POLITECNICO DI TORINO

Master's Degree in Electronic Engineering

Master's Degree Thesis

FPGA-Based Signal Processing Architectures for 5G Wireless Using High-Level Synthesis

Supervisors

Prof. Luciano LAVAGNO

Prof. Mihai Teodor LAZARESCU

Prof. Roberto QUASSO

Candidate

Yudi QIN

October 2023

Abstract

The rapid advancement of technology, coupled with the widespread adoption of the Internet of Things (IoT), has driven the quest for novel solutions that significantly improve speed and resource efficiency, surpassing traditional implementations. In hardware design, High-Level Synthesis (HLS) techniques present an efficient avenue for generating hardware designs using high-level programming languages, notably C/C++. HLS tools scrutinize design specifications and autonomously create hardware implementations aligned with performance requirements. The main objective of our research was to accelerate the 3GPP 5G Channel Model. We specifically aimed at enhancing the channel model using High-Level Synthesis (HLS) tools designed for Xilinx and Intel FPGA platforms. Given the complex nature of the project, each team member focused on discrete components of the channel model. My primary contribution centered on the Oversample_Filter section. This component processes a large-sized input matrix using the Upsample function and Matlab's Finite Impulse Response (FIR) function. I first designed the Oversamp_Filter in C++ to mirror the Matlab algorithm. Subsequently, I employed Vitis HLS to optimize the C++ code, ensuring optimal throughput to align with other parts of the channel model. In the end, the generated Register Transfer Level (RTL) design was subjected to simulation tests and on-board validation before its incorporation into the channel chain.

Table of Contents

List of Tables IV

List of Figures V

Acronyms VIII

1 Introduction 1
- 1.1 5G Protocol Stack . . . 1
  - 1.1.1 5G Physical Layer . . . 3
  - 1.1.2 Channel Simulation . . . 4
- 1.2 Board Description . . . 8
  - 1.2.1 FPGA . . . 9
- 1.3 Vitis Unified Software Platform . . . 12
  - 1.3.1 Vitis HLS . . . 15
- 1.4 Thesis Structure . . . 18

2 Oversample_Filter Block 19
- 2.1 Matlab Reference . . . 19
  - 2.1.1 Upsample . . . 19
  - 2.1.2 Filter . . . 20
- 2.2 C++ Code and C-Simulation . . . 23
  - 2.2.1 bin_read.h . . . 24
  - 2.2.2 BVector_Filter_OneSample_Sub.h . . . 25
  - 2.2.3 BVector_Filter_OneSample_Sub.cpp . . . 25

3 Synthesis 37
- 3.1 Pipelining . . . 38
  - 3.1.1 Principle . . . 38
  - 3.1.2 Syntax . . . 41
  - 3.1.3 Application . . . 41
- 3.2 Unroll . . . 42

3.2.1 Principle . . . 42
3.2.2 Syntax . . . 44
3.2.3 Application . . . 45
3.3 Array_Partition . . . 45
3.3.1 Principle . . . 45
3.3.2 Syntax . . . 47
3.3.3 Application . . . 48
3.4 Inline . . . 49
3.4.1 Principle . . . 49
3.4.2 Syntax . . . 49
3.4.3 Application . . . 49
3.5 Loop_Tripcount . . . 51
3.5.1 Principle . . . 51
3.5.2 Syntax . . . 51
3.5.3 Application . . . 52
3.6 Dataflow . . . 52
3.6.1 Principle . . . 52
3.6.2 Syntax . . . 56
3.6.3 Application . . . 56
3.7 Interface . . . 57
3.7.1 Syntax . . . 58
3.7.2 Application . . . 59

4 Result Analysis 61
4.1 Synthesis Result . . . 61
4.2 Co-Simulation . . . 63

Bibliography 64

List of Tables

1.1 Alveo U280 Data Center 9

4.1 The throughput of Input and Output ports 63
4.2 The usage of resource 63

List of Figures

1.1 NR radio interface protocol . . . 3
1.2 Link Level Simulator for 5G NR . . . 6
1.3 The Channel Model Framework . . . 7
1.4 Vitis Development Flow . . . 13
1.5 Vitis HLS Development Flow . . . 16

2.1 bin_read.h code . . . 24
2.2 Top function flow chat . . . 26
2.3 Fir function flow chat . . . 27
2.4 Composition of the function . . . 28
2.5 Content of Top function . . . 29
2.6 Upsample C++ code . . . 31
2.7 Filter C++ code . . . 33
2.8 Filter C++ code . . . 34

3.1 Function pipelining behavior . . . 38
3.2 Loop pipelining behavior . . . 39
3.3 Function and Loop Pipelining Behavior . . . 40
3.4 Loop Pipelining with Rewind Option . . . 40
3.5 Using Pipeline in Loop . . . 41
3.6 Using Pipeline in Function . . . 42
3.7 Loop Unrolling Details . . . 43
3.8 Using Unroll in Function . . . 45
3.9 Array Partitioning . . . 46
3.10 Partitioning Array Dimensions . . . 47
3.11 Using Unroll in Function . . . 48
3.12 Using Inline in Function . . . 50
3.13 Using Loop_Tripcount in Function . . . 52
3.14 Sequential Functional Description . . . 53
3.15 Parallel Process Architecture . . . 53
3.16 Dataflow Optimization . . . 54

3.17 Using Loop_Tripcount in Function 56
3.18 C-argument Type 58
3.19 Using interface in Function 59

4.1 Synthesis Report 62

Acronyms

AI
artificial intelligence

3GPP
The 3rd Generation Partnership Project

HLS
High-level synthesis

FPGA
Field Programmable Gate Array

RTL
Register Transfer Level

GPU
Graphics processing unit

HDL
Hardware description language

LUT
Look Up table

DSP
Digital Signal Processor

FF
Flip-Flop

Chapter 1

Introduction

1.1 5G Protocol Stack

In today's world, there is a pressing need for a new network infrastructure that can provide reliable services to a multitude of devices. This infrastructure must offer stable connections, increased bandwidth, and minimal latency. These requirements serve as the foundation for the fifth generation of wireless networks, commonly known as 5G. By utilizing higher frequencies, 5G enables communication at faster data rates, although over shorter distances. To overcome this limitation, multiple antennas are implemented to enhance capacity and signal quality. Additionally, 5G allows operators to partition a physical network into multiple virtual networks based on usage requirements[1]. The Third Generation Partnership Project (3GPP) has defined a new radio interface for 5G known as New Radio (NR). While NR builds upon some features and structures of LTE, it is not obligated to maintain backward compatibility, thereby leveraging much higher frequencies. This expansion in frequency spectrum allows for wider bandwidth and higher data rates. However, communication at higher frequencies is susceptible to increased radio-channel attenuation, which poses a challenge in terms of network coverage. This challenge is addressed by employing multiple antennas for communication, further promoting the beam-centric design of NR[2].

The initial specifications for 5G NR were published in December 2017, supporting the Non-Standalone (NSA) mode, in which 5G-compliant user equipment relies on existing LTE networks for initial access and mobility. Subsequently, in June 2018, the Standalone (SA) versions of 5G NR specifications were finalized, enabling independent operation without reliance on LTE.

The purpose of this chapter is to provide some big picture to been able to understand the radio protocol stack. Most of the fundamental idea in this page comes from 3GPP 38.300[3]

There are two main components in 5G NR network:

UE (mobile subscriber) and **gNB** (base station).

gNBs establish a connection with the 5G Core network on the backend. The communication from gNB to the UE is referred to as the downlink, utilizing PBCH, PDSCH, and PDCCH channels to transmit various data and control information. Conversely, the link from the UE to the gNB is termed the uplink, employing PRACH, PUSCH, and PUCCH channels.

In 5G NR there are various physical channels in the downlink (from gNB to UE) and uplink (from UE to gNB):

Downlink channels: PDSCH, PDCCH, PBCH. **Uplink channels:** PRACH, PUSCH, PUCCH.

In both downlink and uplink scenarios, specific physical signals play critical roles. Front-loaded DMRS (Demodulation Reference Signal) serves as a common element for both PDSCH and PUSCH channels. The use of Orthogonal Frequency Division Multiplexing (OFDM) with a cyclic prefix (CP) is a standard practice in both downlink and uplink communications. Furthermore, the uplink employs DFT Spread OFDM with CP to extend coverage, with CP length tailored to the specific requirements. The 5G NR system operates within two frequency ranges: FR1 (Sub 6GHz) and FR2 (millimeter-wave range, 24.25 to 52.6 GHz). NR adopts flexible subcarrier spacing, derived from the fundamental 15 KHz subcarrier spacing used in LTE.

The NR Radio Protocol Stack Architecture shares many similarities with the LTE Radio Protocol Stack Architecture. As in LTE/WCDMA systems, the NR radio protocol stack is divided into two distinct segments based on the type of data they handle. Signaling messages are processed through the C-plane stack, while user data flows through the U-plane stack. Although the U-plane and C-plane share a common underlying structure, illustrated in Figure 1.1, the components above PHY/MAC/RLC/PDCP differ between the two segments

The 5G Protocol Stack can be divided in layer 1, layer 2 and layer 3:

- The 5G layer-1 is PHYSICAL layer.
- The 5G layer-2 include MAC, RLC and PDCP.
- The 5G layer-3 is RRC layer.

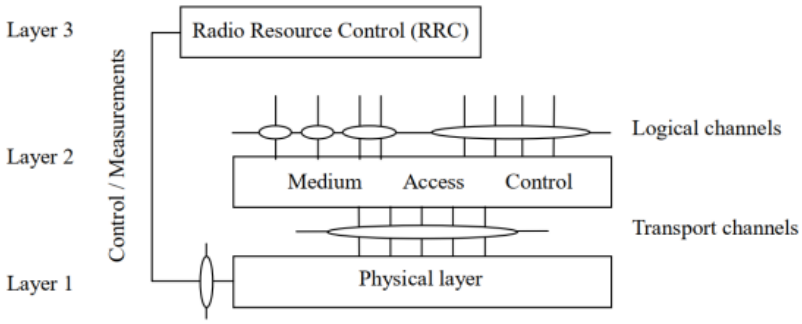


Figure 1.1: NR radio interface protocol

Figure 1.1 illustrates the architecture of the NR radio interface protocol, primarily focusing on the physical layer (Layer 1). The physical layer interfaces with the Medium Access Control (MAC) sub-layer of Layer 2 and the Radio Resource Control (RRC) Layer of Layer 3. The circles connecting different layers and sub-layers represent Service Access Points (SAPs).

The physical layer provides a transport channel to the MAC, defining how information is transmitted across the radio interface. MAC, in turn, offers various logical channels to the Radio Link Control (RLC) sub-layer of Layer 2, each designed for specific types of information transfer.

In the case of the User Plane (U-Plane), an additional layer called SDAP resides at the top of the radio stack, connecting to the User Plane Function (UPF). In contrast, for the Control Plane (C-Plane), two layers, RRC and NAS, sit atop the stack. The NAS layer establishes a connection with the Access and Mobility Management Function (AMF).

1.1.1 5G Physical Layer

The physical layer provides data transport services to higher layers, with access to these services achieved through the utilization of transport channels managed by the MAC sub-layer. A transport block refers to the data exchanged between the MAC layer and the physical layer. NR Layer 1 is designed in a bandwidth-agnostic manner, based on resource blocks, enabling it to dynamically adapt to various spectrum allocations.

Following are the functions of 5G layer 1 i.e. PHYSICAL (PHY) Layer[4].

- Error detection on the transport channel and indication to higher layers.
- FEC encoding/decoding of the transport channel.

- Hybrid ARQ soft-combining.
- Rate matching of the coded transport channel to physical channels.
- Mapping of the coded transport channel onto physical channels.
- Power weighting of physical channels.
- Modulation and demodulation of physical channels.
- Frequency and time synchronisation.
- Radio characteristics measurements and indication to higher layers.
- Multiple Input Multiple Output (MIMO) antenna processing.
- Transmit Diversity (TX diversity).
- Digital and Analog Beamforming.
- RF processing.

1.1.2 Channel Simulation

Channel simulation plays a crucial role in the functional and performance verification of models during the network planning phase. In the context of channel modeling, we refer to the medium that exists between typical transmission and receiving stations, comprising two groups of antennas—one for transmitting and the other for receiving—interconnected by a propagation channel. This propagation channel represents the environment through which radio waves travel from the transmitting antenna to the receiving antenna.

The significance of channel simulation lies in its ability to assess the performance of communication systems. The objective of a model is to replicate physical reality with an appropriate level of detail, striking a balance between accuracy and complexity. Channel simulators serve as tools for modeling routing protocol performance, analyzing traffic flows, and evaluating communication system efficiency using real-world parameters within a virtual environment. Understanding the impact of a physical channel in a real-world setting is of paramount importance.

Developing a channel model begins with defining the requirements of the target system to be described. This is especially critical in wireless systems, given the substantial variations in the propagation medium across space, time, and frequency.

The outcomes of simulations are employed as input by planning tools, commonly used by network operators, to determine network infrastructure (network nodes) and configuration. Channel simulation enables the creation of a theoretical model that assesses the performance of actual devices.

Many network operators develop these models by conducting simulations and subsequently testing commercial devices in both laboratory and field settings to verify that the expected performance levels are achieved. 3GPP and other standardization bodies leverage channel simulators to craft theoretical models of communication systems. However, it's worth noting that accurate 5G channel model simulations demand significant computational resources and extended execution times when performed on general-purpose processors.

One approach to expedite the execution and reduce simulation time is through hardware acceleration. Achieving homogeneity in the environment facilitates the utilization of High-Level Synthesis optimizations, enabling the derivation of technological requirements and the selection of optimal solutions, architectures, and systems.

To accelerate channel functions using an FPGA that supports complex simulations involving a higher number of antenna elements and various UE speeds, the design is divided into two major components: host and kernel code.

In our case, the link between trasmitter and receiver is modelled as follows:

```
% OFDM signal generation
[ tb10 ] = ofdm_modulation_ul(env,tb9);
% tb10 = OFDM signal in time domain including CP
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Transmitter Front-End (Oversampling and filtering)
[ tb11 ] = oversample_filter_ul(env,tb10);
% tb11 = OFDM signal oversampled and filtered
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Phase Noise (PN) insertion
[ tb12, Jmf ] = nr_phase_noise_insertion_ul(env,tb11);
% tb12 = OFDM signal affected by phase noise
% Jmf = phase noise spectrum
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Rayleigh fading channel
[ tb13, sinr_pre, Hid ] = fading_channel(env,ch,tb12);
% tb13 = OFDM signal at radio channel output
% sinr_pre = pre-detection SINR per antenna in dB
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Receiver Front-End (Filtering and downsampling)
[ tb14 ] = downsample_filter(env,tb13);
% tb14 = received sig  nal at RX Front-end output
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Analog Beamforming (Grid of Beams)
[ tb15, beam_id ] = nr_analog_beamforming_ul(env,tb14);
beam_stat(:,SINR_ITER,counter+1) = beam_id;
% tb15 = OFDM signal after Analog Beamforming
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% OFDM signal demodulation
[ tb16 ] = ofdm_demodulation_ul(env,tb15);
% tb16 = received signal after FFT and CP cut
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Figure 1.2: Link Level Simulator for 5G NR

The host code is responsible for managing control and data-related tasks, including memory allocation on the device, receiving data from a client via a socket, launching the kernel, and transmitting results back to the client through another socket. In contrast, the kernel code is designed to be highly data-parallel and computationally intensive, meticulously optimized for execution on the target FPGA.

The entire channel model likes as follows: There are five blocks of this channel

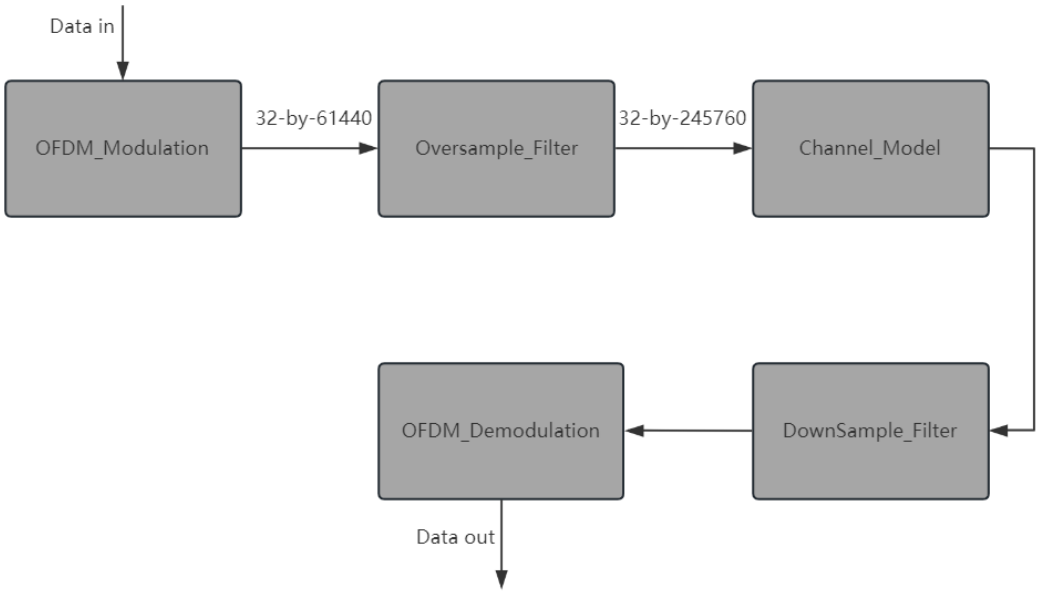


Figure 1.3: The Channel Model Framework

model, this thesis focus on the Oversample_Filter block only. For this block, the input data is from the previous block and the output result will sent to the lower block. Therefor, the processing rate of this block should correspond to the up and down block.The input matrix size of the Oversample_Filter block is 32-by-61440, the output size is 32-by61440*32,all datas are complex<double> type.

1.2 Board Description

The Alveo U280 board serves as the designated platform for the project detailed within this study. Engineered to address the dynamic requirements of contemporary data centers, the AMD Alveo U280 Data Center accelerator cards offer a versatile solution[5].

Constructed utilizing the advanced AMD 16nm UltraScale+ architecture, the Alveo U280 presents an impressive capacity of 8GB of HBM2 memory, delivering a remarkable bandwidth of 460 GB/s. This configuration facilitates exceptional and flexible acceleration tailored for applications characterized by memory-bound and compute-intensive operations. Such applications encompass a wide spectrum including database management, analytical processes, and machine learning inference.

This board comprises a Virtex UltraScale+ FPGA accompanied by dual HBM2 stacks, amounting to a total of eight HBM2 dies. Facilitating connectivity between the FPGA and HBM2 dies are 32 distinct parallel channels. Each of these channels boasts a capacity of 256MB, aggregating to a cumulative total of 8GB. Furthermore, the U280 acceleration card is equipped with PCI Express 4.0 support, harnessing the contemporary server interconnect framework to optimize communication with high-bandwidth host processors.

Contemporary memory interfaces offer access via numerous banks that feature dedicated channels, such as high bandwidth memory (HBM) lanes or double data rate (DDR) channels. As a result, the array's access bandwidth can be enhanced by distributing it across the diverse memory interfaces, or banks, accessible on the board.

In order to increase the onchip memory bandwidth to match the requirements of the data computations, the same considerations also apply to onchip BRAM banks.

Within the Xilinx device, the architecture comprises of two distinct FPGA partitions: Shell and User. The Shell partition, a stationary sector, establishes fundamental infrastructure for the platform, encompassing elements such as PCIe connectivity, board management, sensors, clocking, and reset mechanisms. Conversely, the User partition, a dynamic sector, encompasses the user-compiled binary labeled as .xclbin. During execution, this binary is loaded by the XRT (Xilinx Runtime) to facilitate the operational process.

RTL kernels denote personalized logic modules meticulously crafted by developers and subsequently programmed into the dynamic partition. In the context of this manuscript, the term "kernels" pertains to the specific functions that designers construct and incorporate within the dynamic domain of the Alveo accelerator card.

Card Specifications	U280
DRAM Memory	
HBM2 Total Capacity	8GB
HBM2 Total Bandwidth	460GB/s
DDR Format	2x16GB 72b DIMM DDR4
DDR Memory Capacity	32GB
DDR Total Bandwidth	38GB/s
SRAM Memory	
Internal SRAM Capacity	41MB
Internal SRAM Total Bandwidth	30TB/s
Interfaces	
PCI Expresss	Gen4x8 with CCIX
Network Interfaces	2xQSFP28 (100GbE)
Logic Resources	
Look-up Tables (LUTs)	1,079,000
Power	
Maximum Total Power	225W

Table 1.1: Alveo U280 Data Center

1.2.1 FPGA

At its essence, Xilinx has engineered the Alveo series of PCIe Data Center accelerator cards, harnessing the foundational power of FPGAs[6].

An FPGA (field-programmable gate array) is an integrated circuit (IC) that incorporates configurable logic blocks (CLBs) along with additional elements. This IC can be programmed by users through a bitstream created by synthesis tools, effectively translating into a tangible circuit the digital function of their choice. The designation "field-programmable" underscores the FPGA's capacity for customization, in contrast to conventional ICs whose functionality is permanently etched by the manufacturer.

Through the generation of multiple instances of these operations, FPGAs demonstrate a distinctive aptitude for the concurrent execution of functions. This inherent capability positions them exceptionally well to function as hardware accelerators within contexts characterized by substantial degrees of parallelism. Assemblages of configurable hardware units, arrayed and interconnected as necessary, facilitate the construction of highly efficient, application-specific architectures tailored to diverse domains.

FPGAs are available in a range of dimensions, each featuring distinct quantities of programmable logic components. Devices of greater scale encompass increased resources, affording designers the capacity to embed augmented parallel circuits, thereby yielding heightened levels of acceleration. This array of device options empowers designers with a spectrum of choices, enabling them to navigate diverse cost-to-performance considerations[7].

The procedure bears resemblance to software programming in that you compose code transformed into a binary file and subsequently loaded onto the FPGA. However, the consequential effect is distinct, as the hardware description language (HDL) induces tangible alterations in the hardware itself, as opposed to solely optimizing the device for software execution. Moreover, this process allows for the fine-tuning of fundamental functions such as memory allocation or power consumption, contingent upon the specific task at hand.

FPGAs provide programmers and designers with a remarkable capacity to customize and modify the computing architecture with exceptional flexibility. This adaptability culminates in the development of domain-specific architectures that closely align with the precise needs of their applications. While FPGAs are not a novel technology, their significance has grown exponentially, primarily fueled by the rapid pace of innovation in fields such as artificial intelligence. Since the elements within FPGAs and the routes connecting them can be configured post-power-up, FPGAs can be repetitively reprogrammed to execute a diverse array of required functions.

FPGA programming leverages a Hardware Description Language (HDL) to manipulate circuits according to the desired device capabilities. This process fundamentally differs from programming GPUs or CPUs, as it doesn't involve crafting a sequential program. Instead, HDL is employed to design circuits and enact physical changes to the hardware in alignment with the intended functionality. CPUs are distinguished by their high flexibility, but their underlying hardware remains fixed post-manufacturing. They rely on software instructions to specify particular operations, such as arithmetic functions, to perform on designated data in memory. The hardware within a CPU must possess the capability to execute a vast spectrum of potential operations, with software instructions dictating the execution path, typically handling one instruction at a time.

In contrast, FPGAs excel in processing extensive datasets concurrently. The advantage of adaptable hardware over CPUs varies depending on the specific application, predominantly hinging on the nature of the computation and its potential for parallelization. Nonetheless, it's not uncommon to witness a performance enhancement of up to 20 times when comparing FPGA implementations with CPU counterparts for tasks that lend themselves well to parallelization.

The architectural design of FPGAs renders them a highly effective solution for hardware acceleration. In contrast to devices such as ASICs and GPUs, which

employ conventional methods that involve transitioning between programming and memory, FPGAs excel in use cases demanding real-time data processing. The substantial power requirements for storage and retrieval tasks in ASICs and GPUs often lead to performance bottlenecks.

FPGAs, thanks to their reconfigurability tailored to specific functions, exhibit greater flexibility compared to Application Specific Integrated Circuits (ASICs). However, this flexibility comes at the expense of increased power consumption and physical footprint. While they surpass general-purpose processors in terms of efficiency, programming FPGAs is typically more intricate, and their flexibility is somewhat diminished. In summary, FPGAs emerge as a compelling choice for hardware acceleration, offering a unique balance of configurability and efficiency, especially in scenarios where real-time data processing is pivotal.

In contrast to ASICs and GPUs, FPGAs operate without the need for frequent transitions between memory and programming. This inherent feature significantly enhances the efficiency of data storage and retrieval processes. Moreover, the FPGA architecture offers remarkable flexibility, permitting users to tailor power consumption according to specific task requirements. Unlike GPUs, which consist of fixed processing cores responsible for fetching and executing instructions, FPGAs possess a versatile architecture that directly maps code to physical logic circuitry. However, similar to GPUs, it remains imperative for users to acquire a foundational understanding of these principles to effectively architect their code for optimal performance outcomes.

This flexibility proves invaluable in offloading energy-intensive tasks to one or multiple FPGAs, relieving the burden on conventional CPUs or other devices. Furthermore, due to the reprogrammable nature of many FPGAs, implementing upgrades and fine-tuning a hardware acceleration system becomes a straightforward process. While FPGAs traditionally resided within the purview of hardware engineers, today, AI researchers and software programmers have access to innovative platforms that bridge the gap, making FPGA programming feel akin to conventional software development. With the appropriate tools, you can discover FPGA programming solutions that align with your existing knowledge of both software and hardware, offering a smoother and more accessible path to harnessing the power of FPGAs.

In the context of this study, the FPGAs under examination, although designed for data center applications, lack specialized support for double-precision floating-point addition and multiplication. As previously discussed and in line with the chosen IP components, it was deemed more advantageous to utilize floating-point data rather than double precision. To facilitate the creation of our design, Xilinx offers a comprehensive suite of tools tailored to assist software developers at every stage of the FPGA programming journey.

1.3 Vitis Unified Software Platform

The Vitis unified software platform serves as a development ecosystem tailored for heterogeneous applications, providing support for Xilinx devices, including the Alveo Data Center Accelerator cards. This integrated platform seamlessly merges all facets of Xilinx's hardware and software development into a unified environment, enabling the utilization of standard C/C++ for both software and hardware components.

The Vitis suite of tools offers an array of functionalities encompassing compilation, linking, profiling, and debugging, catering to heterogeneous systems within various design paradigms. These design flows encompass Data Center application acceleration, RTL kernel design, Embedded System development, as well as conventional embedded hardware and software design.

Within the Vitis environment, heterogeneous systems encompass a broad spectrum of components, including software applications executed on x86 host processors or Arm embedded processors. These systems also involve compute kernels functioning within programmable-logic (PL) regions or Versal AI Engine arrays. Furthermore, extensible platform designs are pivotal in establishing the structural groundwork for the creation and operation of these heterogeneous systems.

The software development toolset comprises essential components, including compilers and cross-compilers for constructing your software application. Debugging tools are instrumental in identifying and rectifying issues within your system design. Program analyzers facilitate the profiling and in-depth analysis of your application's performance. To establish connectivity between your software program and the target platform, Xilinx Runtime (XRT) offers an API and drivers. XRT manages transactions and data transfers between the software application and the hardware design, streamlining their interaction.
The development flow works in parallel on two ways:

- **Application Compilation** using G++ to generate the host.exe file.
- **Kernel Compilation** to obtain the .xclbin file needed.

The host program, developed in C/C++ and employing the XRT native API, undergoes compilation using the g++ compiler, resulting in the creation of a host executable file designated for execution on the x86 processor. This host program is designed to interface with kernels situated within the programmable-logic (PL) region of the FPGA device.

Vitis HLS stands as a compiler specifically engineered to transform C/C++ source code into a synthesized RTL (Register-Transfer Level) design meticulously tailored for optimal performance on Xilinx FPGA products. For every C++ kernel, it is imperative to undergo synthesis through Vitis HLS to generate a Xilinx object (.xo) file.

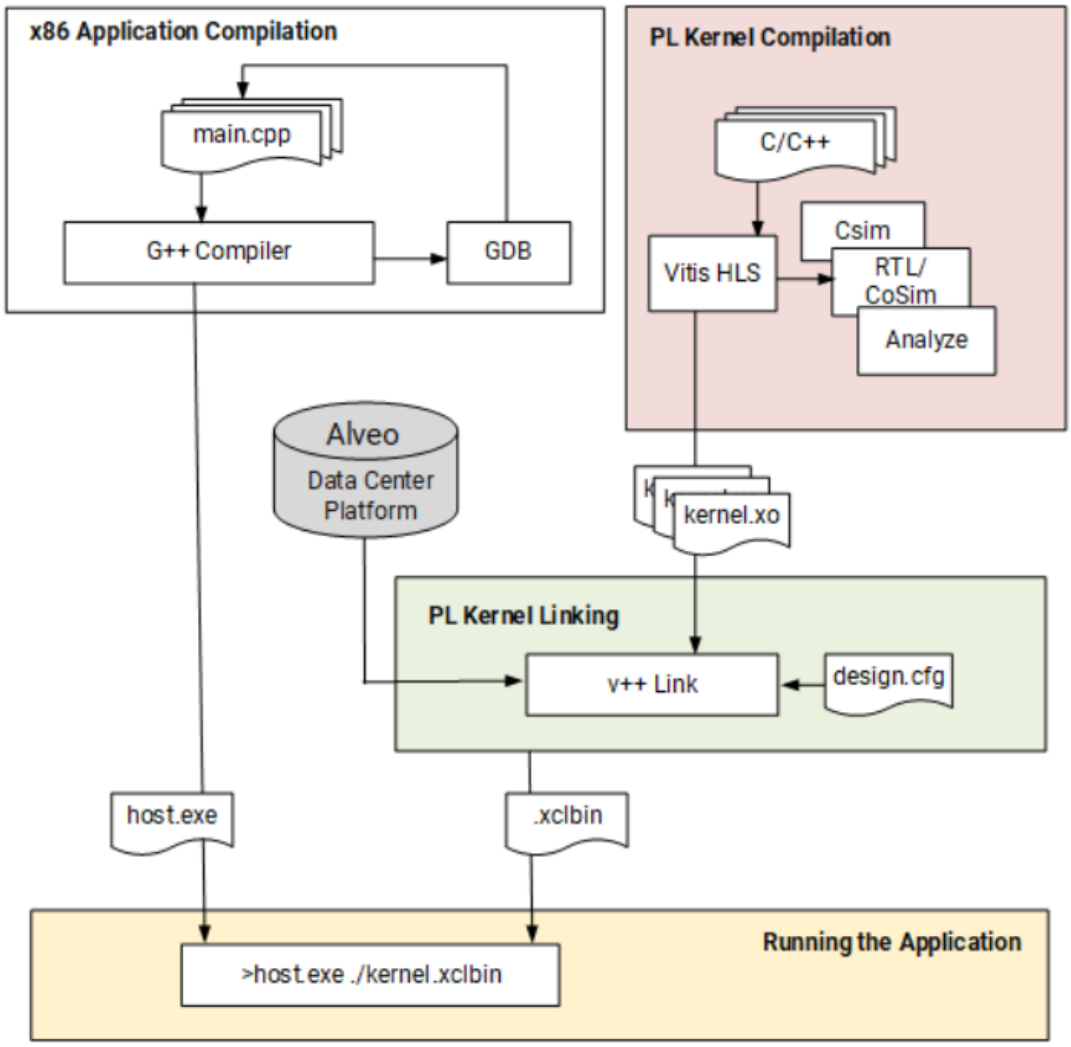


Figure 1.4: Vitis Development Flow

The Vitis accelerated libraries offer meticulously optimized hardware functions that enhance performance with minimal code modifications. This eliminates the necessity of re-implementing your algorithms while enabling you to leverage the advantages of Xilinx's adaptive computing capabilities. These accelerated libraries encompass a wide spectrum of common functions in mathematics, statistics, linear algebra, and digital signal processing (DSP). They are also available for domain-specific applications such as vision and image processing, quantitative finance, database management, data analytics, and data compression.

To delineate the device binary's architecture, a configuration file can be crafted,

specifying various options. These options may encompass details such as the number of instances of a kernel (or Compute Unit) to be constructed within the device binary, the manner in which kernels are interconnected with the global memory or with other kernels, among other factors. Subsequently, this configuration file is provided to the Vitis linker, which utilizes it to generate the .xclbin file.

To generate the .xclbin file, the Vitis linker facilitates the pairing of one or more .xo files, effectively merging them with the specified target hardware platform. This process culminates in the creation of a device binary file (.xclbin) that is subsequently loaded onto the Alveo accelerator card for execution.

A PL kernel (.xo) represents a customizable hardware function that can be integrated into the Programmable Logic (PL) region of an extensible platform, allowing for the definition of specialized hardware components. PL kernels can be constructed using C++ code within Vitis HLS or through RTL code along with the IP packaging feature available in Vivado. The Device Binary (.xclbin) file encapsulates the programmable device image (PDI) for Versal Adaptive Compute Acceleration Platform (ACAP) or the bitstream for Zynq Multi-Processor System-on-Chip (MPSoC), along with essential metadata required for the management and control of the hardware design.

Vitis Accelerated Libraries offer hardware functions meticulously optimized for performance, requiring minimal modifications to existing code, and eliminating the necessity for algorithmic reimplementation in order to leverage the advantages of Xilinx's adaptive computing capabilities. These libraries encompass a broad spectrum of commonly utilized functions, including mathematics, statistical analysis, linear algebra, and digital signal processing (DSP). Moreover, they extend to domain-specific applications such as computer vision, image processing, quantitative finance, database management, data analytics, and data compression.

In order to delineate the device binary's architectural characteristics, a configuration file can be crafted to define various parameters. This includes specifying the number of instances of a kernel (or Compute Unit) to be incorporated within the device binary, configuring interconnections between kernels and the global memory, among other essential aspects. Subsequently, this configuration file is provided to the Vitis linker to facilitate the generation of the .xclbin file.

The Vitis Compiler offers three distinct build targets, each defining the nature and contents of the resultant .xclbin file. Among these targets, two are designated for validation and debugging purposes: software emulation, which facilitates C-based simulation, and hardware emulation, which supports RTL co-simulation. The third target pertains to hardware, intended for generating the final project output for execution on the Alveo card. Importantly, a single host program can seamlessly execute any of the .xclbin targets. During execution, the host program is responsible for loading the .xclbin file, a binary artifact generated by the Vitis Compiler. It's noteworthy that the host application invariably operates on the

CPU and can function either in emulation mode on x86 architecture or on the actual physical accelerator platform.

An RTL (Register-Transfer Level) design tool encompasses the entire hardware design process, from initial design creation to synthesis and implementation. This tool empowers hardware designers to craft hardware designs and export them in the form of a Xilinx Support Archive (.xsa). This archive serves as a versatile hardware container with applications across multiple domains. The Xilinx Support Archive (.xsa) can be employed within the context of fixed or extensible platforms. A 'Fixed Platform' comprises a fully realized hardware design encapsulated within an .xsa file, accompanied by essential software files delineating the operating system, libraries, and boot files. Here, 'fixed' implies that the hardware design is comprehensive and self-contained. Conversely, an 'Extensible Platform' serves as the target environment within the Vitis heterogeneous system design workflow. In this context, 'extensible' signifies the platform's adaptability, enabling further customization through the incorporation of programmable components such as PL kernels and AI Engine graph applications. This customization process results in the creation of an embedded system tailored to specific requirements. Furthermore, the Extensible Platform can also be harnessed for software development, mirroring the capabilities of a Fixed Platform.

1.3.1 Vitis HLS

Xilinx's High-Level Synthesis (HLS) tools provide a vital interface bridging the realms of software and hardware design, aligning with the goal of accelerating applications[8]. In the context of the Vitis application acceleration workflow, Vitis HLS plays a pivotal role in automating substantial portions of the necessary code modifications. This automation facilitates the transformation of C/C++ code into programmable logic, striving to achieve low latency and high throughput.

While FPGA programming traditionally involved Hardware Description Languages (HDLs) like Verilog or VHDL, there is a growing trend towards High-Level Synthesis (HLS) tools. These tools enable the conversion of algorithmic descriptions written in higher-level languages such as C/C++ into lower-level hardware description languages like Verilog or VHDL (RTL), suitable for implementation in the programmable logic (PL) region of Xilinx FPGA devices.

Furthermore, FPGA architectures empower fine-grained control over parallelism in implementation, not only between tasks within a kernel but also within iterations of inner loops. This feature enhances the efficacy and flexibility of hardware design by accepting software languages, easing the logic design and description of intricate computations. The primary advantage of this approach lies in leveraging languages like C/C++ to craft efficient code that can subsequently be translated into hardware. However, achieving desired performance levels often necessitates

additional efforts, such as code refinements to align with the HLS tool's performance objectives.

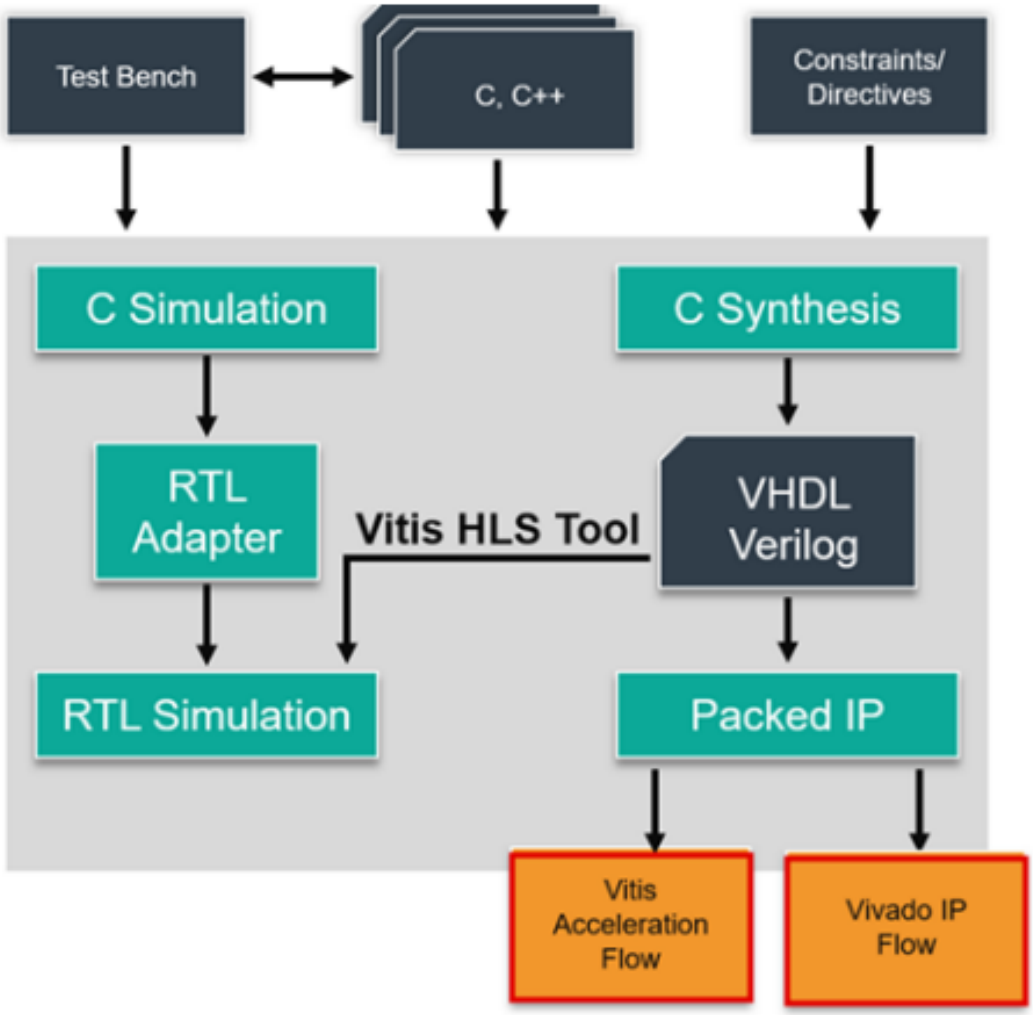


Figure 1.5: Vitis HLS Development Flow

The Vitis HLS kernel development process unfolds in several key steps:

1. Authoring the C/C++ code for the desired function.
2. Verification of the C/C++ code through C-simulation.
3. Kernel construction through C-synthesis.
4. Co-simulation for validation of the generated kernel using C++ outputs.

5. Reviewing HLS synthesis and co-simulation reports to assess performance.

6. Iterative refinement of the previous steps until performance goals are met.

Vitis HLS generates Vivado IP or Vitis Kernel based on specified target flows, design constraints, and optimization pragmas or directives. These optimization directives allow modification and control over internal logic and I/O ports, overriding default tool behaviors. The tool offers a range of pragmas to optimize the RTL design, exploring design spaces to meet specific space and throughput requirements. Properly inferring these pragmas to define function interfaces and pipeline loops and functions within the code is a fundamental aspect of Vitis HLS.

Vitis HLS seamlessly integrates with both the Vivado Design Suite for synthesis, placement, and routing, and the Vitis core development kit for heterogeneous system-level design and application acceleration. In the Vivado IP flow, Vitis HLS supports code customization to implement broader interface standards, aligning with specific design objectives. The generated RTL can be directly utilized as IP within the Vivado tool or Model Composer. Conversely, the Vitis Kernel flow, a bottom-up design approach within the Vitis Application Acceleration Development flow, enforces a more structured set of interfaces.

This structured approach ensures correct-by-construction integration of HLS blocks with Vitis extensible platforms, facilitating seamless integration with the Xilinx Run Time (XRT) software stack. Consequently, the hardware/software integration process is greatly simplified.

1.4 Thesis Structure

After seeing these key points, it's possible to understand the main objective archieve during this thesis:

Chapter 2 Describe the algorithm inside the Oversample_Filter block detailed.

Chapter 3 Describe the principles of optimization statements and their application in C++ code.

Chapter 4 Describe the synthesis result san co-simulation resulte of this design.

Chapter 2

Oversample_Filter Block

2.1 Matlab Reference

As referenced at previous chapter, there are two functions from Matlab:

- **Upsample**
- **Filter**

These two equations are used to process the input matrix, allowing the output result to serve as the input for the next module, enabling seamless information transfer and processing.

2.1.1 Upsample

In MATLAB, the upsample function is used to upsample a signal sequence. Upsampling is a signal processing operation that increases the sampling rate of a signal by inserting zero values or using other interpolation methods, thereby enhancing the temporal resolution of the signal[9]. Its specific functions include:

1. **Increasing the Sampling Rate:** The upsample function increases the signal's sampling rate by inserting new samples between the original sample points, effectively doubling or multiplying the sampling rate. This can be useful for signal reconstruction or resampling to achieve higher precision or capture finer details for subsequent processing.

2. **Preserving the Spectrum:** During the upsampling process, the upsample function widens the frequency spectrum of the original signal. This helps to avoid aliasing distortions caused by low sampling rates and preserves the spectral information of the signal.

3. **Interpolation:** The upsample function typically employs linear interpolation to calculate the values of the inserted samples, ensuring that the increase in sampling rate is achieved without introducing distortion while maintaining the signal's smoothness. Various interpolation methods can be chosen to suit specific application requirements.

The function expresses as:

$$y = upsample(x, n) \tag{2.1}$$

- **n** is the upsampling factor, specified as a positive integer.
- **x** is the input signal.
- **y** is the upsampled output signal.

Because the input matrix is huge, therefore here uses a small matrix as input as example, when the inputs are:

$$x = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \tag{2.2}$$

$$n = 3 \tag{2.3}$$

After upsample function, the sample rate of the input matrix will be improved to 3 times than before, then the output is:

$$y = \begin{bmatrix} a & b \\ 0 & 0 \\ 0 & 0 \\ c & d \\ 0 & 0 \\ 0 & 0 \end{bmatrix} \tag{2.4}$$

In summary, the upsample function is applied in signal processing to increase the temporal resolution and preserve the spectral information of a signal, enabling more precise analysis and subsequent processing.

2.1.2 Filter

MATLAB's filter function is a powerful tool designed for linear filtering operations, widely applied in fields such as signal processing, data smoothing, and frequency analysis[10]. Its basic syntax is:

$$y = filter(b, a, x) \tag{2.5}$$

where b and a represent the coefficients of the rational transfer function, respectively, with x denoting the input signal. By adjusting these coefficients, users

can implement various filtering types, including low-pass, high-pass, band-pass, band-stop, and custom filters, to meet the demands of different applications. The filter function finds extensive utility in tasks like noise removal, data smoothing, and extraction of specific frequency components, thus playing a pivotal role in fields such as signal processing, audio processing, image processing, and control system design. This tool offers researchers and engineers the flexibility to design filters and process data according to specific requirements.

In MATLAB, the filter function supports rational transfer functions, which means you can define a discrete-time system's transfer function by providing coefficients for the numerator and denominator polynomials. This transfer function describes the relationship between the system's input and output, allowing you to perform filtering, analysis, and simulation of the system's behavior [11].

Specifically, the filter function's parameters b and a correspond to the coefficients of the numerator and denominator polynomials of the transfer function, respectively. The numerator polynomial coefficients are defined by b, and the denominator polynomial coefficients are defined by a. This way, you can use rational functions to describe the system's transfer function and then filter signals through that system. A rational transfer function is of the form:

$$Y(z) = \frac{b(1) + b(2)z^{-1} + ... + b(n_b + 1)z^{-n_b}}{1 + a(2)z^{-1} + ... + a(n_a + 1)z^{-n_a}} X(z) \tag{2.6}$$

which handles both FIR and IIR filters. na is the feedback filter order, and nb is the feedforward filter order. Due to normalization, assume a(1) = 1. You also can express the rational transfer function as the difference equation:

$$\begin{aligned} a(1)y(n) =& b(1)x(n) + b(2)x(n-1) + ... + b(n_b + 1)x(n - n_b) \\ & - a(2)y(n-1) - ... - a(n_a + 1)y(n - n_a) \end{aligned} \tag{2.7}$$

Furthermore, you can represent the rational transfer function using its direct-form II transposed implementation, as in the following diagram. Here, na = nb = n-1. The operation of filter at sample m is given by the time-domain difference equations:

$$\begin{aligned} & y(m) = b(1)x(m) + w_1(m-1) \\ & w_1(m) = b(1)x(m) + w_2(m-1) - a(2)y(m) \\ & ... \\ & ... \\ & ... \\ & w_{n-2}(m) = b(n-1)x(m) + wn - 1(m-1) - a(n-1)y(m) \\ & w_{n-1}(m) = b(n)x(m) - a(n)y(m) \end{aligned} \tag{2.8}$$

In this thesis, we only focus on the equations:

$$[y, zf] = filter(b, a, x, zi) \tag{2.9}$$

This function in MATLAB serves as a versatile tool for digital signal processing. It supports both Finite Impulse Response (FIR) and Infinite Impulse Response (IIR) filtering operations. FIR filtering exclusively considers the input signal x and numerator coefficients b, while IIR filtering additionally incorporates denominator coefficients a. The function calculates the filtered output signal y by either performing convolution for FIR or using recursive calculations for IIR. An optional initial state vector, zi, can be provided to set the filter's initial conditions, and the zf variable captures the final state for subsequent filtering. filter facilitates a wide range of filtering tasks, including signal analysis and system modeling, making it an essential tool in MATLAB for digital signal processing applications.

- **b** represents the numerator coefficients (feedforward coefficients) defining the zeros of the filter.
- **a** represents the denominator coefficients (feedback coefficients) defining the poles of the filter.In this thesis, it is 1.
- **x** is the input signal, the digital signal to be filtered.
- **zi** is an optional initial state vector used to specify the filter's initial state. If not provided, a zero state is used. If zi is a matrix or multidimensional array, then the size of the leading dimension must be max(length(a),length(b))-1. The size of each remaining dimension must match the size of the corresponding dimension of x.
- **y** is the filter function returns the filtered output signal
- **zf** is the final state, This final state can be employed as the initial state for subsequent filtering operations.If x is a matrix or multidimensional array, then zf is an array of column vectors of length max(length(a),length(b))-1, such that the number of columns in zf is equivalent to the number of columns in x.

Depending on the provided coefficients and initial state, it can perform various types of filtering operations, including FIR and IIR filtering.

2.2 C++ Code and C-Simulation

There are four files in C++ code, which achieve the functions of MATLAB code and test the result is corresponds to the result of MATLAB code.The C++ code files are:

- **bin_read.h**
- **BVector_Filter_OneSample_Sub.h**
- **BVector_Filter_OneSample_Sub.cpp**
- **BVector_Filter_Test.cpp**

2.2.1 bin_read.h

Due to differences in definitions between the C++ and MATLAB languages, to ensure that the input data matches the MATLAB code used by TIM, we need to read the input data from a .bin file, which consists of binary data. Similarly, for the final testing step, we need to retrieve the output data for comparison from the .bin file. These .bin files contain data in a format compatible with the MATLAB code, ensuring accuracy and consistency. This approach helps mitigate potential issues arising from language disparities and ensures the reproducibility and verifiability of the TIM code. The code is as follows:

```
typedef double                          data_dd;
typedef complex<double>                 data_cd;

std::vector<data_dd> readFile(const char *filename)
{
    // open the file:
    std::streampos fileSize;
    std::ifstream file(filename, std::ios::binary);

    // get its size:
    file.seekg(0, std::ios::end);
    fileSize = file.tellg();
    file.seekg(0, std::ios::beg);

    // read the data:
    std::vector<data_dd> fileData(fileSize);
    file.read((char *)&fileData[0], fileSize);

    return fileData;
}
```

Figure 2.1: bin_read.h code

By employing this file, we ensure that the input data utilized by the C++ code is consistent with the functionality of the MATLAB code. This approach guarantees data compatibility between the two programming languages, promoting accuracy and facilitating seamless data exchange for our computational tasks.

2.2.2 BVector_Filter_OneSample_Sub.h

This file is used to define some global variables, which will be utilized in other files. These global variables possess a shared nature throughout the project, allowing their values to be accessed and modified across different code files. Such utilization of global variables promotes consistency and data sharing throughout the project, while also providing a convenient means of passing information and parameters to facilitate collaboration between various components. By defining and managing these global variables, we can better organize and optimize our code to achieve the project's goals and requirements. In the following sections, we will provide detailed explanations of the definitions and purposes of these global variables, as well as emphasize their significance and roles within the project.

2.2.3 BVector_Filter_OneSample_Sub.cpp

This file serves as the cornerstone of our MATLAB code, playing a pivotal role in the realization of its core functions. It acts as the central hub where complex algorithms, data processing techniques, and critical computations converge to achieve the overarching objectives of our project. The significance of this file lies in its ability to orchestrate and integrate various components of our MATLAB codebase, providing the essential framework upon which the entire software solution is built.

Within this file, we meticulously design and implement the intricate logic and methodologies required to tackle the challenges posed by our research or application. It acts as a repository for essential variables, functions, and data structures, serving as a fundamental reference point for the entirety of our codebase. Through the diligent work conducted within this file, we establish the foundation upon which our MATLAB code executes, ensuring the efficient and accurate execution of complex tasks and analyses.

In the subsequent sections, we will delve into a comprehensive examination of the contents and functionalities encapsulated within this crucial file. We will elucidate the specific algorithms, mathematical models, and data processing techniques employed, highlighting their contributions to the successful achievement of our research objectives. Furthermore, we will explore the interdependencies between this central file and other components of our MATLAB code, demonstrating its role as the linchpin in the pursuit of our project's goals. The flow chat of the file is as follows:

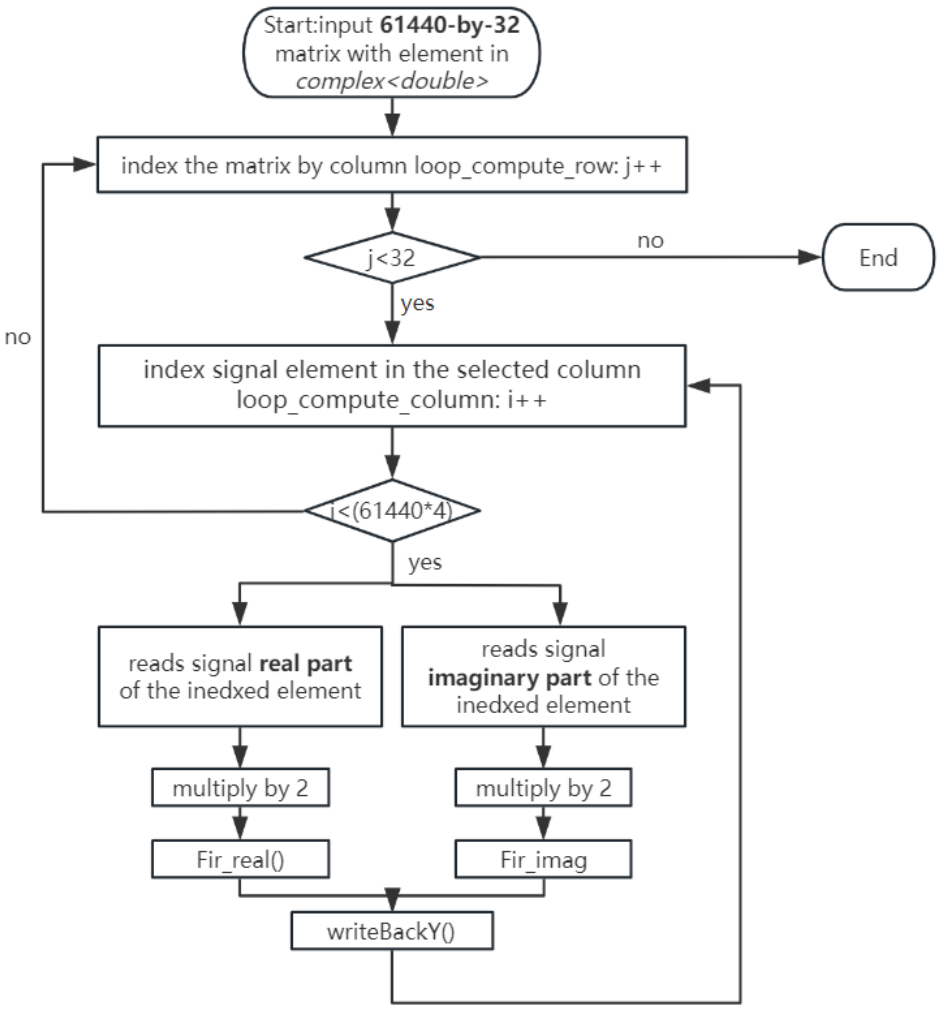


Figure 2.2: Top function flow chat

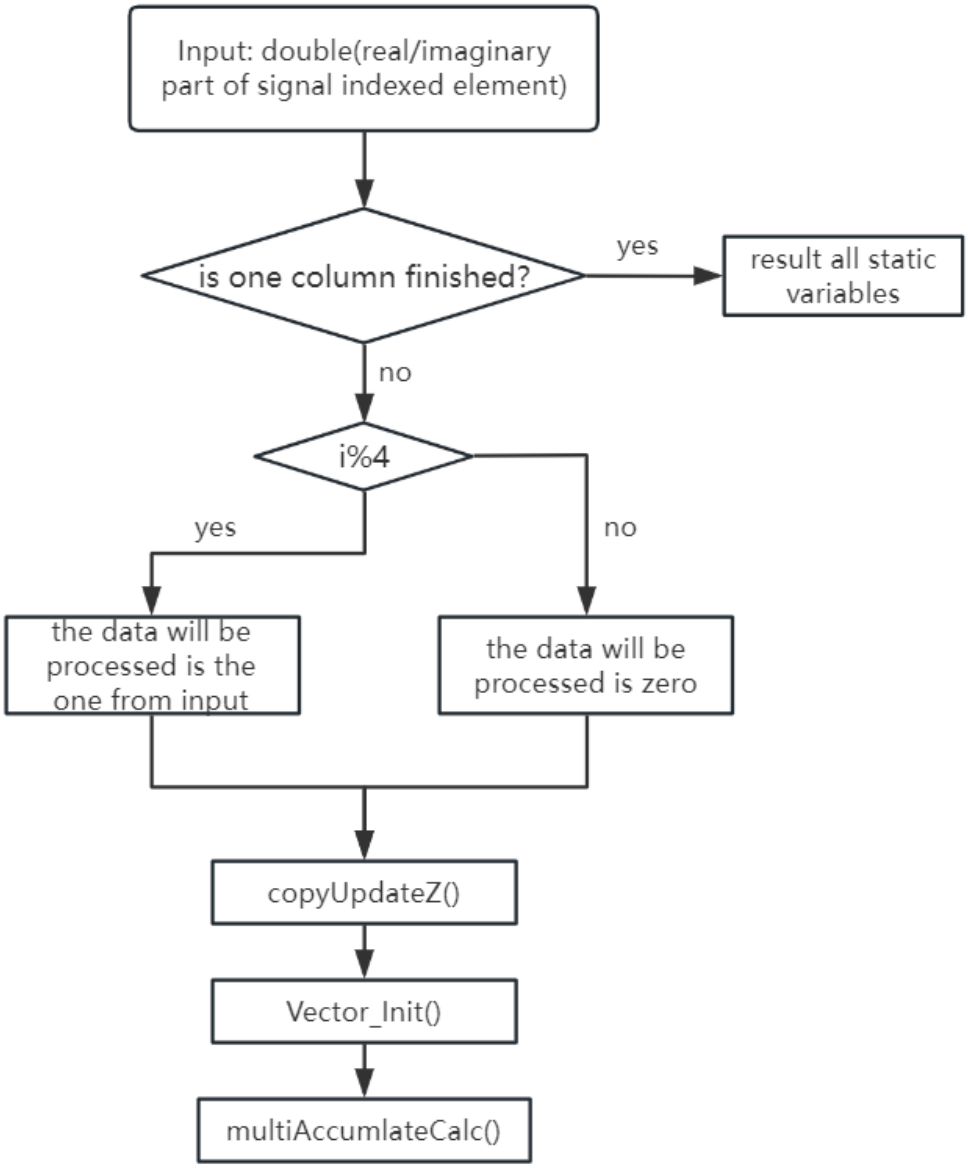


Figure 2.3: Fir function flow chat

The composition of the function and the content of the Top function as follows:

```
// Top function
void Top(data_c *tb_in, data_c *tb_out, data_d *coeff,
    ↪ int N);

//Sub Function
void Fir_real(data_d input_real, data_d B[NUM_COEFF],
    ↪ data_c *Y, int i);
void Fir_imag(data_d input_imag, data_d B[NUM_COEFF],
    ↪ data_c *Y, int i);
void writeBackY(data_c Y, int i, int j, data_c *tb_out);
}
```

Figure 2.4: Composition of the function

```
/* init B*/
    data_d B[NUM_COEFF];
#pragma HLS ARRAY_PARTITION variable=B type=complete dim
  ↪ =1
read_coeff_loop:
    for (int i = 0; i < NUM_COEFF; i++)
    {
        B[i] = coeff[i];
    }
/* end init B*/

//  loop_compute((data_c *)tb_in, B, N, (data_c *)tb_out
  ↪ );
loop_compute_row:
    for (int j = 0; j < NUM_COLUM; j++)
    {
    loop_compute_column:
        for (int i = 0; i < N*4; i++)
        {
            data_c Y;
            data_c input_temp = tb_in[j + i / 4 *
  ↪ NUM_COLUM];

        data_d input_real = input_temp.real() * 2.0;
            data_d input_imag = input_temp.imag() * 2.0;

            Fir_real(input_real, B, &Y, i);
            Fir_imag(input_imag, B, &Y, i);
            writeBackY(Y, i, j, (data_c*)tb_out);
        }
    }
```

Figure 2.5: Content of Top function

- **Main Function Top:** The main function Top serves as the entry point of this code. It accepts input parameters tb_in, tb_out, coeff, and N and is responsible for coordinating the entire signal processing flow.
- **Initialization of Array B:** Within the main function, there is a section of code dedicated to initializing a double-precision floating-point array named B. This array stores the coefficients of a filter, which is a critical component of the signal processing.
- **Main Loop:** The main function contains two nested loops, loop_compute_row and loop_compute_column. These loops collectively handle the processing of input data and write the results back into the output array tb_out.
- **Invocation of Filter Functions:** Inside the main loop, the code calls two filter functions, Fir_real and Fir_imag, to perform filtering operations on the real and imaginary parts of the input data, respectively.
- **Data Processing and Storage:** Within the main loop, after each input data element undergoes filtering, the results are stored in the output array tb_out. The writeBackY function handles the task of correctly writing the results back to their appropriate positions.

Within the Fir subfunction, we implement two fundamental operations: upsample and filter, akin to the corresponding functions in MATLAB. These operations are essential components of our signal processing algorithm, contributing significantly to the accuracy and effectiveness of our data processing pipeline.

C++ Upsample

According to the algorithm of upsample function, the C++ code of it as follows(because the real part is same as the imag part, we use the real part of data to show):

```
loop_compute_row:
    for (int j = 0; j < NUM_COLUM; j++)
    {
    loop_compute_column:
        for (int i = 0; i < N*4; i++)
        {
            data_c input_temp = tb_in[j + i / 4 *
↪ NUM_COLUM];
            data_d input_real = input_temp.real() * 2.0;
            data_d X_real;
            data_d zeros_real = 0.0;
            //upsample
            if(i % 4 == 0)
                X_real = input_real;
            else
                X_real = zeros_real;
        }
    }
```

Figure 2.6: Upsample C++ code

The input matrix X i with size $M \times N$ (the M corresponds 61440 rows, and the N corresponds 32 columns) is:

$$X_{before} = \begin{bmatrix} x_{00} & x_{01} & \dots & x_{0N} \\ x_{10} & x_{11} & \dots & x_{1N} \\ \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots \\ x_{M0} & x_{M1} & \dots & x_{MN} \end{bmatrix} \tag{2.10}$$

After upsampling with the factor 4, the matrix which size is $4M \times N$ will be processed by filter is:

$$X_{after} = \begin{bmatrix} x_{00} & x_{01} & ... & x_{0N} \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ x_{10} & x_{11} & ... & x_{1N} \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ ... & ... & ... & ... \\ ... & ... & ... & ... \\ x_{(M-1)0} & x_{(M-1)1} & ... & x_{(M-1)N} \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \tag{2.11}$$

Therefore, due to the process of upsampling, the size of the output matrix that will be utilized in the next channel block remains the same as that of the input matrix. This aligns with the filter algorithm we previously discussed. In the realm of signal processing, upsampling plays a crucial role. It increases the sampling rate of a signal by inserting zero values between the signal's sample points. This operation results in an output matrix with the same dimensions as the input matrix, ensuring data consistency and compatibility. Maintaining an output matrix of the same size serves the requirements of the filter algorithm we discussed earlier. As mentioned earlier, the filter algorithm necessitates operations on input and output matrices of identical dimensions to effectively process the signal and achieve the desired filtering effects. Through the process of upsampling, we guarantee that the size of the output matrix matches that of the input matrix, establishing a robust foundation for the continuous flow of signal processing. This consistency in matrix size aids in preserving data continuity and ensures the smooth execution of the filter algorithm.

C++ Filter

In accordance with the algorithm for the filter function, the corresponding C++ code is presented below (for simplicity, we illustrate the code using the real part of the data, noting that the real and imaginary parts are processed in the same manner). This filter operates by sequentially processing the input data, column by column. Notably, the data that has undergone processing is returned to a static matrix, which serves as the input variable for subsequent calls to the filter function.

The filter function embodies a critical component of our signal processing pipeline, facilitating the enhancement and manipulation of data. As it traverses

through each column of the input data, the filter meticulously applies the specified filtering operations, ensuring that the desired signal modifications are achieved. This process is carried out iteratively, column by column, ensuring that each data point receives the requisite treatment.

Crucially, the utilization of a static matrix to retain the processed data underscores the iterative nature of this operation. The data's preservation in the static matrix enables seamless continuity when the filter function is called again, ensuring that previous processing steps are considered in subsequent operations. This iterative and data-preserving approach is fundamental to the successful implementation of the filter function within our signal processing framework.
The first part C++ code of the filer like as follows:

```
void Fir_real(data_d input_real, data_d B[NUM_COEFF],
    ↪ data_c *Y, int i)
{
    static int oldest_Z_idx_real = 0;
    static data_d Z_real[NUM_COEFF-1];  // NUM_COEFF =
    ↪ 257
    if (i==0)
    {
        for(int i = 0; i<NUM_COEFF - 1; i++)
        {
            Z_real[i] = 0;
        }
        oldest_Z_idx_real = 0;
    }
}
```

Figure 2.7: Filter C++ code

This section of code is utilized to monitor the state of the variable 'zi.' Upon completion of processing one column of the input matrix, the historical variable 'zi' is cleared, and it assumes the role of a new historical variable initialized with zeros for the subsequent column.

In the context of signal processing, the variable 'zi' (also known as the state variable or historical variable) plays a pivotal role in filter and signal processing algorithms. It is employed to track the past input and output, ensuring the continuity and precision of signal processing.

Once the processing of one column of input data is completed, it is imperative to

reset the historical variable 'zi' to zero. This step is essential as each column of data may exhibit distinct signal characteristics, and, therefore, we aim to commence the processing of each column with a clean slate, free from the influence of previous data.

By clearing 'zi' and initializing it to zero, we ensure that each stage of signal processing begins with a consistent starting point, contributing to the maintenance of data continuity and the accuracy of signal processing. This state management mechanism stands as a critical component in signal processing, laying the foundation for the effective execution of the algorithm.
The second part C++ code of the filer is used to achieve mathematical algorithm, like as follows:

```
void vectorInit_real()
{
    VEC_INIT:
    Y->real(X_real * B[0]);
}
void multiAccumulateCalc_real()
{
    const int unroll_factor = 4;
    data_d part_sum_real;
    MAC:
    for (int i = NUM_COEFF - 1; i != 0; i--)
    {
        const int current_Z_idx_real = (Z_idx_real- i +
↪ (NUM_COEFF-1)) % (NUM_COEFF-1);
        part_sum_real += Z_buffer_real[
↪ current_Z_idx_real] * B[i];
        if(i % unroll_factor == 1)
        {
            Y->real(Y->real() + part_sum_real);
            part_sum_real = 0;
        }
    }
}
```

Figure 2.8: Filter C++ code

The specific algorithm is shown here.Input coefficient B which has k (K=257)values is:

$$B = \begin{bmatrix} b_0 & b_1 & b_2 & b_3 & b_4 & ..... & b_{k-1} \end{bmatrix} \tag{2.12}$$

The history variable zi, which holds (K-1) values, is used to store the data processed by the *Fir* function, originating from the input matrix X. The initial state is as follows:

$$Zi = \begin{bmatrix} z_0 = 0 & z_1 = 0 & z_2 = 0 & z_3 = 0 & z_4 = 0 & ..... & z_{K-2} = 0 \end{bmatrix} \tag{2.13}$$

The input matrix X after upsampling and will be processed with size $M \times N$ (here M corresponds 61440*4 rows and N corresponds 32 columns):

$$X = \begin{bmatrix} x_{00} & x_{01} & x_{02} & x_{03} & ..... & x_{0N} \\ x_{10} & x_{11} & x_{12} & x_{13} & ..... & x_{1N} \\ x_{20} & x_{21} & x_{22} & x_{23} & ..... & x_{2N} \\ x_{30} & x_{31} & x_{32} & x_{33} & ..... & x_{3N} \\ x_{40} & x_{41} & x_{42} & x_{43} & ..... & x_{4N} \\ ... & ... & ... & ... & ..... & ... \\ ... & ... & ... & ... & ..... & ... \\ ... & ... & ... & ... & ..... & ... \\ x_{(M-2)0} & x_{(M-2)1} & x_{(M-2)2} & x_{(M-2)3} & ..... & x_{(M-2)N} \\ x_{(M-1)0} & x_{(M-1)1} & x_{(M-1)2} & x_{(M-1)3} & ..... & x_{(M-1)N} \end{bmatrix} \tag{2.14}$$

When the input data to Fir function is **K**th data of the first column from X, after the *vectorInit* function, zi with current state:

$$Zi = \begin{bmatrix} z_0 = x_0 & z_1 = x_1 & z_2 = x_2 & z_3 = x_3 & z_4 = x_4 & ..... & z_{K-2} = x_{K-2} \end{bmatrix} \tag{2.15}$$

the output will be:

$$X = \begin{bmatrix} x_{00} & x_{01} & x_{02} & x_{03} & ..... & x_{0N} \\ x_{10} & x_{11} & x_{12} & x_{13} & ..... & x_{1N} \\ x_{20} & x_{21} & x_{22} & x_{23} & ..... & x_{2N} \\ x_{30} & x_{31} & x_{32} & x_{33} & ..... & x_{3N} \\ x_{40} & x_{41} & x_{42} & x_{43} & ..... & x_{4N} \\ ... & ... & ... & ... & ..... & ... \\ x_{(K-1)0}b_0 & ... & ... & ... & ..... & ... \\ ... & ... & ... & ... & ..... & ... \\ x_{(M-2)0} & x_{(M-2)1} & x_{(M-2)2} & x_{(M-2)3} & ..... & x_{(M-2)N} \\ x_{(M-1)0} & x_{(M-1)1} & x_{(M-1)2} & x_{(M-1)3} & ..... & x_{(M-1)N} \end{bmatrix} \tag{2.16}$$

Then, by passing the *multiAccumulateCalc* function, the result will be:

$$X = \begin{bmatrix} x_{00} & ..... & x_{0N} \\ x_{10} & ..... & x_{1N} \\ x_{20} & ..... & x_{2N} \\ x_{30} & ..... & x_{3N} \\ x_{40} & ..... & x_{4N} \\ ... & ..... & ... \\ x_{(K-1)0}b_0 + z_{K-2}b_1 + z_{K-3}b_2 + ... + z_0 b_{K-1} & ... & \\ ... & ..... & ... \\ x_{(M-2)0} & ..... & x_{(M-2)N} \\ x_{(M-1)0} & ..... & x_{(M-1)N} \end{bmatrix} \tag{2.17}$$

Each data point within the input matrix yields an identical outcome when processed by the *Fir* function. This indirect approach effectively mirrors the functionality of MATLAB's filter function. Through a meticulous comparison of results obtained from both the C++ and MATLAB implementations, it is evident that the correctness rate stands at an impressive 100%.

This outcome underscores the successful alignment between the two implementations, affirming the fidelity of the C++ code in emulating the behavior of the MATLAB filter function. The uniformity of results, achieved for every data point in the input matrix, serves as a compelling validation of the C++ implementation's accuracy and reliability in replicating the desired filtering behavior originally established in MATLAB.

Chapter 3

Synthesis

Design synthesis is the phase during which code is translated into an actual circuit with lower-level implementations such as gates, flip-flops, and adders. The input design is transformed into a netlist that describes the components used and the interconnections among them[12].

The process of design synthesis commences with a syntax check when provided with an HDL-based design as input. Subsequently, logic is optimized using various techniques, including the elimination of redundant logic, logic simplification, and size reduction, all while simultaneously enhancing its implementation speed.

This leads to the incorporation of pragmas for design optimization, such as latency reduction and throughput maximization, allowing for control over the resources of the final RTL (Register-Transfer Level) code. These pragmas are meant to be directly inserted into the source code and are interpreted directly by the synthesis tool. Optimization directives are embedded within the C source code. If the optimization directives are embedded in the code, they are automatically applied to every solution during re-synthesis.

Key steps in the design synthesis process include:

1. Syntax Check: Accepting designs based on hardware description languages (HDL) as input and conducting syntax checks.

2. Logic Optimization: Optimizing logic using various techniques, including the elimination of redundant logic, logic simplification, and size reduction.

3. Integration of Compilation Directives: Embedding compilation directives (pragma) to achieve design optimization, such as reducing latency, maximizing throughput, and controlling resources.

4. Automatic Application of Optimization Directives: If optimization directives are embedded in the code, they are automatically applied to each solution during re-synthesis.

This design synthesis process transforms abstract design descriptions into hardware-level circuits, laying the foundation for the implementation of high-performance circuit accelerators on programmable hardware platforms.

This thesis will introduce the synthesis methods in the project.

3.1 Pipelining

3.1.1 Principle

Pipelining enables concurrent execution of operations, eliminating the need for each execution step to wait for all previous operations to complete before commencing the next one. This pipelining technique is applicable to both functions and loops, and the enhancements in throughput resulting from function pipelining are illustrated in the following figure.

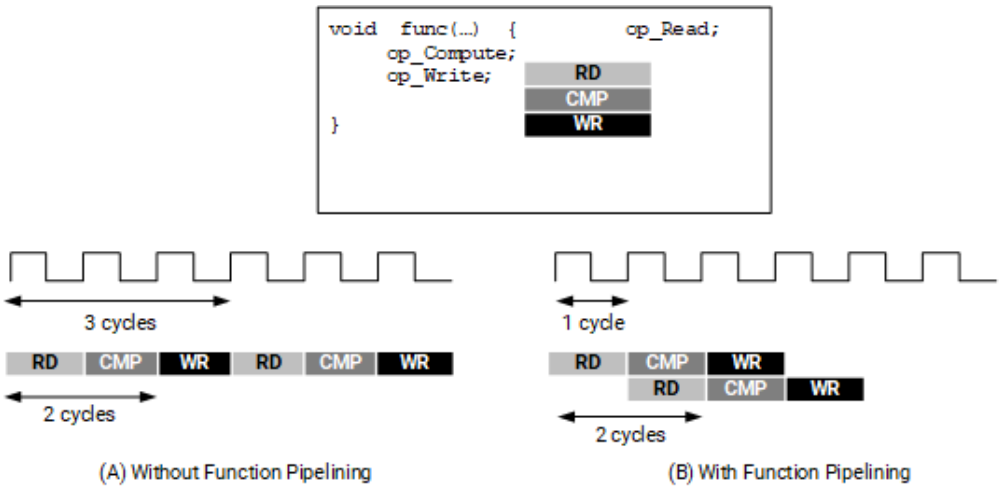


Figure 3.1: Function pipelining behavior

In the absence of pipelining, the function in the above example reads an input every 3 clock cycles and produces an output after 2 clock cycles. The function exhibits an initiation interval (II) of 3 and a latency of 3. However, with pipelining applied in this example, a new input is read every cycle (II=1), while the output latency remains unaltered.

Loop pipelining enables the operations within a loop to overlap and be executed concurrently. In the Figure 3.2, (A) depicts the default sequential operation with a 3-clock-cycle gap between each input read (II=3), requiring 8 clock cycles before the final output write occurs. In the pipelined configuration of the loop, as shown in (B), a fresh input sample is read every cycle (II=1), and the last output is

written after only 4 clock cycles. This substantial improvement applies to both the initiation interval (II) and latency while utilizing the same hardware resources.

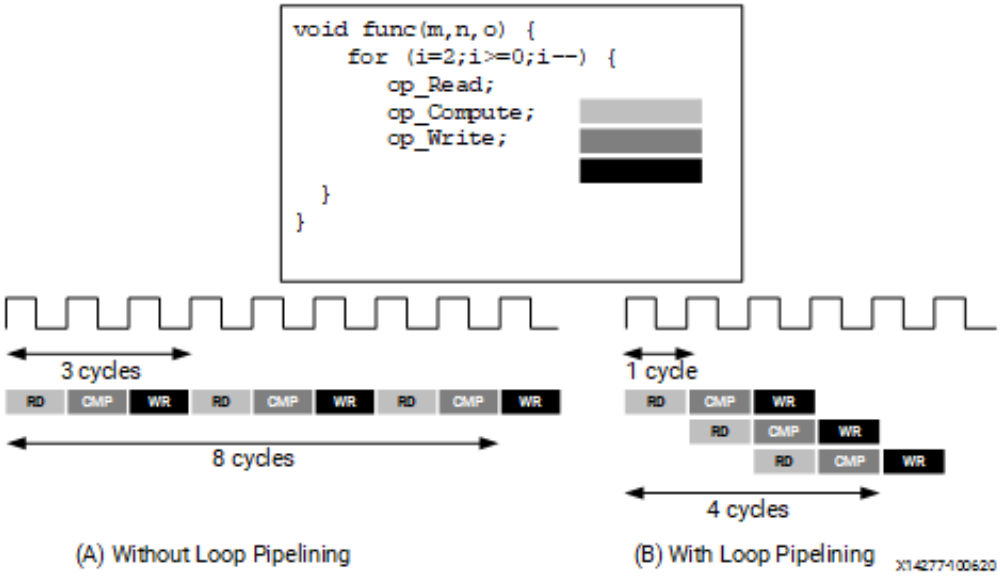


Figure 3.2: Loop pipelining behavior

Pipelining functions or loops is accomplished by employing the PIPELINE directive, which should be indicated within the region encompassing the function or loop body. In cases where the initiation interval (II) is not explicitly defined, it defaults to 1 but can also be explicitly specified.

Pipelining is selectively applied to the designated region and does not extend to the hierarchy below it. However, within this hierarchy, all loops are automatically unrolled. In cases where there are sub-functions within the hierarchy below the specified function, they must be individually subjected to the pipelining process. By pipelining these sub-functions, the upper-level pipelined functions can capitalize on the enhanced pipeline performance. Conversely, any sub-function situated below the top-level pipelined function, which remains non-pipelined, could potentially serve as the limiting factor affecting overall pipeline performance.

There is a difference in how pipelined functions and loops behave:

- In the case of functions, the pipeline runs forever and never ends.
- In the case of loops, the pipeline executes until all iterations of the loop are completed.

The difference between function and loop in pipelining shows in Figure 3.3. As depicted in Figure 3.3, a pipelined function continually processes new inputs and generates fresh outputs. In contrast, for a pipelined loop, there exists a distinct

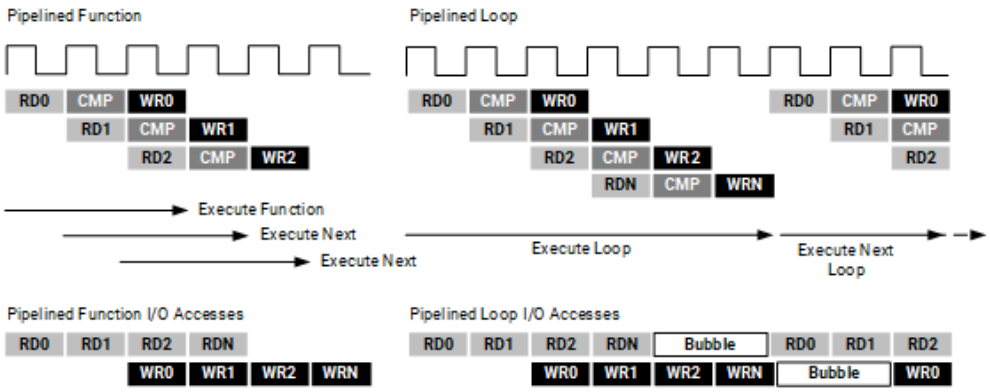


Figure 3.3: Function and Loop Pipelining Behavior

behavior due to the requirement that all loop operations within an iteration must complete before transitioning to the next iteration. This results in what is referred to as a 'bubble' in the data stream—a phase where no new inputs are read during the completion of final iterations, and a phase where no new outputs are written during the initiation of new loop iterations.

To address the challenges illustrated in the preceding figure (Function and Loop Pipelining), the PIPELINE pragma incorporates an optional **rewind** command. This command facilitates the overlapping execution of consecutive calls to the loop, particularly when this loop constitutes the outermost construct within the top function or a dataflow process that undergoes multiple executions.

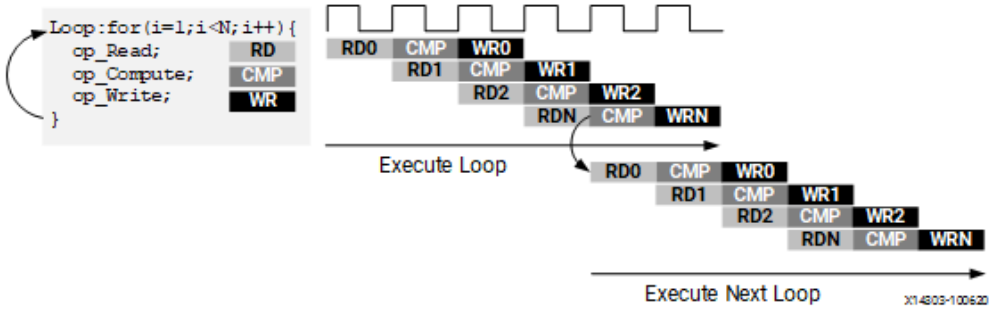


Figure 3.4: Loop Pipelining with Rewind Option

3.1.2 Syntax

The syntax of pipeline pragma is:

```
#pragma HLS pipeline II=<int> off rewind
```

Where:

- **II=<int>:** Specifies the desired initiation interval for the pipeline. The HLS tool tries to meet this request. Based on data dependencies, the actual result might have a larger initiation interval.
- **off:** Optional keyword. Turns off pipeline for a specific loop or function.
- **rewind:** Optional keyword. Enables rewinding as described in Rewinding Pipelined Loops for Performance. This enables continuous loop pipelining with no pause between one execution of the loop ending and the next execution starting. Rewinding is effective only if there is one single loop (or a perfect loop nest) inside the top-level function. Rewinding is effective only if there is one single loop (or a perfect loop nest) inside the top-level function. The code segment before the loop:
  - Is considered as initialization.
  - Is executed only once in the pipeline.
  - Cannot contain any conditional operations (if-else).

3.1.3 Application

This pragma is applied in the code:

```
read_coeff_loop:
    for (int i = 0; i < NUM_COEFF; i++)
    {
#pragma HLS pipeline II = 1 rewind
        B[i] = coeff[i];
    }
```

Figure 3.5: Using Pipeline in Loop

```
1    void Fir_real(data_d input_real, data_d B[NUM_COEFF
   ↪ ], data_c *Y, int i)
2    {
3        ......
4    #pragma HLS PIPELINE II=1
5        .....
6    }
```

Figure 3.6: Using Pipeline in Function

Through the utilization of pipelining, a noteworthy enhancement in throughput becomes evident when invoking the *Fir* function and conducting the *read_coeff_loop* loop. This improvement is especially pronounced when we consider the performance in scenarios where both functions operate concurrently without pipelining. In such cases, the simultaneous execution of these functions exhibits a marked increase in data processing efficiency.

3.2 Unroll

3.2.1 Principle

By default, Vitis HLS maintains loops in a rolled format. In this configuration, each iteration of the loop consumes a hardware resource. Although this approach ensures efficient resource utilization, it can occasionally introduce performance bottlenecks.

Vitis HLS offers the capability to unroll FOR loops, either fully or partially, through the use of the UNROLL pragma or directive.

Figure 3.7 illustrates both the benefits of loop unrolling and the associated considerations when applying this technique. In this scenario, we assume that the arrays a[i], b[i], and c[i] are mapped to block RAMs. This example underscores how effortlessly diverse implementations can be generated through the straightforward application of loop unrolling.

- **Rolled Loop:** When the loop is rolled, each iteration is executed in distinct clock cycles. In this particular implementation, it necessitates four clock cycles, utilizes only a single multiplier, and each block RAM operates as a single-port block RAM.
- **Partially Unrolled Loop:** In this instance, the loop is partially unrolled

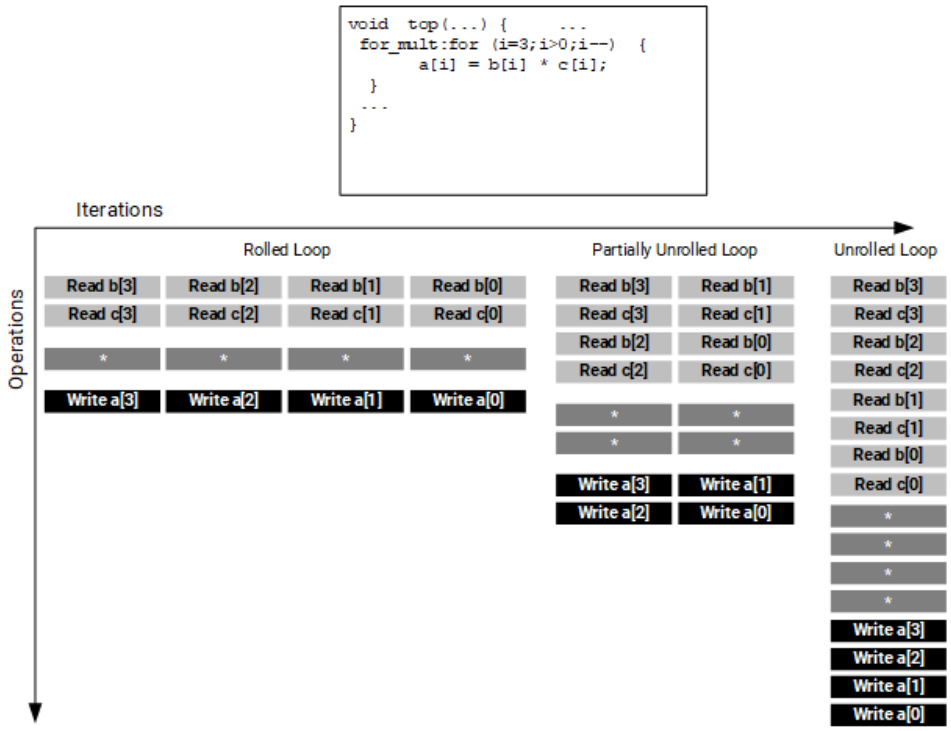


Figure 3.7: Loop Unrolling Details

with a factor of 2. This configuration demands two multipliers and dual-port RAMs, enabling two simultaneous reads or writes to each RAM in a single clock cycle. Remarkably, this implementation only requires 2 clock cycles to finish, representing half the initiation interval and half the latency when compared to the rolled loop version.

- **Unrolled loop:** In the fully unrolled version, all loop operations can be executed within a single clock cycle. However, this implementation mandates the presence of four multipliers. Of greater significance, this configuration necessitates the capability to carry out 4 read and 4 write operations simultaneously within the same clock cycle. Given that a block RAM typically supports a maximum of two ports, this implementation mandates the partitioning of arrays.

To execute loop unrolling, you have the option to employ UNROLL directives for specific loops within the design. Alternatively, you can apply the UNROLL directive at the function level, which results in the unrolling of all loops contained within the function's scope.

When a loop is fully unrolled, all operations are executed concurrently, contingent upon data dependencies and resource availability. However, if operations within

one iteration of the loop rely on results from a prior iteration, parallel execution is not feasible, and they will run as soon as the required data becomes accessible. In the case of a fully unrolled and thoroughly optimized loop, it typically entails the presence of multiple instances of the logic within the loop body.

3.2.2 Syntax

The syntax of unroll pragma is:

```
1    #pragma HLS unroll factor=<N> region skip_exit_check
```

Where:

- **factor=<N>:** Specifies a non-zero integer indicating that partial unrolling is requested. The loop body is repeated the specified number of times, and the iteration information is adjusted accordingly. If factor= is not specified, the loop is fully unrolled.
- **skip_exit_check:** Optional keyword that applies only if partial unrolling is specified with factor=. The elimination of the exit check is dependent on whether the loop iteration count is known or unknown:
  - Fixed bounds: No exit condition check is performed if the iteration count is a multiple of the factor.If the iteration count is not an integer multiple of the factor, the tool:
    - Prevents unrolling.
    - Issues a warning that the exit check must be performed to proceed.
  - Variable bounds: The exit condition check is removed. You must ensure that:
    - The variable bounds is an integer multiple of the factor.
    - No exit check is in fact required.

3.2.3 Application

The unroll pragma is applied like as follows:

```
1     for(int i = 0; i<NUM_COEFF - 1; i++)
2             {
3     #pragma HLS UNROLL  // max=256
4                 Z_real[i] = 0;
5             }
6             oldest_Z_idx_real = 0;
7         }
```

Figure 3.8: Using Unroll in Function

Utilizing the unroll pragma ensures that all iterations within a loop conclude simultaneously. Nonetheless, it is essential to acknowledge that the resource requirements for implementing such a loop will also escalate. Consequently, striking a balance between resource utilization and throughput becomes a critical consideration.

3.3 Array_Partition

3.3.1 Principle

When we optimize our code using pipeline and unroll to improve throughput, the optimized code achieves maximum parallelism. However, this can also lead to constraints on resources, as mentioned earlier. The primary constraint arises from limitations in memory ports, where these optimized codes may struggle to schedule read operations effectively when mapping to hardware resources. This limitation stems from the fact that block RAMs typically cannot simultaneously process multiple data reads or writes due to memory port contention. As a result, Vitis HLS reports may display warnings related to the final Initiation Interval (II), and the optimization results may not meet our expectations.

This concern primarily stems from the use of arrays, which are mapped to block RAM resources, each typically having a maximum of two data ports. Such limitations can constrain the throughput of algorithms that involve frequent read/write (or load/store) operations. To enhance bandwidth, one approach is to partition the array, originally represented as a single block RAM resource, into multiple smaller arrays, effectively increasing the number of available ports.

Arrays are partitioned using the ARRAY_PARTITION directive. Vitis HLS provides three types of array partitioning, as shown in the Figure 3.9. The three styles of partitioning are:

- **block:** The original array is split into equally sized blocks of consecutive elements of the original array.
- **cyclic**: The original array is split into equally sized blocks interleaving the elements of the original array.
- **complete:** The default operation is to split the array into its individual elements. This corresponds to resolving a memory into registers.

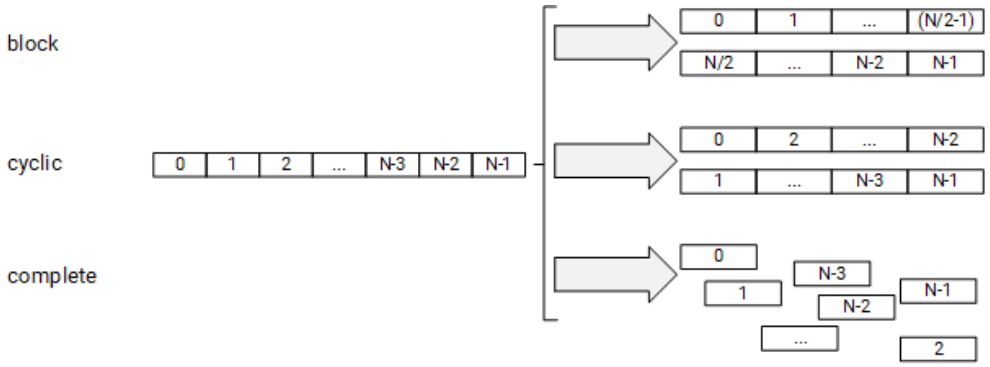


Figure 3.9: Array Partitioning

For block and cyclic partitioning, the 'factor' parameter determines the quantity of arrays generated. In the illustration above, a 'factor' of 2 is employed, effectively dividing the array into two smaller arrays. When the number of elements in the array is not an exact multiple of the 'factor,' the final array may contain fewer elements.

When partitioning multi-dimensional arrays, the dimension option is used to specify which dimension is partitioned, like as follows:

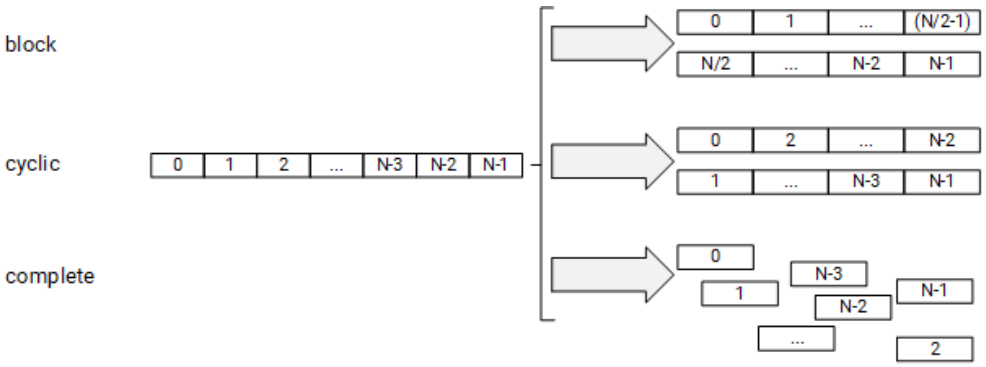


Figure 3.10: Partitioning Array Dimensions

The subsequent illustration illustrates the application of the 'dimension' parameter for partitioning. It visually exemplifies that partitioning dimension 3 leads to the creation of 4 distinct arrays, while partitioning dimension 1 yields 10 separate arrays. In cases where 'dimension' is specified as zero, partitioning occurs across all dimensions.

3.3.2 Syntax

The syntax of array_partition pragma is:

```
1    #pragma HLS array_partition variable=<name> type=<
   ↪ type>  factor=<int>  dim=<int>
```

Where:

- **variable=<name>:** A required argument that specifies the array variable to be partitioned.
- **type=<type>:** Optionally specifies the partition type. The default type is complete. The following types are supported:
  - cyclic

  - block
  - complete

- **factor=<int>:** Specifies the number of smaller arrays that are to be created

3.3.3 Application

The array_partition pragma is applied like as follows:

```
1         data_d B[NUM_COEFF];
2     #pragma HLS ARRAY_PARTITION variable=B type=complete
  ↪   dim=1
3     ......
4     void Fir_real(data_d input_real, data_d B[NUM_COEFF
  ↪ ], data_c *Y, int i)
5     {
6         .......
7         static data_d Z_real[NUM_COEFF-1];  // NUM_COEFF
  ↪   = 257
8     #pragma HLS ARRAY_PARTITION variable=Z_real type=
  ↪ complete
9         data_d Z_buffer_real[NUM_COEFF-1];
10    #pragma HLS ARRAY_PARTITION variable=Z_buffer_real
  ↪ type=complete
11        ......
12    }
```

Figure 3.11: Using Unroll in Function

Array_partition subdivides arrays into smaller arrays or individual elements, resulting in RTL representations containing multiple small memories or registers. This approach effectively reduces the need for large memories on the board, conserving resources. It enhances the efficiency of large memories by effectively increasing the number of memory read and write ports, significantly boosting design throughput. However, the notable drawback is the requirement for additional small memories and/or registers. When considering board-level resource constraints, this trade-off is acceptable.

3.4 Inline

3.4.1 Principle

Inlining optimization is a compiler technique aimed at reducing the overhead of function calls. When a function exists within a program's hierarchy and can be called by other functions, inlining optimization allows the compiler to incorporate the contents of the function directly into the calling function, rather than treating it as an independent entity. This means that at the locations where the function is called, you will no longer see explicit calls to that function, as its code has been inserted into the calling function's code. Regardless of how many times this function is called, it no longer appears as an independent level in the RTL (Register-Transfer Level) hierarchy. In hardware terms, this means that the compiler will choose the same hardware to process this function wherever it is called. This approach effectively conserves area resources, especially when parallel optimization techniques are applied at the call sites of the function. Inlining allows functions sharing to be better controlled. For functions to be shared they must be used within the same level of hierarchy.

3.4.2 Syntax

The syntax of inline pragma is:

```
1    #pragma HLS inline <recursive | off>
```

Where:

- **recursive:** By default, only one level of function inlining is performed, and functions within the specified function are not inlined. The recursive option inlines all functions recursively within the specified function or region.
- **off:** Disables function inlining to prevent specified functions from being inlined. For example, if recursive is specified in a function, this option can prevent a particular called function from being inlined when all others are.

3.4.3 Application

The inline pragma is applied like as follows:

```
1   void Fir_imag(data_d input_imag, data_d B[NUM_COEFF
    ↪ ], data_c *Y, int i)
2   {
3       ...
4       copyUpdateZ_real(X_real, Z_real,
    ↪ oldest_Z_idx_real, Z_idx_real, Z_buffer_real);
5       vectorInit_real(B, X_real, Y);
6       multiAccumulateCalc_real(B, Z_idx_real, Y,
    ↪ Z_buffer_real);
7   }
8
9   void copyUpdateZ_real(data_d X_real, data_d Z_real[
    ↪ NUM_COEFF - 1], int &oldest_Z_idx_real, const int
    ↪ &Z_idx_real, data_d Z_buffer_real[NUM_COEFF - 1])
10  {
11      #pragma HLS INLINE
12      ...
13  }
14
15  void vectorInit_real(data_d B[NUM_COEFF], data_d
    ↪ X_real, data_c *Y)
16  {
17      #pragma HLS INLINE
18      ...
19  }
20
21  void multiAccumulateCalc_real(data_d B[NUM_COEFF],
    ↪ const int &Z_idx_real, data_c *Y, data_d
    ↪ Z_buffer_real[NUM_COEFF - 1])
22  {
23      #pragma HLS INLINE
24      ...
25  }
```

Figure 3.12: Using Inline in Function

The inline optimization technique significantly reduces area resource consumption when calling sub-functions within the same hierarchy. While this may result in a reduction in throughput, it is the most suitable approach when dealing with a large input matrix. When processing large-scale data, optimizing hardware resource utilization is often more critical than improving throughput, as it helps reduce area costs and enhance overall performance efficiency.

3.5 Loop_Tripcount

3.5.1 Principle

When a loop contains variable bounds, certain optimization operations that can be applied by Vitis HLS are disabled because Vitis HLS cannot know when the loop will terminate. This is because the number of loop iterations is determined by the values of variables rather than being a fixed constant. Due to this uncertainty, Vitis HLS restricts the application of certain advanced optimization operations as it cannot accurately estimate the execution time of the loop. These optimization operations typically require knowledge of the loop's iteration count at compile-time to generate the best hardware description. Therefore, when a loop has variable bounds, the compiler limits some of the optimization operations on the loop to ensure that correct hardware descriptions are generated even in cases of uncertain loop termination.

3.5.2 Syntax

The syntax of loop_tripcount pragma is:

```
1    #pragma HLS loop_tripcount min=<int> max=<int> avg=<
   ↪ int>
```

Where:

- **max= <int>:** Specifies the maximum number of loop iterations.
- **min=<int>:** Specifies the minimum number of loop iterations.
- **avg=<int>:** Specifies the average number of loop iterations.

3.5.3 Application

The loop_tripcount pragma is applied like as follows:

```
1     loop_compute_row:
2     for (int j = 0; j < NUM_COLUM; j++)
3     {
4     loop_compute_column:
5         for (int i = 0; i < N*4; i++)
6         {
7             ...
8             #pragma HLS LOOP_TRIPCOUNT  min=61440*4 max
  ↪ =61440*4 avg=61440*4
9             ...
10    }     }
```

Figure 3.13: Using Loop_Tripcount in Function

In Figure 3.13, 'N' is a variable that is not defined with a specific value. Therefore, the optimization directives within the function can only be executed after specifying the bounds of the loop *loop_compute_column* using the loop_tripcount optimization directive. This is because the number of iterations in the loop needs to be determined at compile time to generate the optimal hardware description. Thus, in the presence of variable bounds, the compiler restricts certain optimization operations on the loop to ensure that the correct hardware description is generated even when the loop's end time is uncertain.

3.6 Dataflow

3.6.1 Principle

The DATAFLOW pragma facilitates task-level pipelining, enabling functions and loops to operate concurrently, thereby enhancing the RTL implementation's concurrency and overall design throughput.

In a C++ description, all operations are typically executed sequentially. In the absence of resource-limiting directives like pragma HLS allocation, Vitis HLS strives to minimize latency and enhance concurrency. However, data dependencies can impose limitations. For instance, functions or loops accessing arrays must complete all read/write accesses to those arrays before proceeding, preventing subsequent functions or loops from commencing their operations until the data is

ready. DATAFLOW optimization empowers functions and loops to initiate their operations even before previous functions or loops have completed all of theirs.The dataflow optimization is useful on a set of sequential tasks show as follows: Dataflow

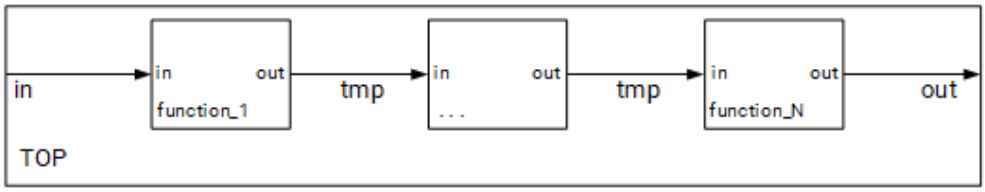


Figure 3.14: Sequential Functional Description

optimization is a potent technique for enhancing both design throughput and latency. It achieves this by transforming a sequence of sequential tasks into an architecture of concurrent processes, as illustrated below: The following diagram illustrates

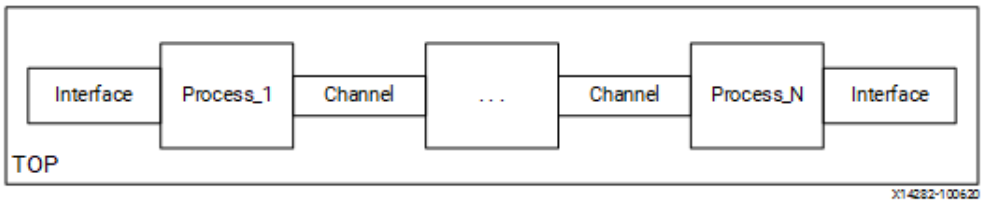


Figure 3.15: Parallel Process Architecture

the impact of dataflow optimization on task execution. In scenario (A), dataflow optimization is not employed, and three functions are executed sequentially. In the preceding iteration, func_c's result must be written back before func_A can write its input data. This results in longer startup time intervals and lower efficiency.

Conversely, in the same example, (B) demonstrates the advantages of dataflow optimization. In the same iteration, func_A and func_B can execute in parallel without waiting for func_A to complete. This significantly reduces the startup time interval, improving performance and efficiency.

However, achieving such parallelism comes with a hardware overhead. When a specific region, like a function body or a loop body, is designated for dataflow optimization in Vitis HLS, the tool scrutinizes the function or loop body. It then generates individual channels to represent the dataflow, storing the results of each task within that region. These channels may take the form of straightforward FIFOs for scalar variables or ping-pong (PIPO) buffers for non-scalar variables such as arrays. Each of these channels includes signals that indicate whether the FIFO or ping-pong buffer is full or empty, forming a data-driven handshaking interface.

By implementing separate FIFOs and/or ping-pong buffers, Vitis HLS allows each task to execute at its own pace, and the throughput is solely dependent on the

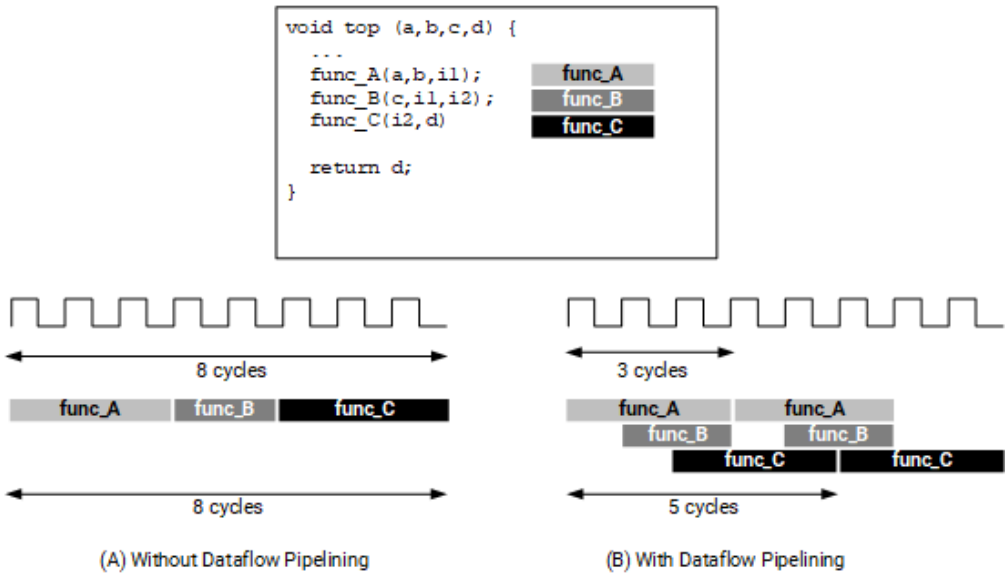


Figure 3.16: Dataflow Optimization

availability of input and output buffers. This approach facilitates a more efficient interleaving of task execution compared to a standard pipelined implementation. However, it does introduce additional FIFO or block RAM registers for the ping-pong buffer, which is the trade-off for these advantages.

Howere,there are some rules we must obey, then we can the dataflow optimization will active:

- **Canonical Forms**
  - The canonical form for a function where sub-functions are not inlined.
  - Dataflow inside a loop body.
    - Initial value declared in the loop header and set to 0.
    - The loop condition is a positive numerical constant or constant function argument.
    - Increment by 1.
    - Dataflow pragma needs to be inside the loop.
- **Canonical Body**
  - Use a local, non-static scalar or array/pointer variable, or local static stream variable. A local variable is declared inside the function body (for dataflow in a function) or loop body (for dataflow inside a loop).

  - A sequence of function calls that pass data forward (with no feedback), from a function to one that is lexically later, under the following conditions:
    * Variables (except scalar) can have only one reading process and one writing process.
    * Use write before read (producer before consumer) if you are using local variables, which then become channels.
    * Use read before write (consumer before producer) if you are using function arguments. Any intra-body anti-dependencies must be preserved by the design.
    * Function return type must be void.
    * No loop-carried dependencies among different processes via variables.
    * No control whatsoever is supported inside a dataflow region, except for function calls (that define processes).

- **Dataflow Optimization Limitations** The DATAFLOW optimization optimizes the flow of data between tasks (functions and loops), and ideally pipelined functions and loops for maximum performance. It does not require these tasks to be chained, one after the other, however there are some limitations in how the data is transferred.
  - Reading from function inputs or writing to function outputs in the middle of the dataflow region.
  - Single-producer-consumer violations.
  - Bypassing tasks and channel sizing.
  - Feedback between tasks.
  - Conditional execution of tasks.
  - Loops with multiple exit conditions.

- **Reading from Inputs/Writing to Outputs** Inputs to the function should be read at the onset of the dataflow region, while outputs should be written at its conclusion. Performing read/write operations on the function's ports within the dataflow region may result in sequential execution of processes, potentially hindering performance due to the absence of overlap.

3.6.2 Syntax

The syntax of dataflow pragma is:

```
#pragma HLS dataflow [disable_start_propagation]
```

Where: **disable_start_propagation**: Optionally disables the creation of a start FIFO used to propagate a start token to an internal process. Such FIFOs can sometimes be a bottleneck for performance.

3.6.3 Application

The loop_tripcount pragma is applied like as follows: By applying dataflow op-

```
void Top(data_c *tb_in, data_c *tb_out, data_d *
↪ coeff, int N)
{
    ...
    #pragma HLS DATAFLOW
    ...
    Fir_real(input_real, B, &Y, i);
    Fir_imag(input_imag, B, &Y, i);
    writeBackY(Y, i, j, (data_c*)tb_out);

}
```

Figure 3.17: Using Loop_Tripcount in Function

timization within the Top function, it becomes possible to achieve maximum parallelism among its three sub-functions, namely *Fir_real*, *Fir_imag*, and *writeBackY*. However, this enhanced parallelism comes at the cost of increased resource utilization. Nevertheless, in pursuit of optimizing throughput, this resource overhead is deemed acceptable.

3.7 Interface

Principle

In C/C++ code, input and output tasks are seamlessly carried out through formal function arguments. In contrast, RTL designs necessitate the completion of these same tasks via designated interface ports, often following specific input/output (I/O) protocols. This transition introduces variations in handling such operations.

In a Vitis HLS design, the top-level function's arguments are transformed into interfaces and ports that consolidate multiple signals. These interfaces and ports define the communication protocol between the HLS design and external components. Vitis HLS automatically generates these interfaces according to industry standards that specify the protocol. The type of interfaces created by Vitis HLS varies depending on factors such as the data type and direction of the top-level function's parameters, the chosen solution's target flow, and the default interface configuration settings set by config_interface. In this design, the RTL-implemented ports are derived from global variables, which are accessible to the top-level function and defined outside its scope. If global variables are accessed but all read and write operations are performed locally within the function, then the resource is instantiated in the RTL design.

The target flows supported by Vitis HLS as described in Vitis HLS Process Overview include:

- The **Vivado IP flow** which is the default flow for the tool.
- The **Vitis Kernel flow**, which is the bottom-up design flow for the Vitis Application Acceleration Development flow.

In this thesis, we only focus on the Vivado IP flow.The Vivado IP flow offers extensive support for a wide range of I/O protocols and handshakes, catering to the diverse requirements of FPGA designs across various applications. This flow accommodates both traditional system design practices, where multiple IPs are integrated into a system, and IPs can be generated using Vitis HLS.Within this IP flow, two modes of control govern the execution of the system:

- Software Control: The system is managed through a software application, which operates on an embedded Arm processor or an external x86 processor. Drivers facilitate access to hardware design components, allowing for the reading and writing of registers in the hardware to regulate the execution of IPs within the system.
- Self Synchronous: In this mode, the IP exposes signals that initiate and terminate the execution of the kernel. These signals are controlled by other

IPs or elements within the system design that oversee the execution of the IPs.

The Vivado IP flow supports memory, stream, and register interface paradigms where each paradigm supports different interface protocols to communicate with the external world. The default interfaces are defined by the C-argument type in the top-level function, and the default paradigm, as shown in the following table:

C-Argument Type	Supported Paradigms	Default Paradigm	Default Interface Protocol		
			Input	Output	Inout
Scalar variable (pass by value)	Register	Register	ap_none	N/A	N/A
Array	Memory, Stream	Memory	ap_memory	ap_memory	ap_memory
Pointer	Memory, Stream, Register	Register	ap_none	ap_vld	ap_ovld
Reference	Register	Register	ap_none	ap_vld	ap_vld
`hls::stream`	Stream	Stream	ap_fifo	ap_fifo	N/A

Figure 3.18: C-argument Type

Because the input of this design is array, according to the table, we only focus on the ap_memory interface. In the context of the Vivado IP flow, ap_memory protocol serves as a communication mechanism with memory resources like BRAM and URAM. This protocol effectively manages both address and data phases. Initially, it issues requests for resource read/write operations and awaits an acknowledgment indicating resource availability. Subsequently, it proceeds to the data transfer phase for reading/writing.

An essential aspect to note about ap_memory is its capability for single-beat data transfers to a singular address, which sets it apart from m_axi, a protocol supporting burst accesses. This characteristic positions ap_memory as a more lightweight option in comparison to alternative protocols.

3.7.1 Syntax

The syntax of Interface pragma is:

```
1    #pragma HLS interface mode=<mode> port=<name> depth
   ↪ =<int>
```

Where:

- **mode=<mode>:** Specifies the interface protocol mode for function arguments, global variables used by the function, or the block-level control protocols. The mode can be specified as one of the following:

  - ap_none: No protocol. The interface is a data port.
  - ap_stable: No protocol. The interface is a data port. The HLS tool assumes the data port is always stable after reset, which allows internal optimizations to remove unnecessary registers.
  - ap_memory: Implements array arguments as a standard RAM interface. If you use the RTL design in the Vivado IP integrator, the memory interface appears as discrete ports.
  - m_axi: Implements all ports as an AXI4 interface.Ect.

- **port=<name>:** Specifies the name of the function argument, function return, or global variable which the INTERFACE pragma applies to.
- **depth=<int>:** Specifies the maximum number of samples for the test bench to process. This setting indicates the maximum size of the FIFO needed in the verification adapter that the HLS tool creates for RTL co-simulation.

3.7.2 Application

The interface pragma is applied like as follows:

```
1    void Top(data_c *tb_in, data_c *tb_out, data_d *
   ↪ coeff, int N)
2    {
3        ...
4        #pragma HLS INTERFACE mode=ap_memory port=tb_in
   ↪ depth=(NUM_ROW_*NUM_COLUM_)*depth_factor
5        #pragma HLS INTERFACE mode=ap_memory port=tb_out
   ↪ depth=((NUM_ROW_*4)*NUM_COLUM_)*depth_factor
6        #pragma HLS INTERFACE mode=ap_memory port=coeff
   ↪ depth=NUM_COEFF_
7        ...
8
9    }
```

Figure 3.19: Using interface in Function

In summary, applying interface to configure and optimize interfaces in Vitis HLS can significantly enhance the performance of FPGA designs. By carefully selecting and configuring interface characteristics, including data transfer modes, data widths, and protocols, we can achieve improved communication efficiency, reduced data transfer latency, and enhanced resource utilization, thus optimizing overall performance. Additionally, this interface-level performance optimization also contributes to simplifying hardware designs, reducing power consumption, and positively impacting the performance of the entire system. Therefore, in FPGA design, appropriately configuring and optimizing interfaces is a crucial step in achieving high performance, efficiency, and reliability.

Chapter 4

Result Analysis

By applying the optimization techniques mentioned in the previous chapter, significant acceleration was achieved for the entire design. These optimization methods encompass dataflow optimization, pipeline optimization, loop unrolling, resource allocation, and interface configuration, among others. Ultimately, we achieved satisfactory results, ensuring that the throughput of the entire design can successfully match the throughput requirements of the upstream and downstream modules within the channel model.

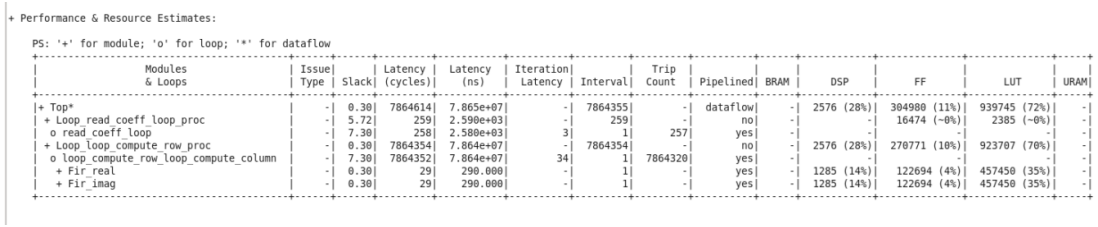
4.1 Synthesis Result

Upon the completion of synthesis, Vitis HLS generates a comprehensive synthesis report for the top-level function. This report provides detailed information about various aspects of the post-synthesis design, including resource utilization, delay analysis, and timing constraints. This synthesis report is invaluable for assessing the design's performance, resource requirements, and timing constraints, helping to confirm whether the performance objectives have been met and whether further optimization or adjustments are needed.

The generation of the synthesis report marks a significant stage in the optimization process, providing essential insights for design validation and further refinement. By continuously optimizing and analyzing the data in the synthesis report, we can ensure that the final hardware design operates with optimal performance on the FPGA, meeting the specific requirements of the application.

In following figure is reported the values relatives to the estimated performance. where:

- **Latency (Max):** The count of clock cycles necessary for the function to calculate all the output values.



+ Performance & Resource Estimates:

PS: '+' for module; 'o' for loop; '*' for dataflow

Modules & Loops	Issue Type	Slack	Latency (cycles)	Latency (ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	URAM
+ Top*	-	0.30	7864614	7.865e+07	-	7864355	-	dataflow	-	2576 (28%)	304980 (11%)	939745 (72%)	-
+ Loop_read_coeff_loop_proc	-	5.72	259	2.590e+03	-	259	-	no	-	-	16474 (~0%)	2385 (~0%)	-
o read_coeff_loop	-	7.30	258	2.580e+03	3	1	257	yes	-	-	-	-	-
+ Loop_loop_compute_row_proc	-	0.30	7864354	7.864e+07	-	7864354	-	no	-	2576 (28%)	270771 (10%)	923707 (70%)	-
o loop_compute_row_loop_compute_column	-	7.30	7864352	7.864e+07	34	1	7864320	yes	-	-	-	-	-
+ Fir_real	-	0.30	29	290.000	-	1	-	yes	-	1285 (14%)	122694 (4%)	457450 (35%)	-
+ Fir_imag	-	0.30	29	290.000	-	1	-	yes	-	1285 (14%)	122694 (4%)	457450 (35%)	-

Figure 4.1: Synthesis Report

- **Initiation interval (II):** The quantity of clock cycles that must elapse before the function becomes ready to accept fresh input data.
- **Loop iteration latency:** The count of clock cycles required to execute a single iteration of the loop.
- **Loop iteration interval:** The quantity of clock cycles preceding the initiation of data processing for the subsequent iteration of the loop.
- **Loop latency:** The count of cycles necessary to complete all iterations within the loop.
- **Resource Utilization:** The quantity of hardware resources needed to realize the design, taking into account the FPGA's available resources, encompassing lookup tables (LUTs), registers, block RAMs, and DSP blocks.

According to the report, the latency of each input data is: The usage of resource

Port	#Data	Latency(cycles)	Throughput
Input	1966080	7864614	4.001
Output	7864320	7864614	1.001

Table 4.1: The throughput of Input and Output ports

is:

DSP	2576(28%)
FF	384980(11%)
DSP	939745(72%)

Table 4.2: The usage of resource

From these figures, it is evident that the optimized design meets the required throughput and maintains a reasonable and efficient utilization of resources.

4.2 Co-Simulation

The purpose of Co-simulation in Vitis HLS is to perform functional verification, timing analysis, and performance evaluation. It achieves these objectives by comparing C/C++ models with hardware designs. The process of co-simulation involves preparing C/C++ models, creating a testbench, executing co-simulation, and comparing the outputs of C/C++ models with hardware designs. The results include reports for functional verification, timing analysis, and performance evaluation, which are used to confirm the correctness of the design, assess compliance with timing requirements, and evaluate performance against application needs. This makes co-simulation a powerful tool for verifying, debugging, and optimizing FPGA designs.

Finally, the design passes the co-simulation.

Bibliography

[1] N. A. Shah. «Optimization and Acceleration of 5G Link Layer Simulator».MS Thesis.» In: (2019) (cit. on p. 1).

[2] Mihai T. Lazarescu Nasir Ali Shah and Luciano Lavagno. «FPGA Acceleration of 3GPP Channel Model Emulator for 5G New Radio». In: (2022) (cit. on p. 1).

[3] *5G; NR; NR and NG-RAN Overall description(3GPP TS 38.300 version 17.0.0 Release 17).* URL: https://www.etsi.org/deliver/etsi_ts/138300_138399/138300/17.00.00_60/ts_138300v170000p.pdf. (cit. on p. 1).

[4] *5G; NR; Physical layer; General description (3GPP TS 38.201 version 17.0.0 Release 17).* URL: https://www.etsi.org/deliver/etsi_ts/138200_138299/138201/17.00.00_60/ts_138201v170000p.pdf. (cit. on p. 3).

[5] *5G; NR; Services provided by the physical layer (3GPP TS 38.202 version 17.3.0 Release 17).* URL: https://www.etsi.org/deliver/etsi_ts/138200_138299/138202/17.03.00_60/ts_138202v170300p.pdf. (cit. on p. 8).

[6] *Alveo U280 Data Center Accelerator Card.* URL: https://www.amd.com/en.html. (cit. on p. 9).

[7] *Programming an FPGA: An Introduction to How It Works.* URL: https://www.xilinx.com/products/silicon-devices/resources/programming-an-fpga-an-introduction-to-how-it-works.html (cit. on p. 10).

[8] *Vitis High-Level Synthesis User Guide.* URL: https://docs.xilinx.com/r/en-US/ug1399-vitis-hls. (cit. on p. 15).

[9] *Increase sample rate by integer factor-MATLAB upsample.* URL: //https://www.mathworks.com/help/signal/ref/upsample.html. (cit. on p. 19).

[10] *1-D digital filter-MATLAB filter.* URL: https://www.mathworks.com/support/search.html?q=filter&page=1. (cit. on p. 20).

[11] *1-D digital filter-MATLAB filter.* URL: https://www.mathworks.com/support/search.html?q=filter&page=1. (cit. on p. 21).

[12] *Vitis High-Level Synthesis User Guide (UG1399).* URL: https://docs.xilinx.com/r/2021.2-English/ug1399-vitis-hls. (cit. on p. 37).