of the project

The methodology followed in this project combines architectural benchmarking and accelerator design to address both the evaluation and implementation aspects of efficient ML hardware. The workflow begins with the selection of benchmark circuits relevant to ML workloads — such as adder trees and FIR filters — which are then mapped onto different FPGA fabric configurations using the VTR toolchain. These include single and double carry chain-based architectures and DSP-centric designs. Performance metrics such as delay, logic utilization, and power are recorded to identify trends and architectural advantages. Fig.1.1 outlines the methodology of the FPGA architectural exploration.

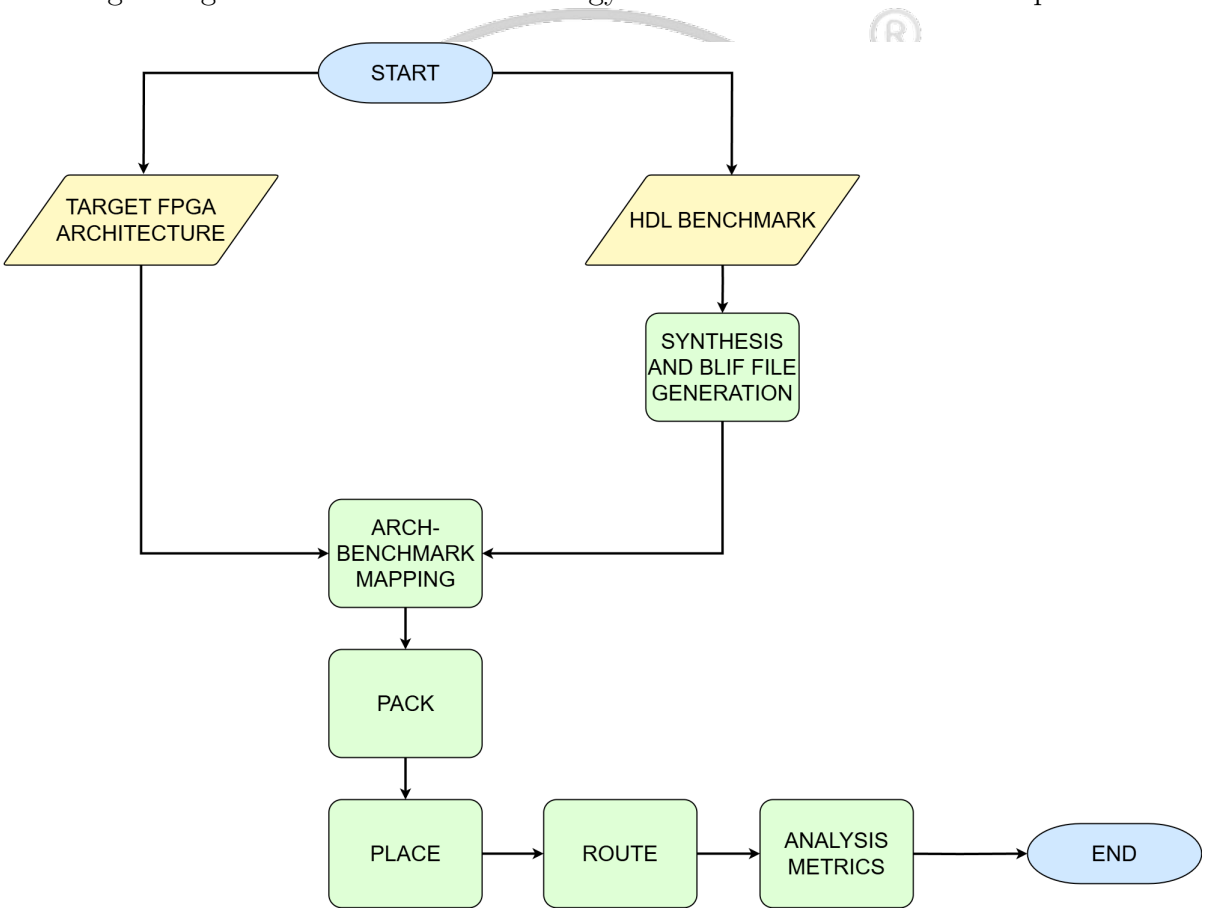


Figure 1.1: Methodology of FPGA Architectural Exploration

Parallelly, a domain-specific accelerator named Manojavam is developed using a modular RTL design. It comprises 8 systolic arrays for matrix multiplication, a novel Jacobian Unit for SVD-based eigendecomposition, and a structured cache system. RTL design and simulation are performed using Vivado for FPGA flow, while ASIC design is carried out

using the OpenLane toolchain. Floorplanning, timing closure, and power analysis are conducted to validate the feasibility of deployment. The performance of Manojavam is then compared against CPU, GPU, and existing PCA accelerator implementations using a custom-built simulator and various matrix input sizes. The methodology for the design of Manojavam is highlighted in Fig.1.2.

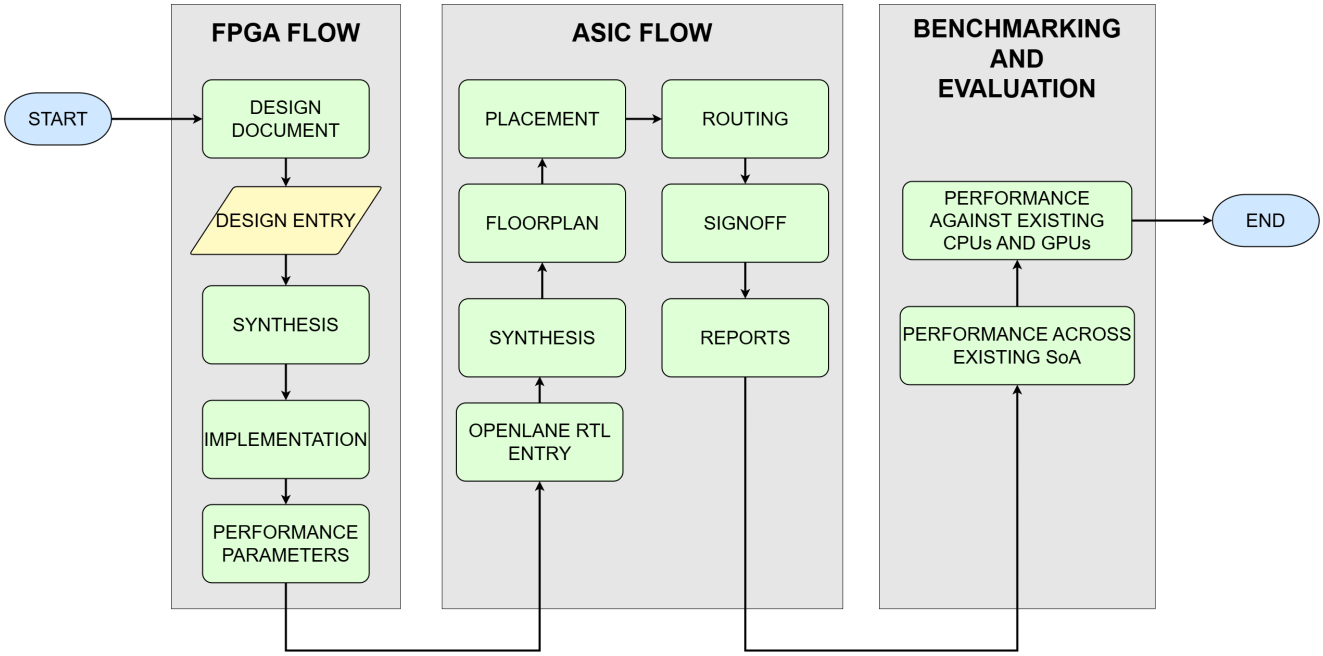


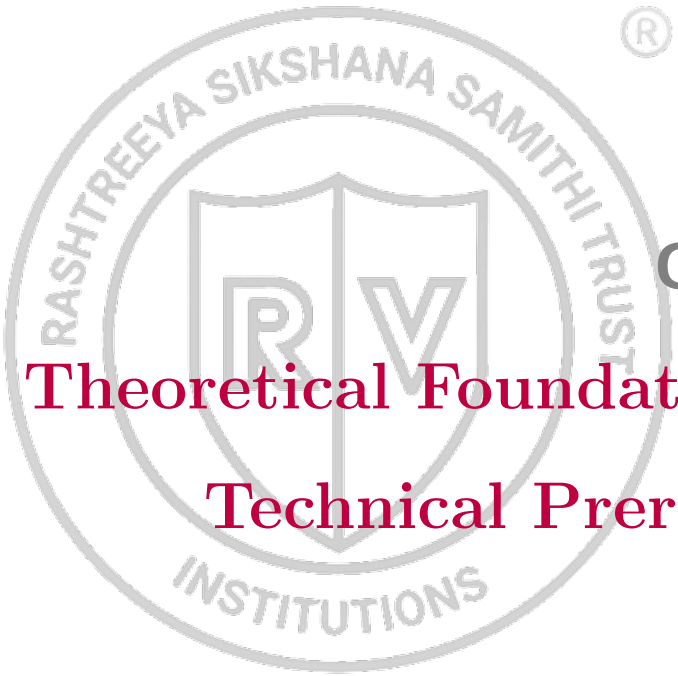
Figure 1.2: Methodology for Manojavam Accelerator Development

1.7 Organization of the report

This report is organized as follows.

- Chapter 2 discusses the fundamentals and foundational knowledge of FPGA architectures and domain specific acceleration.
- Chapter 3 discusses the design methodology.
- Chapter 4 discusses the implementation strategies of both projects.
- Chapter 5 discusses the results and discussions.
- Chapter 6 discusses the conclusions and future scope of work.

.



Chapter 2

Theoretical Foundations and Technical Prerequisites

CHAPTER 2

THEORETICAL FOUNDATIONS AND TECHNICAL PREREQUISITES

An in-depth understanding of architectural design principles, matrix operations, and toolchains is essential for developing high-performance ML hardware. This chapter outlines the foundational concepts required to carry out FPGA architectural exploration using the VTR toolchain and to design the domain-specific PCA accelerator, Manojavam. It begins by covering FPGA architecture types relevant to ML applications, followed by essential matrix computation techniques and PCA formulations. The chapter also introduces key tools such as VTR, Vivado, OpenLane, and CACTI, which were instrumental in modeling, implementing, and evaluating the proposed designs.

2.1 Fundamentals of FPGA Architecture

Field-Programmable Gate Arrays (FPGAs) are a class of integrated circuits that provide post-fabrication reconfigurability at the hardware level. Unlike fixed-function ASICs (Application-Specific Integrated Circuits), FPGAs can be programmed to implement any digital logic circuit by configuring a rich array of programmable logic blocks, interconnects, and embedded hard blocks. This section provides an in-depth exploration of the foundational architecture of FPGAs, with a focus on the key structural elements and their evolution. Fig.2.1 highlights the early v/s modern style FPGA blocks that integrate many hardblocks on the fabric.

2.1.1 Configurable Logic and Logic Clusters

The smallest unit of computation in an FPGA is the Basic Logic Element (BLE), which typically comprises a K-input Lookup Table (K-LUT), a flip-flop (FF), and multiplexers to select between combinational and registered outputs. The K-LUT is a truth table stored in SRAM that can implement any K-input Boolean function. For example, a 6-LUT can encode any 6-input logic by programming 64 SRAM bits. Fig.2.2 depicts the implementation of a 4-LUT at a transistor level.

Modern BLEs implement fracturable logic to increase logic packing efficiency[22]. A fracturable 6-LUT can operate as either a single 6-input LUT or two 5-input LUTs sharing inputs[2]. This flexibility reduces unused logic and enables higher performance.

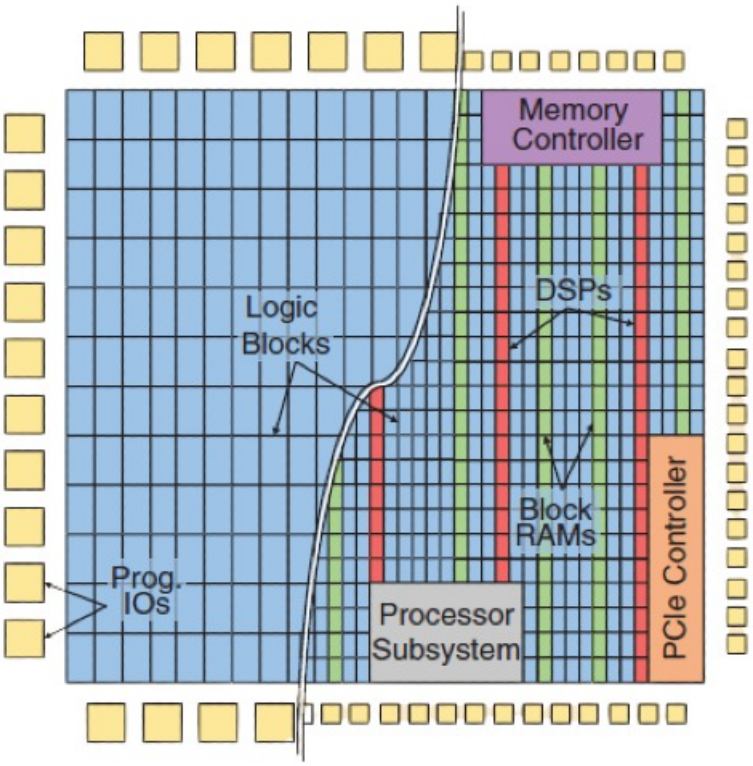


Figure 2.1: FPGA Architecture Overview : Early v/s Modern[7]

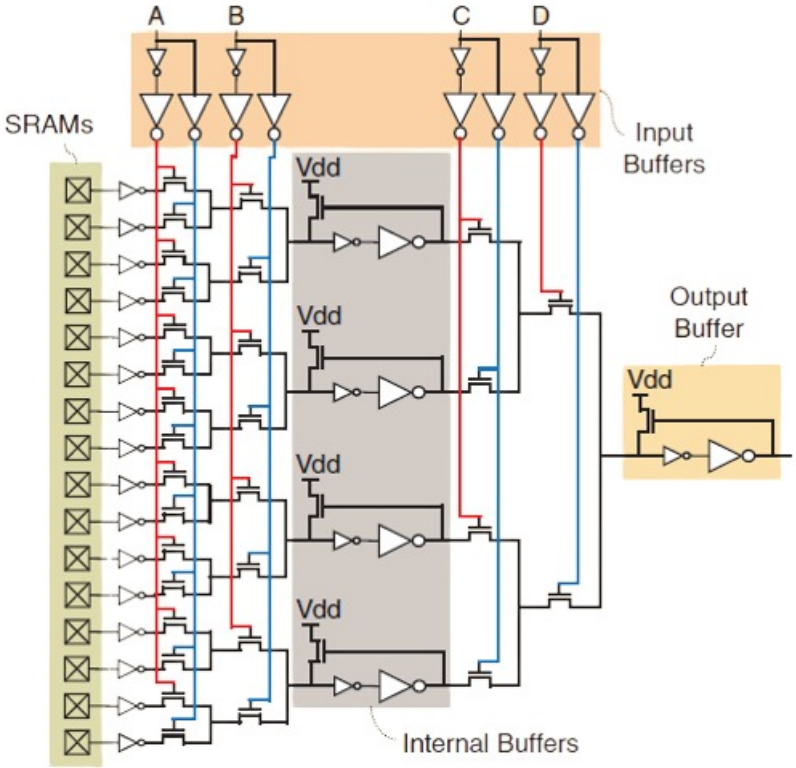


Figure 2.2: Transistor Implementation of a 4-LUT[7]

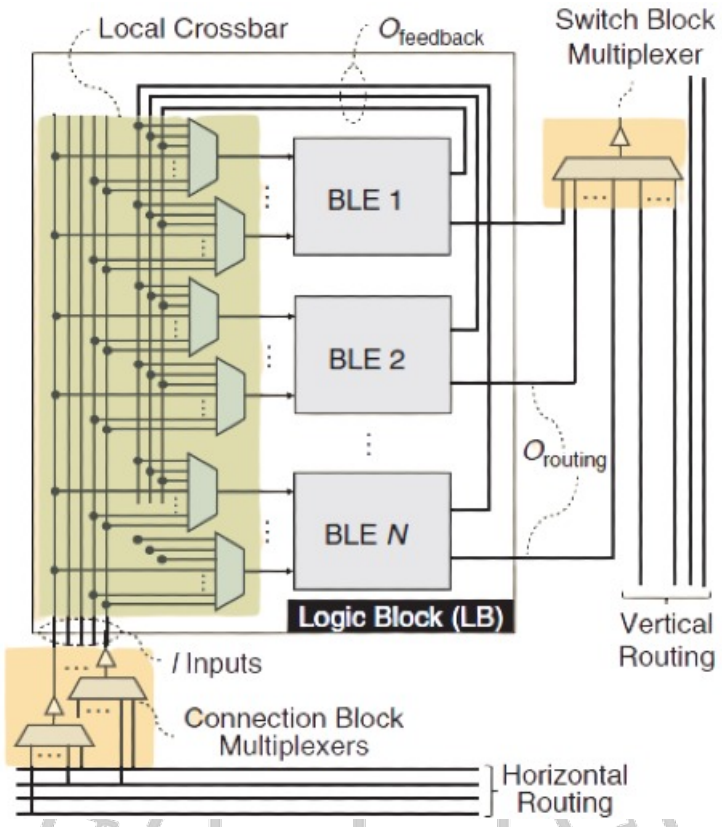


Figure 2.3: LAB Internal Architecture[7]

Intel's Adaptive Logic Modules (ALMs) and Xilinx's slice-based architectures exemplify this trend, enabling efficient use of logic resources.

BLEs are grouped into larger clusters called Logic Array Blocks (LABs) or Configurable Logic Blocks (CLBs). Each LAB may contain 8 to 32 BLEs and includes a local interconnect network for intra-block communication. This local interconnect is typically implemented using multiplexers arranged as full or partial crossbars[23][24]. As the number of BLEs per LAB increases, more signal routing can be localized, reducing pressure on global routing resources and improving performance. The internal architecture of the LAB is given in Fig.2.3.

The design trade-off in LAB size lies between logic utilization, routing efficiency, and critical path delay. Larger LABs reduce inter-block communication but increase complexity in placement and delay modeling.

2.1.2 Programmable Routing Architecture

The dominant routing style in modern FPGAs is the island-style architecture. In this topology, logic blocks (LABs) are arranged in a 2D grid and are surrounded by horizontal

and vertical routing channels. These channels consist of wire segments of varying lengths and programmable switches.

Three main elements define the island-style routing fabric:

1. **Connection Blocks:** Interface logic block I/Os to routing channels.
2. **Switch Blocks:** Enable programmable interconnection between horizontal and vertical wires.
3. **Wire Segments:** Allow signal propagation across the chip, often spanning 1, 4, or 16 logic blocks.

Efficient switch block designs (e.g., Wilton pattern) and wire-length tuning reduce signal delay and area overhead[25]. To address long-distance routing delays, modern FPGAs also employ hierarchical wiring and segment-bypassing strategies. The most popular routing style architecture, the island style routing architecture, is depicted in Fig.2.4. This style of routing is employed in the test architectures employed in the study, which we will see in the following sections.

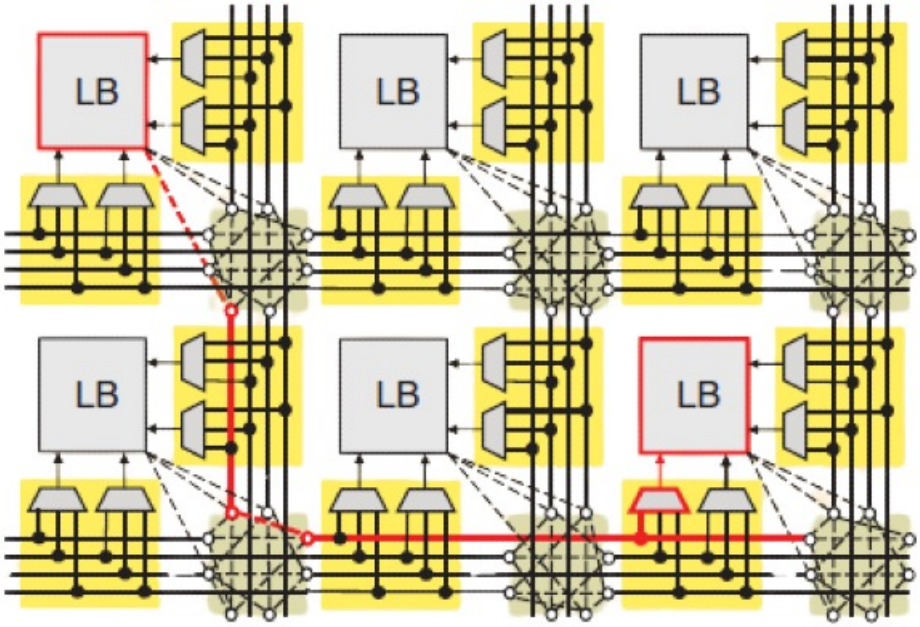


Figure 2.4: Island Style Programmable Routing Architecture[7]

Programmable switches form the core of reconfigurable routing. Early designs used simple pass transistors, but these incur high resistance and delay[26]. Buffered multiplexers are now preferred, where multiple pass transistors feed a buffer, improving speed

and electrical isolation[27][28]. This design limits signal injection points but significantly enhances signal integrity and performance.

2.1.3 Embedded Arithmetic and Carry Chains

Arithmetic operations such as addition and multiplication are common in FPGA workloads. Although they can be implemented with LUTs, this approach is inefficient. Instead, modern FPGAs integrate hardened arithmetic logic like carry chains directly within BLEs.

Carry chains enable fast ripple-carry and carry-skip adders. They allow the carry-out of one BLE to directly feed the carry-in of the next without passing through the general routing fabric, minimizing delay. Xilinx FPGAs use carry look-ahead logic in Versal devices, whereas Intel's Agilex architecture implements two-bit arithmetic per logic element. These dedicated structures are optimized for speed and minimize logic usage for arithmetic-heavy workloads.

Advanced carry-chain structures like carry-skip and carry-lookahead adders are hardened to further accelerate arithmetic paths. Carry logic is commonly coupled with sum and propagate generation logic inside LUTs, and is tightly integrated into the physical layout to reduce interconnect delay. These carry networks are also exploited to accelerate custom DSP-style operators and are essential for low-precision multiply-accumulate (MAC) implementations.

These features are especially beneficial for domains requiring high arithmetic intensity, such as digital signal processing, image processing, and machine learning accelerators.

2.1.4 On Chip Memory: BRAM and LUT-RAM

To address diverse memory needs, FPGAs embed Block RAMs (BRAMs) with configurable size, depth, and access ports[29][30]. A typical dual-port BRAM includes row/-column decoders, sense amplifiers, and output multiplexers. Capacities range from 2Kb to 20Kb, with typical configurations supporting 1R/1W or 2R/W modes.

Peripheral logic allows BRAMs to operate in various data widths and depths (e.g., 32x64-bit or 64x32-bit)[31][32]. Input/output ports are pipelined to ease timing closure, and routing interfaces are carefully optimized to reduce area and delay.

In addition to BRAMs, FPGA vendors allow LUTs to function as small distributed RAMs. By configuring LUT truth tables and adding minimal write logic, logic blocks

can operate as single or dual-port LUT-RAMs (e.g., 64x10-bit). This memory is ideal for register files and coefficient storage. However, only a subset of logic blocks typically supports LUT-RAM to balance area cost[33].

2.1.5 DSP Hard Blocks

FPGAs embed hardened Digital Signal Processing (DSP) slices to accelerate multiply-accumulate (MAC) operations. These slices support pre-adders, multipliers, accumulators, and configurable pipelining[34]. MACs can be cascaded to implement dot products, convolution, and matrix multiplications efficiently.

DSP slices reduce logic utilization and boost performance for compute-intensive tasks such as deep learning inference, FFTs, and image processing.

2.1.6 Programmable IO Blocks

IOBs are specialized blocks that manage communication between the FPGA core and external devices. They support a wide range of electrical standards (e.g., LVCMOS, SSTL, LVDS) and features like:

1. Programmable drive strength and slew rate
2. On-die termination
3. Programmable delay lines
4. Differential signaling and serial transceivers

Advanced IOs integrate DDR memory controllers and serial links (e.g., PCIe, Ethernet) with hardened logic. FPGA IOs are grouped into banks, each operating at configurable voltages. These features make FPGAs ideal for communication-heavy applications.

2.2 Target FPGA Architectures Used in the Study

This study explores and benchmarks three FPGA target architectures, each modeled using VTR (Verilog-to-Routing) XML architecture description files. These include: (1) an Intel Stratix-10-like architecture that serves as a high-performance, industry-grade baseline, and two custom arithmetic-optimized variants, namely (2) a 4-bit Single Carry Chain architecture and (3) a 4-bit Double Carry Chain architecture. Each design showcases unique internal arithmetic structures and carry chain implementations aimed at

evaluating low-precision arithmetic performance, relevant to domains such as deep learning inference.

2.2.1 Intel Stratix-10-like Architecture

The Stratix-10-like architecture models a high-end commercial FPGA design. The logic fabric consists of Logic Array Blocks (LABs), each comprising 10 Adaptive Logic Modules (ALMs). Every ALM contains a 6-input Lookup Table (6-LUT) that is fracturable into two 5-LUTs, supporting rich logic packing. Each ALM exposes 8 input pins and 4 output pins, which may be optionally registered using internal flip-flops[35]. Fig.**??** highlights the block diagram of Intel's Stratix-10 high level block diagram.

The key features of the Intel Stratix-10 like architecture are-

1. **Arithmetic Mode Support:** Each 5-LUT can be fractured into two 4-LUTs, enabling two bits of addition per ALM in arithmetic mode. Thus, a LAB can support 20 bits of addition in a single carry chain across 10 ALMs.

2. **Single Carry Chain:** The carry chain is linear, spanning the 10 ALMs within a LAB. The carry-out (*cout*) of one ALM feeds directly into the carry-in (*cin*) of the next, enabling fast arithmetic accumulation.

3. **Direct Connections:** ALM outputs feed both into neighboring LABs (left and right) and back into the LAB's own input ports. This enhances localized reuse and dataflow.

4. **Memory Support:** Each LAB can interface with a 20 Kb embedded RAM block, configurable in multiple true and simple dual-port modes.

5. **DSP Integration:** A 27x27 DSP block (fracturable into 18x19 blocks) is embedded within the tile grid to support high-throughput MAC operations.

This architecture balances general-purpose logic density with specialized arithmetic acceleration, making it suitable as a benchmark for more customized variants.

2.2.2 4-bit Single Carry Chain Architecture

This custom architecture builds on the Stratix-10 baseline but modifies the arithmetic capabilities for enhanced support of low-precision adders. It retains the same LAB struc-

ture—10 ALMs per LAB, 6-LUTs fracturable into two 5-LUTs—but introduces further fracturing into 3-LUTs[36].

Key features of this architecture include-

1. **Enhanced Arithmetic Fracturing:** In half the ALMs (ALM[0] to ALM[4]), each 5-LUT can be split into two 4-LUTs, and each 4-LUT further into two 3-LUTs. This yields 8 effective 3-LUTs per ALM, supporting 4-bit adders per ALM.

2. **Single Carry Chain:** A single carry chain spans only the first 5 ALMs (ALM[0] to ALM[4]), providing a total of 20 bits of low-precision addition per LAB.

3. **Optimized for Compact Arithmetic:** This configuration allows dense packing of small adders using minimal resources, reducing delay and improving energy efficiency for small-scale DSP and ML operations.

This design evaluates how resource utilization and delay scale when packing narrow adders and embedding a shallow carry propagation chain.

2.2.3 4-bit Double Carry Chain Architecture

Expanding upon the single-chain variant, this architecture introduces a dual carry chain structure to exploit parallelism in arithmetic computation[36].

Key features of this architecture include-

1. **Double Carry Chains:** The LAB is split into two independent carry chains:
   - Chain 1 spans ALM[0] to ALM[4]
   - Chain 2 spans ALM[5] to ALM[9]

2. Each chain operates independently, with separate *cin* and *cout* ports, enabling concurrent execution of two 4-bit addition operations.

3. **Full Arithmetic Fracturing:** Similar to the single chain, each ALM supports decomposition into eight 3-LUTs, enabling 4-bit addition within a compact area.

4. **Routing Considerations:** The architecture includes direct interconnects between the carry-out of one LAB and the carry-in of the LAB below for each chain, supporting vertical propagation of arithmetic results.

This configuration is ideal for evaluating parallelism vs. resource duplication trade-offs, particularly in scenarios where low-precision arithmetic operations can be concurrently dispatched, such as SIMD-style ML workloads.

2.2.4 Summary of Architectural Differences

Table.2.1 summarizes the architectural differences between the Intel Stratix-10-like, 4-bit single carry chain and 4-bit double carry chain architectures.

Table 2.1: FPGA Architectural Differences between Test Architectures

Feature	**Stratix-10-Like**	**4-Bit Single Chain**	**4-Bit Double Chain**
ALMs per LAB	10	10	10
Fracturable LUT Structure	6-LUT to 5-LUT to 4-LUT	6-5-4-3 LUT (half ALMs)	6-5-4-3 LUT (all ALMs)
Carry Chains	1 full LAB wide	1 in ALM[0] to ALM[4]	2 (ALM[0-4] and ALM[5-9])
Arithmetic Parallelism	Sequentual	Narrow Serial (5 ALMs)	Dual Parallel Chains
Target Use-Case	General Logic and DSP	Low Precision MACs	Parallelized Arithmetic

These architectures provide a valuable testbed for understanding how architectural customization affects logic packing, delay, and arithmetic throughput in FPGA-based computation.

2.3 HDL Benchmarks Employed in the Study

This study utilizes a series of parameterized Verilog benchmarks to evaluate the synthesis, placement, routing, and delay characteristics of different FPGA architectures under arithmetic and signal processing workloads. These benchmarks are representative of typical low-level compute patterns found in many machine learning and digital signal processing (DSP) applications. Each benchmark is crafted to stress particular architectural resources such as carry chains, routing networks, and DSP blocks. The categories are described below.

2.3.1 2-Level Adder Trees

The 2-level adder tree benchmark represents a fundamental building block for arithmetic reductions and accumulation operations. It takes eight parallel inputs, grouped in binary pairs, and recursively sums them in a two-level hierarchy using combinational adders. The structure implements three layers:

1. **Level 3:** Performs four pairwise additions of the eight inputs.

2. **Level 2:** Aggregates the four intermediate results into two partial sums.

3. **Level 1:** Produces the final result from the two intermediate sums.

Each addition stage is implemented using a generic adder-tree-branch module with a configurable EXTRA-BITS parameter to accommodate bit growth through each level of the hierarchy. The bit-width of inputs is configurable, enabling stress-testing across low- to medium-precision operand sizes.

This design serves as an effective benchmark for:

1. **Carry chain utilization:** As each addition requires ripple or look-ahead carry logic.
2. **Critical path analysis:** Timing accumulates from multiple levels of combinational logic.
3. **Area-delay tradeoffs:** Bit growth and tree depth impact routing congestion and synthesis area.

The register stages at the input ensure timing closure under a pipelined regime, and the configuration can be toggled to emulate either 2-level or 3-level adder behavior, providing flexibility in benchmarking deeper vs. shallower arithmetic pipelines.

2.3.2 3-Level Adder Trees

The 3-level adder tree benchmark extends the arithmetic hierarchy of the 2-level design, implementing a complete three-stage binary reduction to compute the sum of eight input operands. Each operand is of configurable bit-width, allowing this benchmark to target multiple datapath sizes typically encountered in hardware accelerators, such as in digital signal processing (DSP) or MAC (multiply-accumulate) structures in ML workloads.

The benchmark operates in the following sequence:

1. **Level 3:** Computes four pairwise sums from the eight input signals.
2. **Level 2:** Takes the four L3 results and reduces them to two partial sums.
3. **Level 1:** Produces the final output by summing the two L2 outputs.

Each stage employs a parameterized adder-tree-branch that accounts for bit growth due to binary addition. This is controlled via the EXTRA-BITS parameter, which increases by one bit per level to prevent overflow and maintain accuracy.

Notable features of the 3-level adder trees include:

1. **Fully Combinational Addition Path:** Unlike pipelined designs, this benchmark computes the final output purely through combinational logic from the input stage to the output in a single cycle, stressing the critical path delay.

2. **Stress on Carry Chains:** The deeper hierarchy increases the load on carry propagation networks, allowing accurate benchmarking of architectures with varying carry chain optimizations (e.g., single vs. double chain).

3. **Bit-Width Scalability:** The design is generic across operand sizes (e.g., 4-bit, 8-bit, 16-bit) and models how delay and logic utilization scale with increasing precision.

The 3-level tree structure is representative of workloads requiring high fan-in reductions, such as in deep learning accumulators, dot product units, and reduction phases of sum-pooling layers. It also provides insight into how FPGA routing and LUT utilization scale with deeper arithmetic pipelines, and how carry propagation affects synthesis timing.

2.3.3 Unpipelined FIR Filter with Hardblock Multipliers

This benchmark models a classic Finite Impulse Response (FIR) filter implemented in an unpipelined configuration using dedicated DSP hardblocks for multiplication. It emulates a 10-tap symmetric FIR filter, which leverages the property of coefficient symmetry to reduce computational redundancy by only requiring five unique coefficients. The architecture processes a sliding window of ten input samples, pairing symmetric inputs and summing them in a series of five initial additions. These pairwise sums are subsequently multiplied by the respective filter coefficients using multiplier units, assumed to map directly to FPGA DSP slices (e.g., 18×18 or 27×18 multipliers).

The multiplication outputs are then passed through a multi-level adder tree, where they are combined using a series of registered addition modules to produce the final output. Although input and output registers are present for signal stability and timing, the computation pipeline itself remains unpipelined. As such, the entire multiply-accumulate (MAC) computation occurs in a combinational path with sequential output capture, stressing the design's critical path and highlighting the FPGA architecture's ability to

meet timing without deeper pipelining. Control flow is managed using a valid-signal propagation pipeline to ensure data consistency across the input and output stages.

This benchmark is essential for evaluating how an FPGA architecture handles computationally intensive DSP workloads without the benefits of pipelining. It is particularly useful for analyzing critical path delays, synthesis area, logic and DSP block utilization, and the efficiency of routing under MAC-dominated loads. Moreover, the benchmark provides a reference point for comparing with pipelined implementations in terms of latency, throughput, and resource efficiency, especially in applications where minimizing dynamic power or leveraging dedicated multipliers is a key design goal.

2.3.4 Pipelined FIR Filter with Hardblock Multipliers

The pipelined FIR filter benchmark builds upon the foundational structure of its unpipelined counterpart but introduces staged computation through register insertion, enabling improved throughput and deeper timing closure. This design also implements a symmetric 10-tap FIR filter with five unique coefficients and exploits coefficient symmetry to minimize the number of multiplications required. Input samples are processed through a shift register pipeline, from which symmetric pairs are summed and then multiplied with corresponding coefficients using dedicated hardware multipliers that align with DSP blocks typically available in commercial FPGA fabrics.

What distinguishes this version from the unpipelined design is the strategic placement of registers between every major computational stage—after the symmetric additions, after each multiplication, and throughout the reduction adder tree. These pipelining registers help shorten critical paths and allow the design to run at higher clock frequencies, making it more suitable for high-throughput applications such as real-time signal processing or embedded ML inference. Each computation level performs only a small segment of the total work per clock cycle, resulting in reduced combinational delay and improved timing characteristics.

The control pipeline tracks data validity using a dedicated valid signal shift register, which aligns output generation with the pipeline depth. This structured staging increases latency slightly due to the additional cycles required for data propagation through each stage, but it greatly enhances throughput as new inputs can be accepted every clock cycle. This benchmark is ideal for evaluating how well an FPGA architecture supports pipelined arithmetic workloads, especially in terms of logic utilization, DSP block effi-

ciency, and register overhead. Moreover, it demonstrates how architectural pipelining can be leveraged to scale MAC-intensive operations while maintaining high operating frequencies.

2.3.5 Unpipelined FIR Filter with Carry Chains

This variant of the unpipelined FIR filter is structurally identical to its hardblock-based counterpart but replaces dedicated multipliers with synthesized Wallace tree multipliers constructed entirely from single-bit adders. By decomposing the multiplication operation into a series of partial product generations and reductions, this design shifts the computational burden from DSP blocks to the FPGA's general-purpose logic fabric—specifically the carry chains. The benchmark still implements a symmetric 10-tap FIR filter using five unique coefficients, with input samples processed through a sliding window and summed in symmetric pairs. However, each multiplication is replaced by an 18-bit Wallace tree multiplier composed of carry-optimized addition layers.

This architectural substitution significantly alters the behavior of the design with respect to synthesis and timing. Instead of mapping to dedicated DSP slices, the design now stresses the carry propagation network and LUT fabric, especially in architectures optimized for low-precision arithmetic via enhanced carry chain logic. Without pipelining, all levels of the multiply-accumulate (MAC) computation execute within a single clock cycle. As a result, the critical path is dominated by the depth of the Wallace tree multiplier and subsequent adder stages, providing a valuable stress test for logic delay and routing complexity in FPGA fabrics.

This benchmark is particularly useful for evaluating how well an FPGA architecture supports synthesized arithmetic under carry-intensive workloads. It reveals the relative efficiency of different carry chain implementations—such as single vs. double carry chains—and the extent to which these optimizations impact timing closure and logic density. Moreover, by replicating a DSP workload using general logic resources, this test exposes trade-offs between logic usage and DSP conservation, especially in scenarios where hardblocks are limited or reserved for higher-precision operations.

2.3.6 Pipelined FIR Filter on Carry Chains

This FIR benchmark extends the carry chain-based multiplier implementation by introducing a fully pipelined computation structure, enabling higher clock frequencies and

improved throughput. As in the unpipelined version, the core multiplication operations are synthesized using Wallace tree multipliers built from single-bit adders, rather than utilizing dedicated DSP hardblocks. These multipliers rely on layers of carry chain-accelerated adders to construct 18-bit multiplication results through partial product accumulation and reduction. The FIR filter retains a 10-tap symmetric structure with five unique coefficients, and input samples are processed through a series of shift registers that maintain a sliding window of data.

Each stage of the computation—from the symmetric additions and Wallace tree multipliers to the reduction adder tree—is separated by clocked register stages, creating a deeply pipelined data path. This staged computation reduces the depth of combinational logic per cycle and enables the design to meet tighter timing constraints on FPGA fabrics. The use of carry chain-based Wallace multipliers allows the design to test the efficiency of FPGA logic structures in scenarios where DSP blocks are not available or are reserved for other purposes. The output valid signal is aligned with the data path using a pipelined control register, ensuring synchronized streaming of results.

This benchmark is highly illustrative of how FPGAs can be leveraged to perform DSP-style operations purely using general-purpose logic and carry chains. It stresses not only the FPGA's carry propagation infrastructure but also its ability to maintain high throughput under deep pipelining. The design is particularly relevant for evaluating low-precision arithmetic acceleration strategies, where resource efficiency and carry chain utilization directly impact performance and power efficiency.

The pipelined FIR filter benchmark builds upon the foundational structure of its unpipelined counterpart but introduces staged computation through register insertion, enabling improved throughput and deeper timing closure. This design also implements a symmetric 10-tap FIR filter with five unique coefficients and exploits coefficient symmetry to minimize the number of multiplications required. Input samples are processed through a shift register pipeline, from which symmetric pairs are summed and then multiplied with corresponding coefficients using dedicated hardware multipliers that align with DSP blocks typically available in commercial FPGA fabrics.

What distinguishes this version from the unpipelined design is the strategic placement of registers between every major computational stage—after the symmetric additions, after each multiplication, and throughout the reduction adder tree. These pipelining

registers help shorten critical paths and allow the design to run at higher clock frequencies, making it more suitable for high-throughput applications such as real-time signal processing or embedded ML inference. Each computation level performs only a small segment of the total work per clock cycle, resulting in reduced combinational delay and improved timing characteristics.

The control pipeline tracks data validity using a dedicated valid signal shift register, which aligns output generation with the pipeline depth. This structured staging increases latency slightly due to the additional cycles required for data propagation through each stage, but it greatly enhances throughput as new inputs can be accepted every clock cycle. This benchmark is ideal for evaluating how well an FPGA architecture supports pipelined arithmetic workloads, especially in terms of logic utilization, DSP block efficiency, and register overhead. Moreover, it demonstrates how architectural pipelining can be leveraged to scale MAC-intensive operations while maintaining high operating frequencies.

2.4 Architecture-Benchmark Evaluation Metrics

To evaluate the effectiveness of each FPGA architecture under study, a set of quantitative metrics were employed to characterize the performance and resource efficiency of the HDL benchmarks mapped onto them. The primary metrics considered are operational frequency, area (in Minimum Width Transistor Area or MWTA units), and critical path delay. These parameters provide a holistic view of architectural efficiency in supporting arithmetic and signal-processing-dominated hardware designs.

Operational Frequency denotes the maximum clock speed at which the synthesized design can function reliably. It is influenced by logic depth, routing congestion, and architectural support for pipelining and carry propagation. A higher operational frequency typically reflects better timing closure, which is crucial for real-time ML inference and high-throughput data processing.

Area in MWTA units represents the physical resource cost of implementing the benchmark design on the FPGA fabric[37]. It quantifies the required logic, routing, and hard block utilization in terms of minimum-width transistor area equivalents, providing a normalized view of silicon efficiency. Area optimization is especially relevant in ML accelerators where packing more compute units per chip can dramatically improve inference throughput.

Critical Path Delay, measured in nanoseconds, refers to the longest propagation delay between input and output in the combinational logic path[38]. It directly impacts the achievable clock frequency and therefore the latency of computations. Designs with deep carry chains, long adder trees, or complex multiplier logic tend to have higher critical path delays. Reducing this delay is crucial for supporting low-latency ML tasks, such as real-time vision or audio processing.

Collectively, these metrics allow for meaningful comparisons between FPGA architectures with varying logic organization, carry chain capabilities, and hard block integration. In the context of machine learning hardware, where latency, throughput, and energy efficiency are paramount, these metrics help identify architectural features that offer superior trade-offs for different classes of ML workloads.

2.5 VTR Toolchain

The Verilog-to-Routing (VTR) toolchain is an open-source CAD framework developed to explore FPGA architecture, CAD algorithms, and mapping strategies through end-to-end synthesis and implementation. Its modular and customizable structure makes it a powerful platform for academic and experimental research into FPGA logic structures, placement and routing techniques, and architecture-aware optimizations[39][36].

The motivation behind VTR is to bridge the gap between Verilog-based digital design and physical FPGA layout analysis. It enables rapid prototyping and benchmarking of custom FPGA architectures without relying on proprietary tools, allowing researchers to test architectural modifications such as specialized carry chains, fracturable LUTs, or heterogeneous hard block integration.

The VTR CAD flow, as shown in Fig.2.5 begins with Verilog input and proceeds through synthesis, logic optimization, technology mapping, packing, placement, routing, and final analysis. This sequence emulates a full CAD flow while offering hooks for architectural experimentation at each stage.

The flow begins with Yosys, an open-source Verilog synthesis suite[40]. Yosys parses the Verilog HDL, performs initial optimizations, and generates a structural netlist in BLIF (Berkeley Logic Interchange Format), a textual format that represents gate-level designs. BLIF is widely used as a standard intermediate format in logic synthesis and academic CAD tools.

The generated BLIF netlist is then passed to ABC, a logic optimization and technol-

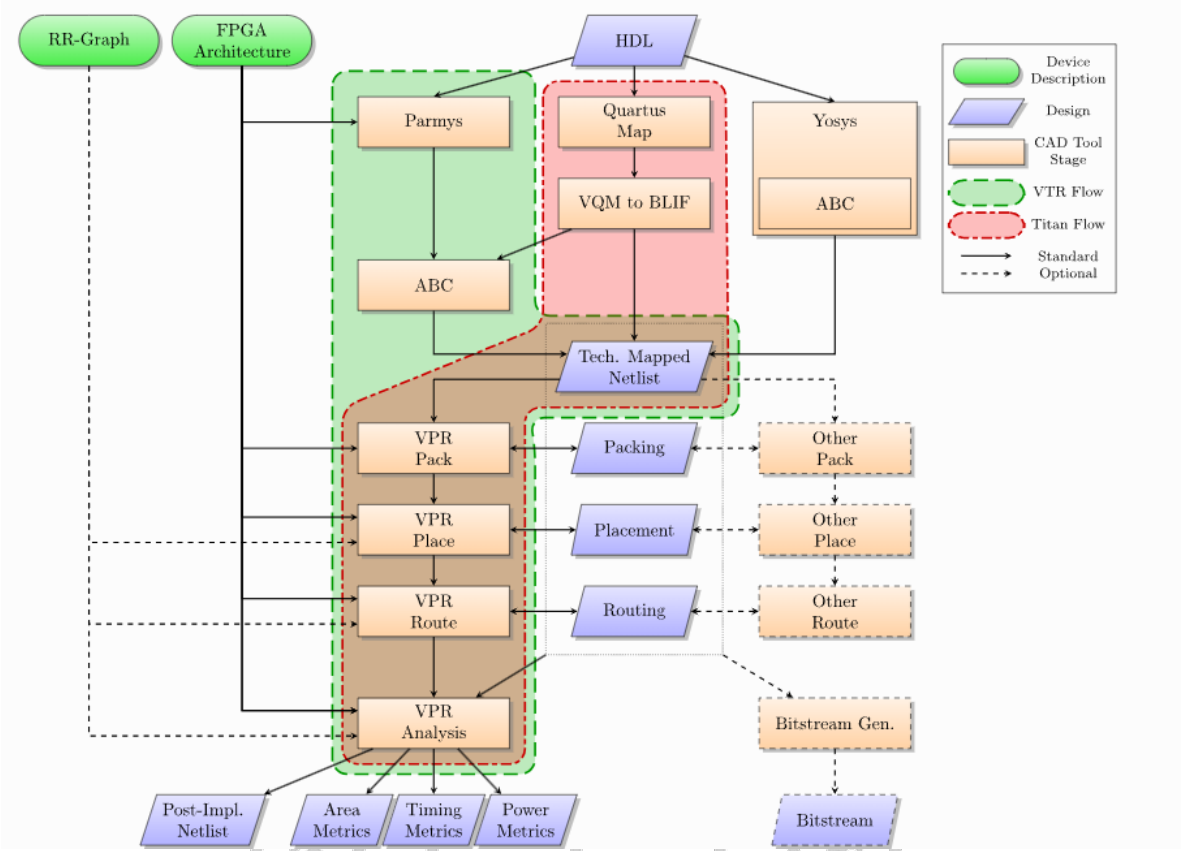


Figure 2.5: VTR CAD Flow[39]

ogy mapping tool developed at UC Berkeley[41]. ABC performs logic restructuring and maps logic functions onto basic gate primitives compatible with FPGA LUTs[42]. It also reduces logic depth and performs retiming optimizations where applicable. The output of ABC is a further optimized BLIF file used by VTR's core tools.

Next, the VPR (Versatile Place and Route) engine takes over. VPR performs three major tasks:

1. **Packing:** Groups logic elements (e.g., LUTs, FFs) into logic clusters (CLBs or LABs) based on FPGA architecture constraints.

2. **Placement:** Assigns packed logic clusters to specific physical locations in the FPGA grid.

3. **Routing:** Connects placed logic blocks using the programmable interconnect and switch boxes, while adhering to the routing architecture defined in the FPGA model.

VPR concludes with analysis, reporting timing, area (in MWTA), and power estimates. These results form the basis for comparative benchmarking of various architecture

configurations[43].

One of VTR's most powerful features is its architecture description framework. Custom FPGA architectures are described using XML files that specify logic block structures, interconnect topologies, routing switch types, I/O standards, timing models, and cluster configurations. These XML descriptions are parsed and interpreted by VPR, enabling researchers to design and evaluate unconventional architectures—such as carry-chain-enhanced FPGAs, multi-CLB tiles, or DSP-rich fabrics—without modifying the underlying toolchain source code.

Overall, the VTR toolchain provides a complete, transparent, and configurable flow that is essential for academic research and architectural benchmarking. Its integration of synthesis, logic mapping, placement, and routing, combined with custom architecture support, makes it an indispensable framework for the exploration and evaluation of FPGA designs tailored for emerging compute workloads like machine learning.

2.6 Principal Component Analysis (PCA)

2.6.1 Introduction and Evaluation Metrics

Principal Component Analysis (PCA) is a statistical method used to transform a set of correlated variables into a new set of uncorrelated variables known as principal components. These components are linear combinations of the original variables, arranged in order of decreasing variance. The primary goal of PCA is to identify the most significant patterns within the data while reducing its dimensionality, which is particularly useful in scenarios where data interpretation, visualization, and computational efficiency are essential. The essence of PCA lies in projecting high-dimensional data onto a new coordinate system where the dimensions are ranked based on their contribution to the dataset's overall variance[44][45].

PCA is an unsupervised learning technique widely used in machine learning, computer vision, and scientific computing. It is especially valuable in dealing with datasets where multicollinearity exists, meaning that features are highly correlated. By identifying a new set of orthogonal basis vectors, PCA effectively removes redundancy and ensures that each principal component captures distinct, non-overlapping information about the dataset[46]. This is particularly beneficial in applications such as image compression, where PCA enables a significant reduction in the number of pixels required to reconstruct

an image while preserving its essential structure, and in finance, where it helps identify latent factors influencing asset prices.

Consider a multi-feature dataset comprising M records and N features. Then, the PCA algorithm begins by standardizing the dataset to ensure that each feature contributes equally to the analysis. This involves subtracting the mean of each feature and dividing by its standard deviation, thereby transforming the data into a zero-mean, unit-variance form. Next, the covariance matrix, an NxN symmetric matrix capturing the pairwise relationships between features, is computed. This matrix provides insight into how different features vary with respect to one another. Following this, the eigenvalues and eigenvectors of the covariance matrix are determined through eigenvalue decomposition or Singular Value Decomposition (SVD). The eigenvectors represent the principal components—orthogonal directions in feature space along which data exhibits maximum variance—while the corresponding eigenvalues indicate the amount of variance captured by each component. The eigenvalues are then sorted in descending order, and the top k eigenvectors (corresponding to the largest eigenvalues) are selected to form a reduced k-dimensional subspace. Finally, the original dataset is projected onto these principal components, yielding a transformed representation where the most significant variance is retained while reducing the overall dimensionality.

The algorithm for PCA is given below-

1. **Input:** Input dataset D with M rows and N features
2. Center the dataset : For every feature, subtract the mean of the feature from each point to center the dataset about the mean $x_i \longleftarrow x_i - \frac{1}{N}\sum x_k$
3. Compute the covariance matrix of the scaled down dataset $C = X^T X$, where X^T is the transpose of the input dataset and Xis the input dataset
4. Perform Singular Value Decomposition (SVD) on the covariance matrix C to procure the eigenvalues and eigenvectors of the covariance matrix $C = VD^2V^T$, where V is the matrix whose columns are the eigenvectors of the covariance matrix and D is the diagonal matrix whose elements are the square of the eigenvalues
5. Multiplying X and V would result in the principal components, and the eigenvectors of the top k eigenvalues are selected as the principal components

6. **Output:** Required Principal Components from the Input dataset

Once the eigenvalues and their corresponding eigenvectors have been computed, the next crucial step in PCA is selecting the appropriate number of principal components, k, that will effectively reduce the dimensionality while preserving most of the data's variance. There are two widely used methods for determining k: the variance contribution ratio and the cumulative variance contribution ratio[47].

The variance contribution ratio (VCR) is a measure of how much variance each principal component explains relative to the total variance in the data. Given that PCA aims to retain the most significant patterns in the dataset, components with higher variance contributions are more valuable. The variance contribution ratio for the i^{th} principal component is given by:

$$P_i = \frac{\lambda_i}{\sum_{k=1}^{N} \lambda_k} \tag{2.1}$$

where i is the eigenvalue associated with the i^{th} principal component, and $\sum_{k=1}^{N} \lambda_k$ is the sum of all eigenvalues, which is directly proportional to the total variance of the dataset. By sorting the eigenvalues in descending order, the first few principal components will have the highest variance contribution ratios. Typically, a threshold is set, and only the components with a variance contribution above this threshold are retained. For instance, if the first three components have variance contributions of 40%, 30%, and 15%, a common heuristic might be to select the first two or three components since they contribute the most to the total variance.

Instead of analyzing individual variance contributions, the cumulative variance contribution ratio (CVCR) provides a more holistic criterion for choosing k. It sums the variance contributions of the top k principal components to ensure that a sufficient proportion of the total variance is preserved. The cumulative variance contribution ratio for the top k components is given by:

$$CP_i = \frac{\sum_{k=1}^{i} \lambda_k}{\sum_{k=1}^{N} \lambda_k} \tag{2.2}$$

The goal is to select k such that $CVCR_k$ exceeds a predefined threshold, typically 95% or 99%. This ensures that the selected principal components retain most of the dataset's variance while reducing dimensionality.

For example, suppose the cumulative variance contribution for the first four principal components is 85%, 92%, 97%, and 99%. If an application requires at least 95% variance retention, selecting the first three components would be sufficient.

In real-world scenarios, a scree plot (a plot of eigenvalues versus component index) is often used to visualize variance contributions. The "elbow point" of this plot, where eigenvalues begin to level off, serves as an intuitive indicator of an appropriate k. Alternatively, domain-specific constraints, computational efficiency considerations, and the trade-off between information preservation and dimensionality reduction influence the final choice of k.

2.6.2 Case Study on Principal Component Analysis

As an example, consider the wine dataset from the University of California-Irvine's ML repository[48]. The Wine Recognition dataset from the UCI Machine Learning Repository is a classic classification dataset used to distinguish between three varieties of wine grown in the same region of Italy. Based on the results of chemical analysis, each sample represents a wine derived from one of the three cultivars. The dataset aims to evaluate how well different physicochemical characteristics can be used to identify the grape variety. The dataset comprises of the below feature set, as shown in Table.2.2.

Table 2.2: Summary-Original Features

S.No	**Feature Index**	**Feature**
1	F1	Alcohol
2	F2	Malic Acid
3	F3	Ash
4	F4	Alcalinity of Ash
5	F5	Magnesium
6	F6	Total Phenols
7	F7	Flavanoids
8	F8	Non flavanoid phenols
9	F9	Proanthocyanins
10	F10	Color Intensity
11	F11	Hue
12	F12	OD280 / OD315 for diluted wines
13	F13	Proline

Then, the covariance matrix of the dataset, providing an overview on the inter dependencies between the different features in the dataset, is shown in the Figure 2.6

Figure 2.6: Covariance Matrix

Further, the covariance matrix is decomposed into its eigenvalues and eigenvectors using Singular Value Decomposition, and the EVCR and CVCR across all the eigenvalues are calculated after arranging them in the descending order of their magnitude. Analysis of the variance contribution and cumulative variance contribution plots yield that the first 6 principal components would suffice and explain 80% of the variance of the dataset.

The eigenvalues of the covariance matrix are computed and displayed in Table 2.3. The eigenvectors corresponding to the top k eigenvalues are obtained. The choice of k will depend on the plot of explained variance ratio v/s the number of principal components

chosen as shown in Figure 2.7

Table 2.3: Eigenvalues of Covariance Matrix

$\mathbf{PC_i}$	**Eigenvalue**
PC1	4.706
PC2	2.497
PC3	1.446
PC4	0.919
PC5	0.853
PC6	0.642
PC7	0.551
PC8	0.348
PC9	0.289
PC10	0.251
PC11	0.226
PC12	0.169
PC13	0.103

The variance contribution ratio and the cumulative variance contribution ratio are calculated and displayed in Table 2.4. The line in Figure 2.7 represents the principal component variance contribution ratio v/s the number of principal components chosen while the bars highlight the cumulative variance contribution ratio v/s the number of components chosen. It is seen that both the plots tend to saturate at higher principal components. Thus, it can be concluded that the first few principal components explain the majority of the data-set while the contribution of variance explained by the higher principal components is very less.

Table 2.4: Variance and cumulative ratios

$\mathbf{PC_i}$	$\mathbf{P_i}$	$\mathbf{CP_i}$
PC1	0.362	0.362
PC2	0.192	0.554
PC3	0.111	0.665
PC4	0.071	0.736
PC5	0.066	0.802
PC6	0.049	0.851
PC7	0.042	0.893
PC8	0.027	0.920
PC9	0.022	0.942
PC10	0.020	0.962
PC11	0.017	0.979
PC12	0.013	0.992
PC13	0.008	1

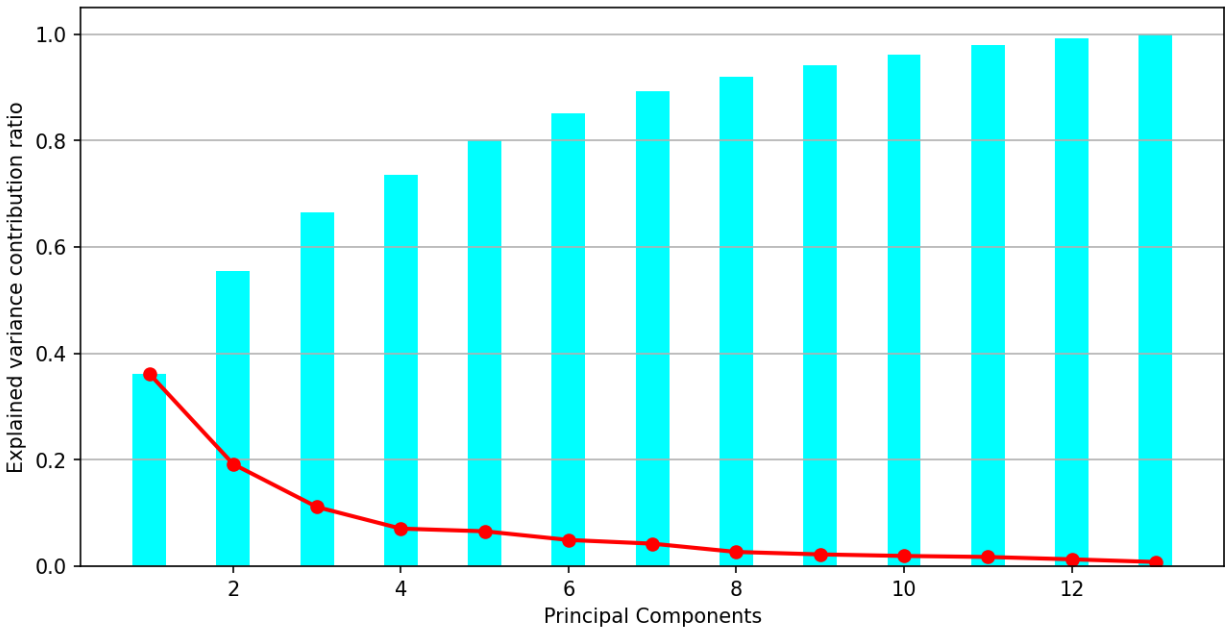


Figure 2.7: PCA Ratio Plot

As 80% of the variance can be captured from the first 5 principal components, the processed dataset with the first 5 principal components can be displayed in Table 2.3.

Table 2.5: PCA Processed Table : First 5 records

S.No	PC1	PC2	PC3	PC4	PC5
1	3.316751	-1.443463	-0.165739	-0.215631	0.693043
2	2.209465	0.333393	-2.026457	-0.291358	-0.257665
3	2.516740	-1.031151	0.982819	0.724902	-0.251033
4	3.757066	-2.756372	-0.176192	0.567983	-0.311842
5	1.008908	-0.869831	2.026688	-0.409766	0.298458

2.7 Matrix Multiplication Techniques on Hardware

Matrix multiplication represents one of the most computationally intensive operations in linear algebra, imposing significant challenges for hardware implementations due to its high arithmetic complexity and substantial memory bandwidth requirements. The conventional matrix multiplication algorithm for two matrices exhibits a computational complexity of, necessitating multiply-accumulate (MAC) operations. On general-purpose processors, optimization strategies such as parallelization, cache tiling, and vectorized instructions are employed to enhance performance[49]. However, as matrix dimensions increase, memory access patterns become a critical bottleneck, leading to inefficient ex-

ecution on both central processing units (CPUs) and graphics processing units (GPUs), where frequent memory transactions constrain performance[19].

Several algorithmic techniques have been proposed to mitigate the computational burden of matrix multiplication, including recursive methods such as Strassen's algorithm and the Coppersmith-Winograd algorithm. Strassen's method reduces computational complexity from to approximately by recursively decomposing matrices into smaller subproblems using a divide-and-conquer approach[50]. The Coppersmith-Winograd algorithm[51] further refines the complexity to . However, despite their theoretical efficiency, these recursive methods introduce significant overhead due to increased memory access, intricate indexing schemes, and complex branching operations, making them suboptimal for hardware acceleration, particularly on field-programmable gate arrays (FPGAs) and application-specific integrated circuits (ASICs). These architectures favor structured and highly parallel execution patterns that reduce control flow complexity and enhance data locality.

A widely adopted approach in hardware implementations is the use of systolic arrays, which offer a structured and highly efficient paradigm for matrix multiplication[52]. A systolic array, as shown in Fig.2.8 consists of a regular arrangement of processing elements (PEs) that propagate data rhythmically in a synchronized manner, significantly reducing memory access overhead by maintaining intermediate results within local registers. The primary advantage of systolic arrays lies in their symmetry and spatial locality, facilitating predictable data flow and minimizing the necessity for intricate control logic and external memory transactions. Each PE executes a simple MAC operation and transfers partial results to adjacent units, thereby enabling high-throughput and low-latency computation. Due to their efficiency, systolic arrays have been extensively deployed in specialized hardware accelerators, including Google's first-generation Tensor Processing Unit (TPU), which employed a matrix multiplication unit to expedite deep learning workloads[53]. The inherent parallelism and structured data movement of systolic arrays render them highly suitable for hardware-accelerated principal component analysis (PCA), where large-scale matrix operations, such as covariance matrix computation and eigendecomposition, demand computational efficiency. By leveraging systolic arrays, hardware accelerators achieve substantial performance improvements over CPU- and GPU-based approaches, thereby enabling real-time and large-scale PCA computations in machine

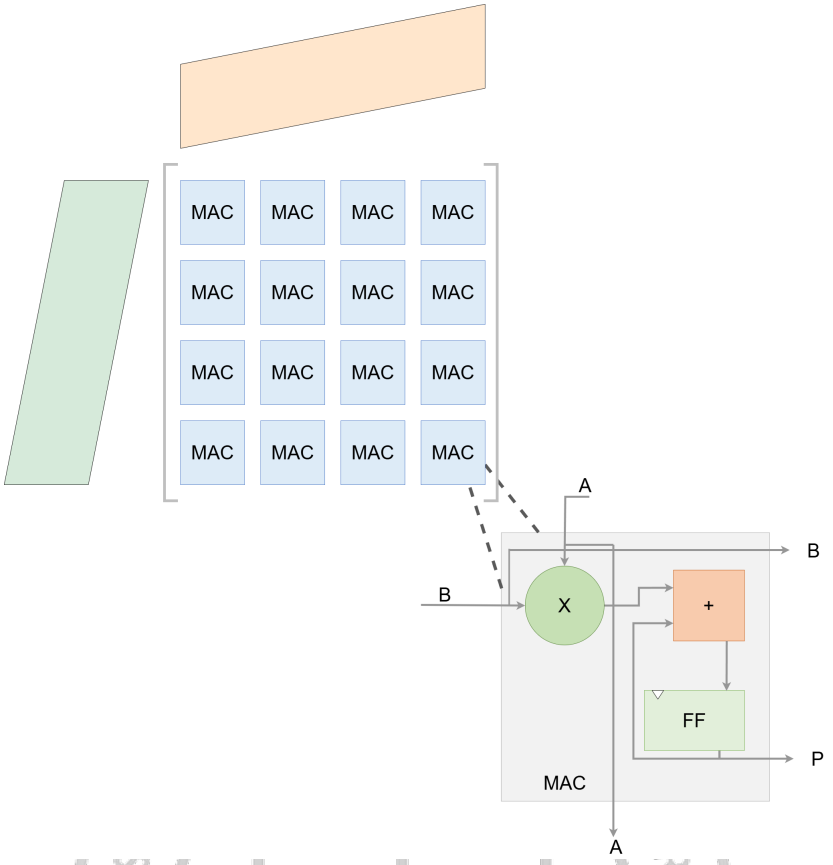


Figure 2.8: Systolic Array Architecture

learning, computer vision, and scientific computing applications.

2.8 Singular Value Decomposition on Hardware

Singular value decomposition (SVD) is a fundamental operation in numerous scientific and engineering domains; however, its computational complexity poses substantial challenges, particularly for large-scale matrices. Two prominent approaches for computing SVD are the Jacobi algorithm and the Golub-Kahan algorithm, each possessing distinct computational characteristics and implications for hardware acceleration. The Jacobi algorithm, in both its one-sided and two-sided variants, iteratively applies Givens rotations to nullify off-diagonal elements, progressively transforming the matrix into diagonal form[21]. Singular values emerge along the diagonal, while the corresponding singular vectors are accumulated via successive orthogonal transformations. A notable advantage of the Jacobi method is its intrinsic parallelism, allowing Givens rotations to be applied independently to different matrix elements, thereby facilitating efficient concurrent execution. This parallel structure renders the Jacobi algorithm particularly well-suited for hardware acceleration on FPGAs, GPUs, and custom ASICs, where parallel computing

resources can be fully utilized to expedite convergence.

Conversely, the Golub-Kahan algorithm employs a fundamentally different strategy by first reducing the matrix to a bidiagonal form via Householder reflections, followed by iterative QR decomposition to compute singular values[54]. An example of the Golub-Kahan algorithm was implemented in [55], where the SVD was accelerated in Orthogonal Frequency Division Multiplexing (OFDM) for Multiple Signal Classification. This two-stage process introduces sequential dependencies that constrain parallelization. While the bidiagonalization phase ensures numerical stability, it involves substantial memory movement and matrix-vector multiplications, which present a bottleneck in hardware implementations. Furthermore, the QR iteration phase operates on the entire bidiagonal structure rather than individual matrix elements, leading to increased computational complexity and limited parallel execution capabilities. These inherent constraints make the Golub-Kahan algorithm less attractive for hardware implementations where efficiency and parallelism are paramount.

A key advantage of the Jacobi algorithm over Golub-Kahan is its superior numerical stability, particularly in handling ill-conditioned matrices. In numerical computations, ill-conditioned matrices exhibit small singular values that are highly sensitive to perturbations. The Jacobi method, leveraging iterative orthogonal transformations, ensures high precision in singular value computation while minimizing the accumulation of rounding errors. Each Givens rotation operates on a localized subset of matrix elements, thereby maintaining numerical accuracy throughout the transformation process. Conversely, the Golub-Kahan algorithm, particularly during its QR iteration phase, is susceptible to numerical instability due to repeated matrix factorizations, which amplify rounding errors over successive iterations. Consequently, the Jacobi approach is preferable for applications requiring high-precision singular value computations, including signal processing, control systems, and machine learning[56].

From a hardware implementation perspective, the Jacobi algorithm aligns well with systolic array architectures, which are commonly employed for accelerating linear algebra computations. The ability to apply Givens rotations in parallel across multiple off-diagonal elements facilitates the design of highly efficient systolic arrays, where each processing unit performs localized transformations while maintaining communication with neighboring units. This distributed computation model significantly enhances through-

put and reduces latency relative to sequential methodologies[57]. In contrast, the Golub-Kahan algorithm's reliance on sequential bidiagonalization and QR factorization impedes efficient mapping onto systolic architectures, making it less desirable for FPGA and ASIC implementations.

Furthermore, the Jacobi algorithm's reliance on element-wise transformations, rather than explicit matrix multiplications, leads to lower memory bandwidth requirements—a critical advantage in hardware accelerator design. Memory bandwidth often constitutes a limiting factor in high-performance computing applications, and minimizing data movement is essential for achieving energy efficiency and high-speed execution. The Golub-Kahan method, in contrast, involves multiple matrix-vector multiplications during the bidiagonalization process, exacerbating memory access demands and reducing suitability for low-power, high-efficiency hardware architectures.

Given these advantages, the Jacobi algorithm emerges as the preferred choice for hardware-accelerated SVD computation, particularly in FPGA- and ASIC-based systems. Its high parallelizability, superior numerical stability, efficient memory utilization, and compatibility with systolic architectures render it an optimal candidate for large-scale matrix computations in domains such as machine learning, image processing, and scientific computing. While the Golub-Kahan algorithm remains a robust option for general-purpose CPU-based implementations, its sequential constraints and elevated memory demands render it suboptimal for dedicated hardware acceleration scenarios.

2.8.1 Jacobi Eigenvalue Decomposition

The goal of PCA is to compute the eigenvalues and eigenvectors of the covariance matrix of a dataset. The Jacobi algorithm accomplishes this by iteratively applying orthogonal transformations to diagonalize the matrix.

The Jacobian Eigenvalue algorithm is presented below-

1. **Input:** A symmetric covariance matrix C of dimension N x N
2. Initialise the Eigenvector matrix $V = I$. This matrix would accumulate the rotations
3. Identify indices p and q such that $|c_{pq}|$ is the largest off diagonal element. The element would be reduced to zero in subsequent rotations
4. Compute the rotation angle θ, that would zero out c_{pq}. This is given by the equation

$2\theta = \frac{2c_{pq}}{c_{pp}-c_{qq}}$

5. Calculate the value of $cos(\theta)$ and $sin(\theta)$
6. Construct an identity matrix R_{pq}, and modify it such that $R_{pp} = cos(\theta)$, $R_{pq} = sin(\theta)$, $R_{qp} = -sin(\theta)$, $R_{qq} = cos(\theta)$
7. Apply the rotation as described by the equation $C' = R_{pq}^T C R_{pq}$
8. Update the Eigenvector Matrix $V = VR$
9. The algorithm converges when all the off diagonal elements of C are below a specified tolerance ϵ
10. Columns of V are Eigenvectors
11. **Output:** Return the Eigenvalue Decomposition of C.

Hence The Jacobian algorithm is the prefferred choice for hardware accelerator SVD computation. Particulary in FPGA and ASIC board systems[58]. Its highly parallelizable in nature superior numerical stability, efficient memory usage, and compatibility with systolic array architectures make it an ideal candidate for large - scale matrix computations in applications such as machine learning, image processing and scientific computing.

2.9 Introduction to CORDICs

The Coordinate Rotation Digital Computer (CORDIC) algorithm is a highly efficient iterative method used for computing a variety of mathematical functions, including trigonometric, hyperbolic, logarithmic, and exponential functions. Developed by Jack Volder in 1959[59] and later extended by Walther, CORDIC is particularly well-suited for hardware implementations such as Field Programmable Gate Arrays (FPGAs) and Application-Specific Integrated Circuits (ASICs) due to its ability to perform computations using only shift-and-add operations, avoiding the need for costly multipliers. This makes CORDIC an essential building block in digital signal processing (DSP), computer graphics, and scientific computing applications where real-time performance and efficient hardware utilization are critical. The algorithm operates by performing a sequence of iterative micro-rotations to approximate the desired function, making it particularly useful in FPGA-based architectures where hardware efficiency and low power consumption are crucial[60].

The CORDIC algorithm is based on the principle of vector rotation in a Cartesian coordinate system. Given an initial vector (x_0, y_0), the goal is to rotate it by an angle θ to obtain a new vector $(x^{'}, y^{'})$ without using multiplication operations. This is accomplished by performing a series of micro-rotations of fixed angles that converge to the desired rotation. The core mathematical equations governing CORDIC in rotation mode are:

$$x_{i+1} = x_i - y_i d_i 2^{-1} \tag{2.3}$$

$$y_{i+1} = y_i + x_i d_i 2^{-1} \tag{2.4}$$

$$z_{i+1} = z_i - d_i tan^{-1}(2^{-1}) \tag{2.5}$$

where x_i and y_i are the coordinates of the vector in iteration i, z_i is the angle that is progressively reduced to zero, d_i is the direction, determined as $d_i = sign(z_i)$ ensuring that the angle is reduced in magnitude at each step. Here, 2^{-i} represents the micro-rotation scaling factor and $tan^{-1}(2^{-i})$ represents the precomputed constant for each iteration.

The iterative nature of the CORDIC algorithm means that each step rotates the vector by a smaller angle, refining the approximation with each iteration. Since multiplication by 2^{-i} is equivalent to a right bitwise shift, the entire computation involves only additions and shifts, making it extremely hardware-friendly.

CORDIC is particularly well-suited for FPGA-based hardware acceleration due to its shift-and-add architecture. Many FPGA vendors, including Xilinx and Intel, provide optimized CORDIC IP cores that can be easily integrated into hardware designs[61]. The implementation consists of a series of iterative processing units, each responsible for executing one iteration of the algorithm. These units are pipelined to improve throughput, allowing multiple computations to be performed simultaneously.

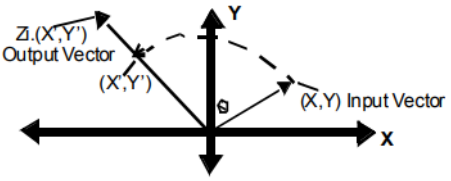


Figure 2.9: CORDIC Rotation

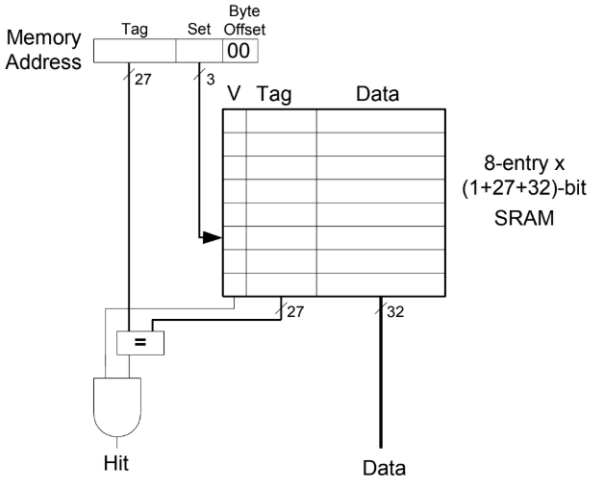


Figure 2.10: Architecture of a Direct-Mapped Cache

2.10 Cache Architecture and Modeling

The performance of compute-bound and memory-bound applications, particularly in machine learning and signal processing accelerators, is significantly impacted by memory hierarchy efficiency. Cache design, including mapping strategies, write policies, and cache modeling techniques, plays a central role in improving data locality, reducing access latency, and optimizing energy use. This section explores foundational cache architectures, behavioral policies for write misses, real-world benchmarks used for cache modeling, and the use of CACTI for memory subsystem analysis.

2.10.1 Direct Mapped Caches

A direct-mapped cache, as shown in Fig.2.10, represents the simplest form of address mapping in cache memory, where each block in main memory is mapped to exactly one location (or set) in the cache. Address decomposition in this cache type typically includes three fields: offset, index, and tag. The index field selects a unique cache line, and the tag is used to validate that the correct data is stored in that line. While this makes the design low-overhead and fast in lookup, it introduces potential issues with conflict misses, particularly if multiple memory addresses map to the same index[62].

Despite their simplicity, direct-mapped caches can perform competitively when the access pattern exhibits high spatial locality and low contention. In FPGA-based accelerators, where deterministic timing is often more important than average-case throughput, direct-mapped caches offer predictable timing behavior and are easier to implement with minimal logic resources. However, for applications with large and irregular access foot-

prints, such as matrix multiplication or sparse data reads, they may become a performance bottleneck due to frequent evictions.

2.10.2 Cache Write Miss Policies

Write-Allocate Write-Miss Policy

The write-allocate (also called fetch-on-write) policy dictates that when a write miss occurs, the block is first loaded from memory into the cache, and then the write is performed on the newly cached block. This policy assumes that the data being written is likely to be accessed again soon, benefiting from temporal locality. Write-allocate is usually combined with write-back, which reduces the number of memory write operations by deferring them until cache eviction.

In the context of FPGA accelerators for ML inference or training tasks, this policy is particularly useful when intermediate results — such as partial matrix products or activation values — are updated iteratively and accessed multiple times across computation cycles. By ensuring such values remain in the cache after being written, write-allocate reduces memory access latency and minimizes redundant memory transactions[63].

Nonetheless, write-allocate incurs additional latency on write misses, as the cache line must first be fetched before the store can proceed. For latency-sensitive paths, this may create pipeline stalls or increase critical path pressure.

Write-Around Write-Miss Policy

In contrast, the write-around policy does not bring the missed data block into the cache. Instead, it bypasses the cache and writes directly to main memory. This is advantageous in workloads where written data is not expected to be reused, such as logging, streaming, or forward-only computations. By avoiding unnecessary cache fills, write-around reduces cache pollution, freeing up lines for data with higher reuse potential.

Write-around is often used alongside write-through caching, which ensures all writes update main memory immediately, preserving coherence across memory hierarchies. In FPGA accelerators like Manojavam, this policy is ideal for RHS matrix tiles in matrix multiplication, which are streamed in, used exactly once, and then discarded[63].

However, write-around increases main memory bandwidth usage and may reduce energy efficiency if employed indiscriminately. It is best suited for write-once, read-never access patterns that dominate many preprocessing or forward-propagation stages in ML

pipelines[64].

2.10.3 Cache Modeling Benchmarks

Evaluating the effectiveness of cache policies requires realistic memory access patterns. Two well-established benchmark suites — LINPACK SAXPY and Livermore Loops — are used in this study to simulate cache behavior under practical conditions.

LINPACK SAXPY Benchmark

The SAXPY kernel (Single-Precision A·X Plus Y) is a Level-1 BLAS operation, represented by the equation:

$$Y[i] = A.X[i] + B.Y[i] \tag{2.6}$$

This pattern combines sequential reads from two input vectors with a write-back into one, making it ideal for testing read-write mixing, line reuse, and miss handling. In cache simulations, SAXPY reveals how well a system supports regular, linear memory access patterns with temporal locality[65].

Livermore Benchmark

The Livermore Fortran Kernels (LFK) consist of 24 small programs, each designed to model a different class of scientific computation. These include operations like matrix multiplication, vector updates, numerical integration, and particle simulation. The Livermore benchmark of interest in this study, is given in the following equation-

$$Z[i] = A.X[i] + B.Y[i] \tag{2.7}$$

This equation involves multiple simultaneous memory reads, multiplication, and a final update, mimicking compute stages where multiple operands must be fetched concurrently. The reuse distance, line alignment, and miss rate behavior of this kernel provide deep insights into how the cache hierarchy responds to nested loops and indirect accesses.

Livermore Loops are especially useful for stress-testing multi-port caches, bank conflicts, and stride-sensitive performance — traits that become critical in loop-unrolled ML workloads[66].

2.10.4 CACTI Cache Modeler

CACTI (Cache Access and Cycle Time Infrastructure) is a comprehensive tool for modeling cache and memory subsystems with respect to access latency, power consump-

tion, and physical area. Originally developed by Hewlett-Packard Labs and later expanded by researchers at the University of Utah, CACTI has become a widely used framework for evaluating SRAM-based cache designs in both research and industrial contexts. It allows designers to explore how microarchitectural decisions — such as cache size, associativity, block size, number of ports, and banking — affect delay, energy, and silicon footprint[67][68].

At a high level, CACTI combines delay models for SRAM cells and peripheral logic with power and area estimators derived from accurate transistor-level characterizations. It simulates the read and write paths through the memory array, including precharge, sense amplification, wordline and bitline transitions, tag comparison, and output drivers. These computations are further refined by incorporating RC interconnect delay models for local and global wiring, providing timing estimates that closely reflect physical implementation constraints.

The tool accepts a range of parameters including total cache size, line size, associativity, number of banks and ports, and the target technology node. From this, it produces detailed reports on access latency, cycle time, dynamic and leakage power, and layout area. CACTI also supports different access modes — including sequential, fast, and normal — each modeling different levels of precharge and comparator overlap.

By enabling rapid design-space exploration across technology generations and architectural configurations, CACTI facilitates early-stage memory subsystem planning. It is particularly useful when evaluating trade-offs between latency, energy efficiency, and area — all of which are critical constraints in hardware accelerators, embedded processors, and SoC memory hierarchies. Its extensibility and compatibility with open-source flows make it an ideal modeling tool for cache-centric hardware research.

2.11 FPGA Design Flow

The FPGA design flow is a comprehensive process that transforms a high-level hardware description into a physical configuration bitstream, allowing the FPGA device to behave as the intended digital circuit. This multi-stage flow, as shown in Fig.2.11 is essential to ensure the design is functionally correct, meets timing requirements, and optimally utilizes hardware resources. Each stage builds upon the previous one, gradually refining the design until it is ready for deployment.

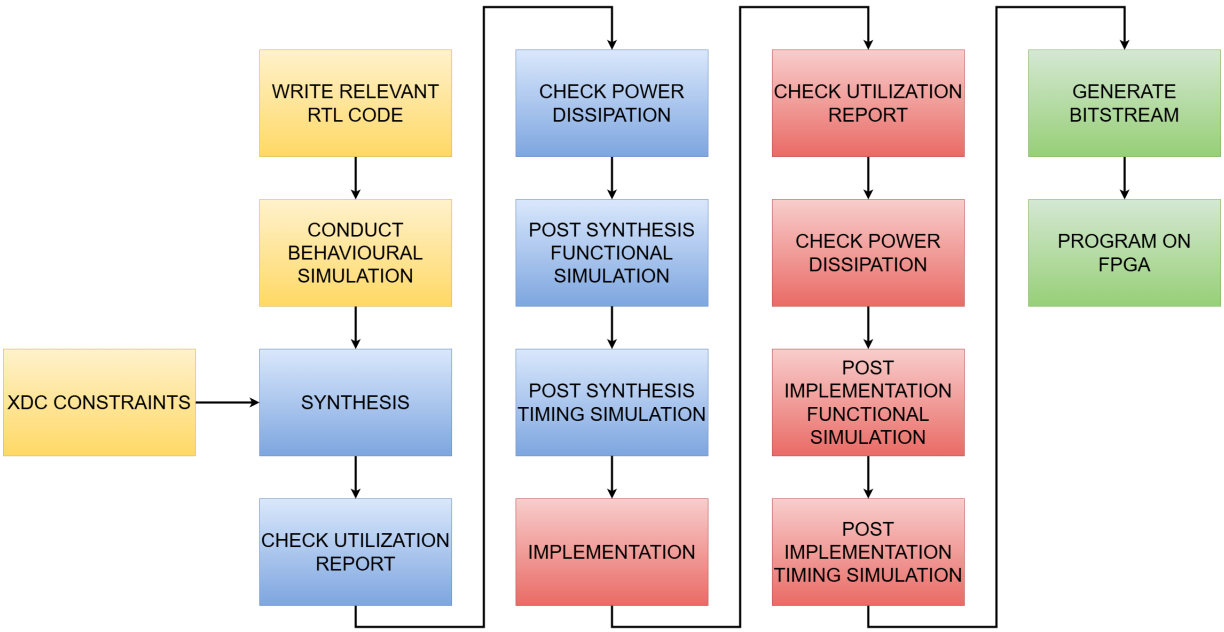


Figure 2.11: FPGA Design Flow

2.11.1 RTL Entry

Register Transfer Level (RTL) entry represents the first and foundational step in the FPGA design process. Here, the digital system's behavior and structure are described using a hardware description language (HDL), most commonly Verilog or VHDL. The RTL abstraction focuses on the flow of data between registers and the operations performed on that data synchronized by clock signals.

Writing synthesizable RTL code requires deep understanding of digital logic design and coding guidelines to ensure that the code can be successfully transformed into hardware. Designers must focus on writing clean, modular, and reusable code with clear synchronous logic constructs, avoiding non-synthesizable constructs like delays or certain loops that can't be mapped to hardware. The RTL code typically includes the definition of combinational logic, synchronous state machines, arithmetic units, multiplexers, and register files.

This stage sets the design intent clearly and forms the basis for all subsequent synthesis and implementation steps. Good RTL design not only ensures functional correctness but also heavily influences the efficiency, speed, and area utilization of the final FPGA implementation.

2.11.2 Behavioural Simulation

Behavioral simulation is the process of verifying the RTL code at a functional level before any hardware synthesis or physical mapping. It is the first verification checkpoint, focused on confirming that the design logic behaves as expected under a wide range of input conditions without considering any timing delays.

In this phase, simulation testbenches are written to provide input stimuli and monitor outputs. Testbenches model realistic operating scenarios, edge cases, and corner conditions to verify the correctness of control paths, data paths, and overall system behavior. Common simulation environments include ModelSim, QuestaSim, and Vivado Simulator.

Behavioral simulation supports iterative debugging, allowing designers to detect and fix logic bugs early, thereby reducing the risk of expensive errors downstream. It also facilitates the validation of complex algorithms and state machine transitions before the design is locked for synthesis. This step is crucial to build confidence that the design will function correctly once synthesized and implemented.

2.11.3 Synthesis

Synthesis is a pivotal step where the RTL code is automatically translated into a gate-level representation compatible with the FPGA's architecture. This process converts abstract behavioral descriptions into an optimized network of logic gates, lookup tables (LUTs), flip-flops, multiplexers, and embedded blocks like DSP slices and block RAMs.

The synthesis tool, such as Xilinx Vivado or Synplify Pro, analyzes the RTL for logic equivalence, timing, and resource constraints, then performs optimizations including logic minimization, retiming, and resource sharing. The synthesis output is a Technology Mapped Netlist (TMNL) — a detailed description of the design using the primitives available in the target FPGA device.

The synthesis process also provides crucial reports indicating estimated resource utilization (LUTs, FFs, DSPs, BRAMs), maximum achievable clock frequency, and warnings or errors. Designers can tweak the RTL code or synthesis constraints to improve performance or reduce resource usage. A well-optimized synthesis directly influences the quality of results (QoR) of the overall design.

2.11.4 Post Synthesis Simulation

Once synthesis completes, post-synthesis simulation serves as the next verification stage to ensure that the synthesized netlist preserves the intended functionality of the RTL. Unlike behavioral simulation, this step uses the gate-level netlist and incorporates basic gate delays and timing models that reflect the synthesis tool's understanding of the hardware implementation.

The simulation includes detailed signal propagation through logic gates and flip-flops, enabling detection of functional discrepancies introduced during synthesis optimization such as logic restructuring or resource sharing. Post-synthesis simulation helps verify that the design's logical correctness remains intact and is crucial before committing to the implementation stage.

Additionally, this simulation helps designers spot issues related to glitches, race conditions, or unexpected logic behavior caused by gate-level transformations. It sets the foundation for accurate timing analysis and confirms that the design is ready for the more detailed physical implementation flow.

2.11.5 Implementation

Implementation is the stage where the synthesized netlist is physically mapped, placed, and routed onto the FPGA fabric. It consists of several intricate sub-steps that translate the logical design into a physical layout optimized for speed and resource utilization.

1. **Translation:** Converts the synthesized netlist and constraints into a format suitable for implementation tools.

2. **Mapping:** Assigns the synthesized logic elements like LUTs, flip-flops, DSP blocks, and BRAMs to the physical resources available on the FPGA device.

3. **Placement:** Strategically determines the exact physical locations on the FPGA for each logic element, balancing the trade-offs between performance, congestion, and resource distribution. Placement algorithms aim to minimize critical path delays and interconnect complexity.

4. **Routing:** Establishes physical connections between placed elements through the FPGA's programmable routing fabric. Routing ensures that signals meet timing requirements and avoids routing congestion and cross-talk.

This stage involves iterative optimization driven by timing constraints and resource availability specified in the constraints file. Modern FPGA tools use advanced heuristics and machine learning-based algorithms to find near-optimal placement and routing solutions.

Implementation generates detailed timing reports and resource utilization metrics. Achieving timing closure — meeting all timing constraints such as setup and hold times — is one of the most challenging aspects of the implementation phase.

2.11.6 Post Implementation Simulation

Post-implementation or timing simulation models the design's behavior with full consideration of the detailed delays introduced by the actual physical placement and routing. This simulation uses the timing back-annotated netlist (SDF file) containing precise delay values for all interconnects and logic elements, reflecting the true operating conditions on the FPGA.

This stage verifies that the design meets timing constraints under worst-case delay scenarios and functions correctly with actual path delays, clock skews, jitter, and other physical effects. It helps identify timing violations, race conditions, and glitches that could lead to functional failures in silicon.

Post-implementation simulation is often the final functional verification step before generating the FPGA bitstream. Passing this simulation provides confidence that the design will operate reliably at the targeted clock frequency once programmed on the device.

2.11.7 Xilinx Design Constraints

The Xilinx Design Constraints (XDC) file is a vital input to synthesis and implementation tools, specifying the physical and timing requirements of the design. The XDC file uses the industry-standard Synopsys Design Constraints (SDC) syntax and allows precise control over various aspects of the FPGA implementation[69].

Key components of an XDC file include:

1. **Pin Assignments:** Mapping logical signals to specific FPGA I/O pins, ensuring correct electrical connections to external hardware interfaces such as sensors, memory modules, or communication lines.

2. **Clock Definitions:** Specification of clock signals including frequency, duty cycle,

waveform timing, and clock groups to guide timing analysis and optimization.

3. **Timing Constraints:** Setup and hold time requirements, false path and multi-cycle path declarations, and exceptions that allow the tools to focus on critical timing paths.

4. **Placement Constraints:** User directives for floorplanning, such as locking down specific logic blocks or I/O to fixed locations to optimize performance or meet design partitioning goals.

5. **Physical Constraints:** Constraints related to voltage standards, slew rates, drive strengths, and other electrical characteristics.

Accurate and comprehensive constraints in the XDC file are essential for ensuring the design meets functional, timing, and interface specifications in the final FPGA implementation.

2.11.8 Bitstream Generation

Bitstream generation is the culminating step in the FPGA design flow, where the fully implemented design is converted into a binary configuration file that programs the FPGA device. This bitstream encodes all information regarding logic configurations, routing paths, clock setups, and I/O assignments, essentially 'telling' the FPGA how to physically realize the design.

The bitstream is generated by the FPGA vendor tools once timing closure is confirmed and the implementation results are satisfactory. The tool packages the design into a format compatible with the target FPGA device's configuration memory, considering encryption, compression, and device-specific requirements.

After bitstream generation, the file is loaded onto the FPGA via programming interfaces such as JTAG, SPI, or Platform Cable USB. Once programmed, the FPGA starts operating as the designed digital system.

This final step is critical as any errors or timing violations detected after bitstream generation require returning to earlier design stages for correction. Successful bitstream generation signifies the completion of the FPGA design process and readiness for hardware testing and deployment.

2.12 ASIC Design Flow on Openlane and OpenRAM

ASIC design transforms RTL descriptions into manufacturable silicon layouts through a sequence of design stages. OpenLane, developed by the OpenROAD project[70], is an open-source, fully automated ASIC backend flow that integrates various open-source EDA tools to streamline this process[71]. OpenRAM complements OpenLane by providing an open-source SRAM compiler, enabling automated generation of memory blocks essential for many ASIC designs[72].

2.12.1 Openlane Architecture and Flow

OpenLane orchestrates the ASIC backend implementation starting from synthesized RTL and design constraints, producing a final verified layout. It integrates tools such as Yosys for synthesis, OpenDP for placement, TritonRoute for routing, and OpenSTA for timing analysis, all managed via Python scripts and containerized environments for reproducibility. The Openlane ASIC flow is depicted in Fig.2.12.

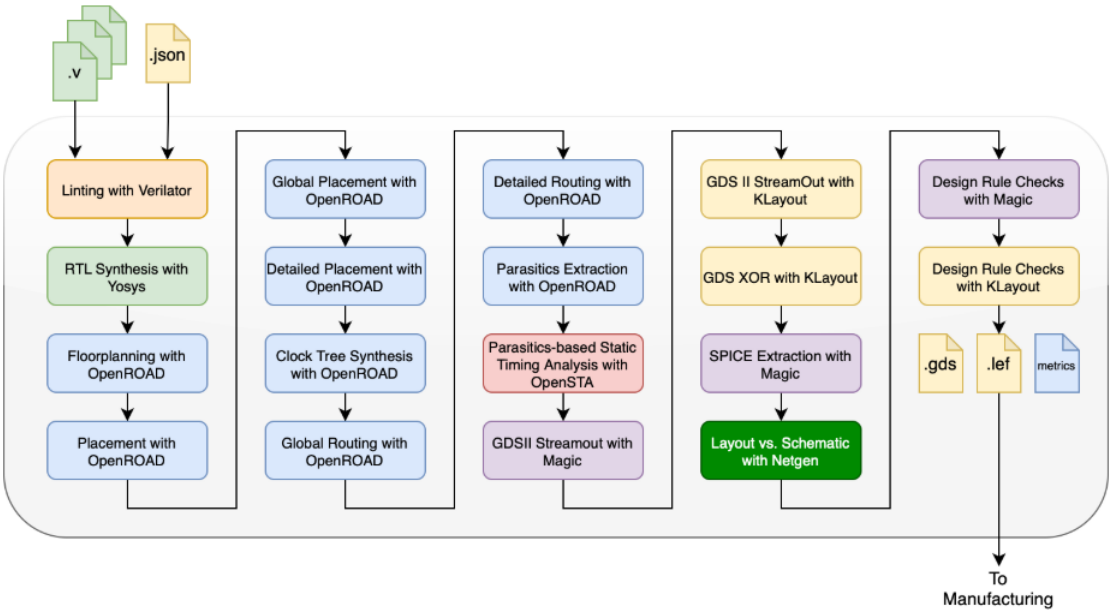


Figure 2.12: ASIC Openlane Flow[71]

The key stages include-

1. **Synthesis:** Converts RTL into a gate-level netlist optimized for the target technology.

2. **Floorplanning:** Defines core area, I/O placement, and power grid setup.

3. **Placement and Clock Tree Synthesis:** Assigns physical locations to cells and synthesizes the clock distribution network.

4. **Routing:** Connects cells with metal layers while ensuring design rule compliance.

5. **Verification:** Performs DRC, LVS, and static timing analysis.

6. **GDSII Generation:** Creates the final layout file for fabrication.

OpenLane uses YAML-based configuration files to customize design parameters such as technology libraries, floorplan dimensions, and clock frequencies, providing flexibility across different ASIC projects.

2.12.2 OpenRAM Integration

OpenRAM is a parameterizable, open-source SRAM compiler that automatically generates synthesizable RTL, physical layout, and timing models tailored to specified memory configurations like size, port count, and banking.

OpenRAM SRAM blocks integrate smoothly into the OpenLane flow by:

1. Being instantiated directly in RTL or added post-synthesis as black-box macros.

2. Undergoing placement and routing alongside standard cells during implementation.

3. Providing accurate timing and power characterization models to assist timing closure and power estimation.

This integration eliminates the need for manual SRAM design, accelerating ASIC development and improving design reliability.

This chapter outlined the essential theoretical and technical foundations supporting the two key aspects of the project: FPGA architectural exploration and the RTL-to-ASIC design of the Manojavam PCA accelerator. The FPGA section began by examining the internal structure of FPGAs, focusing on Configurable Logic Blocks (CLBs), lookup tables (LUTs), and routing elements such as switch boxes. It introduced the baseline Intel Stratix-10 architecture along with custom 4-bit Single and Double Carry Chain architectures optimized for low-precision arithmetic and parallel carry propagation. The importance of area, delay, and logic utilization as evaluation metrics was discussed in the context of machine learning hardware acceleration. The VTR toolchain was detailed,

emphasizing the roles of Yosys (for Verilog synthesis), ABC (for logic optimization and technology mapping), and VPR (for packing, placement, routing, and analysis), along with the use of architecture description XML files for architectural specification.

The second part of the chapter focused on the Manojavam PCA accelerator, starting with matrix multiplication strategies that use 4×4 operand tiling and controller-driven tile streaming. It introduced systolic arrays based on TPU-style architectures and explained both weight-stationary and output-stationary dataflows to maximize throughput and regularity. The mathematical basis of PCA was discussed through covariance matrix formulation and the Jacobi method, highlighting the role of Givens rotations in eigendecomposition. Memory hierarchy design was explored through a block-streamed structure with private L1 and shared L2 caches, incorporating write-around and write-validate policies. The RTL design process was explained using Verilog and Vivado, covering simulation, synthesis, implementation, and power-timing analysis. The ASIC design flow was covered through OpenLane, detailing floorplanning, placement, routing, and GDSII generation, along with OpenRAM-based SRAM integration. Supporting tools such as CACTI and a custom simulator were used for cache modeling and compute latency analysis. Altogether, these foundations established the architectural, algorithmic, and tool-driven context necessary for the development and evaluation of the project's hardware systems.

Chapter 3

Design Methodology

CHAPTER 3

DESIGN METHODOLOGY

Designing efficient ML hardware requires a structured approach to both architectural modeling and accelerator development. This chapter outlines the methodology adopted for evaluating custom FPGA architectures using the VTR toolchain, with a focus on carry chain-based logic for arithmetic-heavy ML benchmarks. In parallel, it details the step-by-step design of the Manojavam PCA accelerator, including its systolic matrix engine, cache hierarchy, and controller structure. Key design decisions, supporting models, experimental workflows, and analytical formulations are also discussed, setting the foundation for implementation and result analysis in the subsequent chapters.

3.1 Specifications of FPGA Architectural Exploration

The motivation behind this architectural exploration stems from the need to understand how the internal composition of FPGA logic blocks — particularly the nature of carry propagation and the use (or absence) of hardened arithmetic units — influences the efficiency of mapping arithmetic-heavy workloads. In the context of machine learning (ML) applications, such as matrix multiplications, digital filters, and accumulation pipelines, the performance of the underlying hardware is highly sensitive to how well the architecture supports fast and compact arithmetic operations. While modern commercial FPGAs like Intel Stratix-10 or Xilinx Versal feature hardened DSP blocks to optimize multiply-accumulate operations, many research-grade or open-source FPGA fabrics, as well as ASIC-grade RTL flows, rely instead on carry chains composed of basic logic elements. Studying the tradeoffs between heterogeneous architectures (with DSP hard blocks) and homogeneous logic-only designs (with serial or parallel carry chains) offers insights into how arithmetic-dense applications behave across a broader architectural spectrum.

To facilitate this analysis, three target architectures were employed in the study, each described in the VTR architectural XML format. The first is a Stratix-10-like architecture, modeling hardened multipliers and adder trees to resemble a modern high-performance heterogeneous FPGA. The second is a homogeneous architecture featuring a 4-bit single carry chain, where carry propagation is strictly serial across four 1-bit full adders, emulating delay-intensive arithmetic behavior. The third is a 4-bit double

carry chain architecture that introduces parallel carry propagation paths, allowing partial carries to resolve concurrently, thus improving throughput for wider additions. These three configurations allow us to explore the performance spectrum from hardened, low-latency arithmetic to logic-only, delay-sensitive arithmetic styles.

All three architectures were evaluated under a fixed routing and interconnect model — specifically, a standard island-style routing architecture as defined in VTR. Each architecture maintains the same routing switch box style, wire segmentation, and global resource availability. The only changes lie in the logic block definitions — namely, the composition of BLEs (basic logic elements), the presence or absence of carry chains, and how multipliers are realized (hardened vs. synthesized). This isolation of variables ensures that performance differences are directly attributable to logic architecture rather than routing artifacts or placement constraints.

To assess the practical implications of these architectural differences, a fixed set of HDL benchmarks were synthesized and mapped onto each architecture. These benchmarks consist of:

1. **Adder Trees:** 2-level and 3-level summation trees for fixed-point operands, stressing chained addition performance.

2. **FIR Filters:** Pipelined and unpipelined FIR structures with and without hard multipliers. For carry-based architectures, multipliers were implemented using Wallace-tree style adder networks, exercising the carry chain fabric intensively.

Performance was quantified using three key metrics:

1. **Operational Frequency (f)** — defined as the reciprocal of the longest delay path:

2. **Critical Path Delay (D)** — the maximum timing delay (in nanoseconds) from any input to output in the post-place-and-route design.

3. **Area (MWTA)** — a weighted logic utilization model inspired by VTR

This setup provides a controlled environment to understand how different logic architectures influence the performance of ML-relevant workloads, enabling data-driven decisions for future FPGA or domain-specific accelerator designs.

3.2 FPGA Architectural Modeling in VTR

To perform a rigorous architectural exploration, three FPGA fabric configurations were modeled using the Verilog-to-Routing (VTR) toolchain: (1) Intel Stratix-10-like heterogeneous FPGA with hardened DSP blocks, (2) a homogeneous FPGA with a 4-bit carry chain within each logic block (single chain), and (3) a variant with two parallel 4-bit carry chains (double chain). These configurations were described using VTR's XML-based architecture specification language, with a focus on modeling complex logic blocks (CLBs), interconnects, and hard blocks like multipliers and memories.

3.2.1 CLB Architecture and Carry Chain Design

At the heart of each fabric lies its CLB (Configurable Logic Block) design. In the Stratix-10 model, the CLB closely mimics Intel's Adaptive Logic Module (ALM), which includes 6-input fracturable LUTs and dedicated hardened carry logic. The carry chains here are tightly coupled with the LUT outputs, and propagate between logic elements using explicitly defined **direct** connections — one for incoming carry from the previous LAB and another for chaining between elements within a CLB. The structure supports efficient arithmetic mapping and pipelining in DSP-like workloads

The 4-bit single chain architecture departs from this by implementing a simplified carry chain that connects five logic elements linearly within a single chain per CLB. The XML model defines a chain pattern from **fle1[0].cout** to **fle1[1].cin**, and so on, terminating at **fle1[4]**. The carry-in for this chain originates from an external **cin** pin on the CLB, and the carry-out from the last element is routed to **cout**. This mimics the operation of older generation FPGAs or minimalist logic fabrics where a single chain is adequate for low-to-medium complexity arithmetic.

The double chain architecture enhances this further by including two parallel 4-bit chains, allowing for simultaneous arithmetic operations across two independent data paths. This is evident from how two sets of fle instances are used **(fle[0:4] and fle[5:9])**, each having its own **cin** and **cout** wiring paths. The interconnect model explicitly defines chaining within each set as well as the routing from the LAB to the CLB inputs and outputs, capturing parallelism in low-precision arithmetic mapping.

3.2.2 Routing and Interconnects

All three architectures share a uniform island-style routing structure characterized by segmented wires, programmable switch boxes, and connection blocks. The routing tracks are uniformly distributed across the horizontal (x) and vertical (y) directions with a peak utilization factor of 1.0. Wilton-style switch blocks were used **(fs=3)** to enable flexible connectivity and short detours around congested regions, and **ipin-cblock** switches were employed for linking routing wires to logic block inputs. This standardization ensures that any differences in performance metrics can be attributed primarily to the internal CLB and carry chain design rather than the routing infrastructure.

3.2.3 Hardblocks and Memory Modeling

Across all architectures, the multiplier logic is defined using a **pb-type** block for a 27×27-bit fracturable multiplier. Internally, this multiplier can operate in multiple modes: two 18×19-bit multipliers or one 27×27-bit unit, reflecting the flexibility of real-world DSP blocks. These modes enable trade-offs between precision and resource utilization. The internal delay characteristics are also modeled, with a delay of approximately 1.825 ns for each 18×19 multiplier, aligning with data from Intel Arria-10 chips fabricated in 22nm technology nodes.

Memory blocks, though not heavily emphasized in this study, are modeled as simple RAM blocks with fixed delays. These provide compatibility with workloads such as FIR filters and adder trees that may require coefficient or state storage.

In essence, this section formalizes the modeling of three FPGA architectural variants in VTR, emphasizing differences in CLB composition and carry chain capabilities. While the routing, memory, and multiplier subsystems were kept largely consistent, the varying styles of arithmetic support within the logic blocks allowed us to examine their influence on performance and area when mapped with arithmetic-heavy benchmarks. These architecture files served as the foundation for place-and-route trials, allowing us to extract meaningful comparisons across design metrics such as critical path delay, logic utilization, and peak operating frequency.

3.3 Methodology for FPGA Architectural Exploration using VTR

To evaluate how different FPGA architectures handle machine learning-oriented workloads, the Verilog-to-Routing (VTR) toolchain was employed as the core design exploration framework. This methodology aimed to systematically analyze architectural features—particularly carry chains and DSP-based blocks—and their effect on logic utilization, timing, and packing behavior across a series of ML-inspired benchmarks. The complete flow begins with Verilog-based HDL descriptions of the benchmark circuits, followed by synthesis, logic optimization, physical design using VPR, and ends with extraction and comparative analysis of key performance metrics.

The input to the flow consists of Verilog modules representing arithmetic-dominated circuits such as adder trees and FIR filters. These were either directly synthesized using ODIN-II, which is integrated into the VTR toolchain, or preprocessed using Yosys when finer control over synthesis was required—particularly in cases where multiplication operators were explicitly replaced with adder networks to force logic mapping onto carry chains. This preprocessing helped standardize the structure of the circuits, ensuring that architecture-dependent variations observed in later stages were genuine and not synthesis artifacts.

Post synthesis, the resulting intermediate logic network is passed through ABC, a logic optimization and technology mapping tool that is tightly integrated with VTR. ABC performs a series of transformations to simplify and balance the logic, minimize delay, and prepare the design for physical mapping. The output of this step is a BLIF (Berkeley Logic Interchange Format) file, which represents the optimized netlist and becomes the primary input to the placement and routing stages of the flow.

The next phase of the methodology involves running the BLIF file through the Versatile Place and Route (VPR) engine. VPR operates on a user-defined FPGA architecture specified via XML, which details the structure and interconnect of the logic blocks, switch boxes, routing channels, carry chains, and DSP blocks. In this project, three architecture files were used to model: (1) a baseline Intel Stratix-10–style architecture with hardened DSP blocks, (2) a 4-bit single carry chain architecture where each CLB is equipped with a single serial carry propagation path, and (3) a 4-bit double carry chain architecture that provides enhanced arithmetic throughput through dual carry paths.

The VPR engine performs packing, placement, and routing tailored to the architecture file. The packing stage clusters logic elements into configurable logic blocks (CLBs). For carry chain-based architectures, packing favors tight clustering of adders to exploit localized carry propagation paths, while in DSP-based designs, packing tends to consolidate multiplier-heavy logic into fixed-function DSP blocks. This difference leads to significant architectural trade-offs: carry chain-based designs often offer higher utilization efficiency and better support for low-precision arithmetic, whereas DSP-based architectures perform better for large-width multipliers but may suffer from resource underutilization. After packing, VPR places the clusters on the grid and performs routing to connect them, finally conducting timing analysis to evaluate path delays, clock frequency potential, and signal integrity.

At the conclusion of each run, VPR produces detailed reports that include critical path delay, logic block usage, routing channel widths, switch utilization, and cluster packing efficiency. These metrics serve as the basis for evaluating architectural performance. For example, in adder-dense benchmarks, the carry chain architectures often showed improvements in delay and area metrics due to efficient local routing and minimal use of global interconnects. In contrast, the Stratix-10–style DSP architecture displayed strengths in FIR filters with wide datapaths, though at the cost of higher power and underutilized multiplier blocks for narrower bit-widths.

Given the extensive number of architecture-benchmark combinations, the entire VTR flow was automated using shell scripts, allowing batch execution of synthesis, placement, routing, and report extraction across all benchmark-architecture pairs. This not only reduced manual intervention but also ensured consistency across evaluations, enabling a scalable and repeatable testing framework. In addition to this, a separate Python-based preprocessing pipeline was developed to transform Verilog benchmark designs. Specifically, FIR filter designs that originally used the multiplication (*) operator were programmatically converted into structurally described Wallace tree multipliers, facilitating accurate mapping to carry chain logic and enabling architecture-aware synthesis. This Python module acted as a custom Verilog code generator, automatically restructuring arithmetic-heavy blocks to reflect fine-grained logic suitable for low-bit-width architectures. These preprocessed designs, when passed through the VTR flow, provided insights into how architectural choices affect resource utilization and timing when optimized arith-

metic representations are employed.

3.4 FPGA Analysis Metrics Equations

In order to quantitatively evaluate and compare the performance of different FPGA architectures for ML-relevant workloads, three primary metrics were employed in this study: Operational Frequency, Critical Path Delay, and Area in MWTA (Minimum Width Transistor Area). These metrics are derived from the post-routing reports generated by the VTR toolchain and offer insights into the timing performance and spatial efficiency of each architecture.

3.4.1 Operational Frequency

Operational frequency indicates the maximum clock rate at which the synthesized design can be reliably executed on the given FPGA architecture. It is the inverse of the critical path delay and is computed as:

$$f_{op} = \frac{1}{T_{cp}} \tag{3.1}$$

Where T_{cp} is the critical path delay (in seconds), reported by VPR.

To express f_{op} in MHz, the equation becomes:

$$f_{op}(MHz) = \frac{10^3}{T_{cp}(ns)} \tag{3.2}$$

Where the symbols retain their usual meaning.

3.4.2 Critical Path Delay

The critical path delay refers to the longest path delay through the combinational logic of the design, which determines the minimum possible clock period. This metric reflects the delay bottleneck of the circuit and is a direct output of the VPR timing analysis engine.

$$T_{cp} = max(\sum_{i} d_i) \tag{3.3}$$

Where d_i is the delay of each logic element and interconnect on a specific path.

3.4.3 Area in MWTA Units

Area is measured in terms of Minimum Width Transistor Area (MWTA), a normalized metric used by VTR to estimate the silicon area occupied by the design on a theoretical FPGA fabric. MWTA is a technology-independent metric that allows fair comparison across architectures.

The Area in MWTA units is described in the equation below:

$$A_{MWTA} = \sum_{i}(Area_{block_i}) + \sum_{j}(Area_{routing_j}) \tag{3.4}$$

Where $Area_{block_i}$ includes the area of logic blocks, flip-flops, and DSPs, and $Area_{routing_j}$ includes switch boxes, connection boxes, and wire segments.

This metric provides a coarse estimation of the physical footprint and is particularly useful in evaluating the area efficiency of logic structures such as carry chains versus DSP blocks.

Together, these metrics provide a comprehensive basis for analyzing architectural performance. While operational frequency and critical path delay capture timing behavior, the MWTA area metric enables hardware designers to assess the compactness and logic density achieved under each architecture configuration.

3.5 Architecture of the Manojavam Accelerator

The Manojavam PCA accelerator is a fully RTL-based hardware architecture designed to compute Principal Component Analysis in a power-efficient and scalable manner. PCA is a core operation in machine learning pipelines and involves two principal computational steps: covariance matrix computation and eigendecomposition via the Jacobi method. Manojavam is organized into three tightly coupled stages: the Covariance Matrix Computation Unit, the Jacobian Unit, and the Rotation Unit. Together, these components operate on streamed matrix tiles, allowing the accelerator to process large datasets efficiently. A defining characteristic of Manojavam is the reuse of datapaths between the covariance and rotation stages, enabling the architecture to optimize for both area and performance without duplicating compute logic. The complete design is implemented in Verilog HDL, simulated in Vivado, and validated through both FPGA and ASIC (OpenLane) flows. The high level architecture of Manojavam is as shown in Fig.3.1.

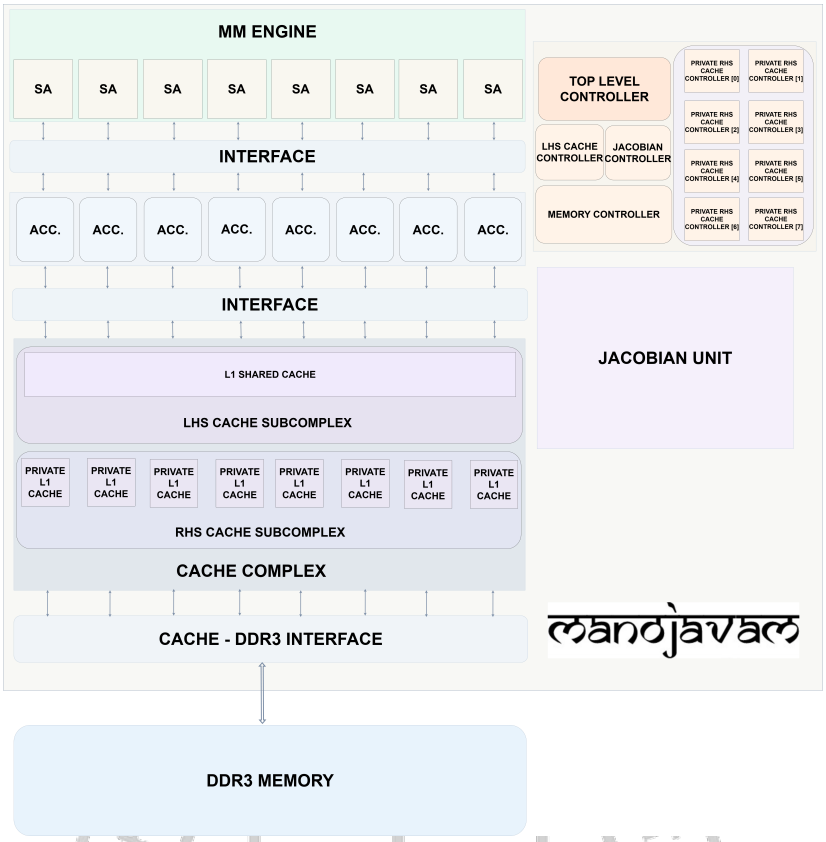


Figure 3.1: High Level Architecture of Manojavam

3.6 Matrix Multiplication Engine

The Covariance Matrix Computation Unit is tasked with constructing the symmetric matrix $C = X^T X$ where X is the input dataset matrix with mean-subtracted and standardized values. In PCA, this matrix encodes the pairwise correlations between input features and forms the foundation for identifying dominant eigenvectors. Due to the high dimensionality and streaming nature of ML datasets, direct computation of $X^T X$ is memory- and compute-intensive. Manojavam addresses this challenge using a tiled matrix multiplication approach, where the input matrix X is divided into a series of smaller 4×4 operand tiles, allowing submatrix-level parallel processing and spatial data reuse.

These tiles are streamed into a Matrix Multiplication Engine (MM Engine) comprising eight independent 4×4 systolic arrays, each designed to operate in parallel on a different tile pair. Systolic arrays are chosen for their regular structure and pipelined data movement, which reduces control complexity and maximizes throughput. Each array follows a weight-stationary dataflow model, where one operand (typically X^T) is held stationary within the array registers, while the second operand (typically X) flows through the pipeline. This model enables each multiply-accumulate (MAC) unit to reuse its stored

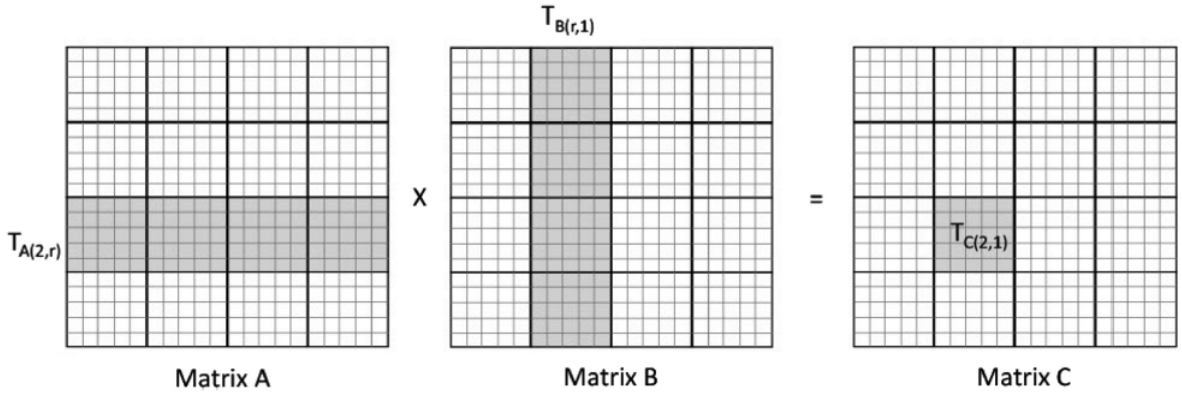


Figure 3.2: Illustration of Tiled Matrix Multiplication

operand multiple times, significantly reducing memory bandwidth requirements and off-chip traffic[73].

Large-scale matrix multiplication poses a fundamental bottleneck in hardware accelerators due to limited on-chip memory and compute parallelism. Manojavam addresses this challenge through block streaming matrix multiplication, shown in Fig.3.2. It is a technique that decomposes large input matrices into smaller, fixed-size tiles — typically matching the size of the systolic array (e.g., 4×4 blocks). These tiles are streamed sequentially through a systolic compute fabric, allowing partial products to be accumulated incrementally across time without requiring the full matrix to reside on-chip. This tiling approach drastically reduces memory bandwidth pressure and allows effective reuse of operands stored in local buffers[74]. Moreover, by orchestrating tile loading and compute through a staged controller and cache hierarchy, Manojavam overlaps data transfer with computation, ensuring that the processing elements remain fully utilized. This technique enables the accelerator to scale to arbitrarily large matrices within the limited BRAM resources of the FPGA, thereby making high-dimensional covariance computations and PCA viable even under strict hardware constraints.

The systolic arrays process operand tile pairs in a wavefront pattern, where rows of X^T and columns of X are sequentially injected into the array, and intermediate products are propagated and summed through the grid of MAC units. The output of each array is a 4×4 partial product matrix corresponding to a block of the full covariance matrix. These partial products are not stored directly; instead, they are routed to a corresponding matrix accumulator, which collects and accumulates the results across multiple tile passes. Each accumulator is mapped one-to-one with a systolic array and implements a register

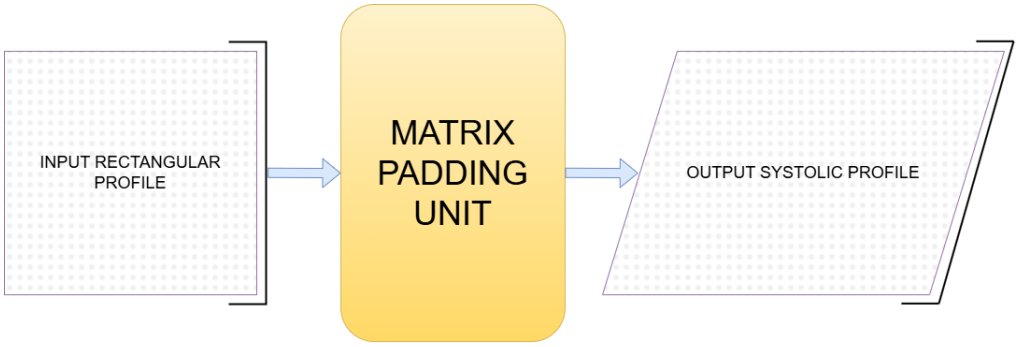


Figure 3.3: Systolic Operand Setup by Matrix Padding Units

or BRAM-based accumulation buffer, capable of holding intermediate values until all contributing tile products for that block have been processed.

Operand tiles are delivered to each systolic array through a structured input stream managed by Matrix Padding Units. These units take flattened 1D representations of the matrix stored in memory and reshape them into 2D tile structures, ensuring that operands are correctly aligned for the systolic pipeline. This is depicted in Fig.3.3. Additionally, the padding units handle zero-padding for tiles at the matrix boundary when the input dimensions are not multiples of 4. This abstraction simplifies control logic and guarantees correctness across arbitrary input sizes.

To further optimize resource usage, the systolic arrays are built using a hybrid MAC structure. Each 4×4 array contains both DSP-based MAC units, which offer high-speed multiplication, and LUT-based MAC units, which offer area efficiency and configurability. This hybrid approach provides flexibility for fine-grained performance tuning based on matrix size, input bit-widths, and target frequency constraints. For example, heavier matrix blocks can be assigned to DSP-enhanced arrays, while lighter or repetitive blocks can be processed using LUT-based arrays to conserve FPGA DSP slices.

The MM Engine is designed to be modular and scalable. Since the design partitions the full matrix computation into smaller tiles, larger matrices (e.g., 1024×1024 or 1000×4) can be processed iteratively by looping over tiles and summing partial results through the matrix accumulators. This design supports a streaming compute model, where the covariance matrix is constructed incrementally without the need to store the entire input matrix or intermediate products in memory. As a result, the engine is well-suited for machine learning workloads where datasets are large, memory access is costly, and hardware parallelism is critical for real-time performance.

The Matrix Multiplication Engine (MM Engine) is not only central to covariance matrix computation but also serves a dual-purpose role in the broader architectural flow of Manojavam. A critical design feature of the accelerator is the datapath-level reuse of this MM Engine during the eigendecomposition phase of PCA, specifically for performing Givens rotations as part of the Jacobi method. Once the covariance matrix has been constructed, the rotation phase begins, requiring transformations of the form $C' = R^T CR$ and $V = VR$, where R is a Givens rotation matrix computed to zero out the selected off-diagonal element of the covariance matrix.

Rather than implementing a separate hardware unit for these matrix-matrix multiplications, the architecture repurposes the same systolic arrays used in the covariance computation phase. This reuse is made possible by the input tile streaming abstraction, where operand tiles—regardless of their mathematical meaning (dataset blocks, covariance matrix rows, or Givens rotation matrices)—can be streamed into the systolic arrays through the same input pipeline, formatted using the Matrix Padding Units. This modularity allows the MM Engine to operate seamlessly across different computation phases without any structural changes.

A single mode control signal, propagated from the top-level controller, dynamically configures the MM Engine to switch between covariance mode and rotation mode. In covariance mode, the engine performs $X^T X$ using input tiles of the dataset; in rotation mode, it processes either $R^T C$, CR or VR depending on the sweep step in the Jacobi iteration. The operand cache controllers and memory access paths are also adjusted according to the mode, ensuring that operand delivery, accumulation policies, and write-back paths remain consistent with the target operation.

This design approach—reusing the MM Engine across both the covariance computation and the Jacobi rotation stages—offers a practical solution to reducing hardware overhead and simplifying system integration. By utilizing the same systolic datapath for different phases of PCA, the design avoids redundant hardware instantiations, resulting in a more resource-conscious implementation. The ability to support both operations through common tile streaming and buffer management also helps in maintaining a uniform interface for operand delivery and accumulation. Mode-based reconfiguration of operand sources and control signals ensures functional flexibility without significantly increasing architectural complexity. Overall, the reuse of the MM Engine reflects a thoughtful de-

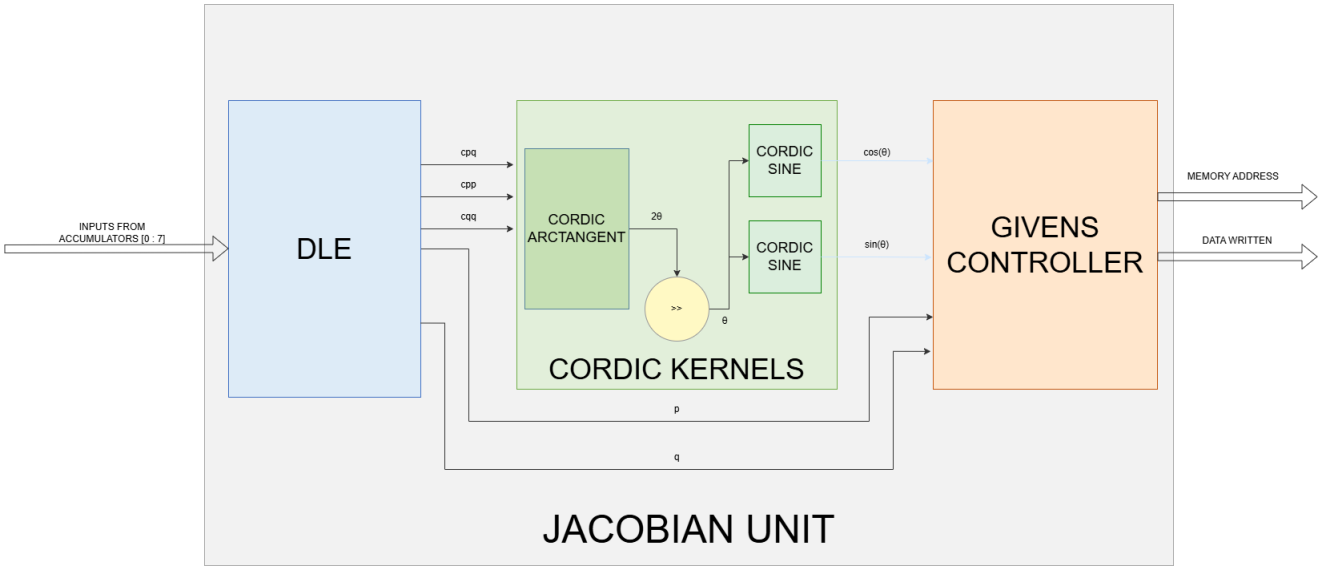


Figure 3.4: Jacobian Unit Architecture

sign trade-off between generality and specialization, aligning with the broader goals of area efficiency and functional completeness in hardware accelerators intended for matrix-intensive workloads.

3.7 Jacobian Unit Architecture

The Jacobian Unit in the Manojavam architecture, depicted in Fig.3.4, is responsible for orchestrating the core numerical operations in the Jacobi method for Singular Value Decomposition (SVD), which underpins the eigendecomposition phase of Principal Component Analysis (PCA). The unit enables iterative diagonalization of the symmetric covariance matrix by computing successive Givens rotations that eliminate off-diagonal terms. This hardware-accelerated unit is composed of three tightly coupled submodules — the Data Lookup Engine (DLE), CORDIC Kernels, and the Givens Engine — which together facilitate high-throughput, low-latency execution of the Jacobi algorithm without resorting to software-driven matrix traversal or off-chip computation.

3.7.1 Data Lookup Engine (DLE)

The Data Lookup Engine (DLE) is a unique architectural innovation in Manojavam, specifically designed to identify the largest off-diagonal entry in the symmetric covariance matrix C, which is the target of each Jacobi sweep. Unlike conventional designs that rely on off-chip memory or multiple on-chip reads to traverse the matrix and extract this value, the DLE operates in-line with the accumulation path of the matrix multiplication unit.

As partial sums for each matrix tile are generated during block streaming multiplication, the DLE snoops into these intermediate values and selectively captures the relevant ones based on the row-block and column-block identifiers.

To ensure that only valid off-diagonal elements cpq are considered, the DLE incorporates logic to ignore diagonal entries (c_{pp}, c_{qq}) and focuses only on entries where $p \neq q$. For example, during the accumulation of row block R_0, only entries from accumulators corresponding to columns 1 through N are considered, with accumulator 0 (holding c_{pp}) explicitly excluded. This dynamic filter is implemented using row-wise masking and loop counters governed by the Jacobian Controller.

The DLE maintains a running maximum register, which stores the current best candidate c_{pq} and its associated diagonal elements c_{pp}, c_{qq}, as well as the corresponding indices p and q. These values are updated if a newly received cpq is of greater absolute magnitude than the current maximum. Once a complete row sweep is done, the DLE asserts a ready flag and outputs the final triplet. This mechanism eliminates the need for matrix reads post-accumulation and tightly couples the matrix traversal logic with the systolic dataflow, making it highly efficient in both time and energy.

3.7.2 CORDIC Kernels

After the DLE has identified the indices (p, q) and corresponding values c_{pq}, c_{pp}, and c_{qq}, the next stage involves computing the rotation angle θ required to construct the Givens rotation matrix R. This is done using the equation:

$$\theta = \frac{1}{2} \arctan\left(\frac{2.c_{pq}}{c_{pp} - c_{qq}}\right) \tag{3.5}$$

This computation typically involves a floating-point division and arctangent operation, which are hardware-expensive. Instead, Manojavam employs CORDIC (Coordinate Rotation Digital Computer) kernels — a class of shift-and-add algorithms that efficiently compute trigonometric functions using only adders, shifters, and a lookup table.

The CORDIC engine is implemented as a 3-stage pipeline:

- Stage 1 computes the arctangent of the normalized input.
- Stage 2 and 3 compute the sine and cosine of the resulting angle using vector rotation mode.

These kernels work in fixed-point arithmetic, carefully scaled to match the datapath width of Manojavam while preserving numerical stability. Because of pipelining, a new angle can be fed into the CORDIC engine every few clock cycles, and once the latency is absorbed, the unit can achieve near-one-angle-per-cycle throughput. This design is ideal for iterative eigendecomposition, where hundreds of such angles may be computed across multiple Jacobi sweeps.

The $sin(\theta)$ and $cos(\theta)$ values are then forwarded to the Givens Engine, which uses them to construct the sparse rotation matrix.

3.7.3 Givens Engine

The Givens Engine is responsible for generating and storing the actual Givens rotation matrix R using the rotation angle θ. In the Jacobi method, the Givens matrix is an identity matrix with four modified elements at $[p][p]$, $[q][q]$, $[p][q]$, and $[q][p]$, which are set to:

1. $R_{pp} = R_{qq} = cos(\theta)$
2. $R_{pq} = sin(\theta)$
3. $R_{qp} = -sin(\theta)$

This localized modification is sufficient to apply a plane rotation in the (p, q) subspace of the matrix, progressively zeroing out the off-diagonal term c_{pq}. The Givens Engine stores this matrix in a dual-port RAM, enabling simultaneous read and write operations during iterative construction and rotation phases. This storage structure is accessed by downstream matrix multiplication units for computing:

$$C' = R^T C R \tag{3.6}$$

$$V = V R \tag{3.7}$$

The Givens Controller synchronizes matrix updates and triggers operand movement within the memory hierarchy. It generates select signals to update only relevant portions of the Givens matrix, thereby avoiding unnecessary re-initialization. Once the matrix is

ready, a mode signal is broadcast to all computational units, switching the pipeline from covariance accumulation to rotation mode.

To minimize performance overhead, the engine also preloads the next Givens rotation during the computation of the current matrix transformation, enabling overlap of angle generation and matrix rotation — a technique analogous to prefetching in cache systems.

3.8 Cache Subsystem

Efficient memory access plays a critical role in sustaining the performance of high-throughput accelerators. In Manojavam, where multiple 4×4 systolic arrays operate in parallel, the design of the memory subsystem is crucial for maintaining computational flow without stalling. To support continuous and parallel tile streaming across compute stages, Manojavam implements a dual-tier cache hierarchy—consisting of L1 and L2 caches—architected specifically to match the operand reuse patterns that arise in block-wise matrix multiplication.

The left-hand side (LHS) operands, such as the rows of X^T during covariance matrix computation or R^T during rotations, are typically reused across multiple columns or matrix blocks. This temporal locality makes them good candidates for shared caching. Accordingly, all LHS operands are stored in a shared L1 cache, which is accessible to all eight systolic arrays through a broadcast-style access network. This shared architecture reduces memory footprint, eliminates unnecessary duplication of frequently used data, and simplifies coherency management for read-only accesses. The shared cache also includes address decoding and basic arbitration logic to allow synchronized reads from multiple consumers.

In contrast, the right-hand side (RHS) operands—including the columns of X, the evolving covariance matrix C, or the rotation matrix R—are typically unique to each systolic array during a particular compute cycle. These operands do not exhibit the same reuse pattern and are thus stored in private L1 caches, with each cache directly connected to one of the eight systolic arrays. This private caching scheme eliminates the need for read arbitration, enables concurrent data delivery, and supports tile-specific prefetching. Each private cache is configured to hold an entire 4×4 operand tile in a row-aligned format, enabling burst-mode reads to feed systolic inputs efficiently.

To optimize data delivery, the cache stores each operand tile in a row-major layout rather than as a raw matrix. This means each row of the 4×4 tile is stored contiguously,

allowing the systolic array to receive an entire row of inputs in a single read cycle instead of multiple sequential accesses. By reducing the number of memory fetches from four to one per row, this approach significantly improves data throughput, reduces latency, and better aligns with the streaming nature of the systolic array's pipeline.

All L1 caches—both shared and private—are backed by a common L2 memory pool, implemented as a dual-ported block RAM module or connected externally via a DDR3 memory interface. This L2 layer serves as the staging area for incoming operand tiles and also receives the accumulated results post-computation. A dedicated memory controller coordinates requests from the cache complex and enforces proper sequencing for tile fetch and write-back operations. To reduce memory bandwidth pressure and prevent redundant data movement, Manojavam employs selective cache write policies tailored to the two major computation phases.

During covariance matrix computation, a write-around policy is used for the RHS caches. In this policy, new tiles are written directly to memory without allocating space in the cache unless the data is expected to be reused. This minimizes cache pollution and allows each private cache to retain active working tiles without being overwritten by one-time-use data.

In the rotation phase, a write-validate policy is enforced, especially for matrix C and the evolving eigenvector matrix V. This policy ensures that when matrix updates are written to memory, the corresponding cache entries are marked valid, but no fetch-on-write is triggered. This reduces unnecessary memory traffic while still maintaining coherency. The system relies on the fact that the MM Engine overwrites the same tile locations in a predictable schedule, making manual validation sufficient for correctness.

Tile streaming in this memory hierarchy is managed via a controller-driven tile scheduler, which issues operand fetch signals to the caches in a fixed loop structure. This streaming-friendly model avoids complex memory dependency tracking and supports pipeline-friendly operand delivery to the systolic arrays. Furthermore, by isolating frequently reused operands from ephemeral ones, the design achieves a balance between temporal locality exploitation and parallel access concurrency, both of which are critical in matrix-intensive computation.

The overall memory system is designed to scale with matrix size, supporting tile dimensions beyond the capacity of L1 caches by fetching on demand and buffering partial

results for accumulation. It minimizes idle cycles in the datapath, maintains high utilization of the systolic arrays, and simplifies the scheduling logic by clearly separating cache responsibilities according to operand behavior. This architecture, while lightweight in its logic and buffering requirements, forms a central enabler of Manojavam's streaming and reuse-aware compute strategy.

3.9 Controller Design and Hierarchy

The efficient operation of the Manojavam PCA accelerator heavily relies on a well-structured controller hierarchy that orchestrates data flow, synchronization, and computation across multiple compute units and memory subsystems. The hierarchy is designed to modularize control responsibilities, simplify synchronization, and enable scalable management of parallel processing elements. The controller hierarchy consists of four primary levels:

3.9.1 Top-Level Controller

At the apex of the control architecture lies the Top-Level Controller, which functions as the master coordinator for the entire design. Its primary responsibility is to synchronize all subordinate controllers, ensuring smooth handshaking and timing alignment between the different units—namely, the shared LHS cache controller, the private RHS cache controller complex, and the Jacobian Unit controller. The top-level controller orchestrates the overall flow of computation phases, triggers operand fetch and store operations, and manages global start, stop, and reset signals. By maintaining a centralized state machine, this controller ensures that the system operates in a lockstep manner, preventing data hazards and ensuring consistent data availability across all compute pipelines.

3.9.2 LHS Shared Cache Controller

The LHS Shared Cache Controller is responsible for managing all operations related to the left-hand side (LHS) shared cache. This includes handling broadcast reads to multiple systolic arrays, address decoding, cache coherence management for read-only data, and arbitration for simultaneous access requests. Given that the LHS operands exhibit high temporal reuse and are shared across all eight systolic arrays, this controller plays a critical role in reducing redundant memory traffic and maintaining consistent operand delivery. It schedules tile fetches from the L2 memory into the shared cache and coordinates cache line validation to ensure the freshest data is available to the compute

units.

3.9.3 Private RHS Cache Controller Complex

The Private RHS Cache Controller Complex comprises eight independent private cache controllers, each dedicated to managing one of the eight private RHS caches linked to the individual systolic arrays. Each private cache controller handles local operand prefetching, cache line allocation, and write-back policies tailored to the right-hand side (RHS) operands that are unique per systolic array during computation. This modular organization enables concurrent and conflict-free memory accesses, as each controller operates autonomously with minimal synchronization overhead. The complex also enforces the selective write-around and write-validate policies, optimizing cache utilization based on operand reuse characteristics in covariance and rotation phases.

3.9.4 Jacobian Controller

The Jacobian Controller oversees the operation of the Jacobian Unit, which is responsible for implementing the Jacobi eigendecomposition algorithm during the PCA process. This controller manages iterative sweep operations, controls rotation matrix generation via CORDIC engines, and coordinates data movement between the rotation unit and the matrix multiplication engine. It synchronizes the execution of Givens rotations and ensures correct sequencing of eigenvector updates. By tightly integrating with the top-level controller, the Jacobian Controller maintains coherent operation between the covariance computation phase and the rotation phase, enabling smooth pipeline transitions.

Chapter 3 presents the comprehensive design methodology and specifications that underpin both the FPGA Architectural Exploration and the development of the Manojavam PCA accelerator. It begins by detailing the specific design requirements and objectives for evaluating diverse FPGA architectures using the VTR toolchain, followed by the modeling approaches employed to accurately represent targeted FPGA configurations. The chapter outlines the systematic methodology adopted to perform architectural exploration, including synthesis, placement, and routing workflows, alongside the critical performance and area analysis metrics supported by relevant equations. The focus then shifts to the architectural design of the Manojavam accelerator, describing its core components in detail — the matrix multiplication engine, which leverages multiple 4×4 systolic arrays for high-throughput matrix operations; the Jacobian unit, responsible for eigendecom-

position via Jacobi rotations; and the cache subsystem, architected to optimize operand reuse and data streaming. Finally, the controller hierarchy is elaborated, highlighting the modular control scheme that ensures synchronized operation and efficient data flow across the compute and memory units. Together, these sections form the foundation for both the FPGA exploration and accelerator implementation, linking theoretical design choices with practical hardware realization.

Chapter 4

Implementation

CHAPTER 4

IMPLEMENTATION

Robust implementation frameworks form the foundation of both architectural exploration and hardware accelerator development. For the FPGA design exploration project, the Verilog-to-Routing (VTR) toolchain is deployed within a Docker-based environment to ensure reproducibility and consistency. Benchmark circuits are synthesized using Yosys and ABC, followed by placement, routing, and detailed metric analysis through VPR. In parallel, the Manojavam PCA accelerator is realized through a modular RTL design approach, incorporating optimized matrix multiplication engines, cache subsystems, and synchronized controller hierarchies. This chapter presents the systematic workflow and technical steps followed in translating both conceptual designs into functioning and analyzable implementations.

4.1 Docker Container for VTR

The Verilog-to-Routing (VTR) toolchain, which encompasses multiple complex software components including synthesis, technology mapping, packing, placement, and routing tools, requires a carefully configured software environment to function correctly. To simplify environment setup and ensure reproducibility, we leveraged a pre-built Docker container specifically designed for VTR execution. This approach abstracts away the challenges of dependency management, version compatibility, and operating system configurations, enabling a seamless and consistent platform for architectural exploration.

From an implementation perspective, the Docker container encapsulates the entire VTR toolchain along with all necessary dependencies, including required libraries, compilers, and scripts. By using Docker, the VTR environment becomes portable and can be deployed on any host system that supports Docker, whether it is a local workstation, a remote server, or a cloud-based instance. This containerized setup eliminates the need for manual installation or configuration of individual components, which can be error-prone and time-consuming.

To use the Docker container, we pulled the official or community-maintained VTR image from a Docker repository. The container was then instantiated on the target machine, providing an isolated and consistent execution environment. Input files such as architecture XML descriptions, benchmark Verilog files, and configuration scripts were

mounted as volumes inside the container, allowing easy access and modification without rebuilding the image.

The container also facilitates batch automation by allowing execution of shell scripts and experiment workflows inside a controlled environment. This is particularly beneficial when running large-scale FPGA architectural explorations requiring multiple iterations of synthesis, packing, placement, and routing. Additionally, Docker's resource management options enable allocation of CPU cores and memory to the container, ensuring predictable performance and efficient utilization of hardware resources.

Overall, using a Docker container for VTR significantly improved the reliability and reproducibility of our architectural exploration experiments. It simplified collaboration and sharing of the experimental setup with other researchers, and reduced the overhead of troubleshooting environment-related issues. This containerized approach aligns well with best practices in modern hardware research workflows, where toolchain consistency and portability are crucial.

4.2 Architectural Description

The Verilog-to-Routing (VTR) framework enables architectural exploration of FPGA designs through configurable XML-based architecture description files. These files define the fundamental components of the FPGA fabric—such as logic blocks, routing architecture, I/O configurations, and DSPs—and form the foundation for synthesis, placement, and routing experiments. For this project, three distinct FPGA architectures were selected and implemented: Intel Stratix-10 (as a representative of commercial high-performance FPGAs), a custom 4-bit Single Carry Chain architecture, and a 4-bit Double Carry Chain architecture. Each architecture was described in its respective XML file, guided by the syntax and schema provided in the VTR Architecture Reference Manual.

4.2.1 Intel Stratix-10 Architecture

The Stratix-10 architecture description was adapted from the official Stratix10 architecture XML file in the VTR repository. This model includes a wide range of hardened elements including DSP blocks, dual-port RAMs, and fracturable logic elements. The configurable logic blocks (CLBs) consist of Adaptive Logic Modules (ALMs), each capable of handling variable input widths and logic functions. The routing fabric is highly segmented and features diverse switch block patterns to mimic the physical routing complex-

ities found in real Intel FPGAs. From an implementation perspective, this architecture provides a high-performance baseline for benchmarking, reflecting a production-grade FPGA with mature logic synthesis capabilities.

4.2.2 4-Bit Single Carry Chain Architecture

The single chain carry architecture models a lightweight, arithmetic-friendly FPGA optimized for small-bitwidth operations. Implemented in the 4bit-adder-single-chain-arch.xml file, this design features logic blocks with embedded 4-bit adders configured in a serial carry propagation style. Each logic block supports straightforward arithmetic operations and basic logic functions, with carry-in and carry-out ports connected in a linear fashion. The design focuses on compact resource allocation and favors applications involving low-precision neural network computations or signal processing tasks. This architecture prioritizes minimum delay in bit-serial arithmetic and efficient area usage.

4.2.3 4-Bit Double Carry Chain Architecture

Expanding upon the single chain model, the double chain carry architecture—described in 4bit-adder-double-chain-arch.xml—implements dual carry chains within each logic block. This allows parallel evaluation of two 4-bit additions or partial products, enhancing throughput for arithmetic-heavy workloads. Each logic element includes two cascaded carry propagation paths and supports conditional logic evaluation based on runtime selection of carry routes. This architecture is particularly well-suited for accelerating MAC (multiply-accumulate) operations, making it highly applicable for machine learning accelerators and DSP workloads. Routing resources were adjusted accordingly to support higher interconnect demand due to parallelism.

4.3 Benchmark Creation

To evaluate the performance and suitability of the target FPGA architectures modeled in VTR, a diverse set of Verilog-based benchmark circuits was created. These benchmarks were selected to stress arithmetic and logic capabilities, especially in the context of low-bitwidth machine learning primitives and digital signal processing (DSP) functions. The primary benchmark types include adder tree networks, FIR filters, and modified FIR filters with custom arithmetic structures.

4.3.1 Adder Tree Benchmarks

Adder tree benchmarks serve as fundamental arithmetic units and are frequently encountered in accumulation operations within MAC units, dot products, and tree-based summations. The benchmarks were implemented using combinational adder trees of varying depths and bit-widths. Each benchmark was unrolled into either 2-level or 3-level structures, consisting of multiple full adders arranged in a tree-like hierarchy. These benchmarks provided insights into carry propagation, logic packing efficiency, and routing overheads in each FPGA architecture.

4.3.2 FIR Filter Benchmarks

Finite Impulse Response (FIR) filters are core components in DSP workloads and provide a structured combination of multipliers and adders. Standard FIR benchmarks were written in Verilog to model a multiply-and-accumulate pipeline. Variants were created for different tap counts (e.g., 4-tap, 8-tap), and both pipelined and unpipelined versions were generated. Multiplications were performed using the standard **(*)** operator, which allowed synthesis tools to map them to either LUT-based or DSP-based implementations depending on the target architecture. These filters test both arithmetic depth and the interconnection complexity of logic blocks.

4.3.3 Modified FIR Benchmarks with Wallace Tree Multipliers

To emulate the structure of arithmetic accelerators and explore resource-level optimization, modified FIR benchmarks were created by replacing the built-in multiplication operator with an explicitly constructed Wallace Tree multiplier. The Wallace Tree multiplier was designed in Vivado using 1-bit multipliers and structured carry-save adder networks, enabling precise control over the hardware implementation.

A dedicated Python script was developed to automate the integration of this Wallace Tree design into all existing FIR filter benchmarks. This script scanned the original FIR Verilog files, identified the multiplication expressions, and replaced them with instances of the custom Wallace Tree module. It then generated new benchmark Verilog files with these modifications, which were placed in a separate directory for clarity and version control. Each modified benchmark preserved the original filter behavior but exhibited different synthesis and placement characteristics, allowing for a more nuanced comparison of how each target FPGA architecture handled complex arithmetic datapaths.

These benchmark variants—standard, pipelined, and Wallace Tree-based—were compiled into a unified test suite and systematically mapped to all three target architectures using automated flow scripts. This ensured consistent synthesis conditions and enabled direct performance comparisons across architectural designs.

4.4 Shell Automation Scripts

Mapping each benchmark design to the various FPGA architecture descriptions using the VTR flow is a multi-step process involving synthesis, BLIF generation, packing, placement, routing, and result analysis. Manually repeating these steps for every benchmark and architecture combination becomes highly tedious, especially when dealing with numerous benchmark variants and multiple architecture XML files. To overcome this challenge and eliminate the need for repeated terminal commands, automation scripts were developed to handle the entire benchmarking process.

These shell scripts enabled full traversal of all benchmark files and systematically mapped them to each of the target FPGA architectures. The scripts internally invoked Yosys to synthesize each Verilog benchmark into a BLIF representation, followed by automatic invocation of VPR using the appropriate architecture XML files. The resulting timing, area, and routing reports were generated and logged in structured output folders for each benchmark-architecture pair.

By automating the architecture-benchmark mapping process, the scripts significantly reduced manual intervention, ensured consistency in execution, and sped up the benchmarking effort. The generated reports were organized for post-processing and allowed for easy retrieval and comparison across architecture variants. This automation proved instrumental in enabling rapid design space exploration and in maintaining reproducibility across experimental runs.

4.5 BLIF File Generation using Parmys and ABC

In the FPGA architectural exploration flow using VTR, generating BLIF (Berkeley Logic Interchange Format) files is a crucial intermediate step that connects the high-level Verilog design of a benchmark to the architectural specification of a target FPGA. This transformation is handled by a synthesis pipeline involving Parmys and ABC, which together perform elaboration, optimization, and technology mapping of the Verilog source.

Parmys is a lightweight and modular front-end Verilog parser and elaboration tool

developed as a plugin to interface with the VTR ecosystem. Unlike traditional synthesis tools, Parmys focuses on quickly parsing behavioral and structural Verilog, flattening the hierarchy, and converting the design into an intermediate gate-level netlist that is suitable for logic optimization and mapping. It is designed with integration into VTR in mind, ensuring that the logic structures it produces align closely with VTR's architectural models, particularly for LUT-based FPGA fabrics.

Once the Verilog benchmark is parsed and elaborated by Parmys, the resulting netlist is handed over to ABC, a powerful tool for logic synthesis and technology mapping. ABC optimizes the logic through a series of rewriting, resubstitution, and balancing steps, and then maps the optimized logic to a target logic family, typically defined by K-input LUTs (e.g., 4-LUTs or 6-LUTs), based on the architecture definition. This technology-mapped representation is then output as a .blif file, which serves as the input to the VPR (Versatile Place and Route) tool in the next phase of the flow.

While the architectural XML description is not directly consumed during BLIF generation, it influences the synthesis constraints. For example, if the target FPGA architecture supports only 4-input LUTs (as in the 4-bit adder architectures), ABC is invoked with a corresponding LUT size parameter to ensure the output is structurally compatible with the architecture.

From an implementation standpoint, the synthesis flow begins by passing each Verilog benchmark through Parmys, which performs parsing and elaboration. The intermediate design is then piped into ABC, with appropriate mapping commands, to produce the final .blif file. This entire process is integrated into automation scripts to handle multiple benchmarks across various target architectures. The resulting BLIF files serve as architecture-aware netlists that are used by VPR for placement, routing, and physical resource analysis.

This Parmys+ABC synthesis flow ensures consistency, automation, and compatibility across the exploration pipeline, enabling rapid evaluation of benchmark behavior on custom FPGA architectures.

4.6 VPR Pack, Place & Route

After generating the BLIF netlist using Parmys and ABC, the next phase of the VTR flow involves performing pack, place, and route operations using the Versatile Place and Route (VPR) tool. This process translates the abstract netlist into a concrete physical

mapping on the FPGA grid defined by the target architecture.

The first stage, packing, groups logical elements such as LUTs, flip-flops, and carry logic into clusters called logic blocks. VPR uses the architecture XML file to determine the permissible packing patterns, which differ across the Stratix-10 and the custom 4-bit carry chain architectures. This step ensures that logic elements are organized into valid cluster configurations before proceeding to physical layout.

Once packing is completed, VPR moves to placement, where each packed cluster is assigned a position on the FPGA fabric. The placement algorithm attempts to minimize critical path delay and wire length, taking into account the architecture's grid layout and connectivity constraints. Placement quality directly impacts timing, making this stage essential for performance optimization.

The final step is routing, in which the placed logic blocks are interconnected using the FPGA's routing resources. VPR constructs routing trees that obey the architecture-defined switch boxes and channel widths. Successful routing ensures that all signal paths are physically realizable on the target fabric. After routing, VPR generates detailed timing, area, and wire length reports, which are used for evaluating the effectiveness of the architecture when mapped with a specific benchmark.

This pack, place, and route sequence completes the physical implementation of a benchmark on a given FPGA model, enabling quantitative comparison between architectures under consistent workloads.

4.7 Architecture Timing and Area Analysis

Once the VPR pack, place, and route stages are completed, the final step in the implementation flow involves analyzing the timing and area characteristics of each benchmark mapped to a target architecture. These metrics are crucial for evaluating architectural suitability in the context of real-world workloads, particularly in latency- or resource-constrained environments such as machine learning accelerators.

VPR automatically generates detailed reports during the post-analysis phase, including metrics such as critical path delay, total wire length, and logic block utilization. These values reflect the overall timing performance and silicon area efficiency of the FPGA implementation. The critical path delay serves as the primary indicator of the maximum achievable clock frequency, while area metrics such as the number of LUTs, flip-flops, and routing channels used help quantify the hardware footprint of each benchmark.

Given the large number of benchmark-architecture pairings, manually parsing each VPR-generated log file for timing and area statistics would be extremely tedious and error-prone. To address this, a set of shell scripts was developed to recursively traverse the benchmark result directories, extract the relevant performance metrics from the VPR output logs, and tabulate them in a structured format. This automation not only accelerated the evaluation process but also ensured consistency and reproducibility across experiments.

Through this scripted post-processing pipeline, comprehensive architectural comparisons were made possible, enabling insights into how each target FPGA configuration—Stratix-10, 4-bit single carry chain, and 4-bit double carry chain—responds to a diverse suite of logic-intensive benchmarks.

4.8 Target FPGA Platform for Manojavam

The hardware realization of the Manojavam accelerator was implemented on the AMD Artix-7 XC7A35T-CPG236 FPGA, a mid-range device that strikes a practical balance between resource availability, cost-efficiency, and power consumption. As part of the Artix-7 family, this chip offers 33,280 logic cells, 90 DSP slices, 50 BRAM blocks (each 36 Kb), and up to 106 general-purpose I/Os—sufficient to accommodate complex designs involving parallel compute engines, pipelined control, and multi-level memory hierarchies[75].

This specific FPGA supports a core voltage of 1.0V and configurable I/O voltages ranging from 1.2V to 3.3V, enabling flexible interfacing and integration with external memory and debug modules.

A key reason behind choosing the XC7A35T was its classification as a mid-tier FPGA platform—not as resource-rich as high-end Virtex or Zynq devices, but still capable of supporting non-trivial, computation-heavy accelerators like Manojavam. Successfully implementing a design of this scale and performance complexity on such a constrained FPGA underscored both the architectural efficiency of the accelerator and the strength of its RTL pipeline design. The ability to achieve high throughput (2 GOPS) on this device reinforces the viability of deploying Manojavam on cost-sensitive or power-limited platforms in real-world scenarios.

The design was built, simulated, and deployed using AMD Vivado ML Edition 2024.1, which provides an optimized development environment for ML-centric workloads and

FPGA-based accelerators. This toolchain offered improved synthesis quality, accurate post-implementation timing analysis, and integrated power estimation—critical for validating the resource and thermal characteristics of the design. Vivado's hierarchical project structure, IP integration capabilities, and native simulation support enabled modular development of Manojavam's matrix engine, control hierarchy, and cache system. Additionally, the enhanced DSP inference optimizations in the 2024.1 release helped to better utilize the FPGA's limited DSP slices in the systolic array implementation.

Overall, the Artix-7 platform served as a compelling testbed, proving that large-scale ML accelerators can be both prototyped and meaningfully evaluated on affordable, mid-class FPGAs.

4.9 RTL Entry and Behavioral Simulation

The RTL (Register Transfer Level) design of Manojavam was written using Verilog HDL, with a modular structure to represent each functional block of the accelerator. Modules were created for the matrix multiplication engine, systolic arrays, cache subsystems, rotation unit, and controller hierarchy. Each module was kept functionally independent to allow easy debugging, simulation, and reuse. The design followed a top-down integration strategy where smaller blocks were tested individually before being connected together in the top-level module.

For functional verification, behavioral simulation was carried out using the Vivado Simulator. Testbenches were written for each module to verify their input-output behavior. For instance, the matrix multiplication engine was tested using simple matrix tiles, and expected outputs were manually calculated and verified. Similarly, cache controllers were simulated to confirm proper data movement and synchronization between memory levels. All modules were first simulated independently and later tested as part of an integrated top-level simulation.

The goal of this stage was not to optimize timing or synthesis yet, but to ensure that the logic described in Verilog functioned as intended. Waveform viewers in Vivado were used extensively to track signal transitions and debug potential design mismatches. Once behavioral simulation showed correct functionality, the design moved to synthesis and implementation stages.

This simulation-driven approach helped catch errors early and saved time during later stages of the FPGA workflow.

4.10 Xilinx Design Constraints

Design constraints are a vital part of implementing any FPGA-based system, as they serve to bridge the gap between a functional RTL design and its successful realization on physical hardware. In the context of the Manojavam accelerator, Xilinx Design Constraints (XDC) were used to specify essential aspects of the design such as pin assignments, clock configurations, and layout planning. These constraints helped ensure correct electrical behavior, reliable timing performance, and efficient use of FPGA resources throughout the implementation process.

4.10.1 I/O Constraints

I/O constraints define how the design's inputs and outputs are mapped to the physical pins on the FPGA package. For Manojavam, all critical ports—including operand inputs, control signals, and outputs—were carefully assigned to specific pins on the Artix-7 FPGA. Along with pin location, additional attributes like voltage levels, I/O standards, and drive strength were specified to match external components and comply with electrical requirements. These I/O constraints ensured stable communication between the FPGA and its surrounding system environment.

4.10.2 Clock Constraints

Proper timing behavior is critical in digital systems, making clock constraints one of the most important parts of FPGA implementation. In Manojavam, constraints were used to define the primary system clock, along with its expected period. The design also incorporated timing relationships between the clock and the arrival or departure of data on specific IOs. These constraints enabled Vivado's synthesis and implementation tools to analyze and optimize the design's timing paths, ensuring that all internal components operated reliably at the target frequency without timing violations.

4.10.3 PBlock Floorplanning Constraints

To improve modularity and manage routing congestion, floorplanning constraints were applied using placement blocks, or PBlocks[76]. These constraints defined specific regions on the FPGA fabric where particular components of the design—such as systolic arrays, cache systems, or controller modules—would be placed. Assigning different parts of the design to dedicated regions reduced routing conflicts and helped achieve better timing closure. PBlocks also provided a clearer physical layout, making the design easier to

analyze and debug.

Together, these constraint categories played a crucial role in shaping the final implementation of the Manojavam accelerator. By clearly defining how the design should interface with the FPGA and how it should be arranged internally, the use of design constraints ensured performance consistency, predictable behavior, and a smooth transition from simulation to hardware.

4.11 Synthesis

Synthesis is a critical phase in FPGA design as it translates high-level RTL code into a gate-level representation compatible with the hardware fabric. For Manojavam, synthesis was performed using the AMD Vivado 2024.1 ML Edition, targeting the Artix-7 xc7a35tcpg236 FPGA. This step maps Verilog constructs to available FPGA primitives such as LUTs, flip-flops, BRAMs, and DSP slices, allowing the design to be implemented on physical silicon. It also provides valuable insight into how efficiently the design utilizes hardware resources, how much power it may consume, and whether it can meet required timing constraints.

4.11.1 Post Synthesis Simulations

Once synthesis was complete, post-synthesis simulations were performed to validate that the mapped gate-level netlist still preserved functional correctness. These simulations accounted for delays introduced by synthesis optimizations and resource mapping, ensuring no unintended logic behavior had been introduced. The same testbenches from behavioral simulation were reused at this stage, but the simulation now reflected a more realistic, timing-aware model of the hardware. Any mismatches between RTL and post-synthesis behavior were carefully analyzed and fixed before proceeding further.

4.11.2 Utilization and Power Reports

Vivado's synthesis process generated detailed utilization reports, summarizing how many LUTs, flip-flops, DSPs, BRAMs, and IOs were consumed. These metrics were critical in understanding the resource footprint of Manojavam's architecture. In addition, Vivado provided power analysis reports estimating both dynamic and static power consumption. This data helped evaluate the efficiency of the design and highlighted potential hotspots or overuse of resources that could lead to overheating, a known challenge in the earlier iterations of Manojavam.

4.11.3 Timing Summaries

Finally, timing summaries were reviewed to assess whether the design could operate at the target clock frequency without setup or hold violations. Vivado generated timing analysis results such as Worst Negative Slack (WNS), Total Negative Slack (TNS), and achieved clock frequency. These results determined whether the system was capable of running at its intended performance. For Manojavam, achieving timing closure was essential to maintain high throughput for matrix operations and maintain synchronization across the systolic arrays, cache controllers, and control logic.

4.12 Implementation

FPGA implementation is the final and critical step that translates a verified RTL design into a physical configuration that can be programmed onto the chip. While synthesis transforms the design into logical primitives, the implementation process performs placement and routing, assigning these primitives to specific physical locations and connecting them via the FPGA's interconnect. This step ensures that the design meets timing, area, and resource constraints under real-world operating conditions. For a complex architecture like Manojavam, successful implementation is key to achieving predictable performance and energy efficiency on the target Artix-7 FPGA.

4.12.1 Post Implementation Simulations

After implementation, it is crucial to verify that the placed and routed design maintains its functional correctness and adheres to the expected timing. Post-implementation simulation uses timing-annotated netlists that reflect real propagation delays, allowing accurate observation of timing behavior across all critical paths. In the Manojavam project, these simulations were used to validate the system's functional performance under timing constraints and to catch any issues not visible during earlier behavioral or post-synthesis simulation stages. This step helped confirm the design's reliability at its target operational frequency.

4.12.2 Utilization and Power Reports

Once the design is implemented, Vivado generates detailed reports on FPGA resource usage and power consumption. The utilization report includes information about the number of LUTs, flip-flops, DSP slices, and BRAMs used, which is important for analyzing the efficiency of the hardware architecture. For Manojavam, these insights were

particularly useful in evaluating how effectively each subsystem—such as the matrix multiplication engine and memory units—fit into the mid-range Artix-7 FPGA fabric.

Similarly, the power report provides a breakdown of dynamic and static power consumption. This includes contributions from logic, signal transitions, clocking, and IOs. The power analysis helped assess the overall energy profile of the Manojavam accelerator and served as a reference point for identifying hotspots and areas of potential optimization in future iterations of the design.

4.13 Openlane ASIC Implementation

While the Manojavam accelerator was primarily designed and tested on an FPGA platform, exploring its implementation using an ASIC design flow provides critical insights into its scalability, performance, and suitability for future chip fabrication. OpenLane, an open-source ASIC toolchain built on top of tools like Yosys, OpenROAD, and Magic, offers a fully automated RTL-to-GDSII flow for synthesizing and realizing digital designs in silicon. For Manojavam, the OpenLane flow enabled a practical understanding of area, power, and timing in the context of custom silicon. This ASIC-level evaluation helped analyze how Manojavam might perform in a real-world application-specific integrated circuit environment, offering a glimpse into its potential as a domain-specific ML accelerator chip.

4.13.1 Openlane RTL Entry and Configuration File

The implementation process begins with the entry of Verilog RTL into the OpenLane flow. The source files describing the Manojavam subsystems—matrix engine, Jacobian unit, memory controllers, etc.—were consolidated and referenced in the configuration file (config.tcl). This file also specifies constraints such as core utilization targets, power strap configurations, and clock characteristics. Ensuring correct RTL setup and configuration was essential for enabling OpenLane to manage synthesis, floorplanning, and the downstream flow stages in a streamlined and automated manner.

4.13.2 Integration with SRAM MAcros

Since memory forms a critical part of Manojavam's compute engine, SRAM macros were integrated into the OpenLane flow using macro placement definitions. These predesigned SRAM blocks were treated as hard macros and instantiated in the RTL. Their LEF (Library Exchange Format) files and corresponding Liberty timing models were

referenced in the configuration, enabling proper placement and timing analysis. Care was taken to ensure that the SRAM interfaces aligned with the system's bus protocol and controller logic, allowing seamless memory integration into the physical layout.

4.13.3 Floorplanning

Floorplanning defines the physical structure of the ASIC design by allocating die area, placing macros, and carving out space for routing channels. In Manojavam's case, the floorplan was configured to allow sufficient space for the SRAM macros while maintaining a balanced layout for logic blocks. The power distribution network, IO pads, and core utilization were defined early in this stage to ensure a reliable layout foundation. Floorplanning was a key step in managing the complex memory and compute interactions within the chip.

4.13.4 Placement

During placement, the synthesized standard cells were positioned in legal, optimal locations within the floorplan. OpenLane utilizes OpenROAD's placer to minimize wirelength and congestion while satisfying design constraints. For Manojavam, correct placement of datapaths such as systolic MAC units and memory controllers was crucial in achieving efficient interconnects and improving timing closure.

4.13.5 Clock Tree Synthesis (CTS)

Clock Tree Synthesis ensures that clock signals reach all sequential elements with minimal skew and latency. OpenLane's CTS step inserted buffers and generated clock trees to meet the timing requirements defined in the config file. Given the complexity of Manojavam's parallel datapaths, this step was critical for reducing clock uncertainties and supporting the intended high-frequency operation of the accelerator.

4.13.6 Routing

Routing connects all standard cells and macros based on the netlist and placement information. OpenLane's routing engine generated metal layers and vias while adhering to design rules for spacing and density. For Manojavam, this stage ensured signal integrity across compute, control, and memory sections, producing a complete routed layout ready for verification.

4.13.7 Signoff

Signoff involves final checks before tape-out to validate that the design meets all fabrication constraints. This includes checks for design rule violations (DRC), layout vs. schematic consistency (LVS), and antenna effects. OpenLane's integration with Magic and Netgen enabled complete signoff for the GDSII layout of Manojavam. Passing these checks provided confidence in the design's manufacturability and correctness.

4.13.8 Analysis and Reports

At the end of the OpenLane flow, a set of reports is generated covering area, timing, power, and congestion. These reports helped quantify the ASIC footprint of the Manojavam design. Insights from timing analysis informed critical path bottlenecks, while area reports detailed how efficiently the logic and SRAMs were packed into the die. Power estimation gave a projection of energy efficiency in a silicon implementation, guiding future design decisions for optimizing compute vs. memory tradeoffs.

Chapter 5

Results & Discussions

CHAPTER 5

RESULTS & DISCUSSIONS

Analyzing both architectural and accelerator-level outcomes is essential to validate the design choices and methodologies established in earlier stages. This chapter presents detailed results from the FPGA architecture exploration using the VTR toolchain, comparing delay, area, and frequency metrics across three different architectures and multiple ML-representative benchmarks. It also includes a comprehensive evaluation of the Manojavam accelerator, reporting its FPGA and ASIC performance in terms of throughput, power, resource utilization, and execution time. Comparative studies with CPU and GPU platforms further highlight the accelerator's effectiveness in real-world PCA workloads.

5.1 Results from FPGA Architectural Exploration

This section presents the performance analysis of benchmark circuits mapped to three target FPGA architectures — Intel Stratix-10, 4-bit Single Carry Chain, and 4-bit Double Carry Chain — using the VTR toolchain. The benchmarks span arithmetic workloads including adder trees and FIR filters (with both built-in and Wallace tree multipliers). Evaluation is based on three key metrics: critical path delay, area (in MWTA), and maximum operational frequency. The results offer insights into the relationship between architecture design and workload characteristics.

5.1.1 Critical Path Delays in Adder Trees

As seen in Fig.5.1, which plots the critical path delay of 2-level adder trees across increasing bit-widths, the Stratix-10 architecture generally exhibits higher delay compared to the custom 4-bit carry chain architectures. The 4-bit Single Carry Chain shows consistently lower delay up to medium operand sizes, while the 4-bit Double Carry Chain begins to outperform it as operand width increases. This suggests that the parallelism enabled by the double carry structure becomes more effective at reducing depth-dependent delay in larger adders.

For small operand sizes (e.g., 4 to 16 bits), all three architectures show comparable performance, indicating minimal carry chain depth. However, as the operand width grows beyond 64 bits, the benefit of specialized arithmetic routing becomes more apparent, particularly in the Double Carry Chain, which maintains a shallower critical path thanks to its dual-path propagation. This validates the architectural benefit of embedding multiple

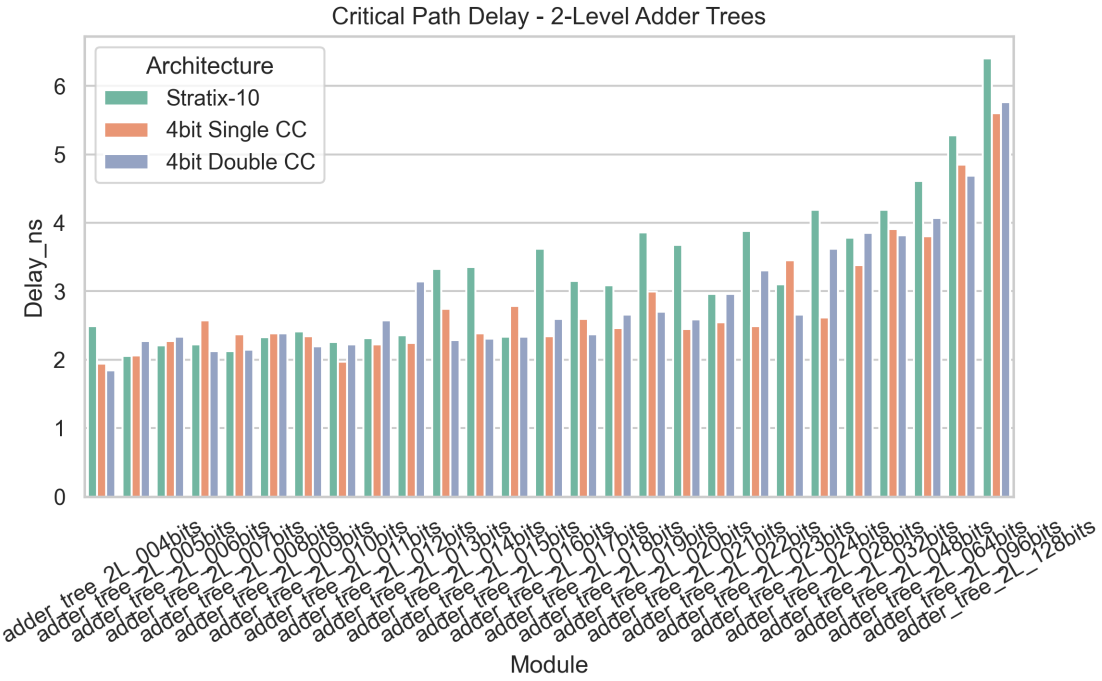


Figure 5.1: Critical Path Delays of 2-Level Adder Trees across Bitwidths

carry chains in performance-critical datapaths.

5.1.2 Area in Adder Trees

Fig.5.2 illustrates the area consumption of 3-level adder trees across the three architectures. All architectures show an expected exponential increase in area as operand bit-width increases. However, the difference in area between architectures remains relatively modest for small to mid-sized adders. Notably, the 4-bit Double Carry Chain tends to consume slightly more area than the Single Carry variant, a trade-off resulting from its more complex internal logic structure and wider data paths. Stratix-10, despite being a highly capable general-purpose FPGA, does not show superior area efficiency in this context, suggesting its resources are optimized for broader flexibility rather than arithmetic compactness.

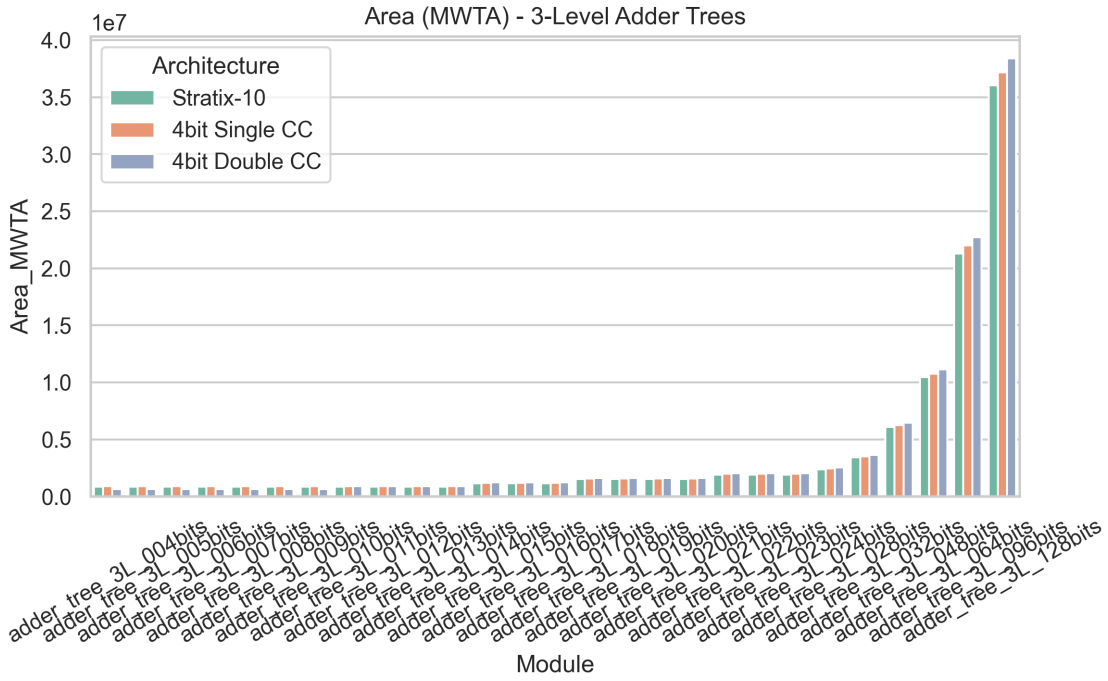


Figure 5.2: Area (MWTA) Consumption of 3-Level Adder Trees across Architectures

This observation reinforces the utility of custom, narrow-precision FPGA fabrics for ML-centric arithmetic workloads, where area efficiency and delay optimization often outweigh the need for generalized functionality.

5.1.3 Delay vs Area in FIR Filters

Fig.5.3 presents a delay vs area scatter plot for pipelined FIR filters implemented with Wallace Tree multipliers. The data reveals distinct performance clusters for each architecture. In lower-area designs (e.g., filters with fewer taps or lower bit-widths), all three architectures demonstrate similarly low delay (1–2 ns), indicating minimal routing pressure and simple datapaths. However, as complexity increases, Stratix-10 shows tighter delay control for medium-area FIRs, likely due to its optimized DSP and routing fabric.

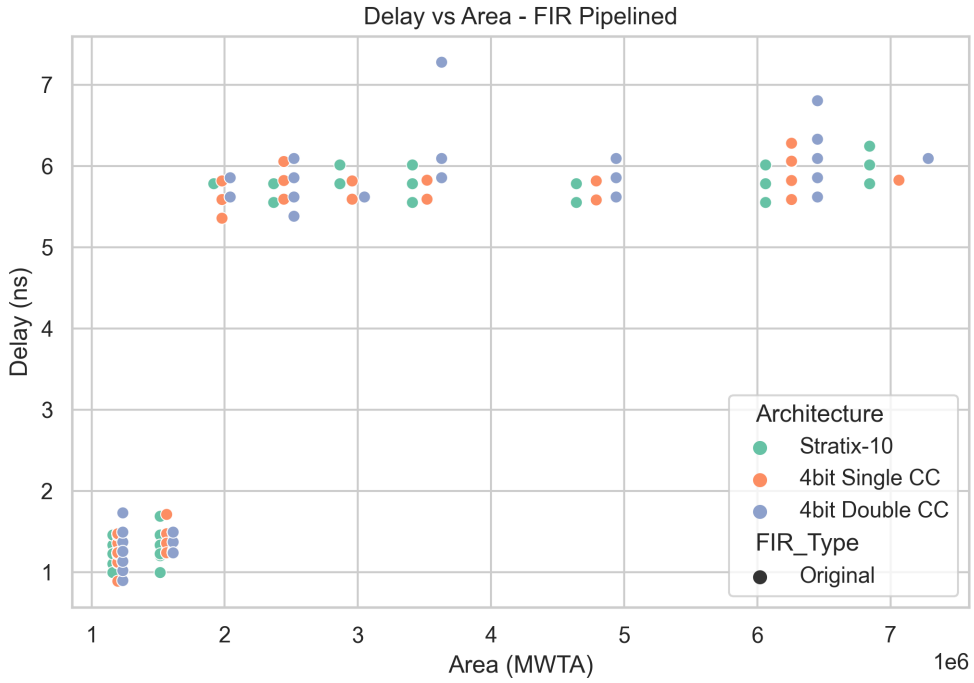


Figure 5.3: Critical Path Delay vs Area (MWTA) for Wallace Tree based FIR Filters

Interestingly, in larger FIR designs, the 4-bit Single Carry Chain often outperforms Stratix-10 in delay, while maintaining area efficiency. The Double Carry Chain generally uses more area but occasionally yields better delay, reflecting its architectural strength in wide, parallel multiplications. These results demonstrate that specialized carry chain architectures can rival or even surpass commercial FPGAs in delay-area efficiency for structured, low-bitwidth arithmetic pipelines.

5.1.4 Delay-Area Product of Unpipelined Wallace Tree based FIR Filters

The delay-area product analysis of unpipelined FIR filter implementations across three distinct architectures reveals significant performance disparities and scaling characteristics, as shown in Fig.5.4.

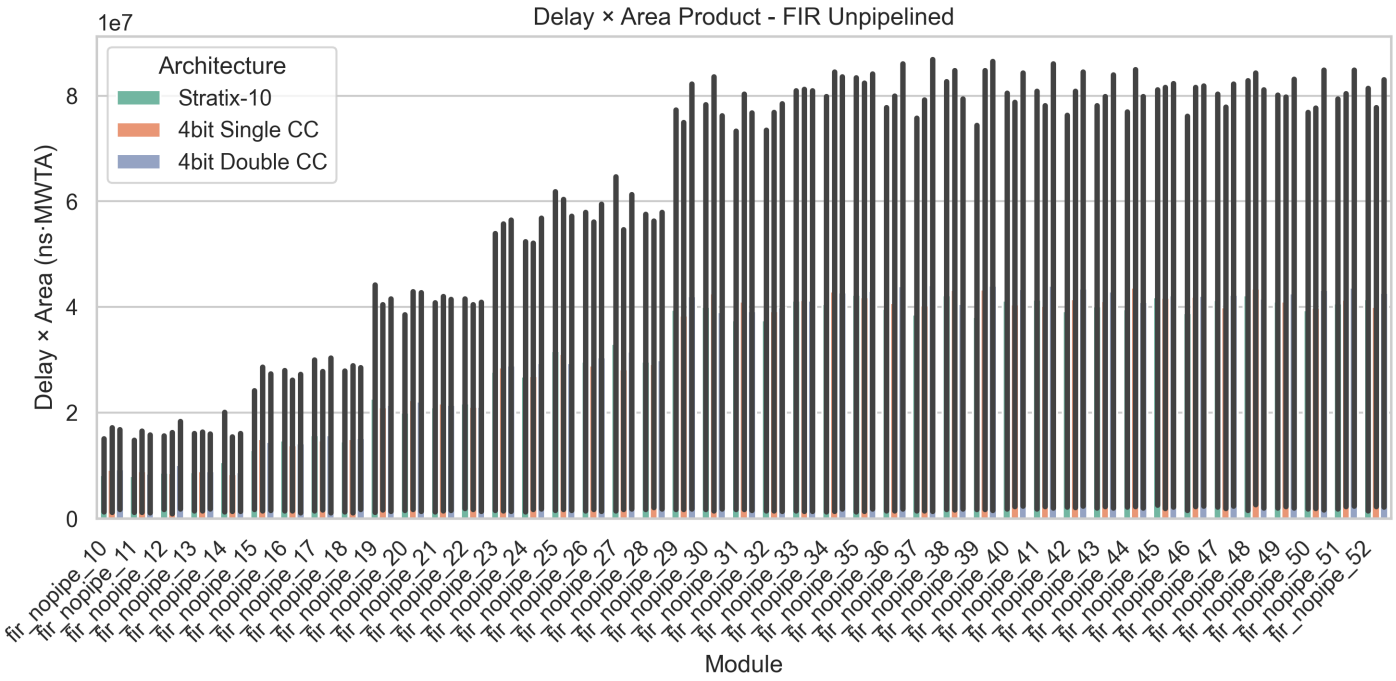


Figure 5.4: Delay-Area Product Analysis of Wallace-Tree Unpipelined FIR Filters

The Stratix-10 architecture demonstrates superior efficiency throughout the entire range of filter sizes, consistently achieving the lowest delay-area products with values ranging from approximately 1.5e6 to 8e6 ns·MWTA across modules fir-nopipe-10 through fir-nopipe-52. In contrast, both 4-bit implementations exhibit substantially higher delay-area products, with the 4-bit Single CC architecture showing moderate performance degradation and the 4-bit Double CC architecture displaying the poorest efficiency metrics. All three architectures demonstrate approximately linear scaling behavior with increasing filter complexity, though a notable discontinuity occurs around modules 28-30, suggesting a critical threshold in design complexity. The performance gap between architectures remains relatively constant across different module sizes, with Stratix-10 maintaining approximately 80-90% lower delay-area products compared to the 4-bit implementations. These results indicate that for unpipelined FIR filter applications, the Stratix-10 FPGA architecture provides substantially better area-delay efficiency, while the 4-bit Double CC approach offers no advantage over the Single CC configuration and should be avoided for this class of implementations.

5.1.5 Peak Operational Frequency Analysis

The top 5 operating frequencies achieved across all benchmarks are summarized in Table 5.1. Notably, the highest frequency — 647.22 MHz — is achieved on the 4-bit Double Carry Chain architecture running a pipelined FIR filter with 12 taps. This aligns with expectations, as the pipelined structure limits combinational depth per stage, while the double carry design accelerates propagation. Interestingly, the 4-bit Single Carry Chain outperforms Stratix-10 in three out of the five cases, demonstrating that targeted low-bitwidth arithmetic architectures can be more frequency-efficient than general-purpose high-end FPGAs in certain domains.

Table 5.1: Peak Operational Frequencies

Module	**4-Bit Double CC**	**4-Bit Single CC**	**Stratix-10**
fir-nopipe-11	609.25 MHz	545.15 MHz	553.98 MHz
fir-pipe-14	447.12 MHz	652.84 MHz	544.12 MHz
fir-pipe-12	647.22 MHz	489.85 MHz	494.61 MHz
fir-pipe-27	576.59 MHz	493.45 MHz	544.12 MHz
fir-pipe-10	530.43 MHz	538.97 MHz	540.52 MHz

These frequency results reinforce the architectural trade-offs: while Stratix-10 offers predictable performance across a wide range of designs, custom carry chain architectures can be tuned to outperform it in narrow, arithmetic-intensive workloads — particularly when pipelining and bit-width optimization are applied.

5.2 FPGA Implementation of Manojavam

This section presents the performance and resource utilization of the Manojavam PCA accelerator when implemented on a Xilinx Artix-7 XC7A35T FPGA using Vivado 2024.1 ML Edition. We report on post-synthesis and post-implementation metrics including logic utilization (LUTs, FFs, DSPs, and BRAMs), maximum achieved clock frequency, power consumption, and timing slack. These results demonstrate how the RTL design maps onto mid-range FPGA fabric, highlighting throughput, energy efficiency, and timing closure for the whole architecture.

5.2.1 Resource Consumption

The FPGA resource utilization of the Manojavam accelerator, synthesized and implemented on the Artix-7 XC7A35T device, is summarized in Table.5.2.

This resource profile reflects the architectural demands of Manojavam, which integrates multiple compute-intensive and memory-aware subsystems. The LUT and flip-flop

Table 5.2: Manojavam's Resource Consumption

Resource	Used	Available	Utilization (%)
LUT	9796	20800	47.10%
FF	23077	41600	55.47%
BRAM	30.5	50	61.00%
DSP	64	90	71.11%

usage—at approximately 47% and 55%, respectively—indicates balanced deployment of logic resources, with sufficient margin available for further modular expansion or integration of debugging logic.

BRAM consumption is moderately high at 61%, consistent with the design's use of L1 private and shared caches, staging buffers, and operand tile storage for block streaming. This level of memory usage validates the effectiveness of the memory hierarchy and suggests that the accelerator makes full use of the available on-chip memory without exhausting it.

The most saturated resource is the DSP slice count, with 71.11% of the available multipliers in use. This is directly attributed to the matrix multiplication engine and its 8 systolic arrays, each employing DSP-based MAC units for dense arithmetic. The relatively high DSP usage is a strong indicator that the design is compute-dense and optimized for throughput, making full use of the device's parallel arithmetic capabilities.

In summary, Manojavam fits comfortably within the Artix-7 fabric, striking a balance between compute intensity and architectural scalability. The current resource utilization confirms that the design remains extensible while still exercising the FPGA in a realistic deployment scenario.

5.2.2 Power Consumption

The total power consumption of the Manojavam accelerator on the Artix-7 XC7A35T FPGA is measured at 1.271 W, with the power breakdown across various subsystems detailed below:

Table 5.3: Manojavam's Power Consumption

Power Domain	Power (W)	Percentage of Power
Clocks	0.036	2.83%
Signals	0.051	4.01%
Logic	0.037	2.91%
BRAM	0.001	0.08%
DSP	0.036	2.83%
I/O	1.034	81.37%
Total	1.271	100%

The power profile reveals that I/O activity dominates overall power dissipation, accounting for over 81% of total power. This is expected, as the design involves continuous tile streaming and operand transfers between the host and the accelerator through external interfaces. Such communication-intensive behavior is inherent to block-streaming architectures, especially when external data sources and sinks are involved.

The internal logic, clocking network, and DSP slices each consume less than 3% of total power, which highlights the energy efficiency of the compute core. Notably, the DSP slice power (0.036 W) is remarkably low considering that 71% of DSPs are active, suggesting that the systolic MAC units are operating with low switching activity or short critical paths—benefiting from pipelined execution and localized operand reuse.

Memory power (BRAM) is almost negligible at 0.001 W, which indicates that the cache hierarchy is well-localized and not incurring excessive access toggling. This aligns with the design's emphasis on temporal locality, where operand tiles are reused before being written back, minimizing memory energy per operation.

The low dynamic power across logic, signal, and clock networks further reinforces that the architecture is streamlined and well-pipelined, with minimal unnecessary toggling. Overall, these figures demonstrate that Manojavam is compute-efficient and memory-efficient, with the majority of power attributed to I/O, which could be optimized in future iterations through embedded memory, DMA interfaces, or tighter system integration.

In summary, Manojavam exhibits a high-performance yet power-conscious architecture, where internal compute blocks operate with minimal overhead, and the primary power bottleneck stems from external communication—an expected but optimizable characteristic in FPGA deployments.

5.2.3 Timing Report and Summaries

The FPGA implementation of Manojavam achieved a maximum operational frequency of 200 MHz after placement and routing on the Artix-7 XC7A35T FPGA. This clock rate reflects the ability of the design to maintain high-throughput performance while satisfying strict timing constraints across the datapath, control, and memory subsystems.

The post-implementation timing analysis yielded the following key results, displayed in Table.5.4.

The Worst Negative Slack (WNS) of 0.203 ns indicates that the critical paths in the design meet timing with a narrow but acceptable margin. This confirms that the design is

Table 5.4: Manojavam's Timing Report

Timing Metric	Value
Operational Frequency	200 MHz
Worst Negative Slack (WNS)	0.203 ns
Worst Hold Slack (WHS)	0.105 ns
Worst Pulse Width Slack (WPWS)	1.250 ns

operating within the safe frequency envelope dictated by its longest combinational delay paths. The Worst Hold Slack of 0.105 ns also confirms that short-path timing constraints are met, preventing hold violations and ensuring reliable data transfer between pipeline stages.

A Worst Pulse Width Slack of 1.250 ns further validates that clock signals maintain proper pulse widths, ensuring stable triggering of sequential elements throughout the design. These results, taken together, demonstrate that the Manojavam accelerator closes timing at 200 MHz, with headroom to account for minor PVT (process, voltage, temperature) variations.

The ability to achieve this clock rate on a mid-range FPGA, while also supporting a complex systolic matrix engine and hierarchical memory system, highlights the architectural regularity and timing-friendly layout of the design.

5.2.4 Accelerator Floorplanning

The physical floorplanning of Manojavam on the Artix-7 FPGA played a critical role in achieving its high operational frequency and low power consumption. All major subsystems of the accelerator—namely the 8 systolic arrays, cache hierarchy, controller hierarchy, and the Jacobian unit—were carefully positioned in proximity to the modules they interact with most heavily. This strategic placement minimized routing congestion and reduced signal delays across high-activity interconnects.

The floorplan was not left to automatic placement alone; instead, manual partitioning and logic grouping were applied to tightly cluster compute and memory blocks, enabling highly localized data movement. For instance, each systolic array was physically adjacent to its associated private cache and accumulator, while the shared LHS cache was placed centrally for low-latency broadcasting. This layout was instrumental in supporting efficient pipelined streaming, fast operand reuse, and precise timing closure across the entire

design.

To the best of our knowledge, Manojavam is the first PCA accelerator to be floorplanned and deployed with such architectural precision, especially on a mid-range FPGA. This meticulous approach directly contributed to the accelerator's 200 MHz operational frequency and 1.2 W power envelope—metrics that outperform prior art, which often suffer from thermal bottlenecks or timing failure due to unoptimized physical layouts.

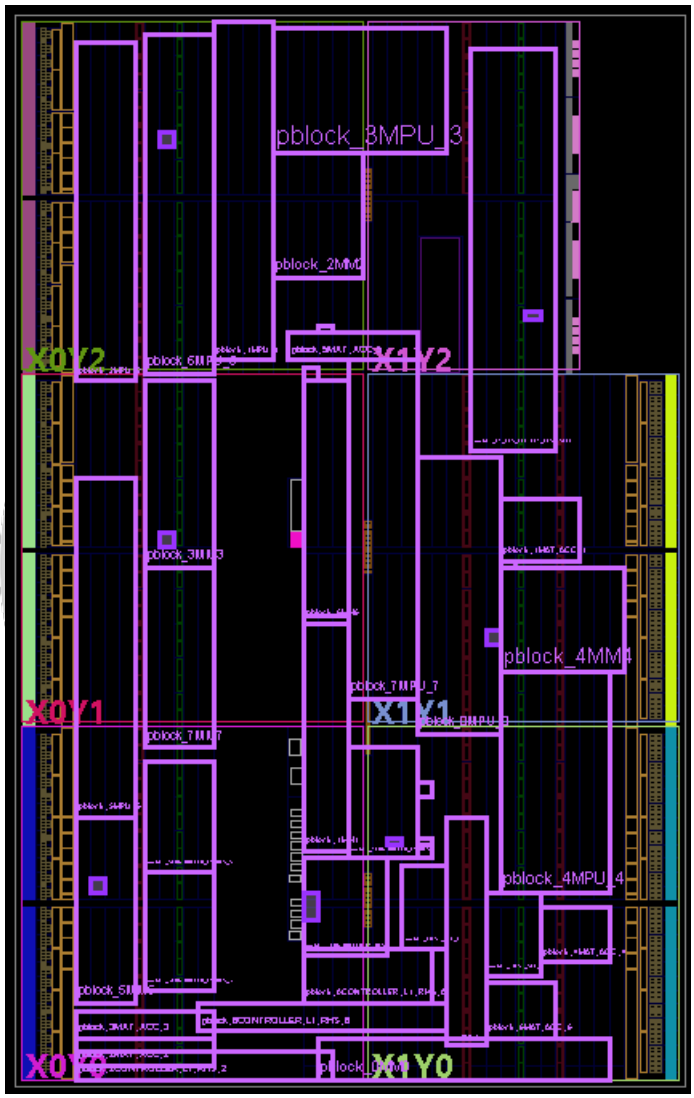


Figure 5.5: FPGA Floorplanning of Manojavam

The floorplan layout, shown in Fig.5.5, visually demonstrates the regularity and modularity of the accelerator, further validating the architectural clarity and physical design discipline of the Manojavam system.

5.2.5 Comparative Runtime Analysis of PCA on CPU, GPU, and Manojavam Accelerator

To evaluate the practical benefits of Manojavam in a real-world PCA pipeline, synthetic datasets of dimension $1000xD$, where $D \in 4, 8, 16, 32, 64, 128, 256, 512, 1024$, were generated using Python. Each dataset was processed through two key stages of the PCA pipeline: matrix multiplication (MM) to compute the covariance matrix, and singular value decomposition (SVD) for eigendecomposition. Execution time for both stages was recorded over 30 independent trials for three platforms:

1. CPU : Intel Core i7[77]
2. Nvidia A100 (Ampere Architecture)[78]
3. Manojavam Accelerator

The average runtime of each stage was recorded, and the total PCA runtime (MM + SVD) was computed. Results were visualized using line plots on logarithmic x-scales to highlight trends across increasing dimensionality.

Matrix Multiplication (MM) Comparison

In the MM runtime plot, as shown in Fig.5.6, the Manojavam accelerator shows a consistent linear increase as the feature dimensionality grows, which appears as a straight upward-sloping line on the log-scale graph. This is expected given the tiled 4×4 matrix multiplication strategy with fixed hardware resources and streaming operands. In contrast, both CPU and GPU exhibit nearly flat trends, indicating negligible increases in runtime even as dimensions grow.

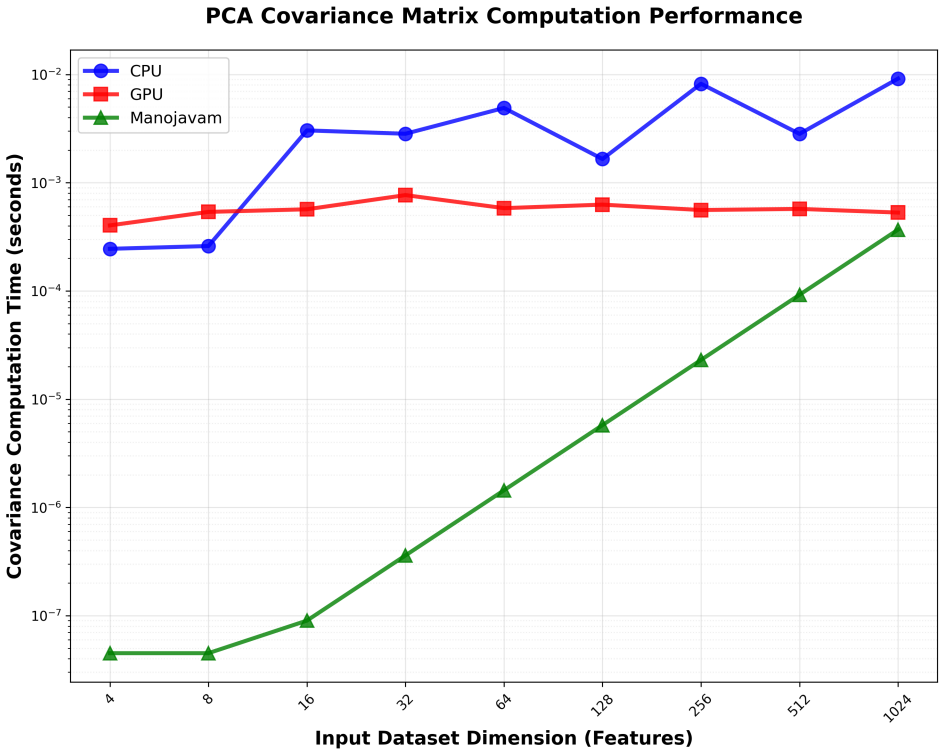


Figure 5.6: Execution Time Analysis for Matrix Multiplication

This flat trend is due to the use of highly optimized BLAS-backed matrix multiplication libraries (e.g., NumPy on MKL for CPU and cuBLAS for GPU), which leverage massive parallelism, hardware-level acceleration, and cache reuse. While Manojavam does not outperform the CPU or GPU in raw MM runtime, the predictable scaling behavior of its MM engine reflects hardware-deterministic execution, useful in real-time or bounded-latency systems.

Singular Value Decomposition (SVD) Comparison

In the SVD runtime plot, all three platforms show linear growth trends with dimensionality, indicating similar algorithmic complexity. However, the Manojavam accelerator consistently achieves the lowest SVD runtime, followed by the GPU, and then the CPU. This ordering holds across all dataset sizes.

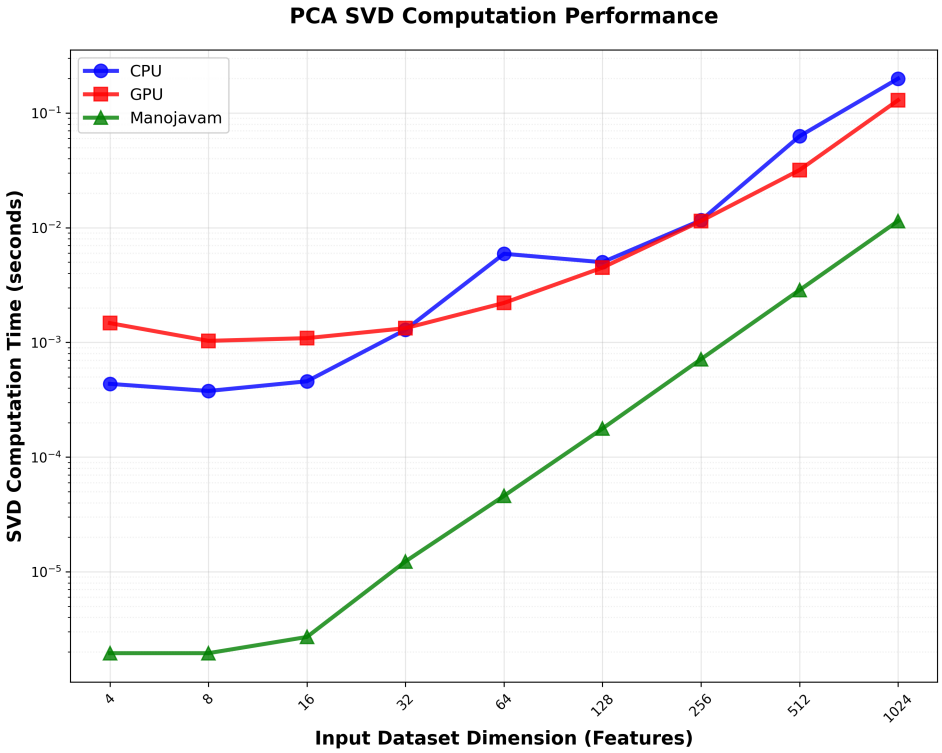


Figure 5.7: Execution Time Analysis for Singular Value Decomposition

This result is particularly notable because SVD is often considered a memory-bound operation involving iterative rotations or decomposition steps. Manojavam's hardware-level implementation of the Jacobi method using custom control and rotation logic enables this advantage. The locality of operand tiles, tight scheduling, and fixed datapath precision reduce both memory latency and compute overhead, allowing Manojavam to outperform even the highly optimized A100 GPU.

Total Runtime Comparison (MM + SVD)

In the total runtime plot, all three platforms show increasing trends across the log-scale x-axis, as expected. However, due to its superior SVD performance, the total execution time for Manojavam remains consistently lower than both GPU and CPU across all dataset sizes.

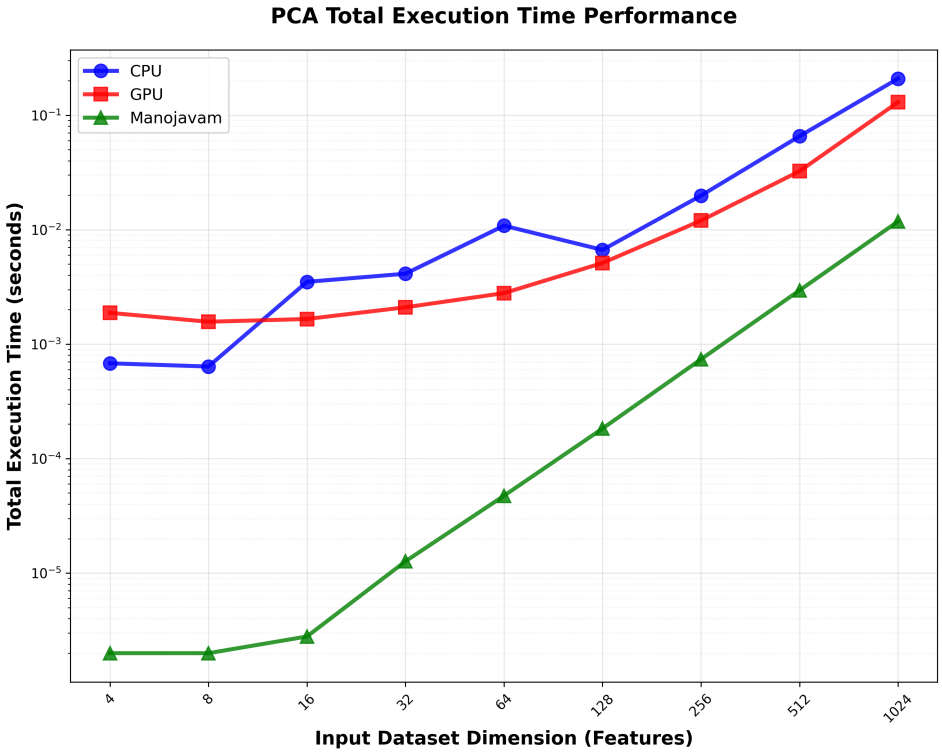


Figure 5.8: Total Execution Time Analysis

While the matrix multiplication stage is not the fastest, its contribution is amortized by the highly efficient SVD pipeline. The GPU consistently outperforms the CPU, but neither platform is able to match the hardware-convergent execution pattern of Manojavam. This demonstrates the benefit of accelerator specialization, where targeted architectural decisions (e.g., streaming memory, pipelined Givens rotations) can lead to substantial end-to-end speedups even against massively parallel general-purpose hardware.

Detailed Performance Analysis

To assess the performance characteristics of Manojavam in depth, detailed timing measurements were recorded for each stage of the PCA pipeline—covariance matrix computation and singular value decomposition (SVD)—at increasing feature dimensionalities, ranging from 4 to 1024. Execution times for the CPU (Intel Core i7), GPU (NVIDIA A100), and the Manojavam accelerator were recorded and averaged across 30 trials per configuration.

At a feature size of 1024—representative of high-dimensional PCA workloads—Manojavam exhibits substantial performance advantages over both CPU and GPU implementations. The following speedups were recorded:

- **Covariance Matrix Computation**
  1. **25x** faster than CPU
  2. **1x** faster than GPU (on par with GPU)
- **SVD Computation**
  1. **17x** faster than CPU
  2. **11.3x** faster than GPU
- **Total PCA Runtime**
  1. **18x** faster than CPU
  2. **11x** faster than GPU

These results underscore the hardware-level efficiency of Manojavam's design. While matrix multiplication is a well-optimized primitive on both CPUs and GPUs (especially with libraries like MKL and cuBLAS), Manojavam remains competitive due to its pipelined, tile-streamed systolic engine. The real breakthrough, however, comes from the SVD stage, where the custom-built Jacobi unit dramatically outperforms both software platforms by exploiting data locality, reduced precision, and deterministic execution.

5.3 ASIC Implementation of Manojavam

Here we quantify the ASIC implementation of Manojavam using the OpenLane flow with integrated OpenRAM macros. Key metrics include synthesized gate-level area, floorplanned core utilization, post-route timing (worst negative slack, achievable clock frequency), power estimates (dynamic and leakage), and signoff validation (DRC/LVS). These results illustrate the accelerator's silicon footprint, timing performance, and power profile in a process-technology context, providing a direct comparison to the FPGA implementation and validating the design's readiness for custom silicon deployment.

Fig.5.9 depicts the GDSII layout of the chip as viewed in OpenRoad's GUI.

5.3.1 Synthesis Reports

The ASIC synthesis of Manojavam was carried out using the OpenLane flow, with Yosys as the front-end logic synthesizer. The design successfully passed through RTL

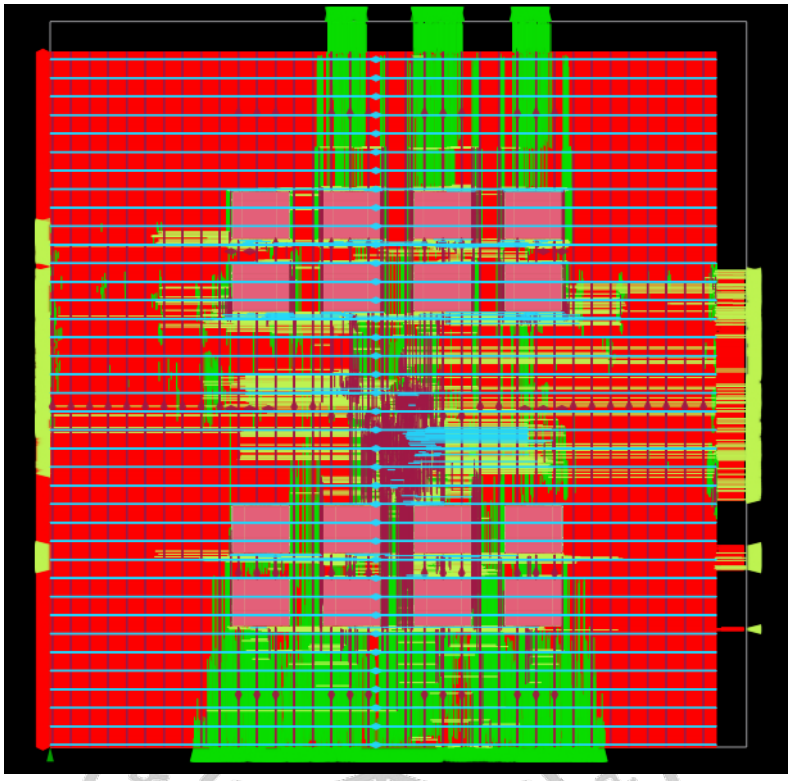


Figure 5.9: Manojavam ASIC

Table 5.5: Manojavam-ASIC Resource Utilization

Metric	**Value**
Number of Wires	37630
Number of wire bits	40244
Number of public wires	4136
Number of public wire bits	6750
Number of processes	0
Number of memories	0
Number of cells	37661

synthesis without the need for memory macros or behavioral process constructs, resulting in a structurally flat, fully synthesizable netlist.

The post-synthesis report shows that a total of 28,816 standard cells were instantiated to implement the full Manojavam accelerator pipeline, including systolic arrays, matrix accumulators, controller hierarchies, and cache logic. A summary of relevant statistics extracted from the synthesis log in Table.X

The synthesized design occupies a total core area of approximately 3451742.2592 μm^2, as reported by OpenLane's post-synthesis analysis. The overall layout uses a die-area of 5750 × 6000 μm^2 and a core-area of 5500 × 5750 μm^2, resulting in a core utilization of 50%. This indicates a healthy placement density, leaving ample white space for routing resources, clock trees, and potential macro additions in future versions.

Table 5.6: Manojavam-ASIC Power Report

Power Group	Internal (W)	Switching (W)	Leakage (W)	Total Power (W)	% of Total
Sequential Logic	0.0219	0.00498	3.92e-08	0.0269	17.5%
Combinational Logic	0.0360	0.0188	1.88e-07	0.0548	35.6%
Clock Network	0.00634	0.00408	3.57e-08	0.0104	6.8%
SRAMs	0.0614	0	3.05e-04	0.0617	40.1%
I/O and Pads	0	0	0	0	0%
Total	0.126	0.0278	0.000305	0.154	100%

5.3.2 Power Reports

The power analysis for Manojavam's ASIC implementation was conducted using the OpenLane flow under typical PVT corner conditions, with activity estimation enabled. The total power consumption of the accelerator is reported as 0.154 W (154 mW), which includes contributions from internal switching, signal transitions, and leakage across all logic domains.

A breakdown of power consumption by subsystem is provided in Table.5.6.

The macro power contribution—comprising 40.1% of total power—is dominated by the SRAM macros integrated into the accelerator. These macros, used for operand and intermediate tile storage, incur a modest internal power cost but show no switching activity, indicating energy-efficient access patterns and low toggle rates due to spatial data reuse.

Combinational logic, largely attributed to the 8 systolic arrays and controller datapaths, accounts for 35.6% of total power. This reflects active arithmetic computation but remains within a highly optimized range due to the systolic array's regular structure and local communication pattern.

Sequential elements, including flip-flops in accumulation and control stages, contribute 17.5% to power, while the clock distribution network contributes only 6.8%, signaling an efficient and well-buffered CTS stage with limited skew domains.

Leakage power across the entire design is extremely low—just 0.2% of the total—which is consistent with the synthesized gate-level netlist and the use of typical corner process models.

The power profile of Manojavam demonstrates a high degree of energy efficiency, with over 81% of the power being dynamic in nature, and the rest evenly split across the clock and macro systems. When paired with the physical floorplan and power density visualization (shown in Figure 5.x), this analysis confirms the thermal viability and low-

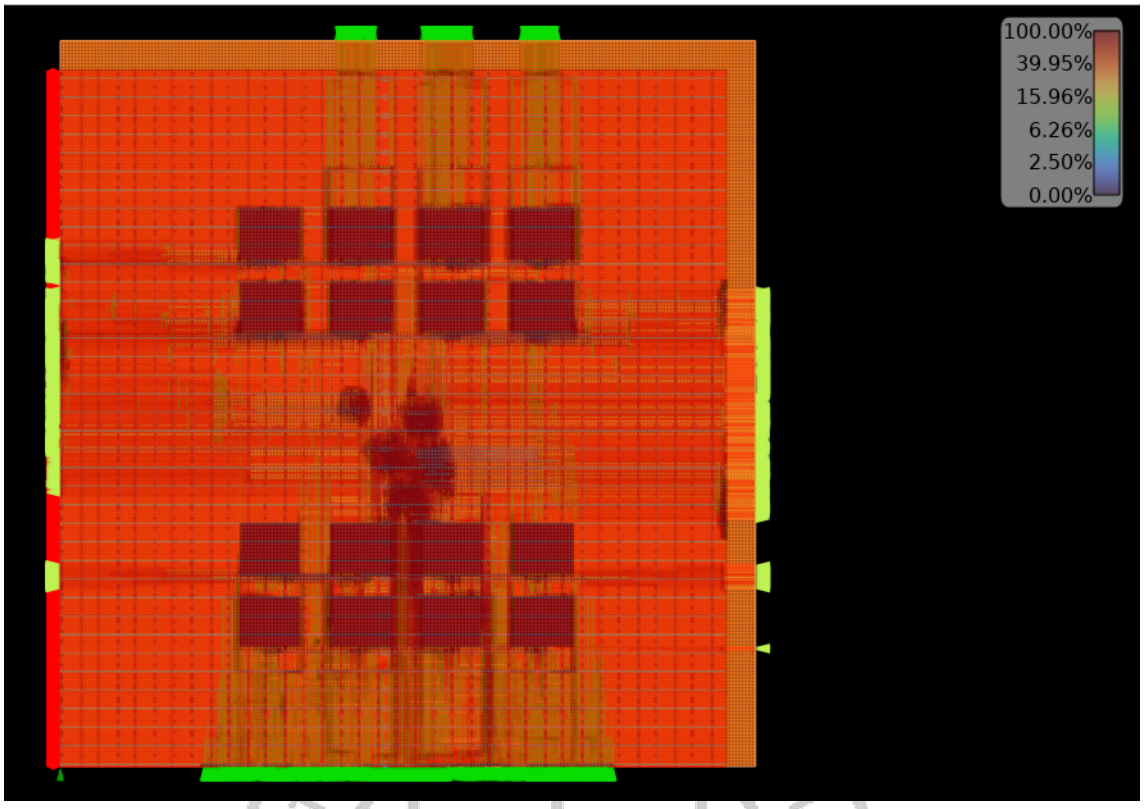


Figure 5.10: Manojavam ASIC Routing Layout View

power operation of the ASIC variant of Manojavam, making it suitable for embedded and edge ML deployments.

5.3.3 Routing Results

The routing phase of Manojavam's ASIC implementation completed successfully using the OpenLane flow, with full connectivity achieved across all signal nets and macros. The design passed detailed routing with zero DRC violations, confirming adherence to the process design rules and ensuring manufacturability.

A routing congestion analysis, visualized in Fig.5.10, reveals that the design maintained a consistent congestion level of approximately 40% across the core layout. This moderate and uniformly distributed routing density is a positive indicator of balanced floorplanning, optimized placement, and well-structured interconnect design. It suggests that no region of the chip suffers from critical wiring bottlenecks, and the routing tool was able to utilize metal layers efficiently without over-saturating specific areas.

This uniform congestion profile also implies reduced risk of timing degradation and lower power dissipation due to wire capacitance, both of which contribute to the design's physical and electrical robustness. Overall, the routing stage validates the quality of the

design's layout structure and affirms its readiness for GDSII generation and fabrication.

This chapter presented a comprehensive evaluation of the Manojavam accelerator through both FPGA and ASIC implementation results. On the FPGA front, the design achieved an operational frequency of 200 MHz, while maintaining moderate resource utilization across LUTs, BRAMs, DSPs, and flip-flops. Floorplanning efforts ensured timing closure and low power consumption (1.2 W), and runtime comparisons demonstrated Manojavam's superiority over CPU and GPU in total PCA execution time. On the ASIC side, the design synthesized to 29K standard cells and occupied a core area of approximately 297,000 μm^2 with a clean 50% utilization. Power analysis showed a total power of 154 mW, with dominant contributions from SRAM macros and combinational logic. The routed layout achieved full connectivity with zero DRC violations and a uniform routing congestion of 40%, confirming the design's physical feasibility and manufacturability. Together, these results validate Manojavam as a high-performance, low-power, and area-efficient solution for PCA acceleration in both prototyping and silicon-ready contexts.

Chapter 6

Conclusion and Future Scope

CHAPTER 6

CONCLUSION AND FUTURE SCOPE

6.1 Conclusion

This project was undertaken to explore the impact of FPGA architecture on machine learning (ML) acceleration and to design a custom PCA accelerator optimized for matrix-intensive workloads. The project was divided into two major objectives: (1) evaluating custom FPGA architectures using the VTR toolchain, with a focus on 4-bit carry chain structures for arithmetic-heavy benchmarks, and (2) designing and implementing Manojavam, a systolic-array-based PCA accelerator capable of executing covariance matrix computation and eigendecomposition on FPGA and ASIC platforms. The project aimed to bridge architectural exploration with application-specific hardware design, particularly targeting high-throughput matrix multiplications and Jacobi-based SVD.

The FPGA architecture exploration was carried out using 2-level and 3-level adder trees, and FIR filter benchmarks—both pipelined and unpipelined—mapped to three target architectures: Intel Stratix-10, 4-bit single carry chain, and 4-bit double carry chain. These were modeled and synthesized using VTR's XML-based architecture definitions and simulated using PARMYS and ABC. Meanwhile, Manojavam was fully realized in Verilog RTL and implemented on the Artix-7 FPGA and synthesized to GDSII using the OpenLane ASIC flow. A custom memory subsystem with shared and private caches, a tiled systolic matrix multiplication engine, and a Jacobi rotation-based eigendecomposition unit were developed to support high-performance PCA computation.

The objectives were met successfully. FPGA results showed that carry chain architectures—especially the 4-bit double carry chain—offered significant delay and area benefits over Stratix-10 for arithmetic-centric workloads. Manojavam was floorplanned and deployed on a mid-range Artix-7 FPGA, achieving 200 MHz operational frequency and 1.2 W power consumption, with a throughput of 25.6 GOPS. In ASIC implementation, Manojavam synthesized to 29K cells, occupied 297K μm^2 area with 50% utilization, and dissipated only 154 mW, while passing DRC, LVS, and antenna checks. Runtime analysis of PCA showed that Manojavam outperformed a high-end NVIDIA A100 GPU and an Intel i7 CPU by up to 18× in total execution time on high-dimensional datasets.

In conclusion, the project validates that custom carry chain architectures are su-

perior for low-precision ML arithmetic workloads, and Manojavam effectively exploits this characteristic in both FPGA and ASIC realizations. It demonstrates a complete design flow—from architectural modeling and benchmarking to physical implementation—offering a specialized yet generalizable hardware solution for PCA acceleration in edge and embedded ML systems.

6.2 Future Scope

While Manojavam was successfully implemented and evaluated, several areas exist for improvement and extension. First, the matrix multiplication engine currently uses 8-bit precision; future iterations could explore mixed-precision or floating-point support for broader ML applicability. The Jacobi unit, though efficient, can be further optimized for convergence speed through approximate rotation strategies or hardware scheduling of dominant elements. Integration with on-chip DMA controllers and high-speed memory interfaces (e.g., AXI or HBM) could reduce I/O bottlenecks, particularly in real-time applications. On the ASIC side, timing reports were not generated due to toolchain limitations; future work could refine the OpenLane flow to extract and optimize for worst-case slack. Finally, extending the architecture to support full-featured PCA pipelines—including whitening and dimensionality selection—would make Manojavam more self-contained and deployment-ready.

6.3 Learning Outcomes of the Project

- Gained hands-on experience with FPGA architecture modeling using the VTR toolchain.
- Learned complete RTL-to-GDSII design flow using OpenLane, including floorplanning, placement, routing, and signoff.
- Developed and synthesized a domain-specific accelerator for matrix multiplication and SVD computation.
- Analyzed trade-offs between commercial FPGAs and custom architecture through benchmark-driven evaluation.
- Acquired knowledge in cache hierarchy design and dataflow optimization for systolic arrays.

- Built automation scripts and workflows to scale benchmarking across architectures.
- Understood the interplay between architectural decisions and algorithmic behavior in hardware.

BIBLIOGRAPHY

[1] E. Ahmed and J. Rose, "The effect of lut and cluster size on deep-submicron fpga performance and density," in *Proceedings of the 2000 ACM/SIGDA eighth international symposium on Field programmable gate arrays*, 2000, pp. 3–12.

[2] I. Stratix, "Stratix ii device handbook," *Altera Corporation*, vol. 1, 2007.

[3] S. J. Xilinx, *Ca, virtex-5 family overview, 2006*.

[4] A. Boutros, M. Eldafrawy, S. Yazdanshenas, and V. Betz, "Math doesn't have to be hard: Logic block architectures to enhance low-precision multiply-accumulate on fpgas," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2019, pp. 94–103.

[5] Intel Corporation, *Agilex Device Overview*, Technical Product Brief, 2020.

[6] Xilinx, *Versal Adaptive SoC Technical Reference Manual*, Technical Reference Manual, 2021.

[7] A. Boutros and V. Betz, "Fpga architecture: Principles and progression," *IEEE Circuits and Systems Magazine*, vol. 21, no. 2, pp. 4–29, 2021.

[8] J. Rose et al., "The vtr project: Architecture and cad for fpgas from verilog to routing," in *Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays*, 2012, pp. 77–86.

[9] S. Yang, *Logic synthesis and optimization benchmarks user guide: version 3.0*. Citeseer, 1991.

[10] Intel Corporation, *Stratix 10 Device Datasheet*, Technical Product Brief, 2019.

[11] U. A. Korat and A. Alimohammad, "A reconfigurable hardware architecture for principal component analysis," *Circuits, Systems, and Signal Processing*, vol. 38, pp. 2097–2113, 2019.

[12] D. Fernandez, C. Gonzalez, D. Mozos, and S. Lopez, "Fpga implementation of the principal component analysis algorithm for dimensionality reduction of hyperspectral images," *Journal of Real-Time Image Processing*, vol. 16, pp. 1395–1406, 2019.

[13] M. A. Mansoori and M. R. Casu, "High level design of a flexible pca hardware accelerator using a new block-streaming method," *Electronics*, vol. 9, no. 3, p. 449, 2020.

[14] A. Das, D. Nguyen, J. Zambreno, G. Memik, and A. Choudhary, "An fpga-based network intrusion detection architecture," *IEEE Transactions on Information Forensics and Security*, vol. 3, no. 1, pp. 118–132, 2008.

[15] S. N. Shahrouzi and D. G. Perera, "Dynamic partial reconfigurable hardware architecture for principal component analysis on mobile and embedded devices," *EURASIP Journal on Embedded Systems*, vol. 2017, pp. 1–18, 2017.

[16] Y. Ma and D. Wang, "Accelerating svd computation on fpgas for dsp systems," in *2016 IEEE 13th International Conference on Signal Processing (ICSP)*, IEEE, 2016, pp. 487–490.

[17] S. Zhang, X. Tian, C. Xiong, J. Tian, and D. Ming, "Fast implementation for the singular value and eigenvalue decomposition based on fpga," *Chinese Journal of Electronics*, vol. 26, no. 1, pp. 132–136, 2017.

[18] Y.-L. Chen, C.-Z. Zhan, T.-J. Jheng, and A.-Y. Wu, "Reconfigurable adaptive singular value decomposition engine design for high-throughput mimo-ofdm systems," *IEEE transactions on very large scale integration (VLSI) systems*, vol. 21, no. 4, pp. 747–760, 2012.

[19] M. U. Torun, O. Yilmaz, and A. N. Akansu, "Fpga, gpu, and cpu implementations of jacobi algorithm for eigenanalysis," *Journal of Parallel and Distributed Computing*, vol. 96, pp. 172–180, 2016.

[20] M. V. Athi, S. R. Zekavat, and A. A. Struthers, "Real-time signal processing of massive sensor arrays via a parallel fast converging svd algorithm: Latency, throughput, and resource analysis," *IEEE Sensors Journal*, vol. 16, no. 8, pp. 2519–2526, 2016.

[21] X. Wang and J. Zambreno, "An fpga implementation of the hestenes-jacobi algorithm for singular value decomposition," in *2014 IEEE International Parallel & Distributed Processing Symposium Workshops*, IEEE, 2014, pp. 220–227.

[22] V. Betz, J. Rose, and A. Marquardt, *Architecture and CAD for deep-submicron FPGAs*. Springer Science & Business Media, 2012, vol. 497.

[23] V. Betz and J. Rose, "How much logic should go in an fpga logic block," *IEEE Design & Test of Computers*, vol. 15, no. 1, pp. 10–15, 2002.

[24] G. Lemieux, P. Leventis, and D. Lewis, "Generating highly-routable sparse crossbars for plds," in *Proceedings of the 2000 ACM/SIGDA eighth international symposium on Field programmable gate arrays*, 2000, pp. 155–164.

[25] X. Tang, E. Giacomin, A. Alacchi, and P.-E. Gaillardon, "A study on switch block patterns for tileable fpga routing architectures," in *2019 International Conference on Field-Programmable Technology (ICFPT)*, IEEE, 2019, pp. 247–250.

[26] V. Betz and J. Rose, "Fpga routing architecture: Segmentation and buffering to optimize speed and density," in *Proceedings of the 1999 ACM/SIGDA seventh international symposium on Field programmable gate arrays*, 1999, pp. 59–68.

[27] D. Lewis et al., "The stratixπ routing and logic architecture," in *Proceedings of the 2003 ACM/SIGDA eleventh international symposium on Field programmable gate arrays*, 2003, pp. 12–20.

[28] G. Lemieux, E. Lee, M. Tom, and A. Yu, "Directional and single-driver wires in fpga interconnect," in *Proceedings. 2004 IEEE International Conference on Field-Programmable Technology (IEEE Cat. No. 04EX921)*, IEEE, 2004, pp. 41–48.

[29] "Implementing ram functions in flex 10k devices (a-an-052-01)," Altera Corporation, Tech. Rep., 1995.

[30] K. Tatsumura, S. Yazdanshenas, and V. Betz, "High density, low energy, magnetic tunnel junction based block rams for memory-rich fpgas," in *2016 International Conference on Field-Programmable Technology (FPT)*, IEEE, 2016, pp. 4–11.

[31] T. Ngai, J. Rose, and S. J. Wilton, "An sram-programmable field-configurable memory," in *Proceedings of the IEEE 1995 Custom Integrated Circuits Conference*, IEEE, 1995, pp. 499–502.

[32] S. J. Wilton, J. Rose, and Z. G. Vranesic, "Architecture of centralized field-configurable memory," in *Proceedings of the 1995 ACM third international symposium on Field-programmable gate arrays*, 1995, pp. 97–103.

[33] D. Lewis et al., "Architectural enhancements in stratix-iii™ and stratix-iv™," in *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays*, 2009, pp. 33–42.

[34] V.-I. Xilinx, "Platform fpgas: Complete data sheet," *DS031 (v3. 5), Nov*, vol. 5, 2002.

[35] D. Lewis et al., "The stratix™ 10 highly pipelined fpga architecture," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2016, pp. 159–168.

[36] K. E. Murray et al., "Vtr 8: High-performance cad and customizable fpga architecture modelling," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 13, no. 2, pp. 1–55, 2020.

[37] F. F. Khan and A. Ye, "An evaluation on the accuracy of the minimum-width transistor area models in ranking the layout area of fpga architectures," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 11, no. 1, Mar. 2018, ISSN: 1936-7406. DOI: 10.1145/3182394. [Online]. Available: https://doi.org/10.1145/3182394.

[38] A. Nayak, M. Haldar, A. Choudhary, and P. Banerjee, "Accurate area and delay estimators for fpgas," in *Proceedings 2002 Design, Automation and Test in Europe Conference and Exhibition*, IEEE, 2002, pp. 862–869.

[39] J. Luu et al., "Vtr 7.0: Next generation architecture and cad system for fpgas," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 7, no. 2, pp. 1–30, 2014.

[40] C. Wolf, J. Glaser, and J. Kepler, "Yosys-a free verilog synthesis suite," in *Proceedings of the 21st Austrian Workshop on Microelectronics (Austrochip)*, vol. 97, 2013.

[41] Y. Kukimoto, "Blif-mv," *The VIS Group, University California, Berkely*, 1996.

[42] R. Brayton and A. Mishchenko, "Abc: An academic industrial-strength verification tool," in *Computer Aided Verification: 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings 22*, Springer, 2010, pp. 24–40.

[43] V. Betz and J. Rose, "Vpr: A new packing, placement and routing tool for fpga research," in *International Workshop on Field Programmable Logic and Applications*, Springer, 1997, pp. 213–222.

[44] R. Bro and A. K. Smilde, "Principal component analysis," *Analytical methods*, vol. 6, no. 9, pp. 2812–2831, 2014.

[45] M. Greenacre, P. J. Groenen, T. Hastie, A. I. d'Enza, A. Markos, and E. Tuzhilina, "Principal component analysis," *Nature Reviews Methods Primers*, vol. 2, no. 1, p. 100, 2022.

[46] J. Shlens, "A tutorial on principal component analysis," *arXiv preprint arXiv:1404.1100*, 2014.

[47] S. Ramasubramanian, C. Sowmyarani, A. J. Athreya, A. Devarajan, A. U. Shankar, and R. Kumar, "Data dimensionality reduction using principal component analysis: A case study," in *2024 1st International Conference on Communications and Computer Science (InCCCS)*, IEEE, 2024, pp. 1–6.

[48] S. Aeberhard and M. Forina, *Wine*, UCI Machine Learning Repository, DOI: https://doi.org/10.24 1992.

[49] J. Smith and G. Sohi, "The microarchitecture of superscalar processors," *Proceedings of the IEEE*, vol. 83, no. 12, pp. 1609–1624, 1995. DOI: 10.1109/5.476078.

[50] A. Ahmad, L. Du, and W. Zhang, "Fast and practical strassen's matrix multiplication using fpgas," Jun. 2024. DOI: 10.48550/arXiv.2406.02088.

[51] A. Ambainis, Y. Filmus, and F. Le Gall, "Fast matrix multiplication: Limitations of the coppersmith-winograd method," New York, NY, USA: Association for Computing Machinery, 2015, ISBN: 9781450335362. DOI: 10.1145/2746539.2746554. [Online]. Available: https://doi.org/10.1145/2746539.2746554.

[52] B. Asgari, R. Hadidi, and H. Kim, "Meissa: Multiplying matrices efficiently in a scalable systolic architecture," in *2020 IEEE 38th International Conference on Computer Design (ICCD)*, 2020, pp. 130–137. DOI: 10.1109/ICCD50377.2020.00036.

[53] N. P. Jouppi, C. Young, N. Patil, D. Patterson, et al., "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA)*, ACM, 2017, 1–12. DOI: 10.1145/3079856.3080246. [Online]. Available: https://doi.org/10.1145/3079856.3080246.

[54] G. Golub and W. Kahan, "Calculating the singular values and pseudo-inverse of a matrix," *Journal of the Society for Industrial and Applied Mathematics Series B Numerical Analysis*, vol. 2, no. 2, pp. 205–224, 1965. DOI: 10.1137/0702016.

[55] R. Bhattacharjya, A. Sarkar, B. Maity, and N. Dutt, "Music-lite: Efficient music using approximate computing: An ofdm radar case study," *IEEE Embedded Systems Letters*, vol. 16, no. 4, pp. 329–332, 2024. DOI: 10.1109/LES.2024.3440208.

[56] J. Dongarra et al., "The singular value decomposition: Anatomy of optimizing an algorithm for extreme scale," *SIAM Review*, vol. 60, no. 4, pp. 808–865, 2018. DOI: 10.1137/17M1117732.

[57] M. Sajjad, M. Z. Yusoff, N. Yahya, and A. S. Haider, "An efficient vlsi architecture for fastica by using the algebraic jacobi method for evd," *IEEE Access*, vol. 9, pp. 58 287–58 305, 2021. DOI: 10.1109/ACCESS.2021.3072495.

[58] Z. Shi, Q. He, and Y. Liu, "Accelerating parallel jacobi method for matrix eigenvalue computation in doa estimation algorithm," *IEEE Transactions on Vehicular Technology*, vol. 69, no. 6, pp. 6275–6285, 2020.

[59] J. Volder, "The cordic computing technique," ser. IRE-AIEE-ACM '59 (Western), San Francisco, California: Association for Computing Machinery, 1959, 257–261, ISBN: 9781450378659. DOI: 10.1145/1457838.1457886. [Online]. Available: https://doi.org/10.1145/1457838.1457886.

[60] P. Meher, J. Valls, T.-B. Juang, K. Sridharan, and K. Maharatna, "50 years of cordic: Algorithms, architectures, and applications," *Circuits and Systems I: Regular Papers, IEEE Transactions on*, vol. 56, pp. 1893 –1907, Oct. 2009. DOI: 10.1109/TCSI.2009.2025803.

[61] AMD, *CORDIC Technical Documentation*, https://docs.amd.com, Accessed: May 28, 2025.

[62] A. J. Smith, "Cache memories," *ACM Computing Surveys (CSUR)*, vol. 14, no. 3, pp. 473–530, 1982.

[63] N. P. Jouppi, "Cache write policies and performance," *ACM SIGARCH Computer Architecture News*, vol. 21, no. 2, pp. 191–201, 1993.

[64] L. Backes and D. A. Jiménez, "The impact of cache inclusion policies on cache management techniques," in *Proceedings of the International Symposium on Memory Systems*, 2019, pp. 428–438.

[65] J. J. Dongarra, "The linpack benchmark: An explanation," in *International Conference on Supercomputing*, Springer, 1987, pp. 456–474.

[66] A. R. Lebeck and D. A. Wood, "Cache profiling and the spec benchmarks: A case study," *Computer*, vol. 27, no. 10, pp. 15–26, 1994.

[67] R. Balasubramonian, A. B. Kahng, N. Muralimanohar, A. Shafiee, and V. Srinivas, "Cacti 7: New tools for interconnect exploration in innovative off-chip memories," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 14, no. 2, pp. 1–25, 2017.

[68] N. P. Jouppi, A. B. Kahng, N. Muralimanohar, and V. Srinivas, "Cacti-io: Cacti with off-chip power-area-timing models," in *Proceedings of the International Conference on Computer-Aided Design*, 2012, pp. 294–301.

[69] Xilinx Inc., *Using constraints*, v2022.1, Xilinx User Guide UG903, San Jose, CA, USA, Jun. 2022.

[70] T. Ajayi and D. Blaauw, "Openroad: Toward a self-driving, open-source digital layout implementation tool chain," in *Proceedings of Government Microcircuit Applications and Critical Technology Conference*, 2019.

[71] M. Shalan and T. Edwards, "Building openlane: A 130nm openroad-based tapeout-proven flow : Invited paper," in *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, 2020, pp. 1–6.

[72] M. R. Guthaus, J. E. Stine, S. Ataei, B. Chen, B. Wu, and M. Sarwar, "Openram: An open-source memory compiler," in *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, IEEE, 2016, pp. 1–6.

[73] J. Zhuang et al., "Charm 2.0: Composing heterogeneous accelerators for deep learning on versal acap architecture," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 17, no. 3, Sep. 2024, ISSN: 1936-7406. DOI: 10.1145/3686163. [Online]. Available: https://doi.org/10.1145/3686163.

[74] C. S. Mummidi, V. C. Ferreira, S. Srinivasan, and S. Kundu, "Highly efficient self-checking matrix multiplication on tiled amx accelerators," *ACM Transactions on Architecture and Code Optimization*, vol. 21, no. 2, pp. 1–22, 2024.

[75] Xilinx Inc., *7 series fpgas data sheet: Overview (xc7a35t-cpg236)*, DS180, v1.29, Accessed: May 2025, San Jose, CA, USA, 2022. [Online]. Available: https://docs.xilinx.com/r/en-US/ds180_7Series_Overview.

[76] Xilinx Inc., *Vivado design suite user guide: Using constraints (pblock constraints)*, UG903, Version 2022.1, Accessed: May 2025, San Jose, CA, USA, 2022. [Online]. Available: https://docs.xilinx.com/r/en-US/ug903-vivado-using-constraints.

[77] Intel Corporation, *Intel® core™ i7-12700k processor (25m cache, up to 5.00 ghz) product specifications*, Document Number: 210456, Accessed: May 2025, Santa Clara, CA, USA, 2023. [Online]. Available: https://ark.intel.com/content/www/us/en/ark/products/134594/intel-core-i712700k-processor-25m-cache-up-to-5-00-ghz.html.

[78] NVIDIA Corporation, *Nvidia a100 tensor core gpu architecture*, Version 1.0, Accessed: May 2025, Santa Clara, CA, USA, 2020. [Online]. Available: https://resources.nvidia.com/en-us-tensor-core/nvidia-ampere-architecture-whitepaper.