Politecnico di Torino

ScuDo
Scuola di Dottorato ~ Doctoral School
WHAT YOU ARE, TAKES YOU FAR

Doctoral Dissertation
Doctoral Program in Electrical, Electronics and Communications Engineering
(35^{th}cycle)

Heterogeneous Acceleration for 5G New Radio Channel Modelling

Using FPGAs and GPUs

By

Nasir Ali Shah

******

Supervisor(s):
Prof. Luciano Lavagno, Supervisor
Eng. Roberto Quasso, Industrial Supervisor

Doctoral Examination Committee:
Prof. Roberto Passerone, Referee, University of Trento, Italy
Dr. Osama B. Tariq, Referee, Newcastle University, UK
Prof. Mohammad Mozumdar, California State University, USA
Prof. Mario R. Casu, Politecnico di Torino,Italy
Prof. Mihai T. Lazarescu, Politecnico di Torino, Italy

Politecnico di Torino
2023

Declaration

I hereby declare that, the contents and organization of this dissertation constitute my own original work and does not compromise in any way the rights of third parties, including those relating to the security of personal data.

Nasir Ali Shah
2023

* This dissertation is presented in partial fulfillment of the requirements for **Ph.D. degree** in the Graduate School of Politecnico di Torino (ScuDo).

I would like to dedicate this thesis to my loving parents and my beloved wife, whose unwavering support has been instrumental in my journey. I am forever grateful for your encouragement and sacrifices. Their belief in my abilities and relentless encouragement have propelled me forward. I am deeply grateful for their presence in my life, and this thesis stands as a testament to their unwavering dedication and invaluable contributions.

Acknowledgements

Completing my doctoral thesis fills me with a sense of accomplishment, and writing this acknowledgment serves as the final touch. Undertaking this doctoral journey has profoundly influenced both my research and personal growth. Thus, I would like to take a moment to acknowledge and express my gratitude to those who have provided unwavering support and assistance throughout this transformative period. Without the guidance of my supervisors, the invaluable contributions of my research group members, and the unwavering support of my family and friends, this work would not have reached its completion.

I would like to express my heartfelt gratitude and deep appreciation to my advisor Prof. Luciano Lavagno and Prof. Mihai Teodor Lazarescu for their invaluable assistance and humble guidance throughout the duration of my Ph.D. Their unwavering support, valuable insights, and approachable nature have greatly contributed to my academic and personal growth. I am truly grateful for their mentorship and the positive impact they have had on my research journey. I could not have imagined having better supervisors.

Further, I extend my sincere gratitude to my external advisors Eng. Scarpina Salvatore and Eng. Roberto Quasso for their assistance at every stage of the research project. I would like to acknowledge TIM for providing the necessary resources and financial support for this research.

I would like to express my sincere appreciation to my lab mates from the HLS group for their invaluable support and collaboration throughout my Ph.D. Their expertise and shared insights have greatly enriched my research experience. I am grateful for the fruitful discussions, the exchange of ideas, and the sense of community we have built together. Their contributions have played a significant role in the success of my thesis, and I am honored to have had the opportunity to work alongside such talented individuals.

Finally, I would like to express my deepest gratitude to my parents and my wife for their unwavering love, support, and sacrifices throughout my Ph.D. journey. Their constant encouragement, belief in my abilities, and understanding of the demanding nature of doctoral research have been the cornerstone of my success.

Abstract

The channel model is the most computationally demanding element in link-level simulations for multiple-input and multiple-output (MIMO)-based fifth-generation new radio (5G-NR) communication systems. Accurately modeling the wireless channel is crucial for developing and assessing wireless networks beyond 5G-NR. The use of realistic geometry-based channel models, such as the three-dimensional spatial channel model (3D-SCM), requires more computational resources for simulation. Channel emulation is employed to validate the functionality and performance of channel models during the network planning phase. General-purpose central processing unit (CPU)-based emulation platforms have limitations in accurately replicating propagation environments because they are either too simplified or have impractical execution time. Hardware accelerators based on specialized computing platforms such as FPGAs and GPUs can be employed to alleviate the load of complex computations and enhance the quality of results.

This study aims to tackle this matter by investigating diverse methodologies and optimization techniques for building an efficient hardware accelerator from a high-level specification. The process of developing applications for specialized architectures is intricate and requires thorough knowledge of hardware design languages and target architectures.

The first part of this study proposes an efficient re-configurable implementation of the 3rd Generation Partnership Project (3GPP) 3D-SCM for 5G-NR on Xilinx and Intel FPGA platforms using high-level synthesis (HLS)-based design flow. It explores the effect of various HLS optimization techniques on the total latency and hardware resource utilization on the target acceleration platforms. By using the proposed methodologies, the accelerated designs on Xilinx Alveo U280 and Intel Arria 10 FPGA achieved speedups of **65X** and **95X**, respectively, compared to the baseline CPU implementation. This speedup enhances to **173X** by optimizing the

design to utilize specialized resources present on Xilinx FPGA, such as UltraRAM (URAM) and High-Bandwidth Memory (HBM).

This study's second part focuses on accelerating the 3GPP channel model using GPU platforms. This study investigates different optimization techniques to exploit the parallelism and memory hierarchy of the GPU, specifically focusing on CUDA-based approaches. The experimental results demonstrate that the developed system achieves a significant speedup of approximately **240X** over CPU-based implementation. The GPU design exhibits a 33.3 % increase in single precision performance compared to the design accelerated on a datacenter-class FPGA. However, it also consumes 7.5 % more energy.

Contents

List of Figures

List of Tables

Acronyms

Abbreviation	Explanation
2D	Two-dimensional
2D-SCM	Two-dimensional spatial channel model
3D	Three-dimensional
3D-SCM	Three-dimensional spatial channel model
3D-UMa	3D urban macro
3GPP	3Rd Generation Partnership Project
4G	Fourth-generation
5G	Fifth-generation
5G-NR	Fifth-generation new radio
AOA	Azimuth angle of arrival
AOD	Azimuth angle of departure
ASIC	Application-specific integrated circuit
ASO	Application-specific optimizations
BRAM	Block RAM
BS	Base station
BSP	Board support package
CDFG	Control and data flow graph
CDL	Cluster delay line
CIR	Channel impulse response
CLB	Configurable logic block
CPU	General-purpose central processing unit
CU	Compute unit
CUDA	Compute Unified Device Architecture
D2D	Device-to-device
DDR	Double Data Rate

Abbreviation	Explanation
DP	Double-precision
DRAM	Dynamic RAM
DSP	Digital signal processor
FIR	Finite impulse response
FP	Floating-point
FPGA	Field programmable gate array
gNB	Next generation NodeB
GPU	Graphics processing unit
GSCM	Geometry-based stochastic channel model
HBM	High-Bandwidth Memory
HDL	Hardware Description Language
HLL	High-level language
HLS	High-level synthesis
II	Initiation interval
InH	Indoor-Office
IOT	Internet-of-things
LOS	Line-of-sight
LSP	Large-scale parameter
LTE	Long-term evolution
LUT	Look-up tables
MIMO	Multiple-input and multiple-output
MPC	Multipath component
MS	Mobile station
NFV	Network functions virtualization
NGSM	Non-geometrical stochastic model
NLOS	Non-line-of-sight
O2I	Outdoor to indoor
OOB	Out-of-the-box
OpenCL	Open Computing Language
PAS	Power angular spectrum
PE	Processing element
PSO	Platform-specific optimizations
RF	Radio frequency
RMa	Rural-Macrocell

Abbreviation	Explanation
RTL	Register-Transfer-Level
SCM	Spatial channel model
SIMD	Single instruction multiple data
SM	Streaming multiprocessor
SP	Single-precision
SRAM	Static RAM
SSP	Small-scale parameter
TDL	Tapped delay line
TDP	Thermal design profile
UE	User equipment
UMa	Urban-Macrocell
UMi	Urban-Microcell
URAM	UltraRAM
V2V	Vehicle-to-vehicle
WG	Work-group
WI	Work-item
ZOA	Zenith angle of arrival
ZOD	Zenith angle of departure

Chapter 1

Introduction

The emergence of always-on systems, internet-of-things (IOT), self-driving vehicles, and other digital systems, has led to the development of a novel networking infrastructure capable of accommodating the demands of these devices. The available spectrum is becoming increasingly congested because of the massive increase in the number of these devices. Consumers may experience poor service quality, unreliable connections, reduced speeds, or even complete outages in highly congested regions or during peak usage hours. Long-term evolution (LTE) system which was an incremental improvement to fourth-generation (4G) mobile network has already reached its maturity. The LTE technology has undergone a subsequent improvement to LTE-Advanced, which satisfies the established standards for 4G networks. The fifth-generation (5G) standard is currently forthcoming and presents numerous modifications in comparison to its predecessor, 4G. A great deal of focus has been devoted to the physical features of radio communication, including frequency and cell positioning, hence the substitute designation of the standard as fifth-generation new radio (5G-NR) [1]. Mobile broadband is the main motivating factor behind the development of 5G cellular communication network since the enhancement of user data rates has long been a top goal in the research and development of mobile communications systems. Broadband human-oriented communications, time-sensitive applications with ultra-low latency, and massive connectivity for the IOT are just some of the use cases that 5G-NR hopes to address [2].

Before and during the physical deployment phase, new cellular technology must first be tested in a simulation environment. The term "simulation" refers to the

practice of creating a computerized model of a physical or hypothetical system to test its functionality. Mathematical channel models are created to obtain a greater understanding of the factors that impact the transmission of wireless signals through a communication medium [3]. These models also facilitate the extraction and development of channel state information (CSI) from channel impulse response (CIR) such as fading, Doppler, delay, and azimuth [4].

The simulation of channel models has emerged as a vital tool in the mobile network planning process. Accurately modeling the channel is a crucial aspect in the design and evaluation of wireless networks beyond 5G-NR. In order to achieve a precise depiction of propagation conditions, a multitude of radio frequency parameters must be meticulously calibrated. It is necessary to recalculate parameters even after deployment in the event of any changes to the network configuration or propagation conditions, such as alterations to the number or positioning of antennas.

1.1 Wireless Network Planning and Channel Simulation

The process of network planning involves the conceptualization and design of wireless networks prior to their actual physical implementation. The process comprises multiple phases, including network dimensioning, pre-planning, detailed planning, verification, and optimization. The primary objective of the network planning process is to attain the utmost network efficiency while minimizing expenses.

Optimal performance at minimal cost is a difficulty faced by the majority of network operators due to the rising demand for high data-rate network traffic and the growing cost of operating communication networks. For best network performance, planners today often use computer-aided network tools to simplify the tedious task of network planning. Locating the most suitable spots for setting up network nodes is a crucial part of every network design process. This analysis can be performed depending on the network coverage requirement and the network capacity requirement by means of a channel information map. Network planning evaluations can be broken down into three tiers: regional, cluster, and link level [5].

Regional level This includes a high-level assessment to estimate investment cost and population coverage in a given region.

Cluster level This stage consists of immediate-fidelity simulations and estimates bit-rate targets, frequency channels, etc.

Link level This stage of network planning is used to optimize radio frequency parameters such as shadowing, path-losses antenna design, etc.

1.1.1 Wireless channel simulation

When it comes to channel planning, the wireless channel model is the most crucial component. It should be able to model a wide range of frequency bands, a substantial number of design parameters, and a variety of different alternatives for deployment. It is extremely vital to model and simulate the impact that a physical channel has on a transmitted signal due to environment. Because of the significant differences in the propagation medium across location, time, and frequency, it is of even greater significance for wireless communication systems.

Link-level simulation, analysis, and measurements are crucial in creating novel wireless communication systems. Each of these methods has its own set of benefits, thus it is important to use all three of them. Measurements serve as the foundation for channel modeling and give the definitive performance standard for any transceiver architecture. As a result, measurements are required in order to conduct analysis and simulations. However, carrying out measures is not only very expensive but also very time intensive, and it is difficult to adjust them to certain communication scenarios. Analytical models have the capacity to show correlations between the important parameters of a system, which is one of the many benefits of these studies. However, to make analytical models tractable, it is often necessary to apply several restrictive assumptions and simplifications. This reduces the value of analytical conclusions when applied to more realistic settings.

Channel simulators are the tools to evaluate the efficiency and performance of a communication system in a controlled environment. The outcomes of the simulations are employed as input by planning tools, which are adopted by all network operators, to ascertain the network infrastructure, including network nodes, and its configuration. The creation of a theoretical model enables the evaluation of the operational efficiency of real devices. The vast majority of today's network service providers construct these models through the use of computer simulations, as

well as carrying out laboratory and real-world assessments on commercial gadgets to verify the attainment of anticipated outcomes.

Channel simulators are an extremely efficient way for carrying out performance evaluations on the various components of a wireless communication system. The evaluation of compliance with communication standards is crucial in the testing of novel communication algorithms. This enables the determination of how effective they are in meeting the essential communication standard requirements. One of the assessments involves the assessment of system performance in the presence of a communication channel. These simulations are considered to be a more dependable and authentic means of conducting experiments due to their ability to replicate real-life scenarios. However, they are often characterized by their complex nature, high costs, and time-consuming processes, which may render them impractical for deploying a large number of nodes. It is feasible to represent the wireless channel as an finite impulse response (FIR) filter that possesses random coefficients that vary over time. Increasing the number of coefficients can lead to a more effective approach in addressing real-world scenarios [6]. However, this may result in an increase in computational complexity.

Historically, network planners prioritized the implementation of simplified channel models designed for execution on general-purpose central processing unit (CPU) platforms, resulting in significantly longer execution times. An alternative method that has been previously employed to enhance the quality of results and energy efficiency involves the use of emulation platforms based on hardware. Application-specific integrated circuits (ASICs) have been predominantly utilized due to their ability to facilitate high throughput and low power consumption. Nevertheless, this category of circuits lacks flexibility and requires describing the functionality of the application at the Register-Transfer-Level (RTL) through a Hardware Description Language (HDL) like VHDL or Verilog, which can be a challenging undertaking.

Network planning tools must have a high degree of flexibility, as they must be able to adjust their parameters to accommodate various propagation scenario simulations. Hardware acceleration is a technology that shows potential in addressing these requirements [7]. This technology facilitates the rapid creation of a channel model emulator from a high-level description, typically in C/C++/CUDA, and implementing it on a reconfigurable hardware device. The software-controlled emulator can be conveniently configured to simulate various propagation scenarios.

Field programmable gate arrays (FPGAs) are one of the candidate technology which is frequently used as hardware acceleration platform and provide a compelling trade-off between power consumption and performance. FPGAs present a compelling trade-off between power consumption and performance, and facilitate the creation of customized computing architecture by means of a configuration file (also called bitstream). These accelerators can be adopted to new applications by using a different configuration file designed with the required specifications. FPGA-based emulators provides, in theory, a suitable compromise between adaptability and processing power. However, the platform's programmability becomes restricted because FPGAs require an HDL description as part of the conventional design flow.

Another candidate technology for hardware acceleration is graphics processing unit (GPU). With the advancement of technology, the GPU has undergone significant improvements, rendering it a powerful, programmable, and extensively parallel unit [8]. Consequently, it has emerged as a crucial constituent in High performance computing (HPC) systems and is currently the most prominent form of hardware accelerator. Due to their significant computational throughput and parallelism, GPUs have been deemed appropriate for general-purpose computing. Researchers have been using GPUs for a wide variety of scientific applications [9]. Efficient management of GPU resources can be achieved through high-level programming languages (such as Compute Unified Device Architecture (CUDA) from NVIDIA) based on the underlying computing architectures, resulting in improved performance.

The proposed solution aims to tackle the aforementioned problem by introducing a design methodology that utilizes hardware acceleration platforms such as FPGAs and GPUs that are designed to compute the time-varying coefficients of fading channel simulators. By utilizing these acceleration platforms and the suggested approaches, individuals without specialized knowledge in specialized computing architectures can enhance the pace of their simulation advancements in comparison to traditional software. The results of the implementation demonstrate that the suggested methodology facilitates the efficient creation of communication channels, while concurrently decreasing the processing duration. In light of the dispersed nature of the channel, the hardware emulation technologies have higher advantages in terms of performance and power consumption over the CPU-based implementation.

This doctoral thesis seeks to enhance the programmability of heterogeneous FPGA and GPU-based emulation platforms by replacing HDL-based flow with

higher-level ones. Use of high-level synthesis (HLS)high-level language (HLL) methodologies and associated tools to produce specialized architectures enables rapid prototyping [10].

1.2 Thesis Outline

The present doctoral dissertation is structured into two primary sections. The first part is the background, presented in chapter 2 and chapter 3. This section presents an extensive review of the 5G-NR cellular technology, channel modelling concepts and methodologies, standard channel models, hardware acceleration platforms, HLS flow and optimizations, and other relevant concepts. The subsequent section, encompassing chapter 4 to chapter 6, relates to the various contributions to research. The work covers extensive details relating to our proposal of a reconfigurable channel model on FPGA and GPU acceleration platforms compiled from HLL specifications.

Chapter 2: provides the background information on 5G-NR, key benefits associated with the new cellular technology, and a short survey of channel modeling methodologies.
Chapter 3: provides background on three-dimensional (3D) channel modelling and discusses various steps and procedures in generation of 3^{Rd} Generation Partnership Project (3GPP) channel model.
Chapter 4: discusses design flow for FPGA acceleration platforms. It presents a detailed discussion of HLS tools and the associated benefits for modeling hardware designs using a high-level specification. It introduces the two HLS tools from Xilinx and Intel for FPGA-based acceleration. It also discusses a variety of optimizations and strategies that can improve the quality of result for HLS-based designs.
Chapter 5: is dedicated to the design space exploration of Chapter 5 discusses the implementation and design space exploration of the 3GPP 3D channel model on Xilinx and Intel FPGA platforms using HLS tools. HLS-based optimizations and their respective on the implementation are presented. This chapter also analyzes the effort required to accelerate applications on FPGA platforms from different vendors.
Chapter 6: presents the optimization techniques and methodologies for accelerating 3GPP channel model on GPU using NVIDIA CUDA tools. It provides a brief analysis of the achieved performance for GPU compared to that on the FPGA platforms.

Chapter 7: Concludes the thesis and suggests potential future directions

Chapter 2

Fifth-generation Mobile Network and Channel Modeling

The 5G mobile communication network also known as 5G-NR offers significant features, including but not limited to low latency, high data rates, and the ability to accommodate a high density of devices and base stations. The anticipated impact of this new cellular network technology is significant and poised to benefit various sectors, such as corporate networks, public networks, and infrastructure. Despite the challenging economic conditions and geopolitical instabilities, service providers are persistently implementing 5G technology. Ericsson's mobility report [11] projects that there will be a total of 5 billion 5G subscriptions worldwide, representing 54 % of all mobile subscriptions, by the end of 2028.

The adoption rate of 5G subscriptions has surpassed that of 4G, which was launched in 2009. It is anticipated that the new mobile communication technology will attain 1 billion subscriptions earlier than 4G by two years, owing to the timely availability of devices from multiple vendors, and a more rapid decline in prices compared to previous generations of the technologies. Figure 2.1 presents a quantitative analysis of the adoption rates of 5G and 4G subscriptions during the initial years of their respective deployments. Some of the key benefits associated with the new cellular technology are listed as follows.

- The implementation of 5G mobile communication technology is expected to serve as a crucial factor in facilitating the efforts of government entities and policymakers to convert their urban areas into smart cities. This will enable res-

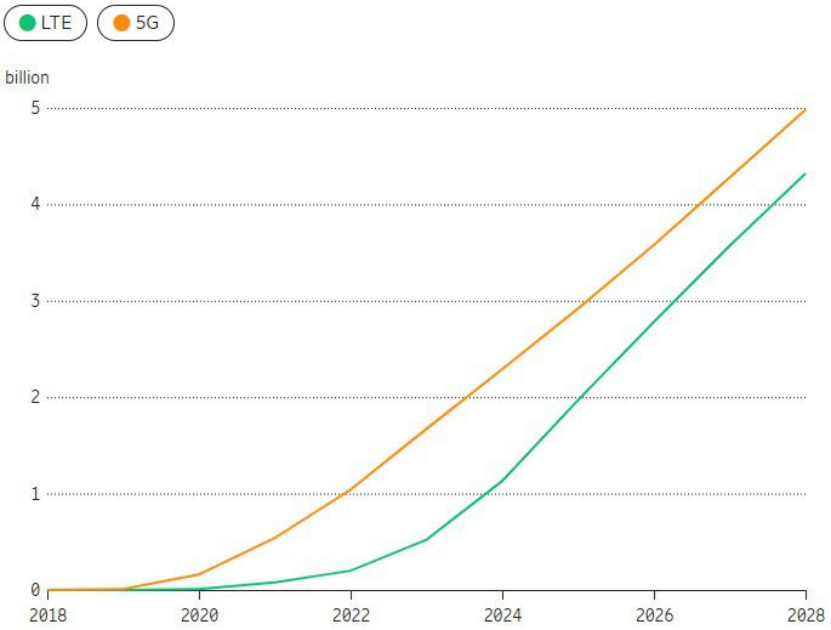


Fig. 2.1 5G and 4G subscription uptake in the first years of deployment [11]

idents and communities to effectively leverage the socioeconomic advantages that are associated with a sophisticated, data-driven digital economy.

- The advent of 5G presents a promising prospect for wireless service providers to expand their offerings beyond mere connectivity services and, instead, create comprehensive solutions and services that cater to the needs of both consumers and various industries across diverse sectors.
- The new wireless communication technology is anticipated to significantly enhance data rates and decrease latency to less than 1 ms, rendering it appropriate for time-sensitive mission-critical services [12]. The high bandwidth available on 5G networks enables the provision of a variety of high-speed broadband services, thereby presenting a viable alternative to last-mile access technologies such as Fiber to the Home (FTTH) or copper connections.
- The new cellular network technology is expected to enhance the overall user experience by providing novel applications and services through faster data transfer rates, as well as substantially improved performance and dependability.
- It adds support for the implementation of virtual networks, also known as network slicing, which is made possible by the 5G technology. This feature enables the creation of subnets that offer connectivity tailored to meet specific requirements. The subnetworks can assign specific characteristics to a segment

of the network, rendering it programmable and enabling the prioritization of connections. This prioritization may involve a higher priority of emergency services over other users, through the application of varying latencies or the elevation of their connection status, thereby safeguarding them from potential disruptions caused by mobile network overloads.

The ITU roadmap for 5G communications systems groups the use case scenarios into three major categories [13];

Enhanced Mobile Broadband (eMBB) The primary application of 5G is considered to be the advancement of long-term evolution (LTE) mobile broadband services, with the aim of delivering improved connectivity, increased throughput, and greater capacity [14]. This particular use case is closely associated with the steady rise in mobile data traffic, which is primarily driven by the growing number of data-intensive devices and multimedia applications. Therefore, eMBB encompasses various applications aimed at enhancing the quality of communication from a human perspective, such as optimizing hotspots to accommodate a larger number of users and expanding coverage for scenarios involving greater mobility.

Ultra-reliable low latency communications (URLLCs) is a communication service designed to effectively transmit packets with strict demands, particularly with regard to availability, latency, and reliability [15]. It promises to facilitate the provision of nascent applications and services. Exemplary services comprise wireless control and automation within industrial factory settings, inter-vehicular communications to enhance safety and efficacy, and the tactile internet. The effective support of verticals is crucial for the success of 5G, as it has the potential to introduce new business opportunities to the telecommunications industry as a whole.

Massive Machine–Type Communications (mMTCs) The purpose of this usage scenario is to facilitate the collection of an enormous number of small data packets from numerous devices in a concurrent manner [16]. The mMTC technology facilitates the deployment of applications that rely on IOT sensors. This enables the utilization of data to optimize energy consumption, enhance work productivity, and enhance the quality of life. This particular use case involves a range of applications including but not limited to IoT, asset tracking, energy monitoring, smart agriculture, smart cities, smart home, and remote monitoring.

Some of the key features promised by the new cellular communication technology are peak data rate, low latency, greater network energy efficiency, improved

spectrum efficiency, increased area traffic capacity, support for higher mobility, higher connection density, improved reliability, and more security and privacy.

2.1 Key Enabling Technologies for 5G

Future 5G networks will need revolutionary breakthrough technology to suit this wide variety of needs. As an added bonus, 5G technology is anticipated to be more welcoming to older technologies than its predecessors, with Wi-Fi, UMTS, and LTE all likely to play significant roles in the 5G ecosystem. A major paradigm shift in the planning of wireless communication systems is needed for this to happen.

2.1.1 Massive-MIMO systems

Massive-multiple-input and multiple-output (MIMO) also known as large-scale antenna systems is a wireless communication technique that employs many transmit and receive antennas to multiplex messages for various devices. The objective is to direct the emitted energy in the desired directions while reducing both inter and intra-cellular interference [17]. In contrast to traditional MIMO systems, massive-MIMO systems typically feature a large number of transmitting and receiving antennas, such as 32 or 64. The utilization of large antenna arrays provides a substantial degree of freedom and versatility in antenna usage, including multi-user beamforming and interference coordination [18].

In the beginning, MIMO technology was conceptualized in two modes, namely single-user MIMO (SU-MIMO) and multi-user MIMO (MU-MIMO). In a given time slot, a conventional SU-MIMO base station (BS) transmits data to a single user-terminal using multiple antennas on both ends. However, BS can transmit data to multiple users simultaneously and the user-terminal is not necessarily required to be equipped with multiple antennas. This allows for better efficiency of the network and reduced equipment cost for user terminals. However, the scalability of MU-MIMO is limited in its initial design, which aimed to target nearly equal numbers of base station antennas and user mobile stations. This is due to the necessity of CSI being available at both ends. Massive MIMO employs channel information that is obtained through direct measurement, as opposed to being assumed. This allows the network

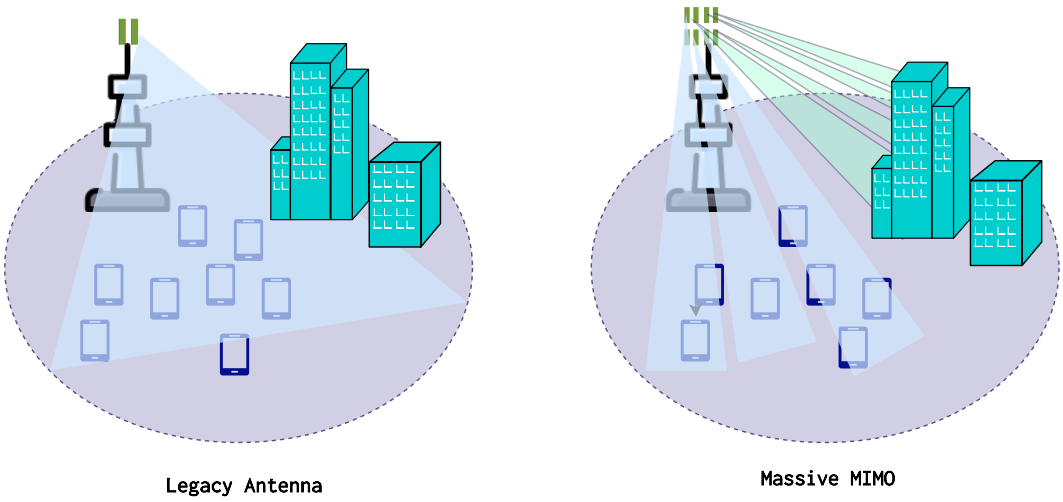


Fig. 2.2 Comparison of legacy antenna and massive-MIMO coverage[19]

to be scalable, as it can effectively utilize any number of BS antennas without any increase in array tolerances. The concept of massive-MIMO is depicted in fig. 2.2.

The idea behind massive-MIMO is to provide the BS with a very high number of antennas so that it may simultaneously serve a relatively smaller number of user terminals. This is accomplished through time division duplexing and calculating only the up-link CSI which is achieved through spatial multiplexing. Massive-MIMO technology supports beamforming, which refers to the capability of directing radio signals in specific directions. It is recommended that massive MIMO systems be constructed using low-power components, typically in the milliwatt range, and at a low cost. This is due to the fact that the necessary transmit power per antenna diminishes as the number of antennas increases. The implementation of beamforming techniques can effectively mitigate the negative impact of fading phenomena, leading to a reduction in latency [20].

Massive-MIMO technology has the potential to significantly enhance capacity by a factor of 10 or more, while concurrently improving radiated energy efficiency by approximately 100 times. The augmentation in capacity is attributed to the implementation of intensive spatial multiplexing techniques in the massive MIMO system. The underlying principle that enables a significant enhancement in energy efficiency is the ability to concentrate energy into highly localized regions in space through the utilization of a multitude of antennas [21].

2.1.2 Network slicing

The implementation of network slicing is considered a highly innovative aspect of the 5G-NR cellular network. The notion of network slicing involves partitioning the network into distinct logical networks that are isolated from one another, which is a crucial aspect of the implementation of the 5G communication technology. The implementation of network slicing enables network providers to establish a tailored virtual network that caters to specific use cases [22]. The customization of each slice to suit specific use cases is facilitated by the virtualized nature of the network, which is less reliant on hardware than previous iterations of cellular technology.

The 3GPP has established a standardized network element, referred to as Dedicated Core (DECOR), which involves the Core Network (CN). The deployment of multiple CNs over a single physical network infrastructure enables mobile network carriers (MNCs) to provide varying levels of flexibility and resource sharing to diverse service consumers. The concept of network slicing surpasses the capabilities of DECOR by granting the MNC complete autonomy in regulating various communication services, each of which has distinct performance prerequisites. Figure 2.3 depicts the concepts of network slicing for various use cases in 5G-NR.

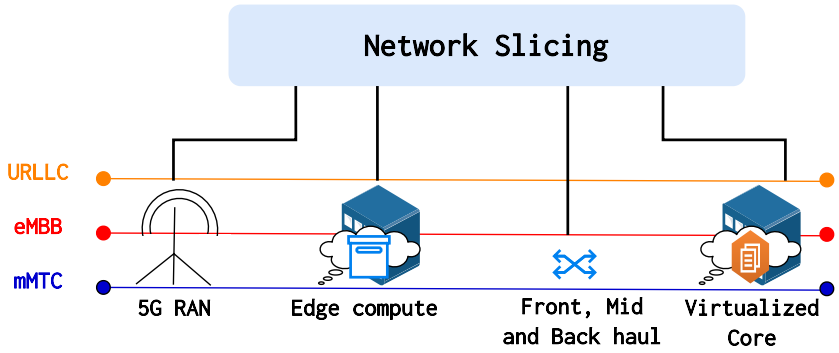


Fig. 2.3 Network slicing in 5G-NR

The feasibility of this objective can be achieved by tools such as network functions virtualization (NFV) and software defined network (SDN). Through the virtualization of network functions, the former enables the MNCs to offer a modular logical architecture and a flexible placement of network functions within its infrastructure. The utilization of the latter enables the MNCs to streamline the forwarding

functions and implement a more sophisticated segregation of control and user plane functionalities.

2.1.3 Network functions virtualization

Deploying a new network service to large operator networks sometimes involves deploying more hardware. Special-purpose equipment is costly, space-consuming, and energy-intensive. Managing, upgrading, and deploying a big network using hardware-based network devices from numerous suppliers is difficult. Thus, adding hardware considerably raises network capital and operational costs [23]. Network operators want to offer new services to increase revenue, yet hardware-based network infrastructures are too expensive and complicated. Software on general-purpose computers can replace special-purpose hardware. The software provides advantages over dedicated hardware. It's flexible and affordable.

To address cost and complexity, the software might integrate network functionalities. The virtualized network function framework (VNF) supports software-based network functions on cheap commodity servers, such as virtual network functions [24]. The concept of "virtualization" has been employed across multiple fields of computer science for many years now. These include the virtualization of servers, disks, and applications. In essence, virtualization serves to create a level of abstraction between a user and a computing resource, thereby concealing the underlying physical attributes of said resource. Virtualization is commonly employed to attain goals such as streamlined resource utilization and simplified management.

The objective of NFV is to revolutionize the network architecture of network operators by means of conventional IT virtualization technology to merge various network equipment categories onto high-volume servers, switches, and storage that comply with industry standards. The vast majority of telecommunication operators are interested in the segregation of network functions from specialized devices and their conversion into software that can be installed on readily available, standard hardware. These components can be placed in data centers, network nodes, and end-user terminals. The process includes the deployment of software-based network functions that are compatible with a variety of server hardware commonly used in the industry. These functions can be relocated or established at different network

locations as needed, without requiring the installation of additional equipment. Some key benefits associated with NFV are;

- The implementation of NFV can lead to a reduction in costs, a decrease in the time required for product updates or releases, and an enhancement in the scalability and resource allocation for applications and services.
- Organizations have the flexibility to avoid being constrained by proprietary, inflexible devices that require on-site visits and substantial resources to implement and customize.
- Virtualization with data center infrastructure allows more to be done with less, making it more efficient. With an improved data center footprint, electricity, and cooling power consumption can be reduced. This is possible because a single server is able to run many virtual network functions, and fewer servers are needed to complete the same task. Software updates can replace truck rolls when network demand changes. Physical network and data center updates are infrequent [25].
- A network that has undergone NFV possesses the capability to promptly and effortlessly adapt to variations in resource requisites in response to fluctuations in incoming traffic to the data center.

2.1.4 Vehicle-to-everything communication

The adoption of advanced traffic information systems, autonomous vehicles, and dependable safety services has facilitated the progress of technology in the domain of vehicles with enhanced features such as low latency, high data rate, and reliability. This technology offers an extensive transmission range and minimal end-to-end latency, making it ideal for cellular networks and includes various vehicular communication scenarios such as vehicle-to-infrastructure, vehicle-to-vehicle, and vehicle-to-pedestrian communication [26].

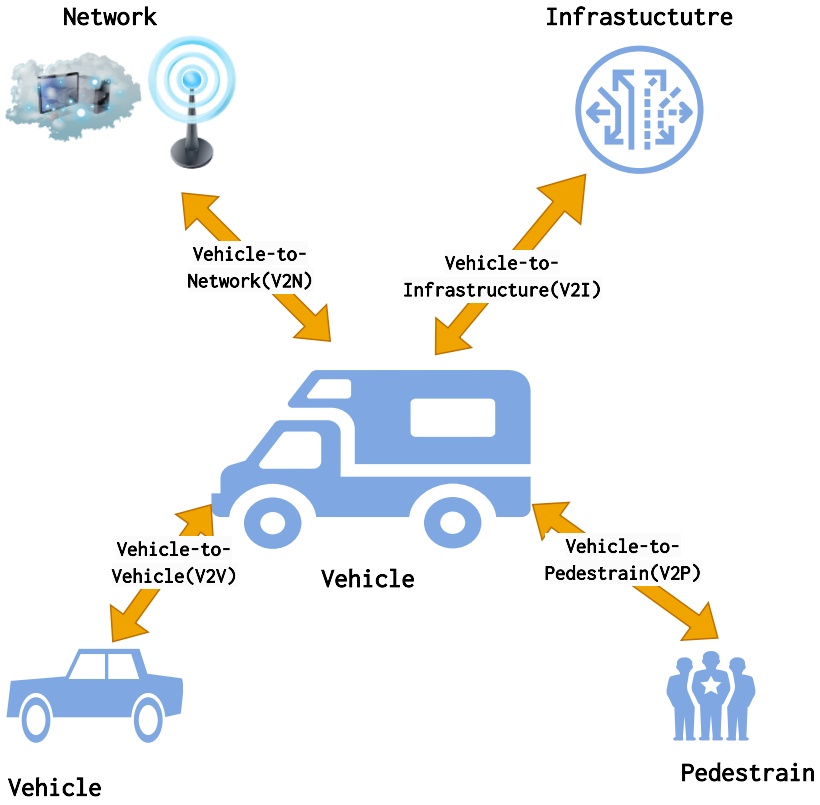


Fig. 2.4 Vehicle-to-everything communication [27]

2.1.5 Device–to–Device communications

Device–to–device communication (D2D) is a pioneering technology in the 5G cellular network that facilitates the establishment of a direct link between devices. D2D communication refers to the transmission of data between two devices in a point-to-point radio technology, without the involvement of a base station. This type of communication is commonly observed in cellular networks that operate on either the in-band or out-band spectrum. This technology is particularly useful during periods of network congestion or at cell boundaries, as it creates an ad hoc mesh network where intermediate devices can serve as relays for other devices. It provides a diverse range of services, that includes safety, traffic offloading, expansion of cellular coverage, reduced battery consumption, dependable communication, and proximity services based on location [28]. Additional advantages of D2D data transmission include its ability to effectively alleviate network congestion and enhance frequency

and space utilization, thereby enabling direct communication between nearby devices without necessitating the use of network infrastructure.

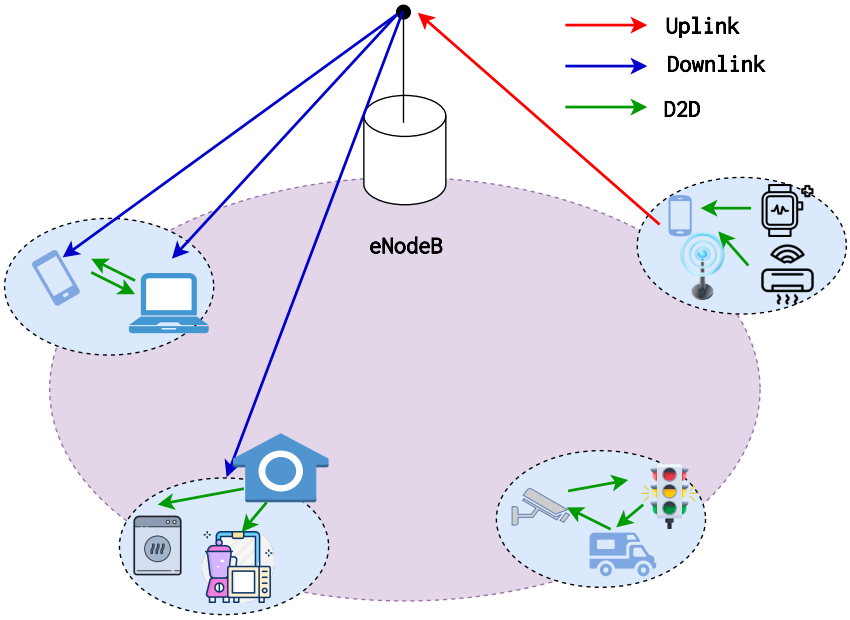


Fig. 2.5 D2D communication [29]

2.2 Wireless Channel Modeling Methodologies

In recent years, several MIMO channel models have been documented, with many of them being derived from empirical measurements. One possible approach to distinguish between the various models is by examining the specific type of channel under consideration, such as narrowband (characterized by flat fading) versus wideband (frequency selectivity) models, as well as time-varying versus time-invariant models, among others [30]. The spatial structure of flat fading MIMO channels provides a complete characterization. Channels with frequency-selectivity involve further modeling of the characteristics of the multipath channel. In the case of time-varying channels, it is necessary to incorporate a model that accounts for the evolution of the channel over time based on specific Doppler characteristics. In addition, it is imperative that the models are straightforward to facilitate their implementation and ensure efficient computational performance [31]. Balancing simplicity and accuracy in wireless channel modeling can be a challenging task.

Channel models reported in the literature can broadly be grouped into the following categories [30];

1. Analytical models
   (a) Propagation models
   (b) Correlation-based models
2. Physical models
   (a) Deterministic channel models
   (b) Stochastic channel models
   (c) Geometry-based stochastic models

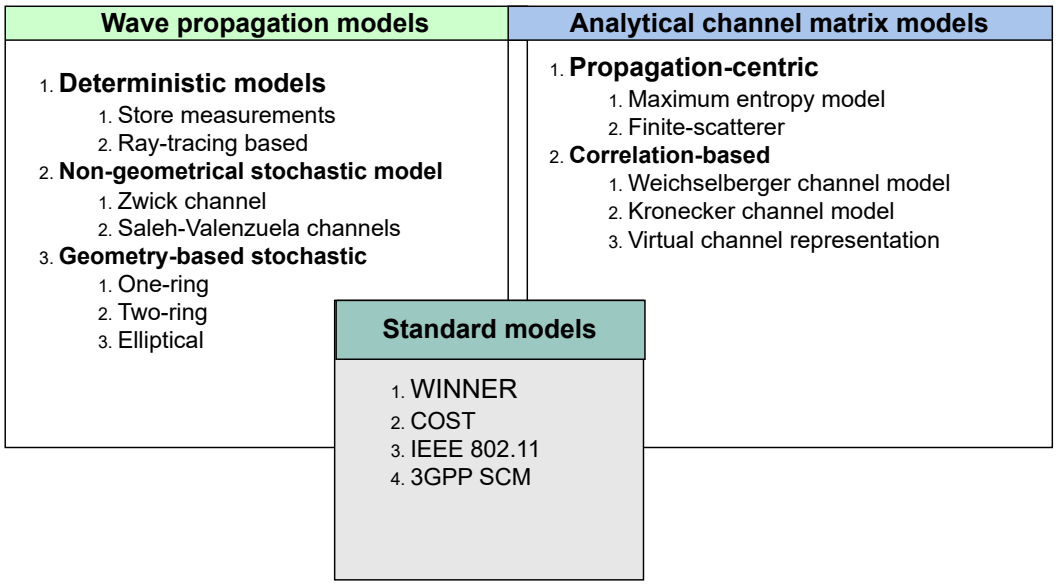


Fig. 2.6 MIMO channel modelling methodologies [32]

2.2.1 Analytical channel models

Analytical models typically represent the impulse response of the radio channel between transmitters and receivers on an individual level but do not offer sufficient insight into wave propagation. Analytical modeling focus on spatial channel coefficient correlation rather than the physical features of the scatterers in the environment [33]. *Propagation-based* and *correlation-based* models are the two sub-categories of analytical models. Maximum entropy model [34] and finite scatterer model [35] are examples of propagation-based models. Kronecker model [36–38] and Weichselberger channel model [39] are examples of correlation-based channel models that

characterize the channel statistically from the correlation between MIMO matrix elements.

2.2.2 Physical channel models

In wireless communication channels, a signal propagates from the transmitter to the receiver along specific geometric pathways and is susceptible to a range of physical phenomena, such as reflection, and scattering [40]. Scattering refers to the phenomenon in which a radio frequency (RF) wave interacts with an object and subsequently disperses into a number of waves. The electromagnetic properties of the environment are employed in wave propagation-based physical models to characterize the channel between the transmitter and receiver. These models consider the radio-wave parameters such as the direction of arrival and departure and delay of MPCs to compute the impulse response of the channel.

Deterministic channel models

These models replicate the channel's transfer characteristics and radio coverage for a specific environment using a well-defined system setup. Deterministic models are valuable tools for channel modeling, as they enable visualization of the propagation mechanism [41]. Nevertheless, they exclusively provide a representation of the specific environment under consideration, and it is often necessary to conduct multiple iterations utilizing various environments [42]. Ray-tracing [43] and stored measurement-data-based models [44] are some examples of deterministic models.

Geometry-based stochastic channel model

In this type of model, scatterers are observed to be stochastically distributed at both ends of the link [45]. GSCMs are constructed by randomly deploying (diffuse or discrete) scatterers while assigning them (scattering) attributes according to specific statistical distributions [46]. The scatterers' signal contributions are determined through simplified ray tracing and then summed at the receiver. The approach not only provides delay and Doppler spectra, but also enables modeling MIMO properties of the channel. GSCM models are flexible in changing antenna influence and environment and are faster when simulating single or double scattering compared

to deterministic ray tracing [47]. The structure of the scattering region provides insights into the tested scenario. For an accurate simulation of the given propagation scenario, it is essential to account for the total number of scatterers in the scattering region and to organize them into clusters. One-ring model [48] and elliptical-ring model [49] are examples of GSCM.

Non-geometrical stochastic model

This category of models solely relies on statistical methods to establish communication paths between the transmitter and receiver, without taking into account the physical geometry of the surrounding environment. Zwick model [50] and Saleh-Valenzuela model [51] are examples of non-geometrical stochastic models. In Zwick model, each MPC is treated individually, while the Saleh-Valenzuela model uses clusters of MPCs.

Table 2.1 Physical vs. Analytical channel models

Parameter	Physical Model	Analytical Model
Approach	Emulates wave propagation and electromagnetic effects	Mathematical simplification using statistics
Accuracy	Very high, closely matches real-world channel	Approximate, depends on formulation accuracy
Complexity	Extremely high, computes detailed physics	Lower complexity, abstractions used
Flexibility	Scenario specific parameters	Can adapt model formulation for scenarios
Simulation speed	Very slow due to intensive computations	Much faster due to simplified model
Calibration	Requires extensive measurements for validation	Validate by fitting model statistics to real-world
Bandwidth	Wideband, captures frequency selective fading	Can be wideband or narrowband
Time-variation	Can model dynamic channels and mobility	Typically static but some support time-variation
Antennas	Accurate modeling of antenna patterns	Abstracted using antenna correlations
Implementation	Require HPC systems, GPU clusters	Can be implemented on standard computers
Use Cases	Final validation, field trials	System-level simulation, parametric analysis

The deterministic models rely on a digital map and leverage electromagnetic wave propagation theory to predict various channel parameters. Nevertheless, the precision of digital maps significantly impacts the effectiveness of such models, although the application of electromagnetic wave propagation theory requires significant processing resources. Parametric channel models for NGSM are obtained by the application of statistical information processing techniques. Despite the fact that the generated models include simplistic structures, they are unable to adequately capture the fundamental characteristics of true propagation environments. In conclusion,

the imitation of stochastically distributed scatterers for multipath fading in GSCM is achieved by utilizing available geometry information. In recent decades, GSCM have gained significant popularity as a modeling approach due to their ability to balance computational complexity with model correctness. Consequently, they have been extensively utilized in the development of 3GPP channel models.

2.2.3 Standard channel models

While developing new radio-communication systems, standardized channel models play a very crucial role in assessing key performance parameters such as bit rate, multiple-access, and signal processing. The establishment of standardized channel models helps a more easy understanding of natural laws and can be helpful in unifying the results of extensive field measurements from scientific research, academia, and industry [52]. Several channel models have been developed by different standardization bodies groups, such as 3GPP [2], WINNER II[53], COST 2100 [54], METIS [55], ITU-R [56], MiWEBA [57] and NYUSIM [58]. These channel models share many similarities and can be grouped into two main categories:

- 3GPP/ITU-based channel models for frequencies below 6 GHz, with modifications to accommodate up to 100 GHz
- NYUSIM [59] based channel models for frequencies ranging from 0.5 GHz to 100 GHz that provide new features and enhancements, such as spatial consistency, mobility, and spherical wave propagation.

COST

The European Cooperation in Science and Technology (COST) has formulated various models, encompassing the directional characteristics of wireless channels, that apply to simulations of MIMO and smart antennas [60]. COST 2100 [54] is a GSCM for frequency bands below 6 GHz was based on the pre-existing COST 259 [61] and COST 273 channel models [62]. Cluster power, delays, and angles in the COST 2100 model are drawn from fixed geometry locations. This model suffers from a limited frequency range and lack of support for scenarios requiring dual mobility, such as device-to-device and vehicular-to-vehicular communication.

METIS

These channel models employ a hybrid approach, combining a map-based model and a stochastic model, to achieve adaptable and scalable channel modeling for mobile and wireless communications [55]. The map-based model was developed through the use of a ray tracing methodology, a simplified 3D geometric representation of a propagation environment, and the addition of random shadowing objects. These models support various propagation scenarios such as diffused scattering, blocking, specular reflection, and diffraction.

IEEE 802.11ay

IEEE 802.11ay is an improved standard for wireless networks that achieves high throughput and power efficiency, building upon the IEEE 802.11ad standard [63]. The wireless transceiver in this design utilizes directional antenna beams to function in the 60 GHz mmWave band [64].

Table 2.2 Comparison of 5G Channel Models

Model	**Scope**	**Frequency bands**	**Scenario Focus**	**Applicability**
3GPP [2]	Cellular networks, standards development	Sub-6 GHz and mmWave	Urban, suburban, rural	Cellular system design and standardization, wireless network planning
COST2100 [54]	Urban and indoor wireless environments modeling	Sub-6 GHz	Urban, indoor	Modeling urban and indoor wireless environments, performance analysis
MIWEBA [57]	Millimeter-wave mobile broadband access	Millimeter-wave	Millimeter-wave (mmWave)	Enhancing mobile broadband access with mmWave, 5G and beyond research
QuaDRiGa [65]	Various wireless communication scenarios	Various frequency	Various (outdoor, indoor, etc.)	Simulating channel conditions in diverse wireless scenarios, research
METIS [55]	Research and development for 5G and beyond	Various frequency	Diverse (urban, rural, etc.)	Shaping the 5G and beyond-5G communication landscape, system design
IMT-2020 [56]	Millimeter-wave-based mobile radio access	Millimeter-wave	Millimeter-wave (mmWave)	Research and development related to mmWave communication, 5G and beyond
NYUSIM [59]	Highly configurable channel simulator	Sub-6 GHz and mmWave	Configurable for various	Simulating channel conditions and system performance, research

2.2.4 3GPP 3D channel model for 5G-NR

The 3GPP has developed a standardized map-based hybrid channel model that combines the ray-tracing method and GSCM. This model for 5G-NR[2] supports channel bandwidth up to 2 GHz and frequencies ranging from 0.5 GHz to 100 GHz. It provides accurate simulation at the cost of higher complexity than alternatives, and can also model additional components, such as oxygen absorption, blockage, large antenna arrays, and spatial consistency. The parameters of the channel are split into two distinct categories, namely deterministic parameters that are acquired through the ray-tracing method, and random parameters obtained through measurement data. This concept has the ability to achieve a balance between complexity and accuracy [66]. In the next chapter, we discuss in detail the 3GPP channel model based on release 15 of TR 38.901 [67].

The 3GPP 3D model combines the key desirable attributes of wideband, time-varying, spatial, and optimized for cellular systems, making it a robust choice for 5G-NR system simulation. Some key benefits of 3GPP channel model are;

Standardization: 3GPP is highly standardized and is the basis for many global 5G deployments. This makes it easier for vendors and operators to adopt it.
Comprehensive Scenarios: 3GPP covers a wide range of scenarios from urban to rural and indoor environments, making it versatile.
Moderate Complexity: While detailed enough to be accurate, 3GPP channel model for 5G is not as computationally intensive as some other models, making it more practical for real-world applications.
Frequency Range: It covers both sub-6 GHz and mmWave frequencies, making it adaptable to various types of 5G deployments.
MIMO Support: 3GPP fully MIMO technologies, which are crucial for 5G's high data rates.
Industry Adoption: Being one of the most widely adopted models, 3GPP has a large ecosystem, which makes it easier for operators to find compatible equipment and solutions.

n summary, the 3GPP 3D channel model is often considered superior due to its balance between complexity and practicality, its wide range of covered scenarios, and its high level of industry adoption and standardization.

Chapter 3

3GPP Channel Model for 5G-NR

A system-level simulation with many detailed scenarios, many configuration parameters, and sophisticated evaluation metrics requires both significant on-chip data storage and high computational power. Interference calculation becomes even more sophisticated with the inclusion of more complex scenarios. Thus, the requirements for system-level simulators must evolve in different directions, such as propagation channel modeling, interference modeling, and clustering. The propagation effect of a wireless channel can be modeled by combining a large-scale propagation model with a small-scale fading model of the channel. The former predicts the characteristics of the wireless channel model that change slowly, such as shadowing and path losses. The latter models the effect due to the Doppler shift or multipath effects on a wireless channel.

The 3GPP was established by the European Telecommunication Standards Institute (ETSI) and various standard development organizations worldwide with the aim of creating novel cellular network technology. The Technical Specification Group for Radio Access Network (TSG RAN) within the 3GPP has undertaken a study item entitled *Study on Channel Model for Frequency Spectrum Above* 6 GHz. This study item pertains to the modeling of channels for frequencies ranging up to 100 GHz. The model involves a range of scenarios, including Urban-Microcell (UMi) street canyon, Urban-Macrocell (UMa), Rural-Macrocell (RMa), and Indoor-Office (InH) environments. The model offers several notable features, including the ability to accommodate a substantial bandwidth of up to 10% of the center frequency, with a maximum limit of 2 GHz. Additionally, the model maintains spatial consistency,

incorporates large antenna arrays, and accounts for blockage and oxygen attenuation phenomena. Furthermore, identical to the METIS approach, a hybrid model was suggested as a more accurate alternative. This model incorporates deterministic environmental factors into the stochastic model.

Analytical studies for 5G often integrate Rayleigh fading with simplified propagation loss models [68]. These models lack the ability to capture the spatial aspect of the channel and cannot be integrated with accurate beamforming models, despite being computationally efficient. The 3GPP channel model has the ability to model interactions with beamforming due to its stochastic properties.

3.1 3D Channel Modeling

Transmission techniques using multi-antenna configurations and MIMO channels are crucial for enhancing the reliability and spectral efficiency of a radio link. For assessing standardized technologies operating with a BS equipped with horizontally arranged antennas, 3GPP has used two-dimensional spatial channel model (2D-SCM) on the horizontal cross-section of wireless channels [69]. The models exhibit a poor representation of the characteristics of an actual channel due to their confinement to a two-dimensional plane. Additionally, the transmission methods for MIMO systems, such as spatial multiplexing, beamforming, and precoding, are restricted solely to the azimuth dimension. To assess communication techniques like vertical sectorization, it is necessary to employ a 3D channel model. A customized narrow elevation beam is designed for individual vertical sectors or specific user equipment (UE) elevations to effectively adjust both the transmission elevation and azimuth for the UE [69].

The model used in this thesis is a 3D channel model [53] and takes into account the elevation angles and the azimuth angle to model small-scale fading effects and correlation among the antenna elements. Figure 3.1a and fig. 3.1b show the different angles used in two-dimensional (2D) and 3D spatial channel model (SCM).

In 3GPP GSCM,a cluster refers to a group of Multipath components (MPCs) that arrive at the receiver within a close time interval and from similar directions. These clusters represent reflections, diffractions, and scatterings of the radio signal from large objects or structures in the environment, such as buildings, hills, or other significant obstacles. The concept of clusters is crucial in understanding and

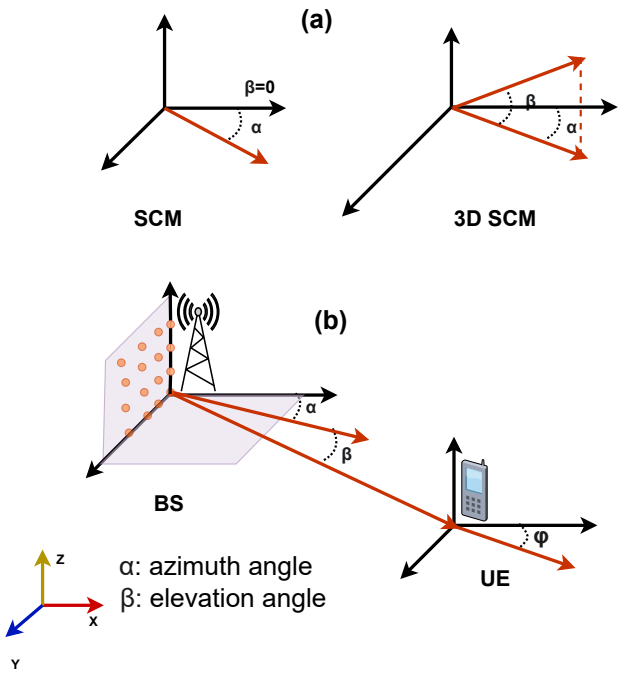


Fig. 3.1 2D SCM to 3D.

modeling the behavior of radio signals in real-world environments. By grouping MPCs into clusters, the 3GPP 3D channel model can more accurately represent the complex interactions between radio waves and the environment, leading to more realistic simulations and better system designs.

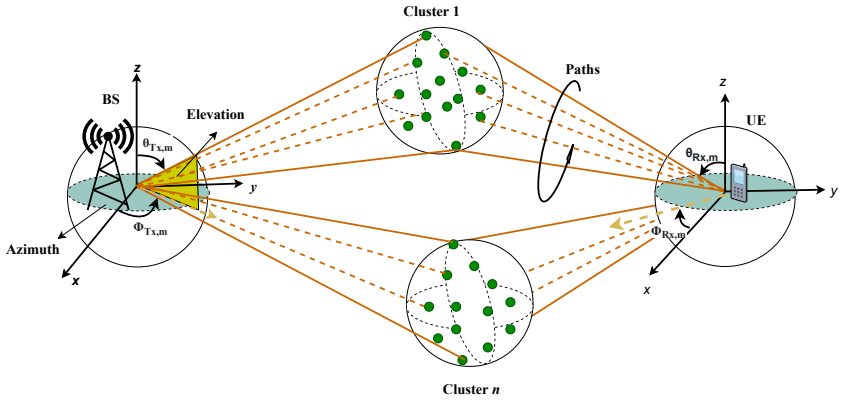


Fig. 3.2 Cluster scattering in 3GPP channel model [59]

Figure 3.2 depicts a stochastic channel model based on geometry for a communication link pairing a transmitter and a receiver. The model incorporates scattering

regions, also known as clusters, which are depicted as large circles consisting of multiple scattering objects. The total number of clusters and rays is subject to variation depending on different scenarios and propagation conditions. Several usage scenarios are defined in the 3GPP specification for elevation beamforming, the urban micro street canyon and open area, 3D urban macro (3D-UMa) with outdoor next generation NodeBs (gNBs), Backhaul, device-to-device (D2D), vehicle-to-vehicle (V2V), and outdoor to indoor (O2I) are examples of some common usage scenarios. For each of these propagation scenarios, different parameters are defined to calculate path losses and microscopic and macroscopic fading. Large-scale parameters (LSPs) are generated for each UE according to the propagation conditions at its location and geographical position. Delay spread, shadow fading,, azimuth angle of arrival (AOA), azimuth angle of departure (AOD), zenith angle of arrival (ZOA), and zenith angle of departure (ZOD) are considered as LSPs, while cluster powers, delays, ZOA, ZOD, and elevation direction are considered as small-scale parameters (SSPs), which change frequently.

3.2 Procedure for Channel Generation

The 3GPP channel model is a hybrid model in which the deterministic part is represented by a path-loss model whereas the stochastic part is modeled by means of LSPs and SSPs. Figure 3.3 shows the steps required for channel generation.

- LSPs generation requires target scenarios selection such as RMa, UMa, UMi, and InH, as well as the network layout such as antenna configuration and velocity parameters. It defines the parameters that fluctuate less frequently over larger distances and mobility such as shadow fading (SF), Ricean K-factor, delay spread (DS), angular spread of arrival and departure in elevation and azimuth (AS), etc.
- SSPs are the parameter that fluctuates over smaller distances and requires remodeling more frequently. It characterizes the propagation of multipath components in terms of power, delay, and angles of arrival and departure in 3D space.

3.2.1 Large-scale fading

This model defines the large-scale link level parameters such as angular spread (AS), delay spread (DS), Ricean K-factor K_R (for line-of-sight (LOS)), shadow fading (SF)

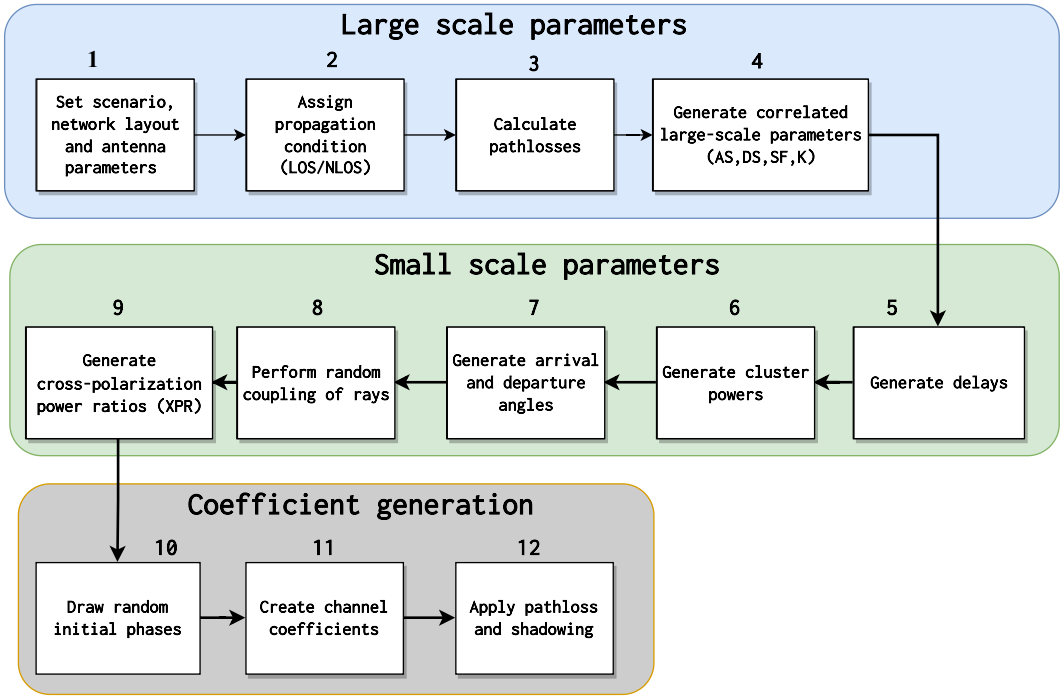
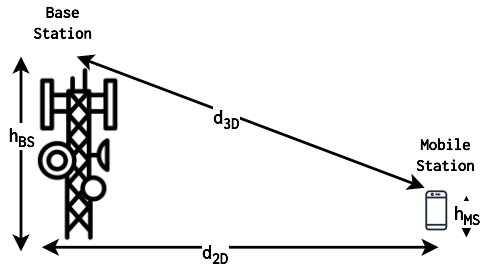


Fig. 3.3 Channel coefficient generation in 3GPP 3D channel model

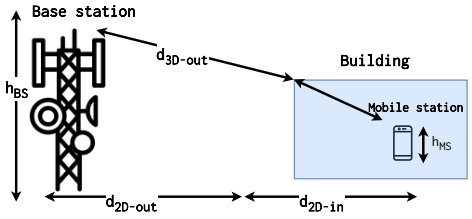
(σ_{SF}), and azimuth and zenith angular spread of arrival and departure (ASA, ZSA, ASD, ZSD) and corresponds to step 4 in 3GPP channel generation procedure fig. 3.3. LSPs in 3GPP specifications are considered as log-normal random variables and provides the mean and variances in [2, Table 7.5-6].

In this step of channel generation, the propagation environment is evaluated for each link between BS and mobile station (MS) through LOS probability. The propagation path-loss model determines the signal strength perceived by the MS in various scenarios, where the higher path-loss corresponds to a weak reception. Path-loss modeling is required to determine the optimal gNB height that would increase the reception strength for the maximum number of MS. As the height of gNB increases, the probability of a direct line-of-sight connection to an MS also increases. However, this comes at the cost of a longer signal path, resulting in greater path loss. Figure 3.4 shows two different scenarios where MS is in LOS in outdoor fig. 3.4a while fig. 3.4b shows BS and MS scenario for indoor environment.

For outdoor MS, the parameter d_{3D} is a straight distance from BS to MS as shown in fig. 3.4a whereas d_{2D} denotes its x-y plane projection. d_{3D} is calculated from horizontal distance d_{2D}, BS antenna height h_{BS} and MS height h_{MS} as



(a) d_{2D} and d_{3D} for outdoor MS



(b) d_{2D-out}, d_{2D-in} and d_{3D-out}, d_{3D-in} for indoor MS

Fig. 3.4 Pathloss distances in outdoor and indoor environments

$$d_{3D} = \sqrt{(d_{2D})^2 + (h_{BS} - h_{MS})^2} \tag{3.1}$$

In case of indoor MS, the parameter d_{3D} is obtained by means of indoor distance from the building to the MS d_{3D-in} and outdoor distance from BS to the building d_{3D-out} as shown in Figure 3.4b. In this way, the total 3D distance for indoor MS is gives as

$$d_{3D} = d_{3D-out} + d_{3D-in} = \sqrt{(d_{2D-out} + d_{2D-in})^2 + (h_{BS} - h_{MS})^2} \tag{3.2}$$

Probability of LOS, P_{LOS} depends on d_{2D-out} for outdoor-to-indoor link and d_{2D} for outdoor-to-outdoor link between BS and MS. [2, Table 7.4.2-1] lists expression for calculation of (P_{LOS})in various propagation environments. In case of RMa, it is calculated as

$$P_{LOS} = \begin{cases} 1 & ,d_{\text{2D-out}} \leq 10\,\text{m} \\ \exp\left(-\frac{d_{\text{2D-out}} - 10}{1000}\right) & ,10\,\text{m} < d_{2D-out} \end{cases} \tag{3.3}$$

Pathloss evaluation is mainly dependent on the distance from BS to MS. Expression for evaluation of path-loss for various propagation conditions is depicted in [2, Table 7.4.1-1].

The path-loss PL_{LOS} for RMa when an LOS is present is computed as follows.

LOS

$$PL_{\text{RMa-LOS}} = \begin{cases} PL_1 & 10\,\text{m} \leq d_{\text{2D}} \leq d_{\text{BP}} \\ PL_2 & d_{\text{BP}} \leq d_{\text{2D}} \leq 10\,\text{km} \end{cases} \tag{3.4}$$

where

$$PL_1 = 20\log_{10}(\frac{40\pi d_{3D} f_c}{3}) + \min(0.03h^{1.72}, 10)\log_{10}(d_{3D}) - \min(0.044h^{1.72}, 14.77) + 0.002\log_{10}(h)d_{3D} \tag{3.5}$$

and

$$PL_2 = PL_1(d_{BP}) + 40\log_{10}(\frac{d_{3D}}{d_{BP}}) \tag{3.6}$$

where h is the average building height in the surrounding, breakpoint distance is $d_{BP} = 4 \times h_{BS} \times h_{MS} \times \frac{f_c}{c}$, f_c is the center frequency in Hz, c = 3.0×10^8 m/sec is the propagation velocity in free space, and h_{BS} and h_{MS} are the effective antenna heights at the BS and the MS, respectively.

There are two main types of correlation between LSPs: intra-MS correlation (correlation between parameters in the same link) and inter-MS correlation (correlation between LSPs on separate links, which can be either intra-cell or inter-cell). For a high number of links,the correlation matrix size increases significantly due to the inclusion of both intra and inter-MS correlations. To reduce the computational complexity, inter-MS correlations are derived first, followed by intra-MS. 3GPP advises using sub-clause 3.3.1 of [53] for generation of LSPs with the square root matrix produced through the utilization of the Cholesky decomposition.

$$\widetilde{s} = \sqrt{C_{M\times M}(0)}\zeta \tag{3.7}$$

where $C_{M\times M}(0)$ represents the correlation matrix, ζ is the identity normal random vector and $\widetilde{s}$ is the normal random variable for the correlation vector.

The LSPs relating to separate BS-MS links exhibit no correlation, whereas the LSPs corresponding to links originating from co-located sectors and terminating at an MS demonstrate identical characteristics. Furthermore, the LSPs pertaining to the connections of MS on different floors are uncorrelated.

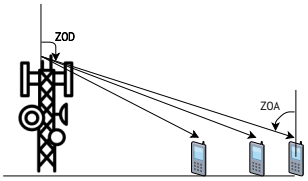
3.2.2 Small-scaling fading

Cluster power, delays, ray parameters, cross-polarization, and angles of arrival and departure are considered as SSPs. We consider a downlink communication

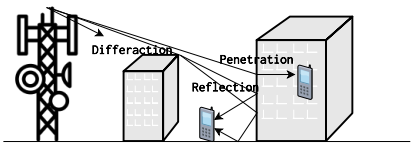
Table 3.1 Notations in 3GPP channel generation

$F_{\mathrm{rx},u,\theta}$	Field pattern of receiving antenna element u in the direction of spherical basis vector $\hat{\theta}$
$F_{\mathrm{rx},u,\varphi}$	Field pattern of receiving antenna element u in the direction of spherical basis vector $\hat{\varphi}$
$F_{\mathrm{tx},s,\theta}$	Field pattern of transmitting antenna element s in the direction of spherical basis vector $\hat{\theta}$
$F_{\mathrm{tx},s,\varphi}$	Field pattern of transmitting antenna element s in the direction of spherical basis vector $\hat{\varphi}$
$\theta_{n,m,\mathrm{ZOD}}$	Zenith angle of departure (ZOD) for ray m in cluster n
$\theta_{n,m,\mathrm{ZOA}}$	Zenith angle of arrival (ZOA) for ray m in cluster n
$\phi_{n,m,\mathrm{AOD}}$	Azimuth angle of departure (AOD) for ray m in cluster n
$\phi_{n,m,\mathrm{AOA}}$	Azimuth angle of arrival (AOA) for ray m in cluster n
$\theta_{n,\mathrm{ZOD}}$	Zenith angle of departure (ZOD) for cluster n
$\theta_{n,\mathrm{ZOA}}$	Zenith angle of arrival (ZOA) for cluster n
$\phi_{n,\mathrm{AOD}}$	Azimuth angle of departure (AOD) for cluster n
$\phi_{n,\mathrm{AOA}}$	Azimuth angle of arrival (AOA) for cluster n
$\theta_{\mathrm{LOS,ZOD}}$	Line-of-sight (LOS) zenith angle of departure (ZOD)
$\theta_{\mathrm{LOS,ZOA}}$	Line-of-sight (LOS) zenith angle of arrival (ZOA)
$\phi_{\mathrm{LOS,AOD}}$	Line-of-sight (LOS) azimuth angle of departure (AOD)
$\phi_{\mathrm{LOS,AOA}}$	Line-of-sight (LOS) azimuth angle of arrival (AOA)
$\kappa_{n,m}$	Cross polarization power ratio for path m and cluster n
$\Phi_{n,m}^{XY}$	Random initial phase
$\hat{r}_{\mathrm{rx},n,m}$	Spherical unit vector of rx element
$\hat{r}_{\mathrm{rx},n,m}$	Spherical unit vector of tx element
$\bar{d}_{\mathrm{rx},u}$	Location vector of rx antenna element u
$\bar{d}_{\mathrm{tx},s}$	Location vector of tx antenna element s
λ_0	Wavelength of carrier frequency
$\bar{v}$	Velocity vector of user equipment (UE)
P_n	nth path power

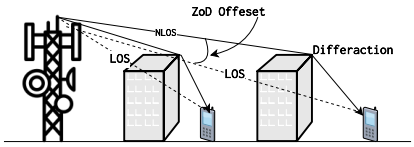
in the following in which the departure angles are defined on BS (also termed as gNB) side whereas arrival angles are at MS (or UE) side. The coefficients of the fast fading channel provide a model for the dynamic fluctuations of wireless channels, which result from the interaction between multipath and the movement of the user equipment. Table 3.1 lists notations used for the realization of small-scale fading using the step-by-step method shown in fig. 3.3. The calculation of channel coefficients for a link connecting a transmitter and a receiver is based on the combined channel impulse responses of the various multiple-path components. Each of these components is distinguished by a path delay and path power, alongside stochastic phases that are introduced during propagation. Additionally, the multiple path components incident path angles, namely AOD and AOA and ZOD and ZOA, are also taken into account. Figure 3.5 shows zenith arrival and departure angles for different scenarios



(a) ZOD in outdoor LOS UEs



(b) ZOD in outdoor NLOS UEs



(c) Outdoor and Indoor NLOS scenarios

Fig. 3.5 Angles of arrival and departure in fast fading model

Cluster delay

The first step in SSP for the fast fading model is the generation of cluster delays. The following steps are performed in order to calculate the propagation delays τ'_n for each cluster with delay scaling r_τ and delay spread DS, using an exponential delay distribution

$$\tau'_n = -r_\tau \times \mathrm{DS} \times \ln(X_n) \tag{3.8}$$

X_n denotes uniform distribution, n is index of cluster $n = 1, \cdots, N$. It is required to sort and normalize the delay vector obtained from (3.8) to ascending order as follows,

$$\tau_n = \mathrm{sort}\left(\tau'_n - \min\left(\tau'_n\right)\right) \tag{3.9}$$

Further scaling of delays is necessary to counteract the impact of LOS using a constant value denoted as C_τ as given by

$$C_\tau = 0.7705 - 0.0433K_R + 0.0002{K_R}^2 + 0.000017{K_R}^3 \tag{3.10}$$

where K_R denotes the Ricean K-factor as generated in step 4 fig. 3.3. The scaled delay can be calculated as

$$\tau_n^{LOS} = \tau_n / C_\tau \tag{3.11}$$

Cluster Powers

The calculation of cluster powers is based on the assumption of a power delay profile that follows a single slope exponential function. The allocation of power is based upon the delay distribution as specified in [2, Table 7.5-6]

$$P'_n = \exp\left(\tau_n \frac{r_\tau - 1}{r_{\tau \mathrm{DS}}}\right) 10^{\frac{-Z_n}{10}} \tag{3.12}$$

Where $Z_n \sim \mathscr{N}\left(0,\ \zeta^2\right)$ and ζ being shadowing factor for each cluster as provided in [2, Table 7.5-6] The cluster powers are then normalized by dividing the previous power P'_n with the total power as given by

$$P_n = \frac{P'_n}{\sum_{n=1}^{N} P'_n} \tag{3.13}$$

If LOS component is also present, an additional component $P_{1,LOS} = \frac{K_R}{K_R+1}$ must be added to the total power.

After obtaining the cluster powers P_n, we proceed to distribute the power among the individual rays. This is achieved by dividing P_n equally among the total number of rays, resulting in $P_{n,m} = \frac{P_n}{M_{ray}}$ where $n = 1, \cdots, N_{cl}$ represents the cluster index and $m = 1, \cdots, M_{ray}$ represents the ray index.

Azimuth and Zenith angles of arrival and departure

Power angular spectrum (PAS) is a metric that characterizes the power distribution in various directions and ascertains the spatial correlation properties of the channel. A wrapped Gaussian distribution is utilized to model the composite in azimuth for all clusters. The generation of arrival and departure angles in azimuth follows a similar procedure [2]. The generation of AOAs involves the application of the inverse Gaussian distribution, utilizing the cluster average power P_n and the root-mean-square azimuth spread of arrival (ASA).

$$\varphi'_{n,AOA} = \frac{2\left(\frac{\mathrm{ASA}}{1.4}\right)\sqrt{-\ln\left(\frac{P_n}{\max(P_n)}\right)}}{C_\varphi} \tag{3.14}$$

The scaling factor denoted as C_φ is defined in relation to the overall number of clusters, and its corresponding values can be found in [2, Table 7.5-2].

The angles can be assigned a positive or negative sign by multiplying them with a random variable X_n that follows a uniform distribution. Additionally, to introduce further random variation, a component Y_n is added as

$$\varphi_{n,AOA} = X_n \varphi'_{n,AOA} + Y_n + \varphi_{LOS,AOA} \tag{3.15}$$

To generate the final cluster angles by adding α_m as offset angle,

$$\varphi_{n,m,AOA} = \varphi_{n,AOA} + c_{ASA}\,\alpha_m \tag{3.16}$$

where C_{ASA} denotes the root-mean-square azimuth spread of arrival for each cluster. The generation of departure angles AOD follows the same procedure as for AOA in (3.14) and (3.16) by replacing azimuth spread of arrival with azimuth spread of departure and C_{ASA} with C_{ASD} from [2, Table 7.5-6].

The Laplacian distribution is utilized to model the composite PAS at the zenith of all clusters. The generation of ZOA involves the application of the inverse Laplacian distribution, taking into account the cluster average power P_n and the root-mean-square zenith spread of arrival (ZSA).

$$\theta'_{n,ZOA} = -\frac{\mathrm{ZSA}\ln\left(\frac{P_n}{\max(P_n)}\right)}{C_\theta} \tag{3.17}$$

where C_θ denotes cluster-wide scaling factor and is given in [2, Table 7.5-4]. Following a similar procedure used for generation of AOA, by introducing a random variable Y_n,

$$\theta_{n,ZOA} = X_n \theta'_{n,ZOA} + Y_n + \overline{\theta}_{ZOA} \tag{3.18}$$

where $\overline{\theta}_{ZOA} = \theta_{LOS,ZOA}$, X_n is uniform random distribution, and $Y_n \sim \mathcal{N}\left(0, \left(\frac{\mathrm{ZSA}}{7}\right)^2\right)$ as random variation. The final ZOA angles are

$$\theta_{n,m,ZOA} = \theta_{n,ZOA} + c_{ZSA}\,\alpha_m \tag{3.19}$$

where α_m is the offset angles, C_{ZSA} is the root-mean-square zenith spread for each cluster. To calculate departure angles in the zenith direction, a similar procedure is followed while replacing ZSA in (3.17) with ZSD and C_{ZSA} in (3.19) with C_{ZSD}.

After calculating azimuth and zenith angles of arrival and departure for each ray, a random coupling among them as given below is performed for each cluster n

- AOA $\varphi_{n,m,AOA}$ with AOD $\varphi_{n,m,AOD}$
- ZOA $\theta_{n,m,ZOA}$ with ZOD $\theta_{n,m,ZOD}$
- AOD $\varphi_{n,m,AOD}$ with ZOD $\theta_{n,m,ZOD}$

Cross-polarization power ratios

The next stage in SSPs computation is the generation of cross-polarization power ratios for each ray m in each cluster n.

$$\kappa_{n,m} = 10^{\frac{X_{n,m}}{10}} \tag{3.20}$$

where $X_{n,m}$ is a Gaussian-distributed random variable and is calculated using values from [2, Table 7.5-6]

Channel coefficient generation

Prior to generating the channel coefficient, it is necessary to randomly select the initial phases, denoted as $\left\{\Phi_{n,m}^{\varphi\varphi}, \Phi_{n,m}^{\theta\varphi}, \Phi_{n,m}^{\varphi\theta}, \Phi_{n,m}^{\theta\theta}\right\}$, from a uniform distribution within the interval $(-\pi, \pi)$.

Considering N cluster scatterers with M resolvable paths each, the CIR for ray m in $n = N-2$ weakest clusters, UE antenna element u, and BS antenna element s is

$$
\begin{aligned}
H_{s,n,m}^{NLOS}(t) = & \\
& \sqrt{\frac{P_n}{M}} \begin{bmatrix} F_{\mathrm{rx},u,\theta}\left(\theta_{n,m,\mathrm{ZOA}}, \phi_{n,m,\mathrm{AOA}}\right) \\ F_{\mathrm{rx},u,\phi}\left(\theta_{n,m,\mathrm{ZOA}}, \phi_{n,m,\mathrm{AOA}}\right) \end{bmatrix}^T \times \begin{bmatrix} e^{j\Phi_{n,m}^{\theta\theta}} & \sqrt{\kappa_{n,m}{}^{-1}}\, e^{j\Phi_{n,m}^{\theta\phi}} \\ \sqrt{\kappa_{n,m}{}^{-1}}\, e^{j\Phi_{n,m}^{\phi\theta}} & e^{j\Phi_{n,m}^{\phi\phi}} \end{bmatrix} \\
& \times \begin{bmatrix} F_{tx,s,\theta}\left(\theta_{n,m,\mathrm{ZOD}}, \phi_{n,m,\mathrm{AOD}}\right) \\ F_{tx,s,\phi}\left(\theta_{n,m,\mathrm{ZOD}}, \phi_{n,m,\mathrm{AOD}}\right) \end{bmatrix} \times e^{j2\pi \frac{\hat{r}_{\mathrm{rx},n,m}^T \cdot \bar{d}_{\mathrm{rx},u}}{\lambda_0}} \times e^{j2\pi \frac{\hat{r}_{tx,n,m}^T \cdot \bar{d}_{tx,s}}{\lambda_0}} \\
& \times e^{j2\pi \frac{\hat{r}_{\mathrm{rx},n,m}^T \cdot \bar{v}}{\lambda_0} t}
\end{aligned}
\tag{3.21}
$$

where $\hat{r}_{\mathrm{rx},n,m}$ is the spherical unit vector with elevation arrival angle $\theta_{n,m,\mathrm{ZOA}}$ and azimuth arrival angle $\varphi_{n,m,\mathrm{AOA}}$. The field patterns of the receiving antenna element u, denoted as $F_{rx,u,\theta}$ and $F_{rx,u,\phi}$, are determined based on equation (3.15) and are measured in the direction of the spherical basis vectors, θ and ϕ, respectively. Similarly, the field patterns of the transmitting antenna element s, denoted as, $F_{tx,s,\theta}$ are measured in the direction of the spherical basis vectors, θ and ϕ, respectively.

For cluster n and ray m within cluster n, the spherical unit vector is given by

$$
\hat{r}_{\mathrm{rx},n,m} = \begin{bmatrix} \sin\theta_{n,m,\mathrm{ZOA}} \cos\phi_{n,m,\mathrm{AOA}} \\ \sin\theta_{n,m,\mathrm{ZOA}} \sin\phi_{n,m,\mathrm{AOA}} \\ \cos\theta_{n,m,\mathrm{ZOA}} \end{bmatrix} \tag{3.22}
$$

Similarly, $\hat{r}_{tx,n,m}$ is the spherical unit vector with elevation departure angle $\theta_{n,m,\mathrm{ZOD}}$ and azimuth departure angle $\phi_{n,m,\mathrm{AOD}}$. For cluster n and ray m within cluster n it is

$$
\hat{r}_{\mathrm{tx},n,m} = \begin{bmatrix} \sin\theta_{n,m,\mathrm{ZOD}} \cos\phi_{n,m,\mathrm{AOD}} \\ \sin\theta_{n,m,\mathrm{ZOD}} \sin\phi_{n,m,\mathrm{AOD}} \\ \cos\theta_{n,m,\mathrm{ZOD}} \end{bmatrix} \tag{3.23}
$$

The Doppler frequency component $v_{n,m}$ depends on the UE speed v is applied to the coefficients of each channel, for each antenna element.

$$
v_{n,m} = \frac{\hat{r}_{rx,n,m}^T \cdot \bar{v}}{\lambda_0}, \text{where } \bar{v} = v.\begin{bmatrix} \sin\theta_v \cos\varphi_v & \sin\theta_v \sin\varphi_v & \cos\theta_v \end{bmatrix}^T \tag{3.24}
$$

The two strongest clusters, namely $n = 1, 2$, exhibit a spread of rays in delay that is distributed across three sub-clusters as follows

- sub-cluster 1 R_1 ={1,2,3,4,5,6,7,8,19,20}
- sub-cluster 2 R_2 = {9,10,11,12,17,18}
- sub-cluster 3 R_3 ={13,14,15,16}

Each of these sub-clusters is characterized by a constant delay offset as given by

$$\tau_{n,1} = \tau_n, \quad \tau_{n,2} = \tau_n + 1.28C_{DS}, \quad \tau_{n,3} = \tau_n + 2.56C_{DS} \tag{3.25}$$

where C_{DS} denotes delay spread of cluster.

Considering the delays and ray mappings listed in [2, Table 7.5-5], the CIR $H_{u,s}^{NLOS}(\tau,t)$ for non-line-of-sight (NLOS) can be computed as

$$\begin{aligned} H_{u,s(\tau,t)}^{NLOS} = &\sum_{n=1}^{2}\sum_{i=1}^{3}\sum_{m\in R_i} H_{s,n,m}^{NLOS}(t)\delta(\tau-\tau_{n,i}) \\ &+ \sum_{n=3}^{N}\sum_{m=1}^{M} H_{s,n,m}^{NLOS}(t)\delta(\tau-\tau_n) \end{aligned} \tag{3.26}$$

where $\delta(.)$ denotes the Dirac's delta function.

Similarly, if there are LOS cluster links, the channel coefficients are computed as follows

$$\begin{aligned} H_{u,s,1}^{\mathrm{LOS}}(t) = &\begin{bmatrix} F_{rx,u,\theta}\left(\theta_{LOS,ZOA},\varphi_{LOS,AOA}\right) \\ F_{rx,u,\varphi}\left(\theta_{LOS,ZOA},\varphi_{LOS,AOA}\right) \end{bmatrix}^T \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} F_{tx,s,\theta}\left(\theta_{LOS,ZOD},\varphi_{LOS,AOD}\right) \\ F_{tx,s,\varphi}\left(\theta_{LOS,ZOD},\varphi_{LOS,AOD}\right) \end{bmatrix} \\ &\times e^{-j2\pi\frac{d_{3D}}{\lambda_0}} \times e^{j2\pi\frac{\hat{r}_{rx,LOS}^T\cdot\bar{d}_{rx,u}}{\lambda_0}} \times e^{j2\pi\frac{\hat{r}_{tx,LOS}^T\cdot\bar{d}_{tx,s}}{\lambda_0}} \times e^{j2\pi\frac{\hat{r}_{rx,LOS}^T\cdot\bar{v}}{\lambda_0}t} \end{aligned} \tag{3.27}$$

The LOS channel coefficients scaled based on the desired Ricean K-factor K_R are computed as

$$H_{u,s}^{LOS}(\tau,t) = \sqrt{\frac{1}{K_{\mathrm{R}}+1}}H_{u,s}^{LOS}(\tau,t) + \sqrt{\frac{K_{\mathrm{R}}}{K_{\mathrm{R}}+1}}H_{u,s,1}^{LOS}(t)\delta\left(t-\tau_1\right) \tag{3.28}$$

3.2.3 Channel models for link-level evaluations

The channel can be either represented as a tapped delay line (TDL) or a cluster delay line (CDL). For simplified evaluation, the TDL model is defined as an impulse response, in which a radio channel is characterized by several delay taps while the CDL model is characterized by the arrival and departure directions in the 3D space which allows better beamforming representation. The TDL model defines the correlation between the antenna elements through a static correlation matrix, whereas the CDL model depends on the geometry of the antenna elements and how the channel propagates. The CDL models cover the entire frequency range spanning 0.5 GHz to 100 GHz, and exhibit a maximum bandwidth of 2 GHz. CDL models may be implemented through the execution of coefficient generation in Step 10 and Step 11 as outlined in [2, Subclause 7.5]. [2, Table 7.5-6] provides the mean and standard deviation of K-factor values, which may serve as a reference for determining appropriate values. To convert the K-factor of a model to $K_{desired}$, one must determine the cluster powers for the Laplacian clusters using

$$P_{n,scaled} = P_{n,model} + K_{model} - K_{desired} \tag{3.29}$$

where $P_{n,model}$ denotes the model path power and $P_{n,scaled}$ is the scaled path power in cluster n. K_{model} K-factor can be defined as

$$K_{\text{model}} = P_{1,\text{model}}^{\text{LOS}} - 10\log_{10} \sum_{n=1}^{N} 10^{P_{\text{n,model}}/10} \tag{3.30}$$

Upon rescaling the powers, it is necessary to re-normalize the delay spread by determining the root-mean-square delay spread after the K-factor adjustment. The delays should be divided by that particular value in order to obtain unit delay spread ($DS = 1$).

Following the detailed explanation of the channel model and its application in network planning, the next emphasis will be on the implemented acceleration techniques that enhance the speedy evaluation of said model on hardware platforms. In the subsequent chapter, we will explore the concept of acceleration flows and the corresponding optimizations for hardware platforms, commencing from a high-level specification.

Chapter 4

FPGA Based Acceleration and HLS

FPGAs are a class of programmable semiconductor devices that rely on configurable logic blocks (CLBs) arranged in a matrix configuration and linked via programmable interconnections. These devices possess the capability to be programmed and configured post-manufacturing to meet specific functional requirements. ASICs constitute another category of semiconductor devices with a fixed micro-architecture. In contrast to FPGAs, ASICs are engineered to perform a predetermined set of functions and are highly optimized for a particular application, however, their functionality is restricted to that specific task for which they are designed in the first place. Conversely, there exist general-purpose processors that are capable of executing nearly all types of applications, albeit with suboptimal performance due to lack of optimization.

FPGAs are deemed to be more flexible than ASICs due to their ability to be reconfigured based on the desired functionality. However, this flexibility comes at the expense of power and area. In general, specialized architectures exhibit higher efficiency compared to their general-purpose counterparts. However, this advantage is offset by the need for more intricate programming and reduced flexibility, which incurs additional costs.

While one-time programmable FPGAs are present in the market, the majority of FPGAs currently employed are based on static RAM (SRAM) technology. These devices consist of a significant quantity of SRAM cells that collectively constitute look-up tabless (LUTs), as well as a large number of registers and programmable routes for interconnections. The ability to update the contents of LUTs and configure

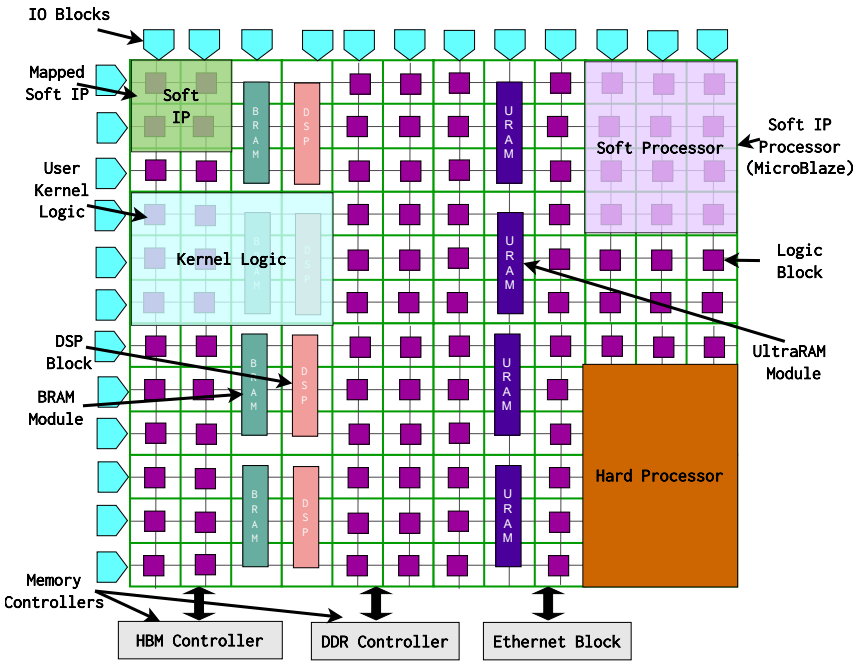


Fig. 4.1 Modern FPGAs with hard logic components [70]

routing accordingly allows designers to easily apply a specific design to a variety of applications. In order to establish connectivity between discrete CLBs and both internal and external systems, a reconfigurable interconnection mechanism is required. A programmable interconnect matrix is employed to establish connections between the CLBs and IO blocks. In addition to LUTs as soft-logic, there exist certain hard-logic components. The aforementioned constituents large memory units such as block RAM (BRAM), digital signal processor (DSP), and a number of memory controllers. Figure 4.1 shows a typical modern FPGA architecture with hard-logic components.

Hard-logic components are employed to execute certain specialized logic functions that would occupy a significant amount of space and be much slower if implemented using LUTs.

4.1 Design Flow

The FPGA design process encompasses several stages, which include design entry, design synthesis, implementation, and device programming.

Design Entry There exist multiple methodologies for the process of design entry. The process can be initiated either by using a schematic, employing an HDL, or a combination of both. When dealing with a system that exhibits a high degree of complexity, it is advisable to commence the design process using HDL as opposed to working with lower-level schematics.

Design Synthesis The process of integrating various design elements into a cohesive and functional whole is commonly referred to as design synthesis. During the design synthesis phase, the code is transformed into a physical circuit through by means of low-level implementation techniques, including gates, flip-flops, and adders. The process of converting the input design involves the creation of a netlist that outlines the components used, and their respective interconnections. The initial step in the design synthesis procedure involves conducting a syntax check on the HDL-based design that has been submitted as input. After the design phase, the logic is refined through the application of various optimization techniques, including but not limited to redundant logic elimination, logic reduction, and size reduction, with the additional goal of enhancing its implementation speed. Upon conducting potential optimization, the design is subsequently mapped onto the relevant technology, extracting timing and area information.

Implementation This particular stage is responsible for establishing the arrangement and structure of the final design that will be implemented as a circuit. The process comprises three distinct stages, namely translation, mapping, placement, and routing. The aforementioned tool consolidates the various design constraints alongside the design netlists. The tool proceeds to perform resource mapping on the FPGA based on the design's specified requirements. Following the process of mapping, the subsequent stage involves the establishment of pathways for the transmission of signals (referred to as routing) and the interconnection of IO interfaces.

Device Programming The result of the implementation phase is a bitstream file that comprises the final design's footprint on the device. Subsequently, the aforementioned file is transferred to the FPGA platform for the execution of the design on the physical FPGA device.

4.2 High-level synthesis

The acquisition of skills in HDL development can be challenging due to the inherent nature of hardware descriptions. The cognitive approach involved in developing a circuit configuration cannot be readily equated to the process of developing software. The marketing challenge in the accelerator domain is particularly significant for FPGA vendors. The process of programming multi-threaded CPUs and GPUs is relatively uncomplicated and exhibits significant similarities between the two. In order to fully utilize the acceleration capabilities of FPGA, it is necessary for developers to acquire proficiency in entirely novel programming languages that possess unique features. As a result of this requirement, leading FPGA producers have created HLS toolchains for software developers without strictly requiring extensive hardware expertise.

The characterization of the circuit is based on its intended functionality rather than its inherent behavior. Subsequently, the compiler tries to infer a functional circuit that aligns with the intended functionality based on the provided description. The development of general-purpose programming languages is a complex and ongoing challenge that has engaged multiple generations of scientists and engineers. The issue remains unresolved to this day.

HLS is a complex subject for compiler engineers due to its intricate requirements, which include the appropriate allocation of tasks across available resources, the establishment of register boundaries to facilitate the formation of pipelines, and the management of recursion and loops. The HLS methodology has been designed to facilitate a seamless transition for designers from high-level specification to HDL implementation while minimizing errors and reducing the required effort to obtain a functional and optimized design. During the initial stages of HLS, its use was limited due to the unavailability of sophisticated tools [71]. In recent times, the latest iteration of HLS tools has garnered considerable attention due to their notable innovations and the improved efficiency of the generated designs.

Several HLS tools have been introduced for rapid prototyping and hardware development for large FPGAs. These tools take as input programs written in C, C++, Open Computing Language (OpenCL) [72–78], and other high-level languages [79–83] alongside some design constraints and pragmas, and translate them into lower-level descriptions such as RTL or HDL with equivalent functionality. This

translation from high-level description into HDL is done by *HLS tools*. The translated design is then transformed into a gate-level description by the *synthesis toolchain*, and mapped onto the hardware resources of the target device. This circuit description is then mapped to the actual locations on the target device to reduce the length of the critical paths through *place and route tools*. The final stage is encoding the circuit description into a binary format (bitstream), which is then used to configure the FPGA on-chip resources and define the initial on-chip SRAM contents.

4.2.1 HLS flow for FPGAs

HLS facilitates the automatic generation of RTL descriptions from high-level specifications, which is essentially an algorithmic representation written in HLLs like C/C++, Matlab, and even Domain-Specific Languages (DSL). Prior to being parsed by the HLS tool, it is common to subject this specification to a test-bench, which is typically composed in the same programming language. A typical HLS tool generates the control and data flow graph (CDFG), which serves as an intermediary depiction of the application. The intermediate representation's operations are scheduled and mapped into hardware resources, taking into account design constraints and the target technology. Optimizing the design is a crucial aspect of high-level synthesis, and numerous techniques are employed to accomplish this task. Subsequently, the RTL description is produced and subsequently subjected to verification through the test-bench. Figure 4.2 depicts a typical HLS based development flow starting from a high-level specification.

HLS-based design process is typically split into two distinct phases. In the first phase, the system is modeled in an HLL as a reference system and tested under a number of constraints. This phase can be performed by software developers with limited hardware design knowledge. The HLS tool initially processes high-level specifications by parsing and transforming them into CDFGs, which serve as an intermediary representation, where operations and their dependencies are represented by nodes alongside control information. These graphs are subsequently supplied to the scheduling and binding procedures, which uses a variety of algorithms. The appropriate algorithms for scheduling and binding are automatically chosen based on the objective of optimizing either performance or area. Hardware designers typically require extensive expertise and experience to effectively optimize a system's performance while balancing resource utilization.

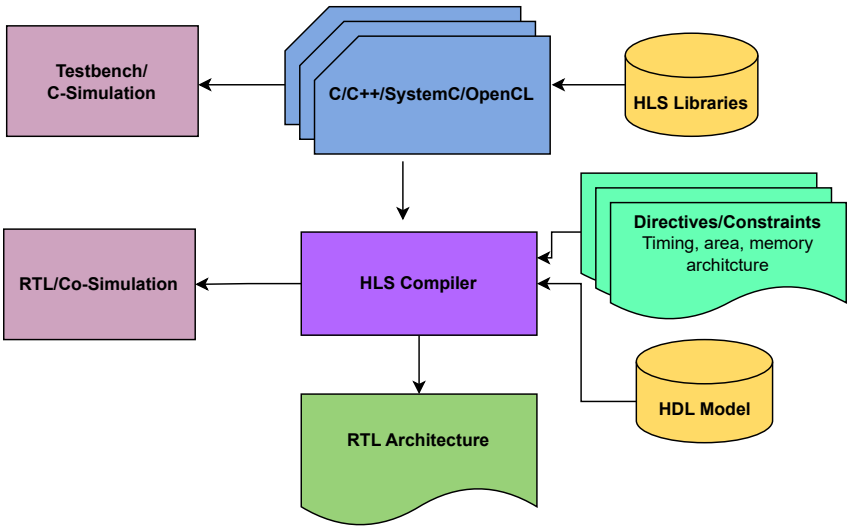


Fig. 4.2 High-level synthesis Flow

The next phase is about the creation of practical hardware architecture and requires a more sophisticated and advanced knowledge of hardware design. HLS performs automated conversion of high-level specifications that adhere to specific hardware-related rules into RTL descriptions. In order to produce high-quality RTL descriptions, it is essential that designers adhere to the prescribed guidelines. Nevertheless, it should be noted that the HLS instructions are reliant on the specific tool being used.

4.2.2 OpenCL in hardware acceleration

OpenCL is a parallel programming language for multicore and heterogeneous computing platforms [84]. OpenCL is developed as an open standard by the Khronos group, thus it has an edge over a similar framework, the CUDA, fully controlled by NVIDIA and only available for its devices. OpenCL is designed so that an application can be *adapted* across different computing platforms. Although OpenCL provides functional portability, *platform-specific optimizations are necessary to exploit most of the target platform computational power*. This allows software programmers to exploit the architectural features of the underlying platforms, such as the distinction between the local on-chip memory, the global memory, and registers, just like they can do for GPUs [85]. An OpenCL application is comprised of one or more *device* or *kernel* functions, and *host* code. Device code is part of the code which is highly

data parallel and computationally intensive and will be executed on the accelerator. Host code is the part that sets up the environment and controls data movement to and from the accelerator device and is executed on a general-purpose CPU.

OpenCL devices include one or more compute units (CUs) and each may contain one or more processing elements (PEs), depending on the platform and the designer implementation choices. OpenCL splits the computations in parallel threads called *work-items (WIs)* which are then combined together in *work-groups (WGs)*. This approach adds support for data-parallel computations and thus some "doall" loop iterations without inter-iteration dependencies (in particular those over WGs), can be mapped to kernel instances that execute in parallel. Not all applications, though, expose high "doall" parallelism at the top of the kernel level. Moreover, FPGA architectures permit finer-grained control over the implementation parallelism, e.g., between tasks within a kernel or iterations of an inner loop. For this reason, OpenCL also offers an execution model, more suitable for CPUs and FPGAs than for GPUs, that executes repeatedly a single instance of the kernel and is called *single work-item kernel*. In this approach, the available parallelism must be defined at a finer grain, using FPGA specific pragmas. A similar approach can also be used to synthesize C or C++ code into a concurrent FPGA implementation, as discussed below. Figure 4.3 shows the main elements of an OpenCL design. It relies on a single instruction

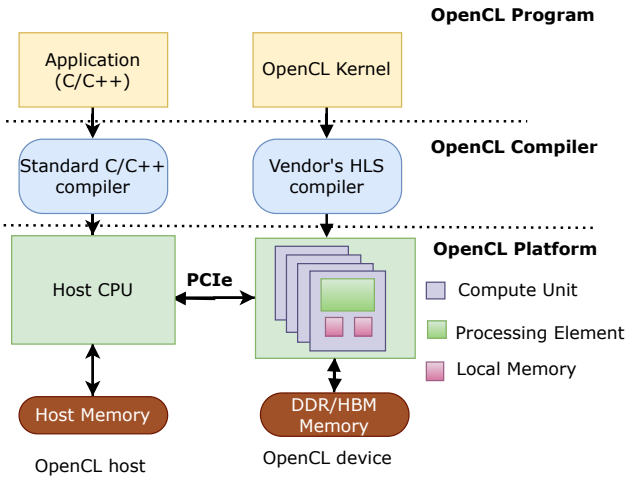


Fig. 4.3 OpenCL based hardware acceleration flow.

multiple data (SIMD) paradigm similar to GPUs to better exploit the hardware platforms. It enables the developers to generate efficient code that fits the architecture of the target device by providing an abstract but non-uniform memory hierarchy.

OpenCL based applications are composed of two major parts; a *host* code which performs the initialization, memory allocation, and data-transfer related tasks and executes on CPU, and *kernel* code which performs the computationally intensive tasks and executes on the acceleration platform.

OpenCL divides memory into different spaces namely *global*, *local*, *private*, and *constant* memory. Global and constant memories are shared among all the CUs in a device and with the host CPU, reside in external dynamic RAM (DRAM), and hence have the highest latency. Local memory is shared among WIs in a WG, has lower latency than global memory, and is often mapped to on-chip SRAM. Each WI finally has its private memory space, which is mapped to the register file and has the lowest latency.

4.2.3 Key advantages of HLS

HLS tools facilitate the automatic generation of the RTL implementation of a design from its high-level specification. This helps in the elimination of errors at various design stages and reduced time to market. Some key advantages of using HLS-based implementation flow are discussed in the following section

Reduced effort in design and verification

The key advantage of HLS-based flow is rapid prototyping, which reduces the designer's effort. HLS significantly alleviates design burdens. Designers do not require extensive knowledge of hardware design to implement specific details. For example, when using HDL, it is essential to explicitly specify the clock cycle level information in the design. Nevertheless, this information is not typically included in HLL specifications. Typically, within the majority of HLS tools, users are able to select the desired clock frequency, whereby the corresponding clock statements will subsequently be inserted into the resultant component in an automated manner. Therefore, the designer concentrates solely on behavior design. Additionally, the code written in HLL is comparatively easier to write. It has been observed that HLS-based design results in a reduction of code lines by more than 20% in comparison to those implemented using RTL language for a design containing 1 million gates. Figure 4.4 shows that the adoption of HLS based flow can reduce the design time by a factor of 2 to 5 compared to conventional RTL based flow.

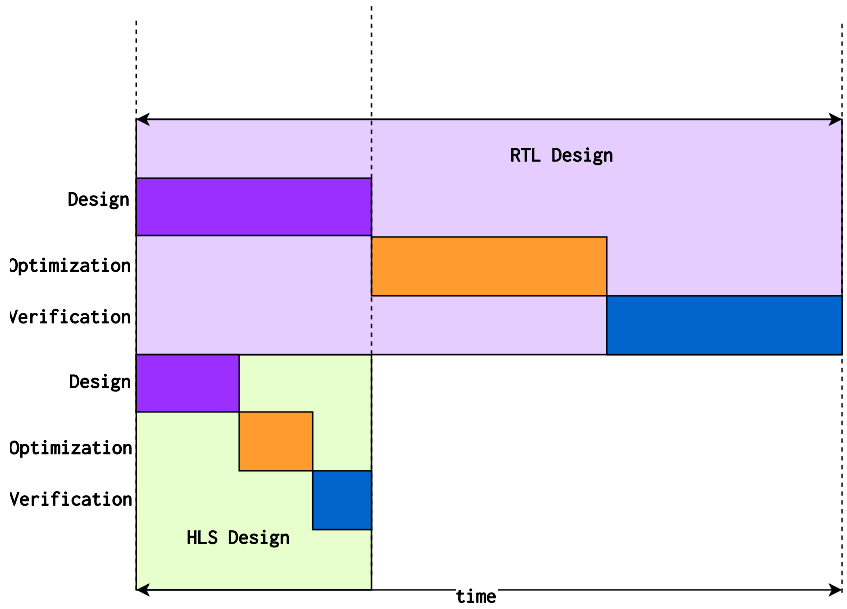


Fig. 4.4 HLS and RTL flow for FPGA design [86]

Even though codes in HLL are typically shorter, the designer can control the micro-architecture of the implemented systems by means of directives with guide the synthesis tool. All the details such as resource allocation, scheduling, operation, and memory binding are set by the designer but implemented in RTL efficiently by the HLS tool. Verification, which usually takes a lot of time, is also done automatically. So, HLS is a useful tool for reducing the amount of work that goes into planning and verification.

Faster design space exploration

The concept of design space exploration refers to the evaluation of hardware microarchitecture design trade-offs while adhering to a specific set of constraints related to FPGA clock frequency, throughput, latency, and area. During the design process, the hardware microarchitecture is typically determined by the RTL engineer. Upon completion of the RTL, an engineer would typically refrain from implementing microarchitecture modifications, such as the addition of pipeline stages, due to the challenging nature of RTL redesign. In contrast, HLS tools streamline the process of design space exploration, which facilitates the ongoing improvement of the microarchitecture of hardware during the design phase. These tools are designed to

insert pipeline registers in an automated manner, taking into account the clock period constraint specified for the HLS target. Figure 4.5 shows various stages of design space exploration for HLS and RTL designs.

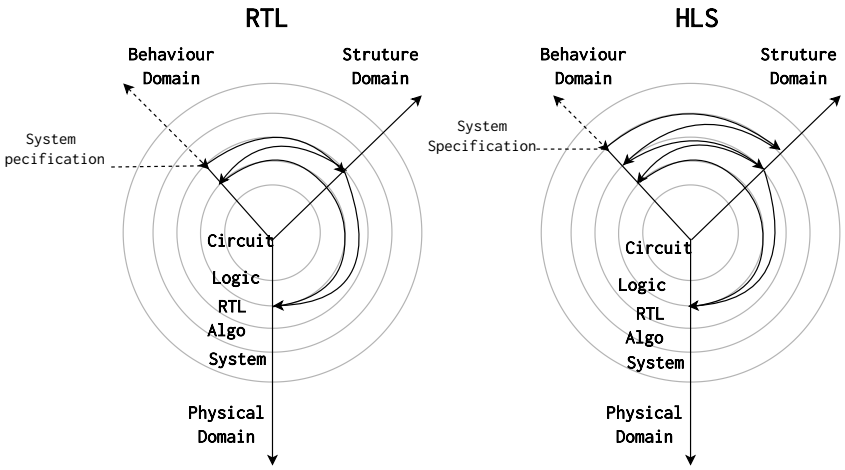


Fig. 4.5 RTL vs HLS design space exploration [87]

HLS runs typically complete in a matter of minutes and enable the designer to get a quick feedback on performance and resource estimate and this avoid long FPGA synthesis runs. The utilization of HLS constraints empowers designers to conduct extensive design space exploration, thereby facilitating the achievement of optimal trade-offs between performance and area for their FPGA designs.

Efficient reuse of designs

HLS tools enable efficient and easy reuse of designs. The majority of RTL designs are optimized to satisfy particular constraints and technology targets, utilizing dedicated processes and state machines, among other techniques. Due to these hard specifications, even minor modifications in the design can result in unexpected behavior and bugs. HLS is considerably more straightforward, where using a high degree of abstraction results in an architecture that is both generic and versatile.

4.3 HLS Tools and Optimizations

Initially, the majority of the HLS tools were aimed at ASIC designs for research objectives only. In the early 1990s, most of the available HLS tools yielded poor quality of results. Since the year 2000, a number OF computer-aided design (CAD) vendors have provided top-notch HLS tools, such as Vitis HLS [72], AutoESL AutoPilot [75], Mentor Catapult C Synthesis [78], Intel FPGA SDK for OpenCL [88], Intel HLS [73], Bluespec [82], Forte Cynthesizer [89], and NEC CyberWorkBench [90]. This chapter offers a brief overview of two commercial HLS tools, namely Vitis HLS [72] and Intel HLS [73]. The majority of our proposed research relies on these tools, and as such, we also present some of the optimization techniques that are available within them.

4.3.1 Vitis HLS

Vitis HLS is a part of Vitis Unified Software Platform [91] development environment, which enables the designers to build accelerated architectures for heterogeneous computing systems. The Vitis platform offers a collection of tools, libraries, and APIs that serve to abstract the complex concepts of hardware programming, thereby facilitating the development of power-efficient, high-performance applications. The platform supports multiple programming languages such as C/C++, OpenCL, and Python, and offers an HLL model, thereby streamlining the process of software development. The Vitis Unified Software Platform employs Vivado as an underlying tool, which is used for generating the hardware design for the FPGA. The Vitis platform offers a suite of software tools and libraries that facilitate the development, testing, and optimization of applications on the FPGA design generated in Vivado by developers. The Vivado and Vitis Unified Software Platforms collaborate to offer a comprehensive solution for designing accelerated applications on Xilinx devices.

4.3.2 Intel FPGA SDK for OpenCL

Intel offers a platform for the generation of bitstreams using OpenCL based kernel function [88]. Intel *AOC* is a source-to-source compiler that takes as input OpenCL based kernels and generated Verilog HDL implementation through the Intel standard toolchain. The Intel FPGA SDK for OpenCL offers the essential application pro-

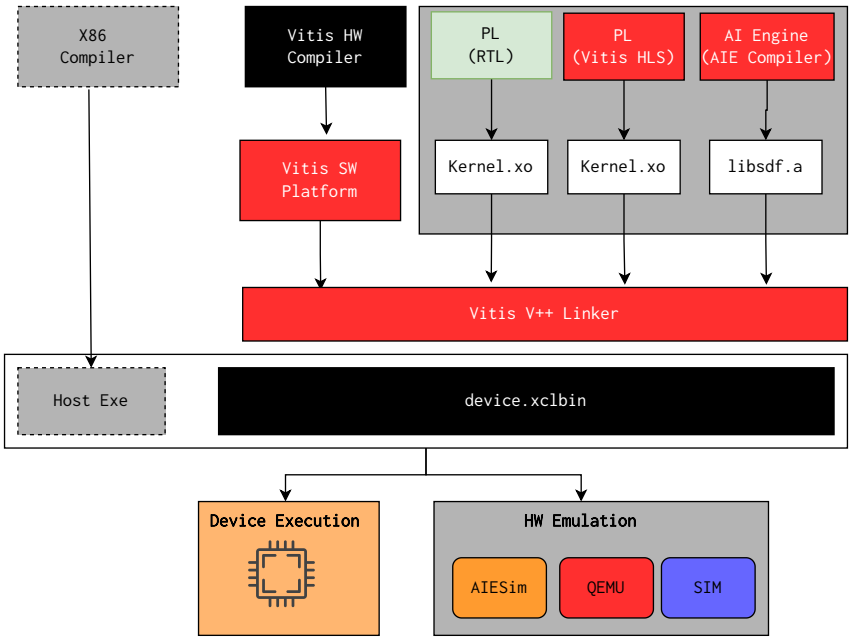


Fig. 4.6 Vitis development flow for acceleration kernels [91]

gramming interfaces and run-time environment to facilitate the programming and utilization of PCIe-attached or System-on-Chip (SoC) FPGAs in a manner akin to that of a GPU or other accelerators. Figure 4.7 depicts the tool flow used for accelerating applications on Intel FPGAs using OpenCL SDK. The procedure for creating OpenCL applications for a heterogeneous computing platform that combines a CPU and FPGA involves utilizing C/C++ to program the CPU and OpenCL to program the FPGA. Both of the aforementioned languages are classified as HLL, as they abstract away the complexities of the underlying hardware. Board support package (BSP) is supplied by board manufacturers and includes the essential IP Cores for facilitating communication between the FPGA, external DDR memory, as well as the necessary PCIe and DMA drivers for enabling communication between the host and the FPGA. This approach alleviates the programmer from manually configuring the IP Cores and generating the drivers, which is common in conventional HDL-based FPGA designs.

The process of synthesizing a C++/OpenCL function into an RTL design through Intel FPGA SDK for OpenCL results in the creation of an IP file. The IP files produced can be integrated into Intel's FPGA design software, namely Platform Designer or Quartus Prime [92]. The bitstream generation process is composed of two major steps. Initially, the OpenCL kernel is compiled HDL code through the

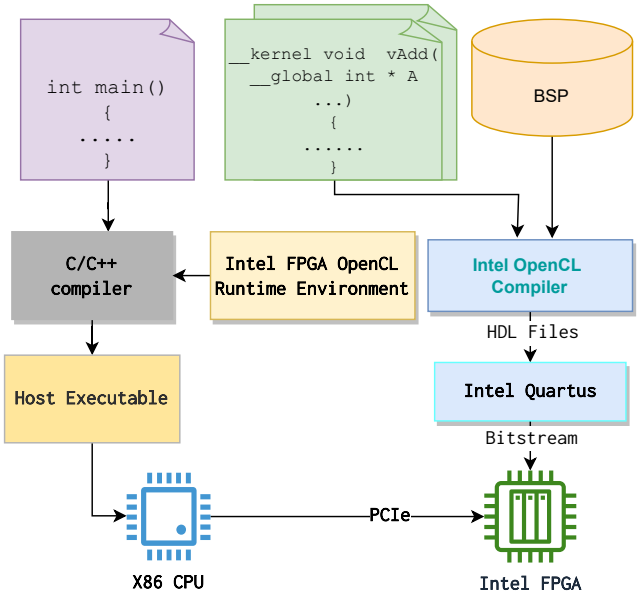


Fig. 4.7 Intel FPGA SDK for OpenCL

utilization of the Intel *AOCL* compiler. The hardware details of the target FPGA are integrated into the resultant HDL code from BSP as shown in fig. 4.7. The following step involves the creation of an FPGA bitstream, which is derived from the compiled HDL code. During this stage, the Quartus toolchain defines all the key implementation details, including the selection of the suitable FPGA frequency. The outcome of this process is the generation of an OpenCL executable including the FPGA bitstream. In contrast, the host code undergoes compilation utilizing a suitable C/C++ compiler.

The process for designing an OpenCL kernel for FPGA begins with the compilation of the *kernel* code for emulation purposes in order to verify its functional corrections. Subsequently, an intermediate compilation may be performed. This step involves the conversion of OpenCL code to Verilog, followed by the generation of a performance report that includes details such as the distance between two iterations of loops. In the process of kernel optimization, it is advisable to prioritize the optimization of the reported bottleneck. Finally, after achieving the expected performance, a bitstream is generated for the target FPGA. It should be noted that the final stage of the process is a time-consuming one, requiring several hours to complete. The flowchart in fig. 4.8 shows the process of compilation starting from OpenCL source code.

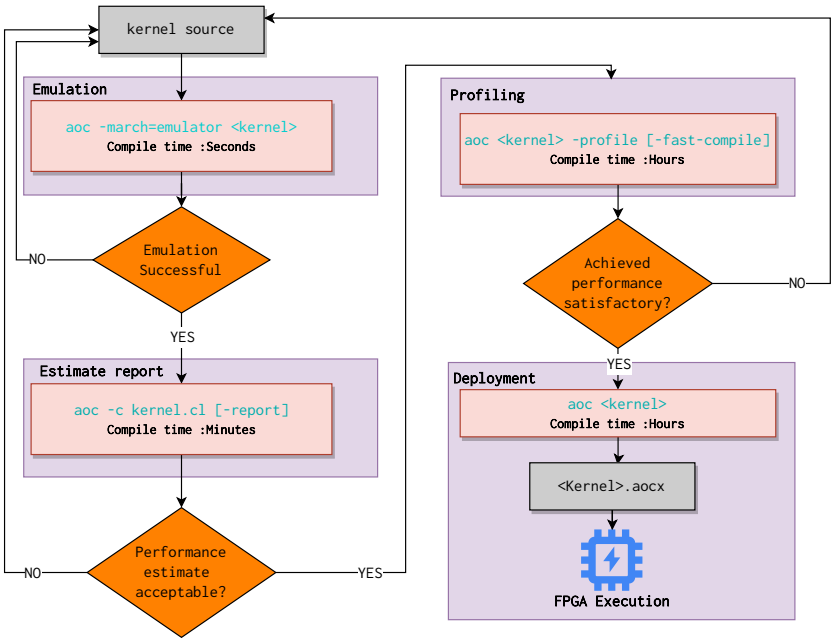


Fig. 4.8 Intel FPGA SDK flow

4.4 Implementation and Optimization for FPGAs

The usage of a high-level implementation-independent model written in OpenCL, C, or C++ brings dual benefits to the FPGAs. On one side, it enables the designers to generate an application-specific hardware architecture instead of using the fixed datapath of a CPU or GPU. On the other side, it brings high-level programming capabilities to hardware design. In order to use the FPGA device efficiently for accelerating an application, the computation bottlenecks have to be identified and then offloaded on the accelerator device, specifying them as kernels. Calculating $H_{u,s,n,m}(\tau,t)$ with (3.26) for different combinations of the input parameter (u,s,n,m) requires extensive computations. This will increase the simulation time significantly and will limit the number of input parameter combinations that can be explored, while still using a reasonable amount of execution time. However, (3.26) offers a very high level of parallelism that can be exploited to significantly speed up the computation using a GPU or FPGA.

4.4.1 Loop based optimizations

Since the channel model considers multiple antennas and scatterers, and hence multiple paths, the implementation is organized as a set of nested loops, often without inter-iteration dependencies (also known as "doall" loops). To exploit the available parallelism, however, the designer has to provide explicit optimization directives and often restructure the original CPU-oriented code, because the out-of-the-box optimization of the HLS tools is insufficient, as discussed below. In the following, we briefly discuss the main loop-based optimization techniques.

Loop pipelining

When a loop is sequentially executed, the next input data are accepted after the previous computation has been fully completed. Some of the resources however can be used much more efficiently by organizing the computation in stages. Pipelining is a form of computation parallelism that splits a sequential operation chain into several stages and introduces storage elements (SRAM or flip-flops) to store the intermediate results. Pipelines are characterized by two primary attributes, namely *latency* and *initiation interval (II)*. Latency is the total number of clock cycles elapsed for input data to reach the exit point. II or *gap* is the number of clock cycles that must elapse before the loop can accept new input data. For a pipeline with II II and latency L that executes N iterations, the total execution time T when operating at frequency f can be described as in [93]

$$T = \frac{L + II \cdot (N-1)}{f}. \tag{4.1}$$

If a design includes two or more chained pipelines, also known as task-level pipelining, the overall II is determined by the slowest one. To achieve maximum performance for a large number of iterations N, it is typically desirable to reduce the II and implement deep pipelines with many stages, hence reducing the overall execution time. Figure 4.9 shows execution of code in listing 4.1 in sequential and pipelined manner.

```
1 void func(m,n,0){
2     for(i=0;i<=2;i++){
3         Read_op;
4         Compute_op;
```

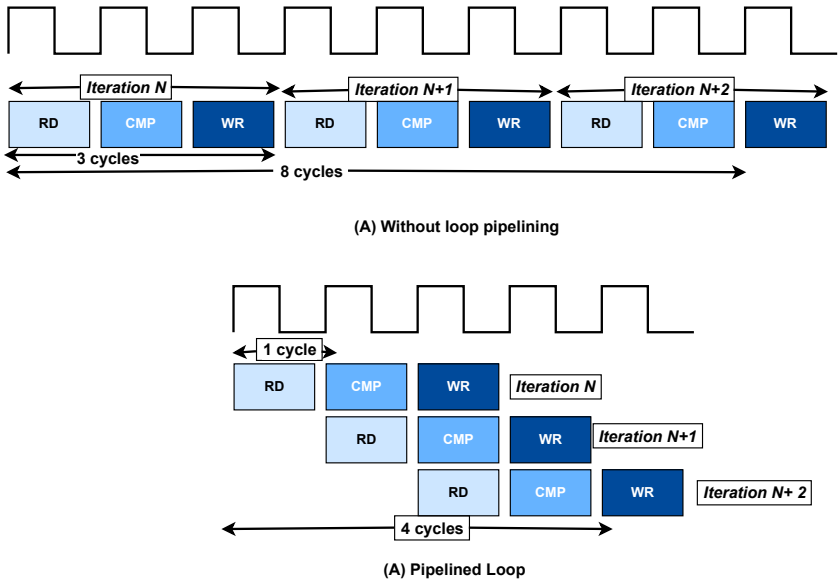


Fig. 4.9 Loop pipelining

```
5           Write_op;
6       }
7   }
```

Listing 4.1 Loop pipelning example

Loop pipelining can be specified in OpenCL kernels with `__attribute__((xcl_pipeline_loop(N)))`, for C/C++ kernels in Xilinx Vitis [91] with `#pragma HLS pipeline II=<N>`, or for Intel FPGA SDK for OpenCL [88] with `#pragma II = <N>`. Pipelining may slightly increase the resource usage due to insertion of extra control logic and intermediate storage elements, but it generally increases the overall design throughput by decoupling it from iteration latency.

Loop unrolling

If there are no data dependencies among the iterations of the loop, the loop execution performance can be improved by executing multiple iterations in parallel. For such a "doall" loop with trip count N, a theoretical speedup of N times, with an increase of resources also by a factor of N, can be achieved by dispatching all the iterations in parallel. If an increase by N of the overall resources is not acceptable, often unrolling is applied partially by creating X copies of the unrolled loop body, where $X < N$. A

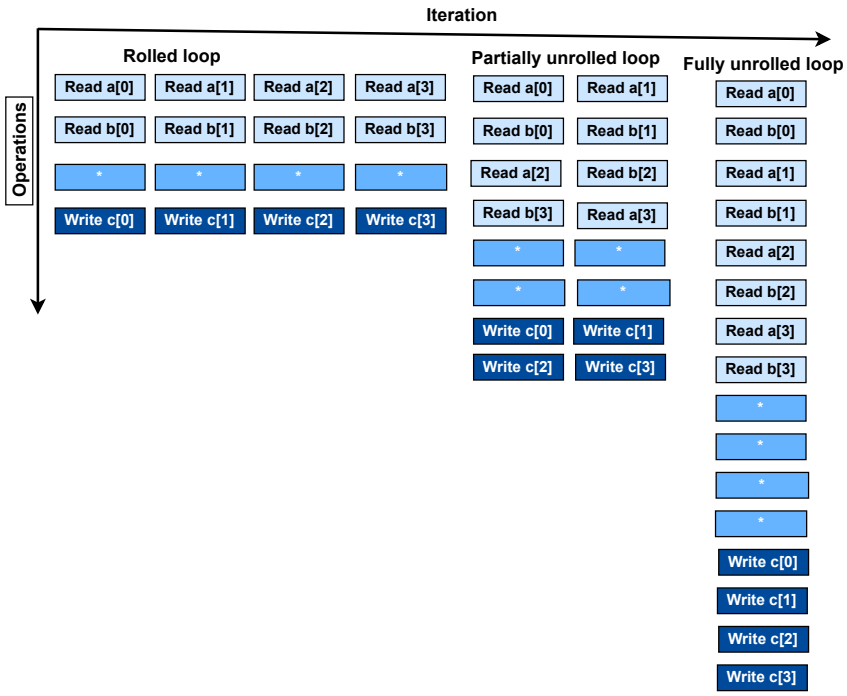


Fig. 4.10 Loop unrolling/vectorization

loop can be fully or partially unrolled depending upon the performance requirements and resource or data availability. Loops can be unrolled by using the `#pragma HLS unroll factor=N` in Vitis HLS or `#pragma unroll N` in Intel HLS, where N is the required number of iterations to be executed in parallel. Figure 4.10 shows the execution of code in listing 4.2 in rolled, partially unrolled, and fully unrolled fashion. Note that unrolling increases the data access parallelism of the loop as well, hence it requires memory architecture restructuring, as discussed below, to achieve the best performance.

```
1 void foo(...){ ...
2     for(i=0;i<=3;i++){
3     a[i]=b[i] * c[i];
4     }
5 }
```

Listing 4.2 Loop unrolling example

Loop tiling

FPGAs have limited on-chip storage resources, which are results required by a given algorithm. In that case, if each iteration of a given loop uses different input, intermediate, and output data, it is possible to split the loop into two nested loops, where the innermost requires a manageable amount of on-chip storage, and transferring only the required data on-chip at each iteration of the outer tiled loop [94]. For a better understanding of the tiling based optimization, listing 4.3 shows an example of nested loops with a large trip count and hence a larger memory footprint.

```
void foo(...){ ...
    for(i=0;i<M;i++){
        for(j=0;j<N;j++){
            a[j] = work(i,j);
                ....
        }
    }
}
```

Listing 4.3 Nested loops example

Loop tiling is applied as shown in listing 4.4, resulting in a smaller memory footprint. This optimization can be used to add support for larger designs on platforms with limited memory resources.

```
void foo(...){ ...
TILE_SIZE =T
    for(j1=0;j1<M;j+=T){
        for(i=0;i<N;i++){
        //smaller memory footprint loop
            for (j2=0; j2<min(M-j1,T); j2++){
                    ....
            }
        }
    }
}
```

Listing 4.4 Tiled loops example

Loop flattening/coalescing

Nested loops can be coalesced into a single loop to improve performance by reducing the overhead of nested loop control. However, in both HLS tools that we consider this can only be done automatically for loops where there is no logic specified between the loop statements, only the innermost has a body and all loop bounds are constant except for the outermost loop bound, which can be variable. listing 4.5 shows the coalesced structure of nested loops in listing 4.3. In Vitis HLS the `#pragma HLS loop_flatten` must be specified inside each coalesced loop, while on the Intel platform, the loops can be coalesced using `#pragma loop_coalesce <loop_nesting_level>` on the outermost loop.

```
void foo(...){ ...
    for(k=0;k<N*M;k++){
        i= k / M;
        j= k % M;
        a[i][j] = work(i, j);
    }
}
```

Listing 4.5 Coalesced loops

4.4.2 Memory oriented optimizations

Off-chip DRAM is required to store most input and output data for the channel model and to communicate with the host. However, DRAM accesses are much slower than the on-chip SRAM accesses (SRAM is also called BRAM on FPGAs). Hence, to compute the CIR with sufficient performance, the input parameters are read from external DRAM to on-chip memory and then accessed repeatedly on-chip by the unrolled pipelined loop bodies. The computation results are written back using the same strategy.

To support the loop optimizations discussed above, the memory hierarchy and off-chip memory interfaces must be optimized. These optimizations include array partitioning, reshaping, banking, and resource allocation:

Data reuse Exploitable parallelism on FPGAs in most cases is limited by the number of off-chip memory ports. If there are multiple accesses to the same data, data

reuse can be exploited by storing them in on-chip buffers, which have low latency and thus reduce total access time [95].

Memory access separation If the innermost loop accesses slow external DRAM frequently, it will have low performance due to off-chip latency and bandwidth limitations, as mentioned above. By separating these memory transfers from computations, they can be both optimized separately to achieve maximum throughput.

Buffering If memory is accessed deep inside the code, it may use the bandwidth inefficiently and degrade the timing and energy performance. To reduce these penalties, access to global memory can be made ahead of the actual kernel computation usage. These accesses read memory in bursts into deep buffers using wider DRAM interfaces than the actual model data type (double-precision floating-point) to fully exploit the parallelism offered by the on-chip DRAM controllers. Performance can also be improved by clocking such memories at higher frequencies (pumping) than the PE.

Memory banking/striping Modern memory interfaces provide access through multiple banks with dedicated access channels, e.g., High-Bandwidth Memory (HBM) lanes or Double Data Rate (DDR) channels. Hence, the access bandwidth of an array can be increased by striping it across the different memory interfaces (banks) available on the board. The same considerations apply to on-chip BRAM banks to increase the on-chip memory bandwidth to match the requirements of the data computations. In HLS, this kind of on-chip memory striping must be performed explicitly by inserting modules to manage data from multiple interfaces. To split data across N banks, on the Intel platform is used the `__attribute__((numbanks(N))` directive on the local memories. In the Xilinx Vitis platform, a memory can be either partitioned completely (into registers) or in a cyclic or block manner using `#pragma HLS array_partition variable=<name> type=<type> factor=<int> dim=<int>`. For off-chip DRAM, on the other hand, a single array must be broken by the designer explicitly into multiple sub-arrays mapped to different HBM or DDR channels, because currently there is no support for HLS automated or aided off-chip memory striping.

Regular memory accesses. Irregular and unaligned access to memory subsystems, in particular to DRAM, leads to severe performance penalties. Hence, memory accesses must be kept carefully aligned, so that they can be combined (*memory coalesced*) by packing the transactions into a single request and making efficient use of the DDR and HBM bandwidths.

In the next chapter, we demonstrate employing a combination of these optimizations to the 3GPP 3D channel model for 5G-NR. The experiment is performed for Xilinx and Intel FPGA platforms, which results in a reduction in execution latency and improved quality of results with respect to the initial CPU-oriented version of the channel model.

Chapter 5

FPGA Acceleration of 3GPP 3D Channel Model

Accurate 5G channel model simulations require very high computational effort and incur long execution times on general-purpose processors. Hardware acceleration of such functions is the way to speed up the execution, reducing the simulation time. Hardware accelerators based on FPGAs improve the runtime performance and system energy efficiency of computationally intensive accurate channel simulators with respect to both CPUs and GPUs [96]. FPGAs can achieve better fine-grained parallelism by customizing the computing engines and memory hierarchy. Civerchia *et al.* [97] studied the optimization of OpenCL designs implementing OFDM module in the 5G stack on FPGA platforms. Alimohammad *et al.* [98] proposed an implementation on FPGA of infinite impulse response (IIR) models for Rayleigh fading channels. Xiao *et al.* [99] studied the use of FPGAs in 5G combined with the neural network optimizations.

Several GSCM emulators have been reported in the literature [100, 101], each with one or more application-specific target scenarios. Hofer *et al.* [100] proposed a parameterized GSCM emulator for FPGAs. It splits the channel into several stationary regions with fixed Doppler frequencies, hence it is not suitable for fast time-varying models like those used in vehicular mobility scenarios. Another emulator for 3D GSCM for fixed-to-mobile channels is presented in [101]. The channel emulator presented in [102] consider a linearly changing Doppler frequency in the stationary regions, but it has non-continuous output fading and hence suffers from accuracy

loss. A ray-tracing-based channel emulator is proposed in [103] with support for dual mobility. The proposed technique relies on pre-computed ray coefficients, hence it introduces errors in the ray amplitudes. Authors in [104] proposed a technique to accelerate the 3GPP channel model by reducing its computational complexity. It considers a single sub-path, which lowers the accuracy, limiting its applicability to real propagation environments. As discussed, most techniques proposed in the literature have some limitations in terms of either accuracy or potential areas of application.

This work started from the application requirements of the Innovation Department of TIM, a major Italian telecommunication provider. Their goal is to exploit the accuracy and generality of the 3GPP GSCM [2] to study the evolution of the radio standard and to maximize the planning quality of mobile networks by means of fast simulation tools, leveraging advanced methods and optimizations for acceleration on FPGA platforms. The key contribution of this work are;

- The work originated from the application needs of TIM's Innovation Department, a leading Italian telecommunication provider.
  Goal: Utilize the accuracy and generality of the 3GPP GSCM to study radio standard evolution and enhance mobile network planning quality.
  Approach: Use fast simulation tools and advanced methods optimized for FPGA platforms.
- The channel model, initially designed for general-purpose CPUs, was adapted for:
  - Xilinx FPGA acceleration platforms.
  - Intel FPGA acceleration platforms.
- A combination of FPGA optimization techniques were applied.
- The author examined the effort needed to utilize various synthesis tools for these platforms.
  - Performance optimization relative to a channel model for general-purpose CPUs.
  - Analyze energy per computation reduction compared to CPU platform implementations.
- The proposed techniques enable the development of fast, efficient, and accurate communication channel model on FPGA platforms.
- The thesis delves into:
  - Using high-performance FPGAs to speed up the channel model in radio link simulators through diverse multi-objective optimizations.

- Enhance the code structure and memory architecture of the 5G-NR channel model on FPGA platforms.
- Maximize parallelism and align memory access with computational capabilities using HLS design environment capabilities.
- Performance analysis of different HLS tools following similar optimization flows.
- Integration of the accelerated channel model into a MATLAB-based simulation system using a socket-based client/server architecture.
- To facilitate shared use by multiple research groups through remote access to accelerated channel simulator.

Part of the work described in this chapter has been already published in Nasir Ali Shah, Mihai T Lazarescu, Roberto Quasso, Salvatore Scarpina, and Luciano Lavagno. FPGA Acceleration of 3GPP Channel Model Emulator for 5G New Radio. IEEE Access, 10:119386–119401, 2022.[105].

5.1 Channel Coefficients Calculation using FPGA

To generate the channel response, the equation for CIR calculation is split into two parts; antenna field pattern and cluster information denoted by $\mathbb{C}$, and user mobility and antenna location information denoted by $\mathbb{V}$ for LOS and NLOS scenarios separately. For NLOS scenario, the cluster information form (3.21) can be calculated as follows.

$$\mathbb{C}_{u,s}^{NLOS} = \underbrace{\sqrt{\frac{P_n}{M}}}_{Power} \underbrace{\begin{bmatrix} F_{\mathrm{rx},u,\theta}\left(\theta_{n,m,\mathrm{ZOA}}, \phi_{n,m,\mathrm{AOA}}\right) \\ F_{\mathrm{rx},u,\phi}\left(\theta_{n,m,\mathrm{ZOA}}, \phi_{n,m,\mathrm{AOA}}\right) \end{bmatrix}^T}_{\text{RX Antenna Pattern}(F_{\mathrm{Rx}})} \times \underbrace{\begin{bmatrix} e^{j\Phi_{n,m}^{\theta\theta}} & \sqrt{\kappa_{n,m}{}^{-1}}\, e^{j\Phi_{n,m}^{\theta\phi}} \\ \sqrt{\kappa_{n,m}{}^{-1}}\, e^{j\Phi_{n,m}^{\phi\theta}} & e^{j\Phi_{n,m}^{\phi\phi}} \end{bmatrix}}_{\text{XPR}} \times \underbrace{\begin{bmatrix} F_{tx,s,\theta}\left(\theta_{n,m,\mathrm{ZOD}}, \phi_{n,m,\mathrm{AOD}}\right) \\ F_{tx,s,\phi}\left(\theta_{n,m,\mathrm{ZOD}}, \phi_{n,m,\mathrm{AOD}}\right) \end{bmatrix}}_{\text{TX Antenna Pattern}(F_{\mathrm{Tx}})} \tag{5.1}$$

For location and mobility information in NLOS scenarios, $\mathbb{V}_{n,m}^{NLOS}$ can be termed as

$$\mathbb{V}_{n,m}^{NLOS} = e^{j2\pi \frac{\hat{r}_{\mathrm{tx},n,m}^{T} \cdot \bar{d}_{\mathrm{rx},u}}{\lambda_0}} \times e^{j2\pi \frac{\hat{r}_{tx,n,m}^{T} \cdot \bar{d}_{tx,s}}{\lambda_0}} \tag{5.2}$$

The array response of the transmitting antenna is $e^{j2\pi \hat{r}_{\mathrm{tx},n,m}^{T} \cdot \bar{d}_{\mathrm{tx},s}}$ and for the receiving antenna is denoted by $e^{j2\pi . \bar{d}_{\mathrm{rx},u}}$, where $\hat{r}_{\mathrm{tx},n,m}$ and $\hat{r}_{\mathrm{rx},n,m}$ represents the wave vectors while $\bar{d}_{\mathrm{rx},u}$ and $\bar{d}_{\mathrm{tx},s}$ represents the respective location vector. When considering user mobility, it is important to note that every ray experiences a phase shift, denoted as $\bar{v}$, as a result of the Doppler effect.

Similarly, the same procedure can be performed for LOS scenario as in (3.27). The cluster information vector $\mathbb{C}_{u,s}^{LOS}$ can be defined as

$$\mathbb{C}_{u,s}^{LOS} = \underbrace{\begin{bmatrix} F_{rx,u,\theta}\left(\theta_{LOS,ZOA}, \varphi_{LOS,AOA}\right) \\ F_{rx,u,\varphi}\left(\theta_{LOS,ZOA}, \varphi_{LOS,AOA}\right) \end{bmatrix}^{T}}_{\text{RX Antenna Pattern}(F_{\mathrm{Rx}})} \times \underbrace{\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}}_{XPR} \times \underbrace{\begin{bmatrix} F_{tx,s,\theta}\left(\theta_{LOS,ZOD}, \varphi_{LOS,AOD}\right) \\ F_{tx,s,\varphi}\left(\theta_{LOS,ZOD}, \varphi_{LOS,AOD}\right) \end{bmatrix}}_{\text{TX Antenna Pattern}(F_{\mathrm{Tx}})} \tag{5.3}$$

Location and mobility information vector $\mathbb{V}_{n,m}^{NLOS}$ in LOS scenarios, can be termed as

$$\mathbb{V}^{LOS} = e^{-j2\pi \frac{d_{3\mathrm{D}}}{\lambda_0}} \times e^{j2\pi \frac{\hat{r}_{rx,LOS}^{T} \cdot \bar{d}_{rx,u}}{\lambda_0}} \times e^{j2\pi \frac{\hat{r}_{tx,LOS}^{T} \cdot \bar{d}_{tx,s}}{\lambda_0}} \tag{5.4}$$

The pseudocode for channel coefficient calculation is described in algorithm 1

The overall channel response can be expressed as the sum of the LOS channel coefficient and the NLOS coefficients as

$$H_{u,s}(t) = H_{u,s,1}(t) + \sum_{n=1}^{N} \sum_{m=1}^{M} H_{u,s,n,m}(t) \tag{5.5}$$

In wireless communication, the signal $x(t)$ that is transmitted experiences a time delay of $x(t-\tau)$ upon arrival at the receiver. The signal that is received is composed of numerous reflections, which lead to the creation of nearly identical replicas of the signal that was initially transmitted. The wireless channel impact can be computed

Algorithm 1 Implementation of the channel impulse response generation in 3GPP channel model

Inputs: $F_{u,s}$, $\hat{r}_{n,m}$, $\kappa_{n,m}$, $\bar{v}$, t
Output: $H_{u,s}(t)$

for $u = 0$ to U **do**
  for $s = 0$ to S **do**
    calculate for $\mathbb{C}_{u,s}^{NLOS}$ as in (5.1)
    if LOS **then**
      calculate for $\mathbb{C}_{u,s}^{LOS}$ as in (5.3)
    end if
  end for
end for
for $n = 0$ to N **do**
  for $m = 0$ to M **do**
    calculate for $\mathbb{V}_{n,m}^{NLOS}$ as in (5.2)
    if LOS **then**
      calculate for $\mathbb{V}^{LOS}$ as in (5.4)
    end if
  end for
end for
for $u = 0$ to U **do**
  for $s = 0$ to S **do**
    for $n = 0$ to N **do**
      for $m = 0$ to M **do**
        $H_{u,s}(t) = \mathbb{C}_{u,s}^{NLOS} \times \mathbb{V}_{n,m}^{NLOS} \times e^{j2\pi\bar{v}} \times t$
      end for
      if LOS **then**
        $H_{u,s}(t) = H_{u,s}(t) + \mathbb{C}_{u,s}^{LOS} \times \mathbb{V}^{LOS} \times e^{j2\pi\bar{v}} \times t$
      end if
    end for
  end for
end for

by the convolution of the channel response $H_{u,s}$ with the input signal $x(t)$ as

$$y(t) = \sum x(t-\tau) H_{u,s}(t,\tau) = (x \circledast H)(t) \tag{5.6}$$

where $\circledast$ denotes convolution operation. The complexity of this operation is heightened due to the fact that during performance evaluation, the input signal $x(t)$ is typically comprised of lengthy vectors. A large size ring buffer is required to store delayed inputs $x(t-\tau)$. Algorithm 2 shows the pseudocode for the application of channel response to the input signal

Algorithm 2 Application of the channel coefficients

Inputs: $x(t)$, $H_{u,s}(t)$, $H_{u,s}(t-\tau)$, N_s
Output: $y(t)$

for $u = 0$ to U **do**
  for $s = 0$ to S **do**
    for $n = 0$ to N **do**
      $\Delta H(t) = \dfrac{H_{u,s}(t) - H_{u,s}(t-\tau)}{N_s}$
    end for
    for $i = 0$ to N_s **do**
      $R_i = x(i * S + u)$ {load in cyclic buffer}
      for $n = 0$ to N **do**
        if fistSubFrame **then**
          tapV = $H_{u,s}(t)$ {current FIR tap}
        else
          tapV = $H_{u,s}(t) + \Delta H(t)$
        end if
        $Acc(t) = Acc(t-\tau) + R_i * tapV$ {convolution as in (5.6)}
      end for
      $y(t) = y(t-\tau) + Acc(t)$
    end for
  end for
end for

5.1.1 FPGA Implementation

When targeting FPGAs using HLS tools, the baseline implementation is often a version of the code that has been developed for CPUs using C/C++. In our case, the channel model was developed in C++ and executed in the MEX co-simulation

environment, with the remaining of the 5G stack being executed inside MATLAB. The design is then ported to the Xilinx Vitis development and Intel FPGA SDK for OpenCL environments.

The design environments and tools used for this study are listed below.

Baseline CPU Platform Uses an *Intel Xeon E5-2660 v4 @2.00GHz* running MATLAB R2021a [106]. The setup consists of a co-simulation environment where the channel model is implemented in C++ and executed in MEX [107], while the rest of the application is being executed in MATLAB.
FPGA Platform 1 Uses the Xilinx Alveo U280 data center accelerator card from *Xilinx UltraScale+* family [108] and consists of three chiplets, also called super logic regions (SLRs), in a single package. This platform contains 30 MB of on-chip UltraRAM (URAM), 4.5 MB of on-chip BRAM, 8 GB of HBM and 32 GB of DDR memory. For fairness of comparison with the other single chiplet platforms, all reported results use only one SLR out of three. Xilinx Vitis Unified Software Platform version 2020.1 is used for the development of the host and kernel code. This platform is referred to as *US+* hereafter.
FPGA Platform 2 Uses the Intel PAC Arria 10GX 1150 [109]. This platform contains 8.2 MB on-chip embedded memory and 8 GB of DDR memory. Intel FPGA SDK for OpenCL version 19.4 is used alongside Intel Quartus Prime Pro 19.2 for the development of the host and kernel code [109, 92]. This platform is referred to as *Arria* hereafter.

The FPGA Platform 1 is based on 16 nm technology node, whereas the FPGA Platform 2 is implemented in 20 nm. Table 5.1 lists the main resources available on these platforms. The final accelerated channel emulator is deployed on these platforms to be used inside the 5G simulation stack.

5.1.2 Roofline model

The Roofline model is a visual performance model used to provide insight into the performance bottlenecks of an application [110]. It is particularly useful for understanding the balance between computation and data movement. When applied to FPGAs, the Roofline model can help in optimizing FPGA designs by identifying performance ceilings and guiding optimization efforts.
To evaluate the performance of the two FPGA platforms, the FPGA Empirical

Table 5.1 FPGA acceleration platforms specification

	US+		Arria
Resource type	**Total**	**Per SLR**	**Total**
FF	2 607 000	869 120	1 708 800
LUTs/ALM	1 303 680	434 560	427 200
DSP	9024	3008	1518
MLAB (KB)	—	—	1587
BRAM (blocks)	4032	1344	2713
BRAM (KB)	18 144	6048	6783
UltraRAM (blocks)	960	320	—
UltraRAM (KB)	34 560	11 520	—

Roofline(FER) benchmarking approach [111] is employed. This approach is derived from the Empirical Roofline Toolkit (ERT) but tailored specifically for FPGA devices [112]. The benchmarking is conducted using HLS tools for FPGA-based accelerators. The purpose of this evaluation is to determine the hardware characteristics of the platforms, enabling the analysis of their architectures using the Roofline methodology. The implementation is based on the Roofline Model and utilizes directive annotated HLS kernels with configurable operational intensity and hardware resource utilization.

Computing architectures are typically defined by a machine balance, denoted as $M_b = \frac{C}{B}$, which represents the ratio of peak compute performance (C) to peak memory bandwidth (B). C refers to the maximum theoretical FLOP/s (floating-point (FP) operations per second), while B represents the maximum theoretical memory bandwidth Byte/s (bytes per second). One of the challenges encountered in the Roofline Model is the fact that FPGA peak performance strongly relies on the specific operations being allocated to the FPGA's available hardware resources. The number of implementable compute cores and the maximum clock frequency are impacted, which in turn affects the theoretical peak performance.

In the context of FPGA platforms, the available resources are commonly divided into two distinct logical components. These components are referred to as the *static* region, responsible for the implementation of hardware related to bitstream loading and PCIe communication, and the *dynamic* region, responsible for the implementation of user designs. This results in a decrease in the total amount of resources that

are accessible for the purpose of implementing the user's design. An FP operation on FPGA can be implemented using different type of resources, such as LUTs or DSPs. The peak achievable compute performance can be limited by either of these resources. To ensure a successful *place and route*, the resource usage should be limited by a factor $\mu < 1$. For an FPGA with total number of available resources R_t, of which R_c are the available for user to implement a FP core with clock frequency f_c, the peak compute performance C of the target FPGA is

$$C = f_c \times \min\left(\frac{R_t^r}{R_c^r} \times \mu_{R^r}\right) \tag{5.7}$$

where r is the resource type available on the target FPGA platform.
Similarly, FPGA platforms offer various types of memories such as DDR, HBM and BRAM. The peak bandwidth B for each of such resource with a bit-width W_r can be computed as [111]:

$$B^r = f_{max} \times W^r \times R_t^r \times \mu_{R^r} \tag{5.8}$$

Where f_{max} is the maximum frequency achieved for the target resource.

5.1.3 ALVEO U280

To improve the timing of the user kernel and ensure place and route, the design was limited to use only one of the three SLRs. SLR0 has a comparable compute and memory resources to that available on the other target FPGA platform, Intel Arria 1150GX. *U280_gen3x16_xdma_202211* platform is used for tests, which leaves 360×10^3 LUTs, 490 BRAMs, 320 URAMs and 2733 DSPs in the dynamic region for user. From the benchmarking of various kernel types on this platform, it is observed that for FP multiplication, 8 DSPs and 167 LUTs for double-precision and 3 DSPs and 104 LUTs for single-precision (SP). Similarly, for FP addition, it utilizes 3 DSPs and 684 LUTs for double-precision and 2 DSPs and 241 LUTs for single-precision. To compute the peak FP performance at $f_c = 300$MHz for a single

SLR, we can use (5.7) as:

$$\begin{aligned} C &= 300\ \text{MHz} \times \min\left(\frac{360.10^3}{164+684} \times 0.7, \quad \frac{2733}{3+8} \times 0.8\right) \\ &= 59 \times 10^9\ \text{FMA/s} \\ &= 119\ \text{GFLOP/s} \end{aligned} \tag{5.9}$$

For single-precision floating-point, by using the values achieved through benchmarking to compute C as;

$$\begin{aligned} C &= 300\ \text{MHz} \times \min\left(\frac{360.10^3}{104+241} \times 0.7, \quad \frac{2733}{2+3} \times 0.8\right) \\ &= 262\ \text{GFLOP/s} \end{aligned} \tag{5.10}$$

In order to measure the peak memory bandwidth while targeting only one SLR, we consider 12 out of total of 32 HBM pseudo-channels. The theoretical maximum bandwidth can be computed using (5.8) as:

$$\begin{aligned} B_{HBM} &= 2 \times 0.9\,\text{GHz} \times 64\ \text{bit} \times 12 \\ &= 172.8\ \text{GB/s} \end{aligned} \tag{5.11}$$

Finally, by means of the values calculated above, computing *machine balance* $M_b = \frac{262\ \text{GFLOP/s}}{172.8\ \text{GB/s}} = 0.69$ making it suitable for memory-bound applications with low arithmetic intensity.

The Alveo U280 has two DDR modules of 16GB each making a total of 32GB clocked at 1.2 GHz. Each of these modules has 4 independent 32-bit memory controllers. By using these values, the peak DDR bandwidth can be calculated as;

$$\begin{aligned} B_{DDR} &= 2 \times f_{max} \times W_{DDR} \times CH_{DDR} \\ &= 2 \times 1.2\,\text{GHz} \times 64bit \times 4 \\ &= 614.4\text{Gbit/s} \\ &= 76.8\text{GB/s} \end{aligned} \tag{5.12}$$

where f_{max} is the frequency of DDR modules, W_{DDR} is the bit-width, and CH_{DDR} is the number of channels multiplied by 2 since the banks are Double Data Rate. This results in the machine balance $M_b = \frac{262\text{GFLOP/s}}{76.8\text{GB/s}} = 3.41$

5.1.4 Intel Arria10 1150GX

This acceleration platform offers various types or resources as reported in Table 5.1. When using the OpenCL based platform for Arria10, the number of resources available for user logic are reduced to 2927×10^3 LUTs, and 2316 BRAMs. From the benchmarking of various kernels on this platform, it is observed that a double-precision FP multiplication requires 4 DSPs and 2238 LUTs while an addition operation is implemented using 1999 LUTs. Thus, by using (5.7), the peak compute performance is

$$\begin{aligned} C &= 240\,\text{MHz} \times \min\left(\frac{2927 \times 10^3}{1999+2238} * 0.7, \frac{1518}{4} * 0.8\right) \\ &= 72.86 \times 10^9 \text{FMA/s} \\ &= 145\ \text{GFLOP/s} \end{aligned} \tag{5.13}$$

To generate a roofline model for this platform for single-precision, values are collected through platform benchmarking. Thus, compute performance can be calculated as;

$$\begin{aligned} C &= 240\,\text{MHz} \times \frac{1518}{1+1} * 0.8 \\ &= 145.73 \times 10^9 \text{FMA/s} \\ &= 291\ \text{GFLOP/s} \end{aligned} \tag{5.14}$$

The second quantity in calculation of machine balance M_b is the peak bandwidth B. Arria10 1150GX FPGA has 2 DDR channels running at 1.2 GHz. Thus, the maximum bandwidth is:

$$\begin{aligned} B_{DDR} &= 2 \times 1.2\,\text{GHz} \times 64\text{bit} \times 2 \\ &= 307.2\ \text{Gbit/s} \\ &= 38.4\ \text{GB/s} \end{aligned} \tag{5.15}$$

For on-chip M20K memory resource, the peak bandwidth is reported as $B_{M20K} = 448 GFLOP/s$. By using the values of peak compute and memory performance, we can build the roofline models. The X-axis represents *arithmetic intensity (AI)* in (FLOP/byte) while the Y-axis shows compute performance (GFLOP/s). The slope

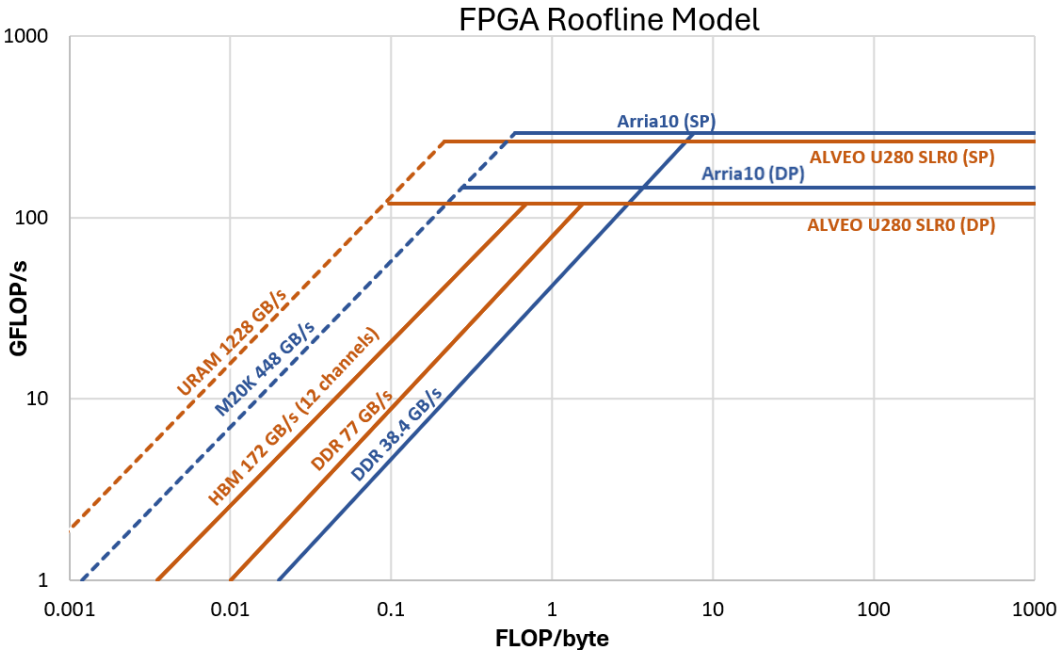


Fig. 5.1 Roofline model for Alveo U280 and Arria 10 FPGAs

lines show memory throughput for various types of resources, where the solid lines represent off-chip memory and dashed lines are used for on-chip bandwidth.

5.1.5 Optimizations for efficient FPGA implementation

The accelerated function (kernel) can be either defined in OpenCL or in C/C++. These kernel modeling styles differ mainly in the way of defining the kernel input parameters and optimization pragmas. OpenCL defines the interfaces to the external DRAM automatically, based on the `__global` memory attribute, whereas in case of C/C++ these are configured via pragmas. Another difference is how optimizations, such as memory partitioning, loop pipelining, and unrolling are specified, and the location where these pragmas should be placed. Finally, OpenCL could in principle allow explicit modeling of data parallelism via WGs and WIs. However, in this project, this opportunity is not exploited because the goal was to exercise finer control over the loop pipelining and unrolling, which is possible only with a single-WI modeling style.

The channel model is implemented as two kernels, where the first part computes the channel coefficients according to Algorithm 1 while the latter applies the coef-

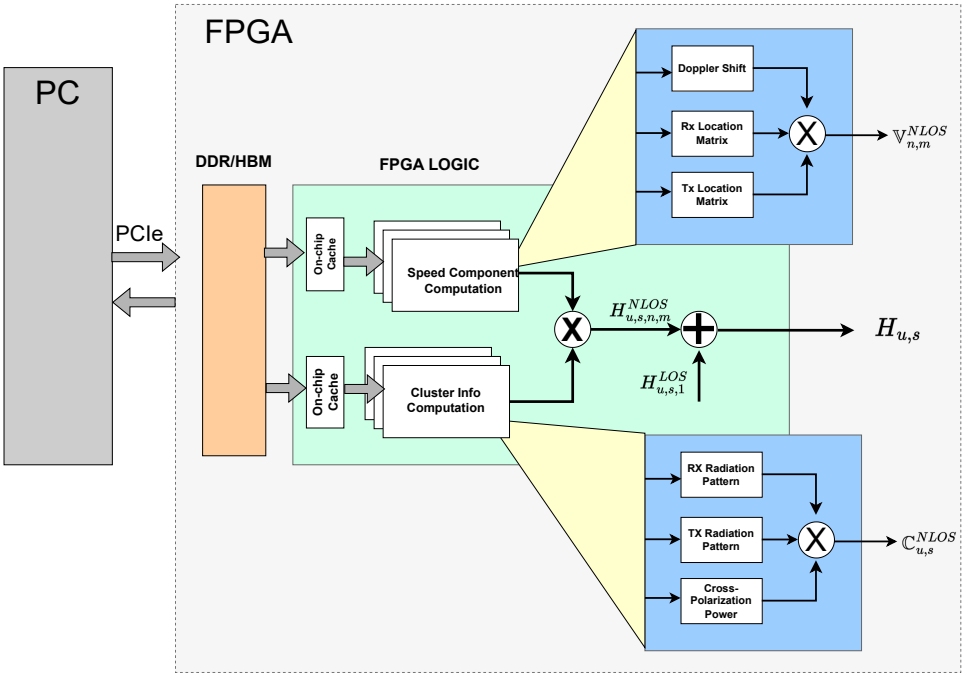


Fig. 5.2 Channel coefficient computation

ficients to each channel sample period. Figure 5.2 To reduce the global memory traffic, the input data is buffed into on-chip local cache ahead of computations. Coefficient recalculation is performed for each time slot. Figure 5.3 applies the channel coefficients to each input stream through a set of FIRs.

Figure 5.4 describes the overall socket-based acceleration architecture used for validation and evaluation in this research. The use of sockets to connect the MATLAB client to the acceleration servers enable to serve multiple remote clients, avoiding having a physical card mounted into the actual physical machine running the instances of MATLAB.

The baseline implementation of the channel model is co-simulated with MATLAB using MEX and used for validation. To accelerate the channel functions using an FPGA which supports complex simulations with higher numbers of antenna elements and more UE speeds, the design is split into two major parts: *host* and *kernel* code. The host code performs tasks related to control and data movement

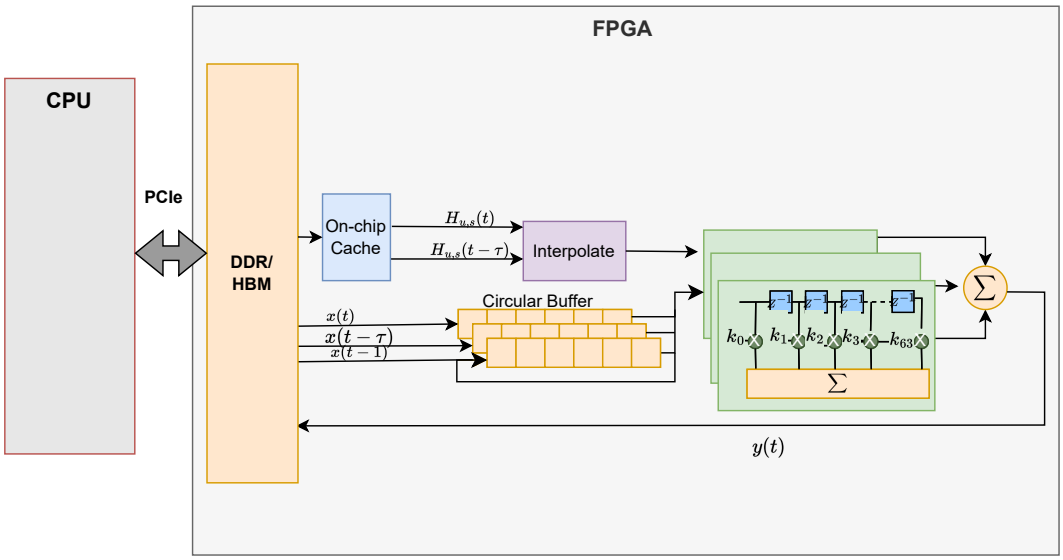


Fig. 5.3 Coefficient application to channel samples

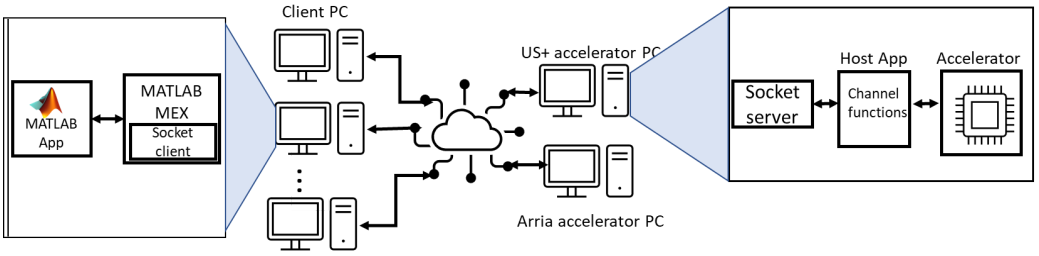


Fig. 5.4 Emulation system setup

such as allocating space on the device memory, receiving data via the socket from a client, launching the kernel and copying results back to the client via another socket. The design is ported to the respective development environments for target FPGA. A new validation step is used to check that the splitting between client code, server host code and server kernel code was performed correctly. This out-of-the-box (OOB) implementation is very inefficient since all the data resides in global DRAM and hence limits the scope of automatic or manual optimizations.

The optimizations adopted in this work are described more in detail in Chapter 4 and can be grouped in the following categories:

Generic HLS optimization These optimizations are generic for any HLS flow and can be adopted on any of the target platforms. These optimizations are further divided into two categories here, highlighting their impact on performance and resource utilization, respectively.

1. **On-chip buffers:** Access to global memory, i.e., off-chip DRAM, is costly in terms of both time and energy. To overcome the memory bottleneck, the data used by the kernel must be copied into low latency on-chip SRAM buffers at the beginning of the kernel execution, and back to DRAM at the end. This brings dual benefits, firstly by issuing wider DRAM access requests than the single words used in the model computation, thus utilizing its full bandwidth, and secondly by reducing the number of such requests by exploiting data reuse. Loop-based optimizations are then applied to make efficient use of on-chip resources. These optimizations include pipelining, unrolling, tiling, and loop coalescing.
2. **Multi-port:** Parallel access to on-chip buffers by unrolled loop bodies is still limited by the number of available ports. To prevent stalling of the computations, these buffers should be partitioned to allow multiple accesses through an adequate number of ports. For example, the FIR coefficients are the result of intermediate computations. These partial results are kept in SRAM buffers with low latency, thus enabling to compute coefficients for all clusters in SIMD fashion, hence reducing the total latency.

Application-specific optimizations (ASO) These types of optimizations are specific to the channel model application and may require modifications of the original CPU oriented algorithm to achieve the best performance. For example, off-chip memory accesses are not aligned initially and hence result in poor bandwidth utilization because of memory stalls. Another optimization is exploiting the algorithm structure

to reduce the number of arithmetic operations. In our case, the iterative computation of an arithmetic sequence is replaced with its closed form, which required only multiplication by a constant, and thus reduced both floating point operations and on-chip memory accesses. This also removed false inter-iteration dependencies and enabled unrolling to parallelize computations.

1. **Regular access pattern:** To improve performance using the optimization mechanisms provided by HLS tools such as loop pipelining, unrolling and loop-flattening, it is essential to enable the tools to understand the access pattern of the underlying design. Random or irregular memory access can result in poor bandwidth utilization, limiting the achievable speedup. In Algorithm 1, line 11 performs memory access to random memory locations depending on the position of cluster scatterers, which is determined at runtime. This is a bottleneck to the achievable parallelism. To solve this, memory accesses are separated into different banks for each cluster.
2. **Closed-form computation:** To reduce the number of arithmetic operations, and hence the memory accesses, the algorithm structure can be exploited. In our case, the iterative computation of an arithmetic sequence was replaced with its closed form, which requires only multiplication by a constant, thus reducing both the floating point operations and the on-chip memory accesses.
3. **False dependence removal:** Memory dependencies occur when a single memory location is both read and written, or written multiple times, within a section of code (typically a loop body). While the true dependencies must be preserved to hold the correctness of computations, false dependencies are the result of *conservative estimations performed by an HLS tool when it cannot exactly analyze access sequences. These dependencies can never occur during actual execution of the code* and can be resolved after a careful manual analysis of the access patterns in the code. False dependence removal pragmas for HLS tools are used to identify such dependencies and improve the effectiveness of loop transformations.

Platform-specific optimizations (PSO) Some FPGA platforms may offer some extra resources, such as off-chip HBM and on-chip URAM on the Xilinx Alveo U280, which can be used to further increase the maximum achievable performance. HBM can be used by specifying a separate interface for each global memory array, and can help to reduce memory contention and bank conflicts. URAM is a special kind of memory that is wider and deeper than the BRAM and can be used to store

Table 5.2 Summary of channel model emulator parameters

Number of polarization	$POL = 2$
Number of elements on H-Planes (Number of Columns)	$N = 4$
Number of elements on V-Planes (Number of Rows)	$M = 4$
Cluster delay line type	CDL B
Model for correlation	Low
Model for delay spread	Very short
Link type	Downlink
Number of subcarrier in Frequency dimension	2048
Carrier frequency	3600 MHz
Sampling frequency	122.88 Hz
Number of TX antennas	$N_{\mathrm{TX}} = 32$
Number of RX antennas	$N_{\mathrm{RX}} = 2$
Oversampling factor	4
Number of clusters	23
Number of rays	20
User speed	$120\,\mathrm{km\,h^{-1}}$
Number of symbols per link	122 880
Transmission time interval	0.25 ms

large data structures. Since in OpenCL currently, it is not possible to control these features, the author port the kernel to C++, which allows for more control over these optimizations, while losing some portability between different FPGA vendors that is afforded by OpenCL. This version of the kernel achieves a much more balanced resource utilization and hence would allow the creation of multiple instances of the kernel on the target FPGA, to simulate multiple channel models concurrently and independently.

5.2 Results and Analysis

The performance analysis of the accelerators before and after optimizations for the two target platforms is performed in this study. The reference parameter values from the 3GPP specification [2] are used for this phase. Table 5.2 lists some of these parameters and the chosen propagation condition for the channel emulator. To measure the efficiency of the accelerated designs, performance metrics based on resource

utilization, latency, and energy consumption are used. The OOB implementation of the channel model on the FPGA platforms is purely a synthesizable version of the original code targeted for CPU execution. Although it is functionally correct, it is very inefficient in terms of computation and memory access. Data reside in the global DRAM and even though the data access is mostly sequential, none of the HLS tools was able to optimize the performance of the memory accesses and exploit the abundant opportunities for data reuse. To reduce frequent accesses to global memory, the data are copied to on-chip buffers before starting the computation core of the kernel and copied back to the global memory at the end of the execution. To achieve computation parallelism, the data should be accessible in parallel. This is limited by the number of access ports available on the requested memory. Memory partitioning allows multiple accesses in parallel at the cost of increased resource utilization. The final implementation combines all these optimizations with algorithm modifications to improve the regularity of the memory accesses and thus simplify the addressing logic.

5.2.1 Latency

Since the primary task of this research is the acceleration of the channel model, the main focus is the reduction of the overall latency of the kernel execution. Table 6.3 lists the latency of the kernel on the baseline CPU and on the various acceleration platforms, using a single SLR for the US+. For comparative analysis, this study reports the speedups achieved after the application of each optimization. In the baseline implementation, the CPU cache provides very good DRAM access bandwidth without any programming effort, but the maximum achievable performance is limited by the number of available computational resources. Hence, the OOB implementation on the two FPGA platforms have lower performance than on the CPU, mainly because of time-consuming memory access requests, since all the data reside in off-chip DRAM. This cost is significantly reduced by copying the data into on-chip buffers before the channel model computation starts. Parallel access to these buffers is still limited by the availability of access ports and hence prevents many HLS optimizations, such as pipelining and unrolling. To overcome this, partitioning on-chip memory, which effectively means using several banks, is used to increase the number of access ports on these buffers and thus enable the HLS tools to schedule more access requests in parallel. Memory bandwidth

Table 5.3 Kernel latency and speed up achieved compared to OOB and CPU implementation, with ASO, PSO, HBM, and URAM

Platform	**Optimization**	**Latency** (s)	**Speedup OOB** (X)	**Speedup CPU** (X)	**Frequency** (MHz)
CPU	Baseline	5.010	32.00	1.00	2000
US+	OOB	160.400	1.00	0.03	285
	On-chip buffers	4.110	39.00	1.02	145
	Multi-port	23.860	7.00	0.21	145
	ASO	0.080	2083.00	65.00	210
	PSO(HBM)	0.033	4860.00	151.00	250
	PSO(HBM+URAM)	0.027	5531.00	173.00	275
Arria	OOB	655.000	1.00	0.01	185
	On-chip buffers	28.000	23.00	0.18	119
	Multi-port	6.390	103.00	0.78	111
	ASO	0.053	12 359.00	95.00	192

utilization, however, is still poor due to the unaligned access patterns, which require a significant amount of multiplexing. ASO are applied to overcome the limiting factors of application, yielding overall **95X** and **65X** speedups on the US+ and Arria platforms respectively compared to the baseline CPU implementation. The regular access pattern enables burst access of on-chip data and hence reduces the overall number of memory access transactions. Closed form computation replaces two read and one write access with integer multiplications and reduces the memory access by 15 %. The last optimization of ASO is highlighting the false memory dependence to the HLS compiler by separating the access targets, which helps achieve II of 1 in the critical loops and hence reduce the overall latency to half.

At this stage, the maximum achievable performance is limited by the number of available FPGA on-chip resources. In addition, the Xilinx US+ platform offers high capacity URAMs and HBM with a number of interfaces that are exploited next, through PSO. To increase the number of global memory access ports, separate HBM interfaces are used to reduce bus and memory controller contention on interfaces and improve bandwidth utilization. The complex input and output data structures of the channel model were split into real and imaginary parts and were assigned each to a

separate HBM channel. A total of 12 HBM channels was used to avoid the interface stalling and to fully pipeline the loops in the implementation. This leads to a speedup of **151X**. Performance is further improved by using URAM resources for some data structures, to better balance the resource utilization of the design and yield an overall **173X** speedup compared to the baseline CPU based implementation.

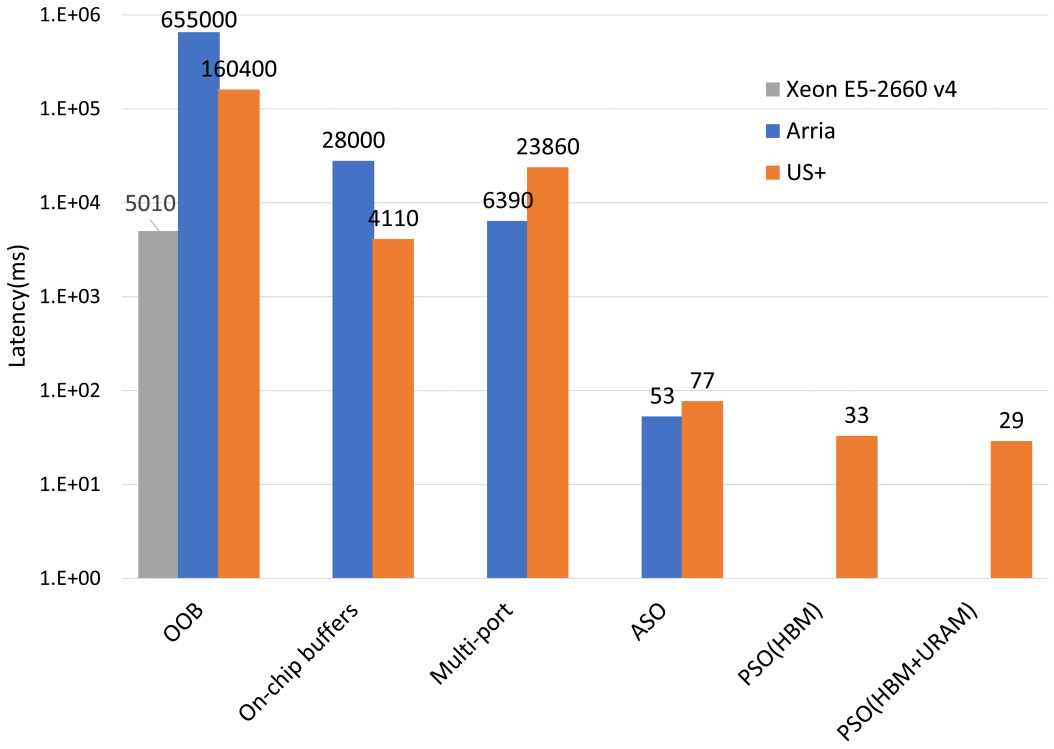


Fig. 5.5 Average execution latency (log scale)

Figure 5.5 shows the latency of the design after various optimizations. As this study follows a step-by-step procedure, the new optimizations are added on top of the ones applied in earlier steps.

To compare the achieved performance with the peak performance of the target platform, we analyzed the channel model performance through benchmarking of the hardware execution. For the channel parameters in Table 5.2, the total number of floating-point operations is 2.53 GFLOP and the total data transfer is 1.26 GB. We

can compute the *Arithmetic Intensity*(AI) as;

$$\begin{aligned}
\text{AI (FLOP/byte)} &= \frac{\text{Total FLOPS (GFLOP)}}{\text{Total Data (GB)}} \\
\text{AI(US+)} &= \frac{\frac{2.53\ (GFLOP)}{0.027\ (s)}}{1.26\ (GB)} = 2.01 \\
\text{Performance (US+)} &= \frac{\text{FLOPs}}{\text{Execution time}} \\
&= \frac{2.53}{0.027} = 93.8\ \text{GFLOP/s} \\
\text{Performance (Arria)} &= \frac{2.53}{0.053} = 46.9\ \text{GFLOP/s}
\end{aligned} \tag{5.16}$$

Roofline model can be used to analyze the achieved performance and determine how efficiently the underlying resources are being utilized.

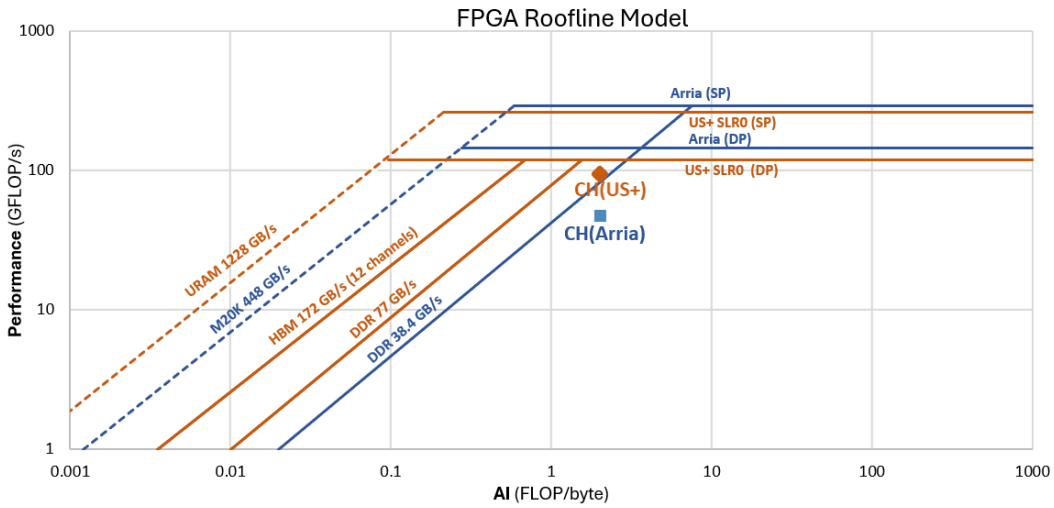


Fig. 5.6 Roofline performance on the target platforms

Figure 5.7 reports execution time for various antenna configurations in downlink and uplink communication scenarios

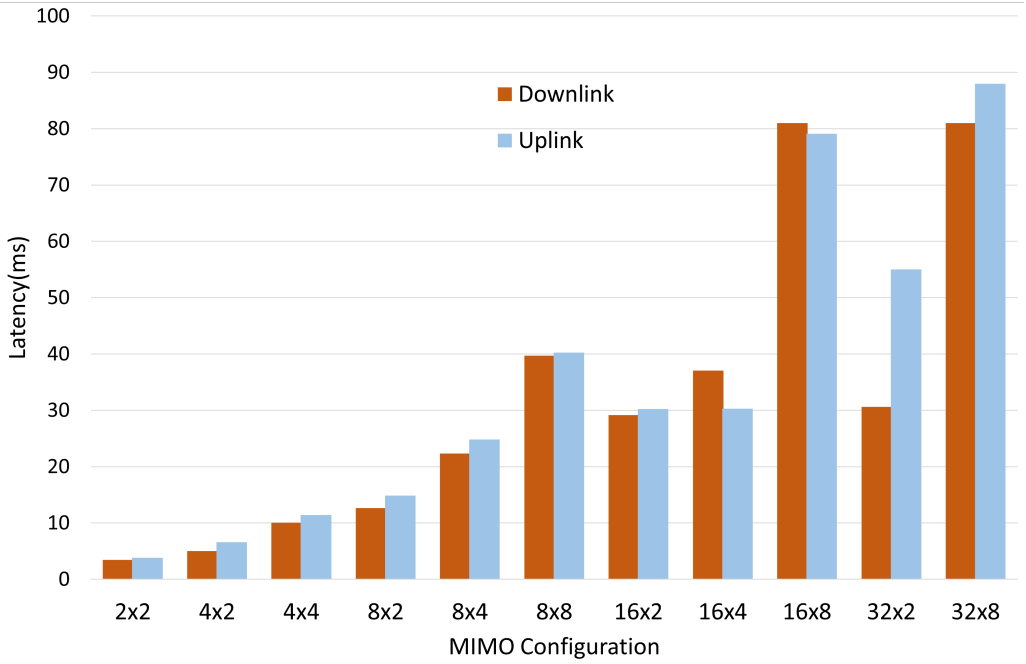


Fig. 5.7 Latency for Uplink and Downlink MIMO configuration

5.2.2 Resource utilization

Optimization pragmas affect the resources used by the accelerated function. Since in the OOB implementation, all data reside in off-chip DRAM, resource utilization is very low for the US+ platform. The HLS tool for the Arria platform on the other hand tries to optimize automatically the memory accesses, but fails due to the unaligned access patterns and inter-iteration dependencies. Table 5.4 lists the resource usage for the various designs. The total available on-chip resources are listed in Table 5.1. To make the implemented designs comparable on FPGA, the available resources on *US+* platform were limited to be allocated on SLR0 only, which has comparable resources to that on Arria platform.

Platform	Optimization	Utilized resources					SLR Utilization (%)				
		FF	LUT	BRAM	URAM	DSP	FF	LUT	BRAM	URAM	DSP
US+	OOB	188901	129038	220	0	59	22	30	16	0	2
	On-chip buffers	338349	207833	1382	0	30	39	48	103	0	1
	Multi-port	318186	190510	1380	0	43	37	44	103	0	1
	ASO	427206	288237	1066	0	816	49	66	79	0	27
	PSO(HBM)	407674	279657	743	0	1530	47	64	55	0	51
	PSO(HBM+URAM)	409292	280272	231	60	1530	47	64	17	19	51
Arria	OOB	227896	123420	702	—	68	13	29	26	—	4
	On-chip buffers	83891	44059	297	—	46	5	10	11	—	3
	Multi-port	86414	45215	354	—	46	5	11	13	—	3
	ASO	379312	194295	1770	—	506	22	45	65	—	33

Table 5.4 Resource Utilization

The next step in the optimization flow, making the on-chip buffers multi-port creates an architecture that is best suited for both HLS tools. This also increases the resource utilization, but it does not yet achieve the best implementation performance due to the unaligned memory accesses and inter-iteration dependencies, which are tackled only by ASO. Resource usage for Intel is reported in Figure 5.8.

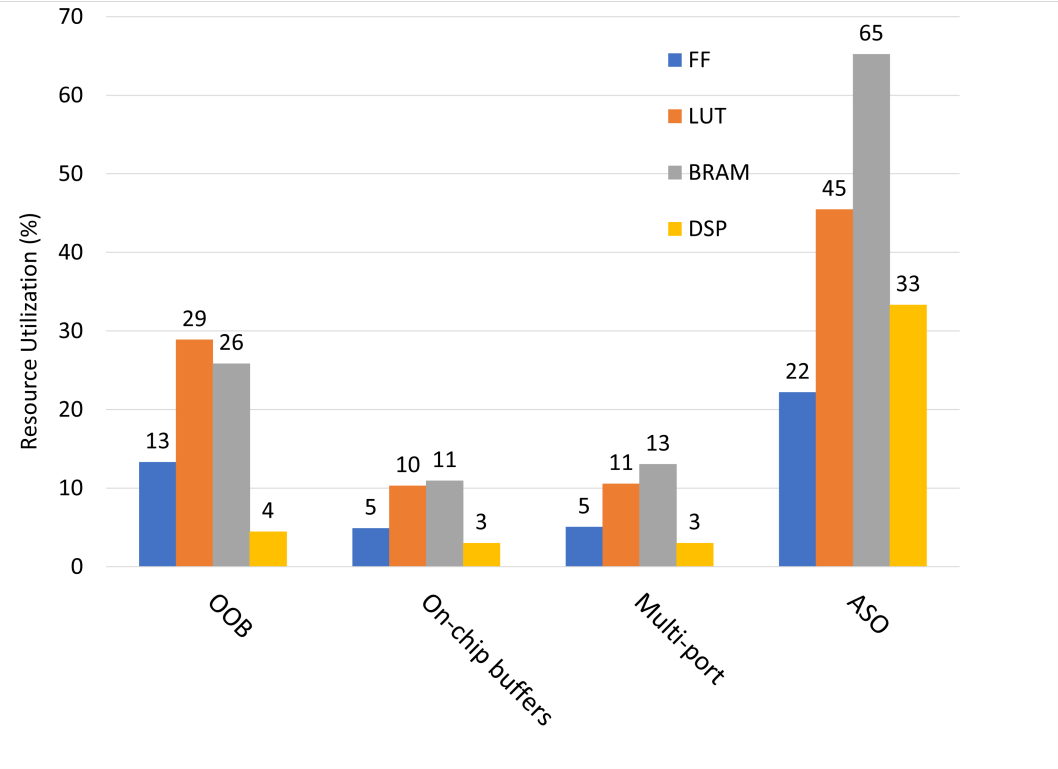


Fig. 5.8 Resource utilization on Arria platform

HBM and URAM resources on US+ platform are then used by platform-specific optimization, to balance the resource utilization and increase even further the achievable performance. Figure 5.9

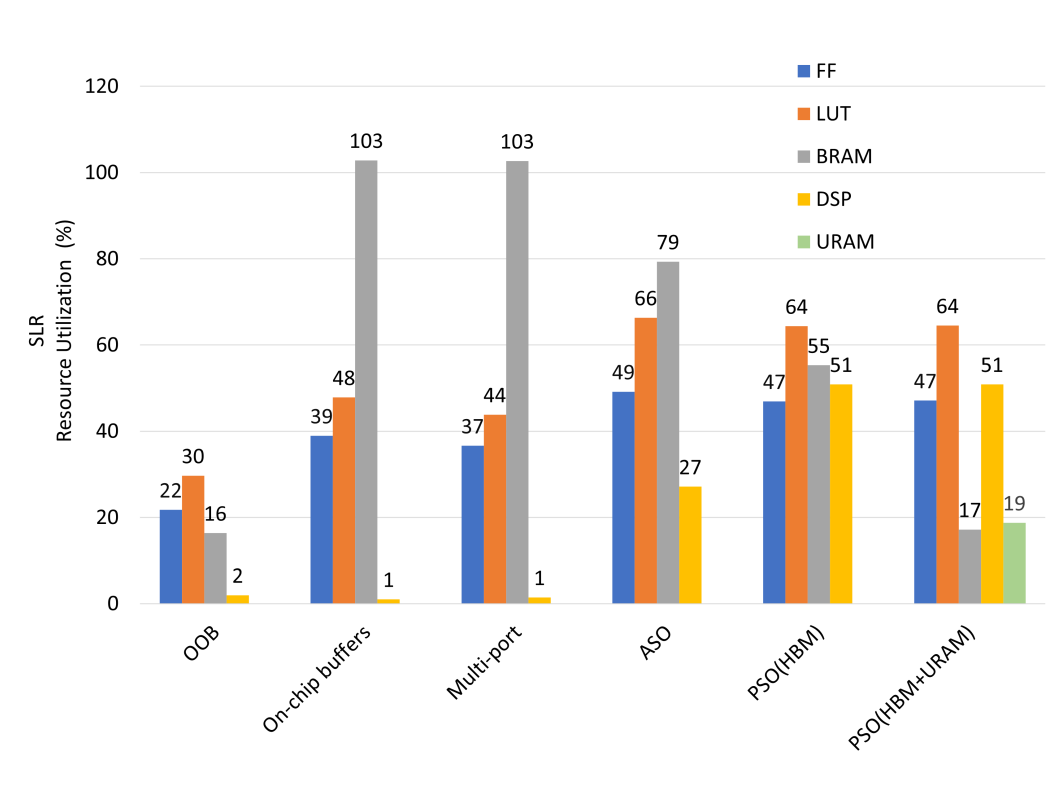


Fig. 5.9 Single SLR resource utilization on US+

shows the percentage utilization of resources on US+. Since all designs tile the on-chip buffering of the DRAM arrays to support large problem sizes, resource usage is not affected by increases in the number of total channel coefficients, which affects only the total latency.

5.2.3 Power and energy

Improving energy efficiency is one of the key advantages of offloading an application function to specialized hardware. General-purpose processors focus on flexibility and hence are not optimized for maximum efficiency for each application. Offloading functions to FPGA hardware accelerators can enhance energy efficiency not only by reducing the execution time but also by using only the required hardware to execute the task. Table 5.5 reports the energy consumption of the design at various optimization levels. The CPU implementation has the highest energy consumption

Table 5.5 Power and energy utilization of baseline and accelerated designs

Platform	**Optimization**	**Power (W)**			**Energy**
		Dynamic	**Static**	**Total**	**(J)**
CPU				105.00	526.05
US+	OOB	10.76	3.44	14.20	**2277.68**
	On-chip buffers	16.35	3.59	19.94	81.95
	Multi-port	21.20	3.73	24.93	594.83
	ASO	36.54	4.20	40.74	3.14
	PSO(HBM)	24.62	3.82	28.44	**0.94**
	PSO(URAM)	23.47	3.78	27.25	**0.79**
Arria	OOB	7.31	5.33	24.28	15903.40
	On-chip buffers	4.13	4.19	19.91	557.20
	Multi-port	4.14	4.16	19.94	127.42
	ASO	18.30	11.17	41.11	**2.18**

due its higher thermal design profile (TDP). The energy consumption is highest for OOB designs because of their higher latency and power-expensive accesses to memory, since all the data reside in DRAM. The optimizations applied help reducing the total latency and making efficient use of the available resources, which also reduce energy consumption. Figure 5.10 shows the energy consumption of implemented design at different optimization stages.

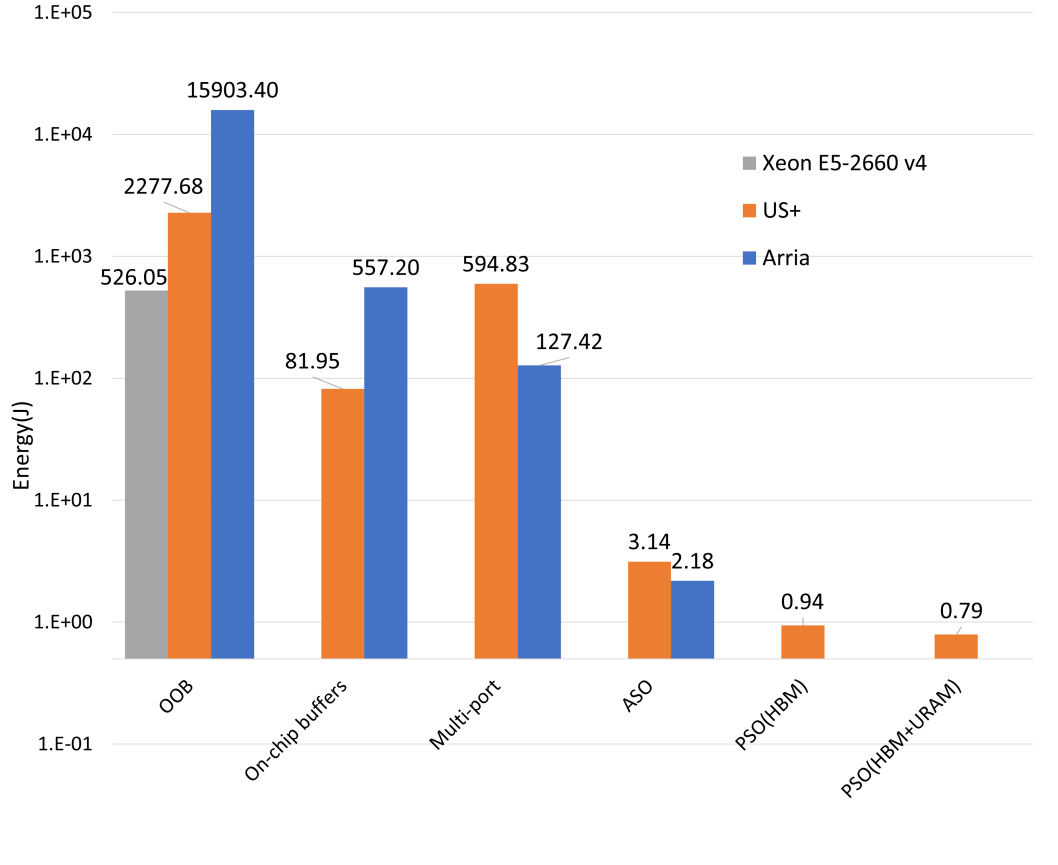


Fig. 5.10 Energy utilization on target platforms (log scale)

For US+ platforms, the energy is reported in fig. 5.11 for a combination of transmitting and receiving antenna elements in both uplink and downlink communication scenarios.

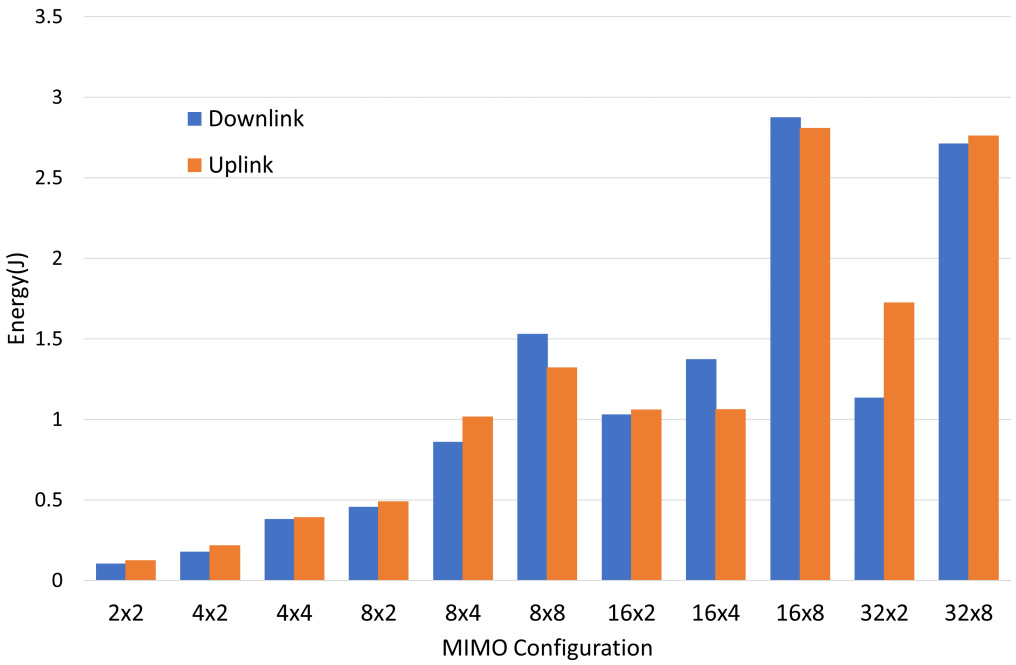


Fig. 5.11 Energy utilization on for various MIMO downlink and uplink configuration on US+ platform

FPGA platforms consume much less power than CPUs with respect to their computational capabilities. Moreover, optimizations for resources also save power and those for latency also reduce energy consumption.

5.3 Conclusion

The simulation of the 3GPP three-dimensional spatial channel model (3D-SCM) channel model can be significantly accelerated using FPGA platforms from different vendors (reported for Xilinx and Intel in this study) by applying a range of optimization techniques. The achievable speedup is limited by the memory, as the bandwidth limit is reached before the computing resource limit. A nominally portable OpenCL implementation allows designing for FPGAs using HLS tools. However, straightforward porting of the original C++ code targeted for a CPU to OpenCL does not reach good out-of-the-box results. A comprehensive set of memory and loop-based

optimization techniques are needed to tackle this challenge and can improve the performance by many orders of magnitude. While the initial implementation was much slower than CPU execution, with optimizations its execution is two orders of magnitude faster.

Using the accelerated channel model, a higher number of parameters can be simulated compared to the model running in MATLAB environment or C++ code on a CPU in a comparable amount of time. Hence, the accelerated model supports the simulation of a wider speed range for UE, and more antenna elements can be considered. To the best of our knowledge, this work is the first implementation of 3GPP 3D-SCM on both Xilinx and Intel FPGA platforms, with a detailed comparison of the achievable results on both. An impressive **95X** and **65X** speedup was achieved on the US+ and Arria platforms compared to the baseline CPU implementation through a combination of generic optimizations alongside application-specific optimizations. The performance on the Xilinx US+ platform improved even further, to **173X**, by exploiting the on-chip URAMs and HBM. Since the data types were kept the same as those in the baseline CPU implementation, i.e. double precision floating point, there was no change in accuracy for the FPGA implementations. This was particularly challenging because the FPGAs considered in this study, despite being both aimed at data center applications lack optimized support for double-precision floating-point adders or multipliers.

Chapter 6

GPU Acceleration of 3GPP 3D Channel Model

The radio link simulation must closely take into account propagation in real-world environments in order to predict and optimize network coverage and minimize post-deployment on-field measurements. Channel simulators are a highly effective means of conducting performance evaluations on the various components of a wireless communication system. The assessment of compatibility with communication standards is crucial in the testing of novel communication algorithms. These assessments involve the analysis of the system's performance in the presence of a wireless communication channel. Simulation of massive MIMO channel models is becoming increasingly important for testing and validation of 5G-NR wireless networks and beyond. However, simulation performance tends to be limited when modeling a large number of antenna elements combined with a complex and realistic representation of propagation conditions. Existing channel model simulators are either too simple to accurately replicate the propagation environment or too computationally expensive to produce meaningful results in a a reasonable amount of time. In addition, most existing simulators are designed for CPU platforms, which have limited parallelism and throughput. Therefore, there is a need for efficient and accurate channel model simulators that can run on parallel platforms such as GPUs, which offer high performance and scalability.

6.1 Related Work

Several hardware-based channel accelerators have been reported in the literature. The authors in [113] introduced a simulation platform that employs GPUs for signal-processing tasks. The platform was subsequently assessed in terms of its runtime performance. The platform incorporates CPU-based simulation logic and GPU-based processing to handle computationally demanding tasks, which aligns with heterogeneous computing methodologies. However, the authors solely take into account events in a single node.

A GPU based wireless channel emulator is proposed in [114]. The authors present a hybrid approach to speed up the channel calculation. However, the authors consider only one delay model (CDL-A) and also only LOS clusters. This limits the applicability of the proposed method since it cannot model the wireless channel accurately for network planning tools.

In FPGA implementation of a 3GPP 3D channel model, a variety of HLS-based optimizations are discussed that are required to achieve acceleration [105]. Recently, [104] proposed an analytical methodology to reduce the complexity of the 3GPP channel model for 5G-NR by reducing the number of sub-paths, thus reducing the computational cost, but it analyzes only a subset of wireless channel propagation characteristics, limiting its application. A GPU-based multipath fading accelerator has been proposed [115], as well as another wireless channel emulator [116]. It lacks complex-valued channel coefficient emulation, which reduces accuracy.

This chapter discusses a GPU-based hardware acceleration for 5G-NR channel model, and show that the proposed GPU accelerator can significantly improve the simulation speed and accuracy over a CPU-based C++ model, and also has higher single precision performance than an FPGA-based accelerator proposed in chapter 5. The main contributions of the chapter are:

- Proposing a GPU-based hardware acceleration for the 3GPP 3D channel model, which is a highly parameterized and realistic channel model for 5G-NR networks;
- Application of various CUDA-based optimization techniques to efficiently utilize GPU resources and increase the overall performance of the channel model simulator;
- Evaluation of the performance and accuracy of the GPU accelerator using benchmark parameters and comparison with both a CPU-based C++ model and a previous design on an FPGA based on the same 16 nm technology node as the GPU;

- Showing that the GPU accelerator can achieve an overall speedup of about 240× compared to the CPU model and 33.3 % higher single precision performance than a comparable FPGA design, while maintaining high accuracy and flexibility.

Part of the work described in this chapter has been already published as Nasir Ali Shah, Mihai T Lazarescu, Roberto Quasso, and Luciano Lavagno. Cuda-optimized gpu acceleration of 3gpp 3d channel model simulations for 5g network planning. Electronics, 12(15):3214, 2023. [117].

6.2 GPU Based Hardware Acceleration

GPUs are a type of SIMD architecture where the same instruction is executed repeatedly on different data in parallel. While CPUs excel in sequential execution performance by executing single operations as a thread as quickly as possible,GPUs are specifically designed to run thousands of threads in parallel for higher throughput and use multi-threading to hide memory latency. Efficient management of GPU resources can be achieved through high-level programming languages based on the underlying computing architectures, resulting in improved performance. Popular parallel computing architectures in the industry include the Open Computing Language (OpenCL) [80], Open Multi-Processing (OpenMP), and CUDA [118], a parallel programming language for managing computations on NVIDIA GPUs. Several code optimization techniques, both generic to GPU code and specific to CUDA, are required to efficiently utilize the on-chip resources and increase the overall performance. CUDA-based acceleration code consists of two components: the *host code*, which runs on the general-purpose CPU and is responsible for memory and device management and a collection of functions called the *kernel code*, which runs on the GPU accelerator device.Threads in CUDA are the unit of computation and are modeled as functions in the kernel code. *They are totally concurrent unless synchronized by the hardware or by the designer.*

In order to efficiently map threads to the architecture of the GPU, they are arranged in 3D clusters called *blocks*. These clusters are then combined into a 3D grid. The CUDA programming model groups a set of 32 threads into a single entity known as a *warp*. Concurrent threads (1) within a warp are automatically synchronized in lockstep by the hardware, while (2) threads within a block can be synchronized via *barriers* by the designer, e.g., to enable all threads to complete data

transfers before starting a computation on those data, and (3) thread blocks cannot be synchronized with each other at all.

When a designer has to port an application that was originally written for a CPU to a GPU, the code must be completely restructured, to *explicitly expose parallel computations and optimize memory accesses*, as the implicit optimizations provided by compilers are usually insufficient. GPU architecture for acceleration and CUDA programming model prospective of GPU are shown in Figure 6.1.

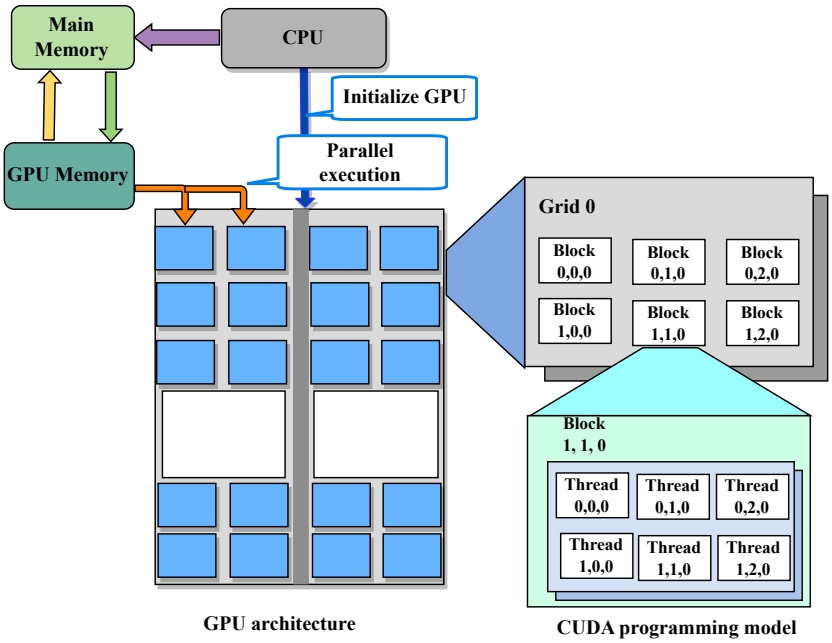


Fig. 6.1 GPU architecture and CUDA programming model

The main characteristics of the GPU programming languages, and of CUDA in particular, are discussed below:

1. allocating arrays to explicit levels in the memory hierarchy.
2. explicitly modeling concurrency via threads.

6.2.1 Global memory

Global memory is the off-chip DRAM available on the GPU board, and it is typically separate from the CPU memory. It is used as a communication buffer for large amounts of data between the CPU and the GPU. It has high latency and relatively

low bandwidth, just like on a CPU, compared with lower levels of the hierarchy. The host code is in charge of transferring data between the host memory space and the global memory. Arrays (less frequently scalars) allocated in global memory must be tagged as `__device__` in CUDA.

6.2.2 Shared memory

Shared memory is an on-chip memory, with low latency and very high bandwidth (like an L1 cache), local to each streaming multiprocessor, and accessible only by threads in the same block. Developers must explicitly specify shared memory data, using the `__shared__` storage attribute to allocate arrays in shared memory, and move data between global and shared memory using kernel code. In our work, threads compute the channel response for each transmitter-receiver antenna port in a cluster and require repeated reading of the input data. Since the CUDA global memory is not fast enough to provide data to all processing elements, a two-step loading mechanism is used. First, the input data is loaded into the on-chip shared memory in a coalesced fashion, and then the data is accessed for CIR computation.

6.2.3 Thread synchronization

Explicit designer-driven thread group synchronization via barriers is the most commonly used synchronization mechanism between otherwise independent threads. It allows, for example, kernel code to transfer data between (1) large and slow off-chip memory and (2) smaller and faster on-chip memory, ensuring that:

- all threads involved in a concurrent set of memory transfers, where each thread copies one or a few words of a large off-chip memory buffer to an on-chip memory one, are finished when computations using the transferred data begin,
- all threads performing parallel computations are finished when the results begin to be transferred back from on-chip memory to off-chip memory.

Implicit automatic thread synchronization occurs in programs with divergent control flows, i.e. where conditional branches in the code may have different outcomes for different threads in a warp. Programmers must carefully consider using conditionals (if-then-else and switch statements) in kernel code, because it may cause significant performance losses on a GPU architecture. If a thread has two nested

if-then-elses, and the conditions are independent, then typically only 25% of each GPU processor can be exploited, since all four combinations of the condition values must be executed *in sequence*, rather than in parallel.

As mentioned above, the CUDA programming model employs three types of thread parallelism:

- parallelism between thread blocks, where synchronization is impossible,
- parallelism within a thread block, where synchronization can be requested by the designer, and
- parallelism within thread warp, where synchronization is automatically ensured by the GPU hardware.

From a hardware perspective, there are three corresponding execution hierarchies: cooperative thread array (CTA) (also known as streaming multiprocessor (SM)), warp, and SIMD lanes. At kernel startup, each thread block is assigned to CTA and each thread is assigned to an SIMD lane. If the block-level explicit synchronization barriers are used, then the CTA hardware will wait for all threads in a block to reach the barrier before any thread is allowed to continue beyond it. Using the warp-level synchronization feature of the CUDA cooperative thread array, threads are synchronized only at the warp level, and other warps can continue to execute. This is especially important in our case because elements in a cluster can be mapped to threads in a warp, and partitioned block into tiles of size equal to warp size. Because each cluster is modeled independently, threads can be synchronized at the warp level, avoiding frequent block-level synchronizations.

6.2.4 Register-based parallel reduction

This programming technique allows a thread to read a register directly from another thread within the same warp and allows them to exchange or broadcast data among each other very efficiently. The idea of parallel reduction is illustrated in Figure 6.2

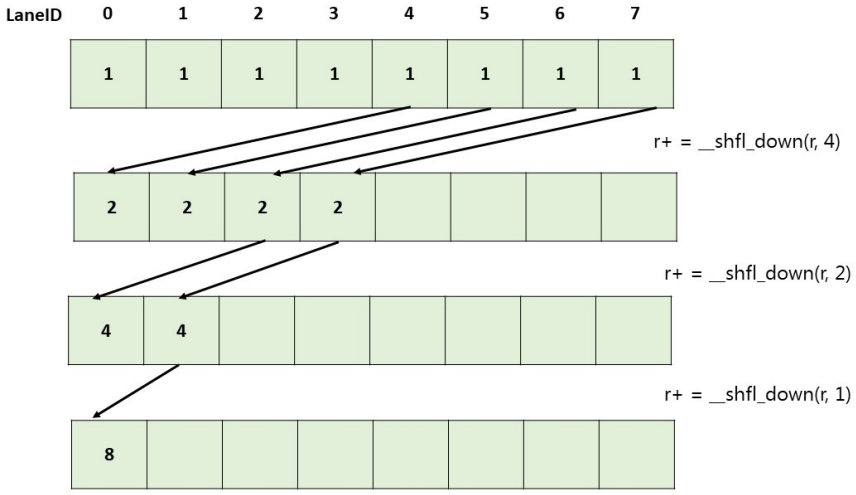


Fig. 6.2 Parallel reduction using registers

where the `__shfl_down()` CUDA instruction calculates the source and the destination of each reduction step, so that in N steps 2^N data elements are reduced via an associative operation (e.g. addition) within a warp, without the need for expensive explicit synchronization barriers (i.e. the maximum value of N for which this can be done with warps of size 32 is 5). The final stage of reduction, beyond the 5 iterations supported by a warp, is performed less efficiently in shared or global memory for all warps belonging to the same block via explicit barrier synchronization. These two kinds of reduction are both exploited in our model to optimize the final accumulation of the results computed by each warp to generate the total CIR.

6.3 Channel Emulator Acceleration on GPU

The geometry-based stochastic channel model consists of two parts: (1) a large-scale fading model that includes path-loss, LOS probability, and additional losses, combined with (2) a small-scale fading model characterized by the CIR (also called "channel coefficients" in the following). In the context of multipath propagation, the received signal is composed of various attenuated replicas of the original transmitted signal. To calculate the channel coefficients, a step-by-step procedure is employed as recommended by 3GPP specifications [2, Fig. 7.5-1].

Cluster-delay line (CDL) serves as a modeling tool in scenarios where the received signal comprises several delayed clusters. Each cluster is composed of multipath components that share a common delay, albeit exhibiting slight variations in angles of departure and arrival. 3GPP has defined various CDL profiles for link-level simulations. For NLOS, three CDL profiles, namely CDL-A, CDL-B, and CDL-C, are defined, while CDL-D and CDL-E are constructed for LOS clusters.

The channel model output computation is a set of FIR filters, one per path. Consequently, the sampled signal at the receiver can be expressed as the sum over paths of a convolution between the taps of this FIR filter and the channel model input signal. This study uses a two-kernel acceleration:

1. a less computation-intensive kernel computes the FIR coefficients, i.e. the CIR, according to (5.5). Its pseudo-code is shown in Listing 6.1.
2. a more computation-intensive FIR kernel that applies the coefficients to each input symbol, as in (5.6). Its pseudo-code is shown in Listing 6.2.

In addition to the two kernels, our accelerator also includes a host code that is written in C++ and executed on the host CPU. It is responsible for interacting with the simulation clients via sockets, performing preliminary model configurations and data transfers with the GPU. The architecture of the proposed accelerated channel model is shown in Figure 6.3.

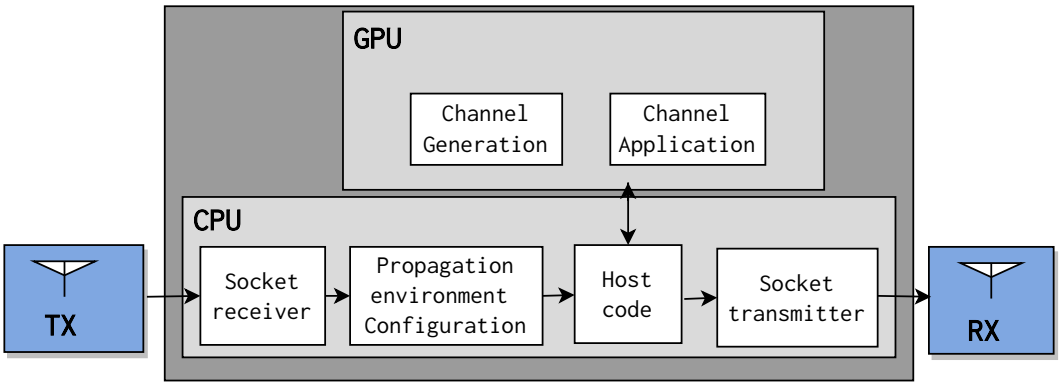


Fig. 6.3 3GPP 3D channel model on GPU

It uses CUDA cooperative groups to eliminate the need for block-level synchronization since each cluster is computed independently. For efficient use of GPU resources, the long chain of computations is split into parts as shown in listing 6.1 where the speed-factor part $\mathbb{V}$ and cluster information part $\mathbb{C}$ are computed in shared

```
__global__ void
calcCIR(H_{u,s}, F_{Rx},F_{Tx},RxLocation, TxLocation)
{
    // CUDA grid with X, Y, Z blocks
    i=Idx.x; tx=Idx.y; rx=Idx.z; l=threadIdx.x;
    cta = this_thread_block(); // create tiled blocks
    thread_tile<32> tile32 = tiled_partition<32>(cta);
    // Considering downlink
    // for uplink, arrival, and departure parameters will be
   swapped
    dopplerSpeed_{n,m} = dopplerSpeed + \hat{r}^{T}_{rx,n,m} \times speed \times \lambda^{-1}
    tile32.sync();
   \mathbb{V}^{NLOS}_{n,m} = 2 \times \pi \times dopplerSpeed_{n,m} \times RxLocation \times TxLocation //compute first
   part of CIR as in Eq. 5.2 and
    tile32.sync();
    \mathbb{C}^{NLOS}_{u,s} = sqrt(P_u/NRAY) \times F_{Rx} \times F_{Tx} \times XPR // Cluster info as in Eq. 5.1
    __syncthreads();            // block sync
    H_{u,s,n,m} = \mathbb{V}^{NLOS}_{n,m} \times \mathbb{C}^{NLOS}_{u,s}
    tile32.sync();
    for each RAY:
    H_{u,s,n,m}(t) = warpReduceSum(H_{u,s,n,m(t)}) //
    if(LOS){
        // load LOS parameters in shared mem
        dopplerSpeed^{LOS}_{n,m} += \hat{r}^{LOS}_{rx,n,m} \times speed \times \lambda^{-1}   // using LOS parameters
        \mathbb{V}^{LOS} = 2 \times \pi \times dopplerSpeed^{LOS}_{n,m} \times RxLocation \times TxLocation // as in Eq.
   5.4
        \mathbb{C}^{LOS}_{u,s} = sqrt(P^{LOS}_u/NRAY) \times F^{LOS}_{Rx} \times F^{LOS}_{Tx}   // compute for LOS as in
   Eq. 5.3
        H^{LOS}_{u,s,l} = \mathbb{V}^{LOS} \times \mathbb{C}^{LOS}_{u,s}
        H^{LOS}_{u,s,1} = warpReduceSum(H^{LOS}_{u,s,l}) / reduction in reg
        tile32.sync();
        H_{u,s} = atomicAdd(H^{NLOS}_{u,s,n,m} + H^{LOS}_{u,s,1}) //Combine LOS and NLOS response
        }
  __syncthreads();
}
```

Listing 6.1 CUDA calcCIR kernel

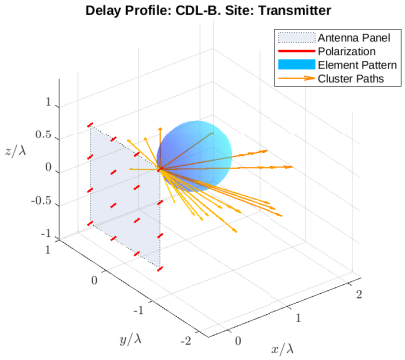
```
__global__ void
applyFIR(y(t),x(t), H_{u,s}(t),H_{u,s}(t−τ),cirBuf,pos)
{
  // CUDA grid with X, Y, Z blocks
  i=Idx.x; tx=Idx.y; rx=Idx.z; l=threadIdx.x;
  cta = this_thread_block(); // create tiled blocks
  thread_tile<32> tile32 = tiled_partition<32>(cta);
  ΔH(t) = H_{u,s}(t) − H_{u,s}(t−τ);      // use shared mem
  regCirBuf_l = cirBuf_{tx,rx,l}          // load circular buffer
  __syncthreads();                         // block sync
  x_l = x(t)_{i,tx};              // load received symbol in shared mem
  index = pos_l;                  // read cluster position from const mem
  tapV = tapV + ΔH(t) + H_{u,s}(t)    // warp-wide tap vector
  tile32.sync();                          // warp-wise soft sync
  tap = regCirBuf_{index} × x_l          // interpolation lines
  tile32.sync();
  acc_l = warpReduceSum_l(tap);           // reduction in regs
  tile32.sync();
  y(t)_{rx} = y(t)_{rx} + acc_l;      // accumulate over Rx
}
```

Listing 6.2 CUDA applyFIR kernel

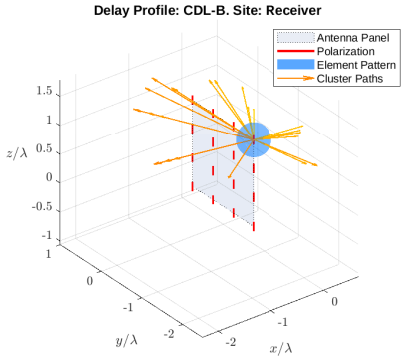
memory. This allows threads to remain active since there is no penalty for context switching. The register-based warp-wise parallel reduction of FIR taps help improve latency and resource utilization.

6.4 Results and Discussion

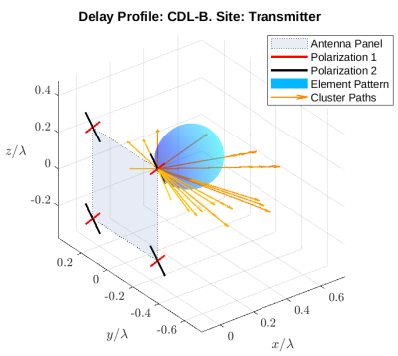
The baseline CPU performance was determined using an Intel Core i7-6900K @3.2 GHz CPU. The baseline channel model is implemented in C++ and runs as a MEX C++ function within a MATLAB R2021a environment. The performance of the channel model is evaluated using the benchmark values in table 5.2. To evaluate the performance for link-level simulations, we consider two CDL profiles, i.e., CDL-B for NLOS clusters and CDL-D for LOS clusters. Figure 6.4 illustrates various MIMO antenna element configurations for single-polarized antennas in Figure 6.4a and Figure 6.4b and dual-polarized arrays in Figure 6.5c and Figure 6.5d on transmitter and receiver end for CDL-B profile. Similarly, the same is reported for CDL-D in Figure 6.5 where fig. 6.5a and Figure 6.5b shows antenna patterns for



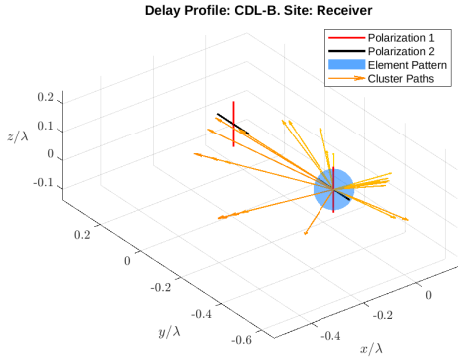
(a) Single-polarized antenna panels on T_x end



(b) Single-polarized antenna panel on R_x end



(c) Dual-polarized antenna panels on T_x end



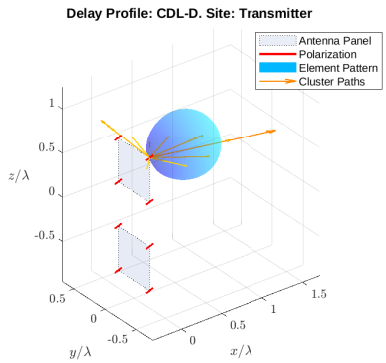
(d) Dual-polarized antennas on R_x end

Fig. 6.4 MIMO antenna configuration in CDL-B profile for NLOS clusters

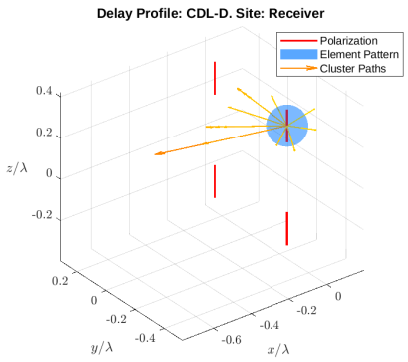
single-polarized arrays and Figure 6.5c and Figure 6.5d illustrates dual-polarized antennas on transmitting and receiving end.

The accelerator discussed in this chapter was developed using the CUDA development tools [118], targeting the NVIDIA GeForce GTX 1070 GPU [119] which features 1920 CUDA cores, 120 texture mapping units (TMUs), 1.5 MB of shared memory, 4 MB of local memory, 8 GB of GDDR5 memory, and 15 SMs.

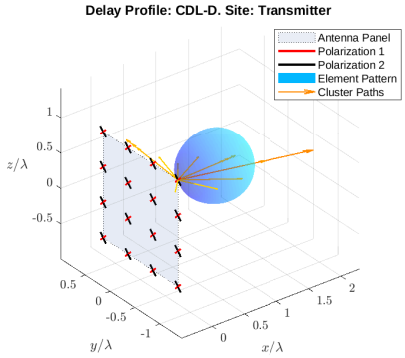
Performance of the GPU accelerator is compared with an FPGA implementation [105], which was developed using the *Vitis Unified Software Platform* [91] for the *AMD Alveo U280* [108]. The FPGA used in [105] is based on the same 16 nm technology node as the GPU and contains 9024 DSP blocks, 41 MB of on-chip static RAM, 1 303 680 look-up tables, and 8 GB of high bandwidth memory (HBM2). Thus, its computational power is comparable to that of the GPU used in this work,



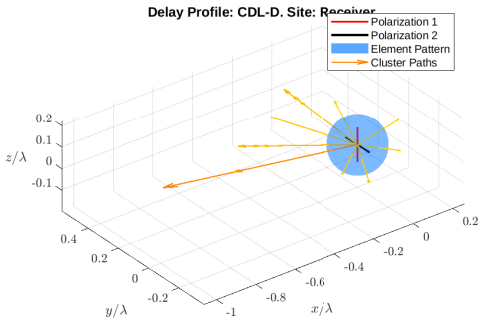
(a) Single-polarized antenna panels on T_x end



(b) Single-polarized antenna panel on R_x end



(c) Dual-polarized antenna panels on T_x end



(d) Dual-polarized antennas on R_x end

Fig. 6.5 MIMO antenna configuration in CDL-D profile for LOS clusters

Table 6.1 Kernel latency for a combination of MIMO elements in CDL-B NLOS

Link type	**Kernel**	**Execution latency Downlink($R_x \times T_x$) , Uplink($T_x \times R_x$)**							
		2×2	**4×4**	**4×8**	**8×8**	**8×16**	**16×16**	**2×32**	**4×32**
Downlink	calcCIR(µs)	5.43	8.22	8.29	10.94	12.93	20.16	9.25	12.67
	applyFIR(ms)	1.72	5.95	11.23	22.04	43.12	85.42	22.12	43.06
Uplink	calcCIR(µs)	5.43	8.00	8.35	11.10	12.58	19.46	8.86	11.23
	applyFIR(ms)	1.72	5.93	11.78	22.13	43.88	85.32	24.93	45.50

Table 6.2 Kernel latency for a combination of MIMO elements in CDL-D LOS

Link type	**Kernel**	**Execution latency Downlink($R_x \times T_x$) , Uplink($T_x \times R_x$)**							
		2×2	**4×4**	**4×8**	**8×8**	**8×16**	**16×16**	**2×32**	**4×32**
Downlink	calcCIR(µs)	6.56	11.71	8.64	10.72	11.52	25.09	8.90	9.47
	applyFIR(ms)	1.09	2.95	5.56	11.01	21.52	42.14	10.90	21.21
Uplink	calcCIR(µs)	8.03	11.58	8.74	10.94	16.54	16.06	7.74	10.85
	applyFIR(ms)	1.37	2.4	4.58	9.62	21.70	33.16	12.46	22.68

since (1) a DSP unit can be used to implement a SP multiply and add, and (2) in FPGA implementation only 1/3 of the total resources are used so that the kernel can fit on one chiplet to avoid routing problems.

The primary goal of this work is to reduce the overall execution time of the channel model under resource constraints. We report the achieved performance for the kernels in Listing 6.1 and Listing 6.2 on GPU platforms. To analyze the performance for both LOS and NLOS scenarios, we consider CDL-B and CDL-D profiles and uplink and downlink connection types. Table 6.1 reports the execution latency for various combination of R_x and T_x antenna elements considering NLOS clusters in CDL-B profile for the parameters listed in Table 5.2.

Figure 6.6 illustrates a comparison of link-level simulation latency on CPU and GPU platforms in the two CDL profiles. The values on the horizontal-axis represent the number of receiving and transmitting antenna elements, where the vertical-axis denotes the total execution time in logarithmic scale. It can be inferred from Figure 6.6 that the GPU implementation greatly reduces the simulation time

Table 6.3 Accelerated kernel latency and energy consumption

Platform	**Latency** (s)	**Speedup** (times)	**Power** (W)	**Energy** (J)
CPU	5.01	N/A	105	526.0
FPGA	0.03	172	31.1	0.96
GPU	0.08	60	52.0	4.37
GPU (SP)	0.02	*240*	40.5	0.85

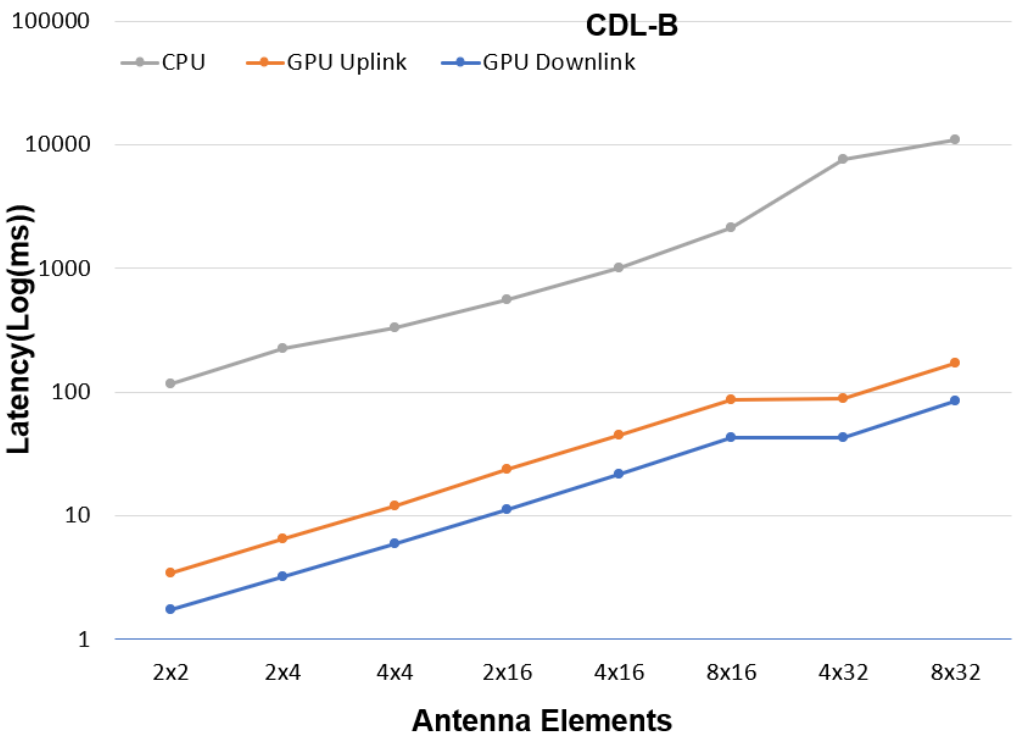
and enables the network planners to simulation more complex propagation scenarios with higher Doppler shift and even more antenna elements.

Table 6.3 reports the achieved performance and energy consumption for FPGA and GPU acceleration platforms for CDL-B delay profile. Overall, the optimizations result in a speedup of **60×** compared to the baseline CPU implementation. The achievable performance is memory-bound due to the limited on-chip shared memory of the GPU, hence the need to repeatedly read large amounts of data from the DRAM rather than storing it on-chip as was done on the FPGA. To analyze the achievable performance without the on-chip memory limit, the precision was reduces to single which resulted in a speedup of **240×**.

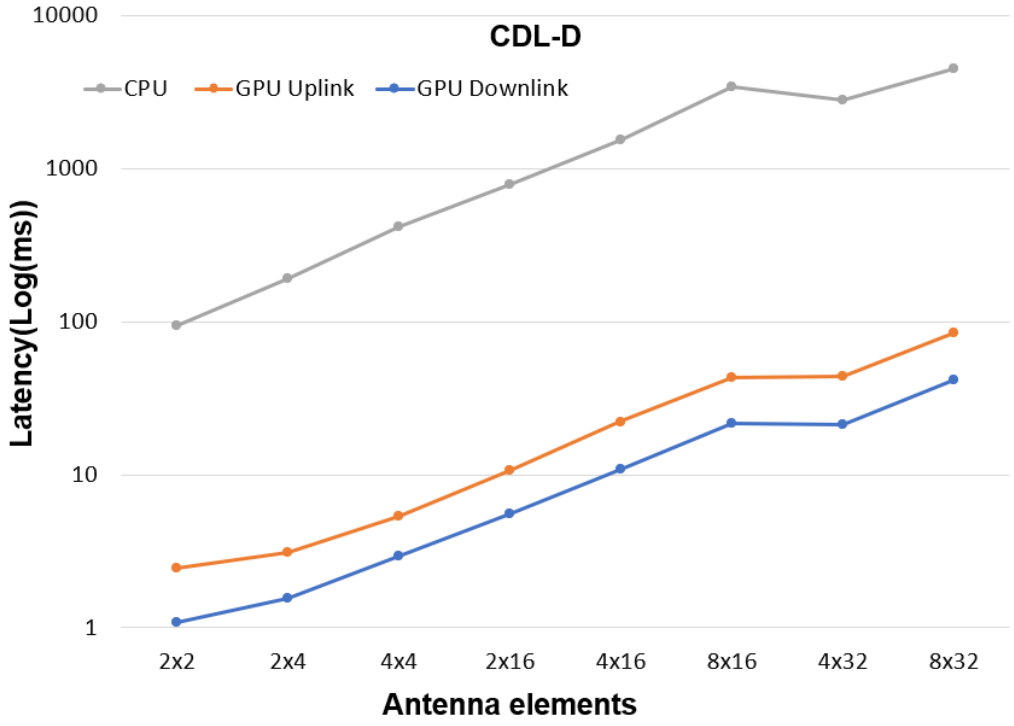
For power analysis, CPU results are calculated based on its thermal design power (TDP) because we have no way to measure its power consumption in real time. Energy consumption is very high due to high execution latency on CPU platform. The energy consumption of the FPGA is lower than that of the GPU because the data is copied only once into on-chip buffers (our FPGA has more on-chip memory than our GPU). The GPU and FPGA power consumption is measured using the respective runtime support. In both cases, they are lower than their respective TDPs because only one-third of the on-chip compute resources were utilized due to memory bandwidth constraints, as shown in Table 6.4. On FPGA platform, only one of the three chiplets (also called SLRs) in the package was used to achieve a good clock period.

6.4.1 Coding Style: CUDA vs. HLS

Although both FPGA and GPU provide parallel computation, writing source code to efficiently program them is very different. In the case of GPU, it is necessary



(a) Latency for CDL-B profile



(b) Latency for CDL-D profile

Fig. 6.6 Execution time on CPU and GPU platforms

Table 6.4 Resource utilization of accelerated designs

Platform	Memory(%)	SM(%)	DSP(%)
FPGA	13.14	N/A	17.24
GPU	40.89	63.61	N/A
GPU(SP)	32.28	42.09	N/A

to explicitly exploit the multithreaded nature of the platform by exploiting the 3-dimensional parallel loop structure of the thread blocks, as shown in Listing 6.1 and Listing 6.2 (note the absence of any explicit loop construct). On the other hand, the accelerated code for the FPGA is actually more similar to the CPU version, with only the addition of (1) loops to transfer data from DRAM to on-chip memory, and (2) loop pipelining, loop unrolling, and array partitioning directives to expose parallelism in the computation and memory architecture in a form appropriate for HLS.

6.5 Conclusion

This work demonstrates an efficient implementation of a 3GPP 3D channel simulation model on GPU platforms. The implementation employs CUDA optimization techniques to efficiently utilize the parallelism and memory hierarchy of the GPU. The proposed GPU accelerator can significantly reduce simulation time for 5G-NR network planning and optimization by approximately 240× compared to a CPU-based C++ model, while maintaining high accuracy. The level of gains in performance is limited by the total amount of on-chip memory, thereby constraining concurrency. The GPU design exhibits better single-precision performance compared to the previously proposed FPGA design, albeit with increased power consumption. It is interesting to note that although FPGAs typically have lower floating-point performance than GPUs, in this case, the FPGA has higher performance due to the larger amount of on-chip memory used to store the data and reduce DRAM accesses, thus offsetting the lower computational performance compared to a GPU for this very memory-intensive channel model. This study showcases the practicality and advantages of employing GPU-based hardware acceleration in 5G-NR channel

model simulations. It also offers a valuable resource for network designers and researchers. Potential future work involves expanding the channel model to accommodate additional propagation scenarios and antenna configurations. Additionally, integrating the channel model with other components of the 5G simulation stack, such as physical layer and link layer models, could be explored.

Chapter 7

Conclusion and Future work

The acceleration of the 3GPP 3D-SCM channel model simulation can be significantly improved through the utilization of FPGA platforms provided by various vendors (specifically, Xilinx and Intel). The attainable boost in speed is constrained by the memory system, as the maximum data transfer rate is reached prior to reaching the limit of computational resources. A portable implementation of the OpenCL enables the development of designs for FPGAs using HLS tools. Nevertheless, the straightforward translation of the initial C++ code designed for a CPU targets to the OpenCL does not yield satisfactory results without further modifications. In order to address this challenge, it is crucial to employ a broad range of optimization techniques that focus on memory management and loop structures. These techniques have the potential to significantly enhance performance by several orders of magnitude.

The HLS tools for Xilinx FPGAs provide a user friendly graphical interface for rapid development and debugging, while there is no such Integrated Development Environment for the Intel FPGAs. However, both tool sets provide detailed design reports that enable micro-architectural optimizations. OOB implementation of code not specifically written for FPGAs is obviously suboptimal, due to the lack of an efficient on-chip memory architecture that is comparable to the cache in a CPU.

Intel FPGA SDK for OpenCL tries to automatically create an optimized memory architecture, but since the algorithm memory access pattern, albeit regular, was hard to analyze, the tool worsens the performance and increases the resource usage. This highlights the need for experienced hardware designers, familiar with HLS tools, who can partially rewrite the top application and manually optimize memory

access. The C++ based kernels for Xilinx FPGAs offer more control over the optimizations, such as the parameters of the DRAM interfaces or the choice of some specific computational resources, than in the nominally more portable OpenCL flow. Although HLS still has some shortcomings compared to hand-crafted RTL implementations, it enables rapid design space exploration and thus ultimately can achieve respectable quality of results with a reasonable design optimization time.

It is noteworthy that, despite the general tendency for FPGAs to exhibit lower floating-point performance compared to GPUs, the current scenario demonstrates a higher performance of the FPGA due to the larger amount of on-chip memory used to store the data and reduce DRAM accesses. Consequently, the lower computational performance of the FPGA is counterbalanced by its capacity to handle the memory-intensive channel model at hand, surpassing that of a GPU.

A potential future extension to this research can be an accelerator that can simulate multiple runs of channel models simultaneously and independently, modeling multiple user equipments, on a single accelerator. Additionally, integrating the channel model with other elements of the 5G simulation stack, such as physical layer and link layer models, could be explored. This would allow to stream the data among all these blocks directly on the accelerator, lowering the DRAM accesses and improving the overall simulation performance.

References

[1] Martin Sauter. *From GSM to LTE-advanced Pro and 5G: An introduction to mobile networks and mobile broadband*. John Wiley & Sons, 2017.

[2] 3GPP. *Study on channel model for frequencies from 0.5 to 100 GHz*, 03 2017. Rev. 14.0.

[3] Apurba Das. *Digital Communication: Principles and system modelling*. Springer Science & Business Media, 2010.

[4] Saud Mobark Aldossari and Kwang-Cheng Chen. Machine learning for wireless communication channel modeling: An overview. *Wireless Personal Communications*, 106:41–70, 2019.

[5] Edward J Oughton, Erik Boch, and Julius Kusuma. Engineering-economic evaluation of diffractive non-line-of-sight backhaul (e3nb): A techno-economic model for 3D wireless backhaul assessment. *arXiv preprint arXiv:2106.04906*, 2021.

[6] ROBERTO Carrasco-Alvarez, JAVIER Vázquez Castillo, A Castillo Atoche, and J Ortegón Aguilar. A fading channel simulator implementation based on GPU computing techniques. *Mathematical Problems in Engineering*, 2015, 2015.

[7] Mark Cummings and Shinichiro Haruyama. FPGA in the software radio. *IEEE communications Magazine*, 37(2):108–112, 1999.

[8] John Nickolls and William J Dally. The GPU computing era. *IEEE micro*, 30(2):56–69, 2010.

[9] Michael Garland, Scott Le Grand, John Nickolls, Joshua Anderson, Jim Hardwick, Scott Morton, Everett Phillips, Yao Zhang, and Vasily Volkov. Parallel computing experiences with CUDA. *IEEE micro*, 28(4):13–27, 2008.

[10] Ganda Stephane Ouedraogo, Matthieu Gautier, and Olivier Sentieys. A frame-based domain-specific language for rapid prototyping of FPGA-based software-defined radios. *EURASIP Journal on Advances in Signal Processing*, 2014:1–15, 2014.

[11] Mobile subscriptions forecast – Mobility Report - Ericsson. https://www.ericsson.com/en/reports-and-papers/mobility-report/dataforecasts/mobile-subscriptions-outlook. (Accessed on 02/20/2023).

[12] Cheng-Xiang Wang, Xuemin Hong, Hanguang Wu, and Wen Xu. Spatial-temporal correlation properties of the 3GPP spatial channel model and the kronecker MIMO channel model. *EURASIP Journal on Wireless Communications and Networking*, 2007:1–9, 2007.

[13] M Series. IMT Vision–Framework and overall objectives of the future development of IMT for 2020 and beyond. *Recommendation ITU*, 2083(0), 2015.

[14] Dakhaz Mustafa Abdullah and Siddeeq Y Ameen. Enhanced mobile broadband (eMBB): A review. *Journal of Information Technology and Informatics*, 1(1):13–19, 2021.

[15] Zexian Li, Mikko A. Uusitalo, Hamidreza Shariatmadari, and Bikramjit Singh. 5G URLLC: Design Challenges and System Concepts. In *2018 15th International Symposium on Wireless Communication Systems (ISWCS)*, pages 1–6, 2018.

[16] Randeep Bhatia, Bhawna Gupta, Steven Benno, Jairo Esteban, Dragan Samardzija, Marcos Tavares, and T V Lakshman. Massive Machine Type Communications over 5G using Lean Protocols and Edge Proxies. In *2018 IEEE 5G World Forum (5GWF)*, pages 462–467, 2018.

[17] Federico Boccardi, Robert W Heath, Angel Lozano, Thomas L Marzetta, and Petar Popovski. Five disruptive technology directions for 5G. *IEEE communications magazine*, 52(2):74–80, 2014.

[18] Thomas L. Marzetta. Massive MIMO: An Introduction. *Bell Labs Technical Journal*, 20:11–22, 2015.

[19] Samsung Shares Massive MIMO Roadmap in New Whitepaper – Samsung Global Newsroom. https://news.samsung.com/global/samsung-shares-massive-mimo-roadmap-in-new-whitepaper. (Accessed on 03/28/2023).

[20] Sundeep Rangan, Theodore S Rappaport, and Elza Erkip. Millimeter-wave cellular wireless networks: Potentials and challenges. *Proceedings of the IEEE*, 102(3):366–385, 2014.

[21] Erik G Larsson, Ove Edfors, Fredrik Tufvesson, and Thomas L Marzetta. Massive MIMO for next generation wireless systems. *IEEE communications magazine*, 52(2):186–195, 2014.

[22] Ting Li, Liqiang Zhao, Fengfei Song, and Chengkang Pan. OAI-based End-to-End Network Slicing. In *2018 IEEE 23rd International Conference on Digital Signal Processing (DSP)*, pages 1–4. IEEE, 2018.

[23] D. Cotroneo, L. De Simone, A.K. Iannillo, A. Lanzaro, R. Natella, Jiang Fan, and Wang Ping. Network Function Virtualization: Challenges and Directions for Reliability Assurance. In *2014 IEEE International Symposium on Software Reliability Engineering Workshops*, pages 37–42, 2014.

[24] Microsoft Word - NFV_White_Paper_ETSI_CM. https://portal.etsi.org/nfv/nfv_white_paper.pdf. (Accessed on 03/28/2023).

[25] Abdelwahab, Sherif and Hamdaoui, Bechir and Guizani, Mohsen and Znati, Taieb. Network function virtualization in 5g. *IEEE Communications Magazine*, 54(4):84–91, 2016.

[26] Mate Boban, Apostolos Kousaridas, Konstantinos Manolakis, Joseph Eichinger, and Wen Xu. Use cases, requirements, and design considerations for 5G V2X. *arXiv preprint arXiv:1712.01754*, 2017.

[27] Khandaker Foysal Haque, Ahmed Abdelgawad, Venkata Prasanth Yanambaka, and Kumar Yelamarthi. LoRa Architecture for V2X Communication: An Experimental Evaluation with Vehicles on the Move. *Sensors*, 20(23), 2020.

[28] Sudhir Sharma, M Deivakani, K Srinivasa Reddy, AK Gnanasekar, and G Aparna. Key enabling technologies of 5G wireless mobile communication. In *Journal of Physics: Conference Series*, volume 1817, page 012003. IOP Publishing, 2021.

[29] Leonardo Militano, Antonino Orsino, Giuseppe Araniti, Michele Nitti, Luigi Atzori, and Antonio Iera. Trusted D2D-based data uploading in in-band narrowband-IoT with social awareness. In *2016 IEEE 27th Annual International Symposium on Personal, Indoor, and Mobile Radio Communications (PIMRC)*, pages 1–6. IEEE, 2016.

[30] Peter Almers, Ernst Bonek, Alister Burr, Nicolai Czink, Mérouane Debbah, Vittorio Degli-Esposti, Helmut Hofstetter, Pekka Kyösti, David Laurenson, Gerald Matz, et al. Survey of channel and radio propagation models for wireless MIMO systems. *EURASIP Journal on Wireless Communications and Networking*, 2007:1–19, 2007.

[31] Jan Zeleny, Fernando Perez-Fontan, and Pavel Pechac. Generalized Propagation Channel Model for 2GHz Low Elevation Links Using a Ray-tracing Method. *Radioengineering*, 24(4), 2015.

[32] M Ozcelik, Nicolai Czink, and Ernst Bonek. What makes a good MIMO channel model? In *2005 IEEE 61st Vehicular Technology Conference*, volume 1, pages 156–160. IEEE, 2005.

[33] Spatial Structure of Multiple Antenna Radio Channels. https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=ac7dc690e898f1183f2ec95c72d62ad27a5e8cd5. (Accessed on 05/31/2023).

[34] Merouane Debbah and Ralf R Muller. MIMO channel modeling and the principle of maximum entropy. *IEEE Transactions on Information Theory*, 51(5):1667–1690, 2005.

[35] Alister G Burr. Capacity bounds and estimates for the finite scatterers MIMO wireless channel. *IEEE Journal on Selected Areas in Communications*, 21(5):812–818, 2003.

[36] Chen-Nee Chuah, Joseph M Kahn, and David Tse. Capacity of multi-antenna array systems in indoor wireless environment. In *IEEE GLOBECOM 1998 (Cat. NO. 98CH36250)*, volume 4, pages 1894–1899. IEEE, 1998.

[37] Dmitry Chizhik, Farrokh Rashid-Farrokhi, Jonathan Ling, and Angel Lozano. Effect of antenna separation on the capacity of BLAST in correlated channels. *IEEE Communications letters*, 4(11):337–339, 2000.

[38] Jean-Philippe Kermoal, Laurent Schumacher, Klaus I Pedersen, Preben E Mogensen, and Frank Frederiksen. A stochastic MIMO radio channel model with experimental validation. *IEEE Journal on selected areas in Communications*, 20(6):1211–1226, 2002.

[39] Werner Weichselberger, Markus Herdin, Huseyin Ozcelik, and Ernst Bonek. A stochastic MIMO channel model with joint correlation of both link ends. *IEEE Transactions on Wireless Communications*, 5(1):90–100, 2006.

[40] Wei Wang, Thomas Jost, Uwe-Carsten Fiebig, and Christian Gentner. Modeling three different types of multipath components for mobile radio channel. In *The 8th European Conference on Antennas and Propagation (EuCAP 2014)*, pages 3065–3069, 2014.

[41] C. Oestges, B. Clerckx, L. Raynaud, and D. Vanhoenacker-Janvier. Deterministic channel modeling and performance simulation of microcellular wideband communication systems. *IEEE Transactions on Vehicular Technology*, 51(6):1422–1430, 2002.

[42] Xiongwen Zhao, Fei Du, Suiyan Geng, Zihao Fu, Zhongyu Wang, Yu Zhang, Zhenyu Zhou, Lei Zhang, and Liuqing Yang. Playback of 5G and beyond measured MIMO channels by an ANN-based modeling and simulation framework. *IEEE journal on selected areas in communications*, 38(9):1945–1954, 2020.

[43] Zhengqing Yun, Zhijun Zhang, and Magdy F Iskander. A ray-tracing method based on the triangular grid approach and application to propagation prediction in urban environments. *IEEE Transactions on Antennas and Propagation*, 50(5):750–758, 2002.

[44] Reiner S Thoma, Dirk Hampicke, Andreas Richter, Gerd Sommerkorn, Axel Schneider, Uwe Trautwein, and Walter Wirnitzer. Identification of time-variant directional mobile radio channels. *IEEE Transactions on Instrumentation and measurement*, 49(2):357–364, 2000.

[45] Josef Fuhl, Andreas F Molisch, and Ernst Bonek. Unified channel model for mobile radio systems with smart antennas. *IEE Proceedings-Radar, Sonar and Navigation*, 145(1):32–41, 1998.

[46] Paul Petrus, Jeffrey H Reed, and Theodore S Rappaport. Geometrical-based statistical macrocell channel model for mobile environments. *IEEE Transactions on Communications*, 50(3):495–502, 2002.

[47] Johan Karedal, Fredrik Tufvesson, Nicolai Czink, Alexander Paier, Charlotte Dumard, Thomas Zemen, Christoph F. Mecklenbrauker, and Andreas F. Molisch. A geometry-based stochastic MIMO model for vehicle-to-vehicle communications. *IEEE Transactions on Wireless Communications*, 8(7):3646–3657, 2009.

[48] Ali Abdi and Mostafa Kaveh. A space-time correlation model for multielement antenna systems in mobile fading channels. *IEEE Journal on Selected Areas in communications*, 20(3):550–560, 2002.

[49] Claude Oestges, Vinko Erceg, and Arogyaswami J Paulraj. A physical scattering model for MIMO macrocellular broadband wireless channels. *IEEE Journal on Selected Areas in Communications*, 21(5):721–729, 2003.

[50] Thomas Zwick, Christian Fischer, and Werner Wiesbeck. A stochastic multipath channel model including path directions for indoor environments. *IEEE journal on Selected Areas in Communications*, 20(6):1178–1192, 2002.

[51] Adel AM Saleh and Reinaldo Valenzuela. A statistical model for indoor multipath propagation. *IEEE Journal on selected areas in communications*, 5(2):128–137, 1987.

[52] Harsh Tataria, Katsuyuki Haneda, Andreas F Molisch, Mansoor Shafi, and Fredrik Tufvesson. Standardization of propagation models for terrestrial cellular systems: A historical perspective. *International Journal of Wireless Information Networks*, 28:20–44, 2021.

[53] Yvo de Jong Bultitude and Terhi Rautiainen. IST-4-027756 WINNER II D1. 1.2 V1. 2 WINNER II Channel Models. *EBITG, TUI, UOULU, CU/CRC, NOKIA, Tech. Rep*, 2007.

[54] Lingfeng Liu, Claude Oestges, Juho Poutanen, Katsuyuki Haneda, Pertti Vainikainen, François Quitin, Fredrik Tufvesson, and Philippe De Doncker. The COST 2100 MIMO channel model. *IEEE Wireless Communications*, 19(6):92–99, 2012.

[55] Vuokko Nurmela, Aki Karttunen, Antti Roivainen, Leszek Raschkowski, Veikko Hovinen, Juha Ylitalo EB, Nobutaka Omaki, Katsutoshi Kusume, Aki Hekkala, Richard Weiler, et al. Deliverable D1. 4 METIS channel models. *Proc. Mobile Wireless Commun. Enablers Inf. Soc.(METIS)*, 1, 2015.

[56] M Series. Guidelines for evaluation of radio interface technologies for IMT-2020. *Report ITU*, 2512:0, 2017.

[57] Richard J Weiler, Michael Peter, Wilhelm Keusgen, Alexander Maltsev, Ingolf Karls, Andrey Pudeyev, Ilya Bolotin, Isabelle Siaud, and Anne-Marie Ulmer-Moll. Quasi-deterministic millimeter-wave channel models in MiWEBA. *EURASIP Journal on Wireless Communications and Networking*, 2016:1–16, 2016.

[58] Shu Sun, George R MacCartney, and Theodore S Rappaport. A novel millimeter-wave channel simulator and applications for 5G wireless communications. In *2017 IEEE international conference on communications (ICC)*, pages 1–7. IEEE, 2017.

[59] Shu Sun, Theodore S Rappaport, Mansoor Shafi, Pan Tang, Jianhua Zhang, and Peter J Smith. Propagation models and performance evaluation for 5G millimeter-wave bands. *IEEE Transactions on Vehicular Technology*, 67(9):8422–8439, 2018.

[60] Dario Bega, Marco Gramaglia, Carlos Bernardos, Albert Banchs, and Xavier Costa-Pérez. Toward the network of the future: From enabling technologies to 5G concepts. *Transactions on Emerging Telecommunications Technologies*, 28:e3205, 06 2017.

[61] Henrik Asplund, Andres Alayon Glazunov, Andreas F. Molisch, Klaus I. Pedersen, and Martin Steinbauer. The COST 259 Directional Channel Model-Part II: Macrocells. *IEEE Transactions on Wireless Communications*, 5(12):3434–3450, 2006.

[62] Nicolai Czink and Claude Oestges. The COST 273 MIMO Channel Model: Three Kinds of Clusters. In *2008 IEEE 10th International Symposium on Spread Spectrum Techniques and Applications*, pages 282–286, 2008.

[63] Alexander Maltsev, Andrey Pudeyev, Artem Lomayev, and Ilya Bolotin. Channel modeling in the next generation mmWave Wi-Fi: IEEE 802.11ay standard. In *European Wireless 2016; 22th European Wireless Conference*, pages 1–8, 2016.

[64] Neeraj Varshney, Jiayi Zhang, Jian Wang, Anuraag Bodi, and Nada Golmie. Link-Level Abstraction of IEEE 802.11ay based on Quasi-Deterministic Channel Model from Measurements. In *2020 IEEE 92nd Vehicular Technology Conference (VTC2020-Fall)*, pages 1–7, 2020.

[65] S Jaeckel, L Raschkowski, K Börner, L Thiele, F Burkhardt, and E Eberlein. Quadriga-quasi deterministic radio channel generator, user manual and documentation. *Fraunhofer Heinrich Hertz Institute, Tech. Rep. v2. 0.0*, 2017.

[66] Jonas Medbo, Kai Börner, Katsuyuki Haneda, Veikko Hovinen, Tetsuro Imai, J Järvelainen, T Jämsä, Aki Karttunen, Katsutoshi Kusume, Jukka Kyröläinen, et al. Channel modelling for the fifth generation mobile communications. In

The 8th European Conference on Antennas and Propagation (EuCAP 2014), pages 219–223. IEEE, 2014.

[67] Qiuming Zhu, Cheng-Xiang Wang, Boyu Hua, Kai Mao, Shan Jiang, and Mengtian Yao. 3GPP TR 38.901 channel model. In *The Wiley 5G Ref: The Essential 5G Reference Online*, pages 1–35. Wiley Press, 2021.

[68] B. Sklar. Rayleigh fading channels in mobile digital communication systems Characterization. *IEEE Communications Magazine*, 35(7):90–100, 1997.

[69] Young-Han Nam, Boon Loong Ng, Krishna Sayana, Yang Li, Jianzhong Zhang, Younsun Kim, and Juho Lee. Full-dimension MIMO (FD-MIMO) for next generation cellular technology. *IEEE Communications Magazine*, 51(6):172–179, 2013.

[70] Gilberto Ochoa-Ruiz, Ouassila Labbani, El-Bay Bourennane, Philippe Soulard, and Sana Cherif. A high-level methodology for automatically generating dynamic partially reconfigurable systems using IP-XACT and the UML MARTE profile. *Design Automation for Embedded Systems*, 16:93–128, 2012.

[71] Grant Martin and Gary Smith. High-level synthesis: Past, present, and future. *IEEE Design & Test of Computers*, 26(4):18–25, 2009.

[72] Vitis HLS. https://www.xilinx.com/products/design-tools/vitis/vitis-hls.html. (Accessed on 06/21/2023).

[73] Melissa Sussmann and Tom Hill. Intel hls compiler: Fast design, coding, and hardware. *White paper*, 2017.

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

[75] Zhiru Zhang, Yiping Fan, Wei Jiang, Guoling Han, Changqi Yang, and Jason Cong. AutoPilot: A platform-based ESL synthesis system. *High-Level Synthesis: From Algorithm to Digital Circuit*, pages 99–112, 2008.

[76] Christian Pilato and Fabrizio Ferrandi. Bambu: A modular framework for the high level synthesis of memory-intensive applications. In *2013 23rd International conference on field programmable logic and applications*, pages 1–4. IEEE, 2013.

[77] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H Anderson, Stephen Brown, and Tomasz Czajkowski. LegUp: high-level synthesis for FPGA-based processor/accelerator systems. In *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, pages 33–36, 2011.

[78] C++/SystemC Synthesis | Siemens Software. https://eda.sw.siemens.com/en-US/ic/catapult-high-level-synthesis/hls/c-cplus/. (Accessed on 06/18/2023).

[79] Razvan Nane, Vlad-Mihai Sima, Bryan Olivier, Roel Meeuws, Yana Yankova, and Koen Bertels. DWARV 2.0: A CoSy-based C-to-VHDL hardware compiler. In *22nd International Conference on Field Programmable Logic and Applications (FPL)*, pages 619–622. IEEE, 2012.

[80] Tomasz S Czajkowski, Utku Aydonat, Dmitry Denisenko, John Freeman, Michael Kinsner, David Neto, Jason Wong, Peter Yiannacouras, and Deshanand P Singh. From OpenCL to high-performance hardware on FPGAs. In *22nd international conference on field programmable logic and applications (FPL)*, pages 531–534. IEEE, 2012.

[81] Maya Gokhale, Jan Stone, Jeff Arnold, and Mirek Kalinowski. Stream-oriented FPGA computing in the Streams-C high level language. In *Proceedings 2000 IEEE symposium on field-programmable custom computing machines (Cat. No. PR00871)*, pages 49–56. IEEE, 2000.

[82] Rishiyur Nikhil. Bluespec System Verilog: efficient, correct RTL from high level specifications. In *Proceedings. Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2004. MEMOCODE'04.*, pages 69–70. IEEE, 2004.

[83] Jerker Hammarberg and Simin Nadjm-Tehrani. Development of safety-critical reconfigurable hardware with Esterel. *Electronic Notes in Theoretical Computer Science*, 80:219–234, 2003.

[84] Lars Struyf, Stijn De Beugher, Dong Hoon Van Uytsel, Frans Kanters, and Toon Goedemé. The battle of the giants-a case study of GPU vs FPGA optimization for real-time image processing. In *International Conference on Pervasive and Embedded Computing and Communication Systems*, volume 2, pages 112–119. SCITEPRESS, 2014.

[85] Fahad Bin Muslim, Liang Ma, Mehdi Roozmeh, and Luciano Lavagno. Efficient fpga implementation of opencl high-performance computing applications via high-level synthesis. *IEEE Access*, 5:2747–2762, 2017.

[86] Joonas Järviluoma. Rapid prototyping from algorithm to fpga prototype. 2015.

[87] Chao Li, Yanjing Bi, Franck Marzani, and Fan Yang. Fast FPGA prototyping for real-time image processing with very high-level synthesis. *Journal of Real-Time Image Processing*, 16:1795–1812, 2019.

[88] Intel FPGA SDK for OpenCL Software Technology. https://www.intel.com/content/www/us/en/software/programmable/sdk-for-opencl/overview.html. (Accessed on 04/22/2023).

[89] Michael Meredith. High-level SystemC synthesis with forte's cynthesizer. *High-level synthesis: from algorithm to digital circuit*, pages 75–97, 2008.

[90] Kazutoshi Wakabayashi. CyberWorkBench: integrated design environment based on C-based behavior synthesis and verification. In *2005 IEEE VLSI-TSA International Symposium on VLSI Design, Automation and Test, 2005.(VLSI-TSA-DAT).*, pages 173–176. IEEE, 2005.

[91] Vitis Software Platform. https://www.xilinx.com/products/design-tools/vitis/vitis-platform.html. (Accessed on 06/21/2023).

[92] Intel Quartus Prime Pro Edition Design. https://www.intel.com/content/www/us/en/content-details/661713/intel-quartus-prime-pro-edition-design-software-version-19-2-for-windows.html. (Accessed on 04/24/2023).

[93] Johannes de Fine Licht, Maciej Besta, Simon Meierhans, and Torsten Hoefler. Transformations of high-level synthesis codes for high-performance computing. *IEEE Transactions on Parallel and Distributed Systems*, 32(5):1014–1029, 2020.

[94] Yufei Ma, Yu Cao, Sarma Vrudhula, and Jae-sun Seo. Optimizing loop operation and dataflow in FPGA acceleration of deep convolutional neural networks. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 45–54, 2017.

[95] Jichen Wang, Jun Lin, and Zhongfeng Wang. Efficient hardware architectures for deep convolutional neural network. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 65(6):1941–1953, 2017.

[96] Ralf Kundel, Kadir Eryigit, Jonas Markussen, Carsten Griwodz, Osama Aboud, Rhaban Hark, and Ralf Steinmetz. Host bypassing: Direct data piping from the network to the hardware accelerator. In *2021 IEEE 14th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*, pages 23–30. IEEE, 2021.

[97] Federico Civerchia, Maxime Pelcat, Luca Maggiani, Koteswararao Kondepu, Piero Castoldi, and Luca Valcarenghi. Is OPENCL driven reconfigurable hardware suitable for virtualising 5g infrastructure? *IEEE Transactions on Network and Service Management*, 17(2):849–863, 2020.

[98] Amirhossein Alimohammad, Saeed Fouladi Fard, Bruce F Cockburn, and Christian Schlegel. A compact single-FPGA fading-channel simulator. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 55(1):84–88, 2008.

[99] Kaitai Xiao and Weijie Zhang. Systematic study on hardware optimization of 5G communication. In *2021 5th International Conference on Computing Methodologies and Communication (ICCMC)*, pages 92–96. IEEE, 2021.

[100] Markus Hofer, Zhinan Xu, Dimitrios Vlastaras, Bernhard Schrenk, David Löschenbrand, Fredrik Tufvesson, and Thomas Zemen. Real-time geometry-based wireless channel emulation. *IEEE Transactions on Vehicular Technology*, 68(2):1631–1645, 2018.

[101] Jingyu Hua, Junfei Yang, Weidang Lu, Limin Meng, and Xutao Yu. Design of universal wireless channel generator accounting for the 3-D scatter distribution and hardware output. *IEEE Transactions on Instrumentation and Measurement*, 64(1):2–13, 2014.

[102] Pengda Huang, Matthew Jordan Tonnemacher, Yongjiu Du, Dinesh Rajan, and Joseph Camp. Towards massive MIMO channel emulation: Channel accuracy versus implementation resources. *IEEE Transactions on Vehicular Technology*, 69(5):4635–4651, 2020.

[103] Yang Yang, LI Tingpeng, CHEN Xiaomin, WANG Manxi, ZHU Qiuming, FENG Ruirui, DUAN Fuqiao, and Taotao Zhang. Real-time ray-based channel generation and emulation for UAV communications. *Chinese Journal of Aeronautics*, 35(9):106–116, 2022.

[104] Egor Endovitskiy, Aleksey Kureev, and Evgeny Khorov. Reducing computational complexity for the 3GPP TR 38.901 MIMO channel model. *IEEE Wireless Communications Letters*, 11(6):1133–1136, 2022.

[105] Nasir Ali Shah, Mihai T Lazarescu, Roberto Quasso, Salvatore Scarpina, and Luciano Lavagno. FPGA Acceleration of 3GPP Channel Model Emulator for 5G New Radio. *IEEE Access*, 10:119386–119401, 2022.

[106] R2021a - MATLAB & Simulink. https://it.mathworks.com/products/new_products/release2021a.html. (Accessed on 04/24/2023).

[107] Build MEX function or engine application - MATLAB mex - MathWorks. https://mathworks.com/help/matlab/ref/mex.html. (Accessed on 04/24/2023).

[108] Alveo U280 Data Center Accelerator Card. https://www.xilinx.com/products/boards-and-kits/alveo/u280.html. (Accessed on 04/24/2023).

[109] Intel Arria 10 GX 1150 FPGA Product Specifications. https://ark.intel.com/content/www/us/en/ark/products/210381/intel-arria-10-gx-1150-fpga.html. (Accessed on 04/24/2023).

[110] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.

[111] Enrico Calore and Sebastiano Fabio Schifano. Fer: A benchmark for the roofline analysis of fpga based hpc accelerators. *IEEE Access*, 10:94220–94234, 2022.

[112] Yu Jung Lo, Samuel Williams, Brian Van Straalen, Terry J Ligocki, Matthew J Cordery, Nicholas J Wright, Mary W Hall, and Leonid Oliker. Roofline model toolkit: A practical tool for architectural and program analysis. In *High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation: 5th International Workshop, PMBS 2014, New Orleans, LA, USA, November 16, 2014. Revised Selected Papers 5*, pages 129–148. Springer, 2015.

[113] Zhiguo Xu and Rajive Bagrodia. GPU-accelerated evaluation platform for high fidelity network modeling. In *21st International Workshop on Principles of Advanced and Distributed Simulation (PADS'07)*, pages 131–140. IEEE, 2007.

[114] Kangning Yan, Nianzu Zhang, Zhengbo Jiang, Yu Sheng, and Yiting Gao. A GPU-based Heterogeneous Computing Method to Speed up Wireless Channel Simulation. In *2022 International Conference on Microwave and Millimeter Wave Technology (ICMMT)*, pages 1–3, 2022.

[115] Ahmed Fathy Abdelrazek, Matthias Kaschub, Christian Blankenhorn, and Marc C Necker. A novel architecture using NVIDIA CUDA to speed up simulation of multi-path fast fading channels. In *VTC Spring 2009-IEEE 69th Vehicular Technology Conference*, pages 1–5. IEEE, 2009.

[116] Kevin C Borries, Glenn Judd, Daniel D Stancil, and Peter Steenkiste. FPGA-based channel simulator for a wireless network emulator. In *VTC Spring 2009-IEEE 69th Vehicular Technology Conference*, pages 1–5. IEEE, 2009.

[117] Nasir Ali Shah, Mihai T Lazarescu, Roberto Quasso, and Luciano Lavagno. Cuda-optimized gpu acceleration of 3gpp 3d channel model simulations for 5g network planning. *Electronics*, 12(15):3214, 2023.

[118] NVIDIA. CUDA Toolkit - Free Tools and Training | NVIDIA Developer. Accessed: 2023-06-02.

[119] NVIDIA. GeForce GTX 1070 Specifications | GeForce, 2021.

Appendix A

3GPP Channel Parameters

Table 7.5-6 **Channel model parameters for UMi-Street Canyon and UMa**

Scenarios		UMi - Street Canyon			UMa		
		LOS	**NLOS**	**O2I**	**LOS**	**NLOS**	**O2I**
Delay spread (DS) lgDS=$\log_{10}$(DS/1s)	μ_{lgDS}	-0.24 $\log_{10}(1+f_c)$ - 7.14	-0.24 $\log_{10}(1+f_c)$ - 6.83	-6.62	-6.955 - 0.0963 $\log_{10}(f_c)$	-6.28 - 0.204 $\log_{10}(f_c)$	-6.62
	σ_{lgDS}	0.38	0.16 $\log_{10}(1+f_c)$ + 0.28	0.32	0.66	0.39	0.32
AOD spread (ASD) lgASD=$\log_{10}$(ASD/1°)	μ_{lgASD}	-0.05 $\log_{10}(1+f_c)$ + 1.21	-0.23 $\log_{10}(1+f_c)$ + 1.53	1.25	1.06 + 0.1114 $\log_{10}(f_c)$	1.5 - 0.1144 $\log_{10}(f_c)$	1.25
	σ_{lgASD}	0.41	0.11 $\log_{10}(1+f_c)$ + 0.33	0.42	0.28	0.28	0.42
AOA spread (ASA) lgASA=$\log_{10}$(ASA/1°)	μ_{lgASA}	-0.08 $\log_{10}(1+f_c)$ + 1.73	-0.08 $\log_{10}(1+f_c)$ + 1.81	1.76	1.81	2.08 - 0.27 $\log_{10}(f_c)$	1.76
	σ_{lgASA}	0.014 $\log_{10}(1+f_c)$ + 0.28	0.05 $\log_{10}(1+f_c)$ + 0.3	0.16	0.20	0.11	0.16
ZOA spread (ZSA) lgZSA=$\log_{10}$(ZSA/1°)	μ_{lgZSA}	-0.1 $\log_{10}(1+f_c)$ + 0.73	-0.04 $\log_{10}(1+f_c)$ + 0.92	1.01	0.95	-0.3236 $\log_{10}(f_c)$ + 1.512	1.01
	σ_{lgZSA}	-0.04 $\log_{10}(1+f_c)$ + 0.34	-0.07 $\log_{10}(1+f_c)$ + 0.41	0.43	0.16	0.16	0.43
Shadow fading (*SF*) [dB]	σ_{SF}	See Table 7.4.1-1	See Table 7.4.1-1	7	See Table 7.4.1-1	See Table 7.4.1-1	7
K-factor (*K*) [dB]	μ_K	9	N/A	N/A	9	N/A	N/A
	σ_K	5	N/A	N/A	3.5	N/A	N/A
Cross-Correlations	*ASD* vs *DS*	0.5	0	0.4	0.4	0.4	0.4
	ASA vs *DS*	0.8	0.4	0.4	0.8	0.6	0.4
	ASA vs *SF*	-0.4	-0.4	0	-0.5	0	0
	ASD vs *SF*	-0.5	0	0.2	-0.5	-0.6	0.2
	DS vs *SF*	-0.4	-0.7	-0.5	-0.4	-0.4	-0.5
	ASD vs *ASA*	0.4	0	0	0	0.4	0
	ASD vs K	-0.2	N/A	N/A	0	N/A	N/A
	ASA vs K	-0.3	N/A	N/A	-0.2	N/A	N/A
	DS vs K	-0.7	N/A	N/A	-0.4	N/A	N/A
	SF vs K	0.5	N/A	N/A	0	N/A	N/A
Cross-Correlations [1)]	*ZSD* vs *SF*	0	0	0	0	0	0
	ZSA vs *SF*	0	0	0	-0.8	-0.4	0
	ZSD vs *K*	0	N/A	N/A	0	N/A	N/A
	ZSA vs *K*	0	N/A	N/A	0	N/A	N/A
	ZSD vs *DS*	0	-0.5	-0.6	-0.2	-0.5	-0.6
	ZSA vs *DS*	0.2	0	-0.2	0	0	-0.2
	ZSD vs *ASD*	0.5	0.5	-0.2	0.5	0.5	-0.2
	ZSA vs *ASD*	0.3	0.5	0	0	-0.1	0
	ZSD vs *ASA*	0	0	0	-0.3	0	0
	ZSA vs *ASA*	0	0.2	0.5	0.4	0	0.5
	ZSD vs *ZSA*	0	0	0.5	0	0	0.5
Delay scaling parameter r_τ		3	2.1	2.2	2.5	2.3	2.2
XPR [dB]	μ_{XPR}	9	8.0	9	8	7	9
	σ_{XPR}	3	3	5	4	3	5
Number of clusters N		12	19	12	12	20	12
Number of rays per cluster M		20	20	20	20	20	20
Cluster *DS* (c_{DS}) in [ns]		5	11	11	max(0.25, 6.5622 -3.4084 $\log_{10}(f_c)$)	max(0.25, 6.5622 -3.4084 $\log_{10}(f_c)$)	11
Cluster *ASD* (c_{ASD}) in [deg]		3	10	5	5	2	5

Cluster *ASA* (c_{ASA}) in [deg]		17	22	8	11	15	8
Cluster *ZSA* (c_{ZSA}) in [deg]		7	7	3	7	7	3
Per cluster shadowing std ζ [dB]		3	3	4	3	3	4
Correlation distance in the horizontal plane [m]	*DS*	7	10	10	30	40	10
	ASD	8	10	11	18	50	11
	ASA	8	9	17	15	50	17
	SF	10	13	7	37	50	7
	K	15	N/A	N/A	12	N/A	N/A
	ZSA	12	10	25	15	50	25
	ZSD	12	10	25	15	50	25

f_c is carrier frequency in GHz; d_{2D} is BS-UT distance in km.

NOTE 1: *DS* = rms delay spread, *ASD* = rms azimuth spread of departure angles, *ASA* = rms azimuth spread of arrival angles, *ZSD* = rms zenith spread of departure angles, *ZSA* = rms zenith spread of arrival angles, *SF* = shadow fading, and *K* = Ricean K-factor.

NOTE 2: The sign of the shadow fading is defined so that positive SF means more received power at UT than predicted by the path loss model.

NOTE 3: All large scale parameters are assumed to have no correlation between different floors.

NOTE 4: The following notation for mean (μ_{lgX}=mean{$\log_{10}$(X) }) and standard deviation (σ_{lgX}=std{$\log_{10}$(X) }) is used for logarithmized parameters X.

NOTE 5: For all considered scenarios the AOD/AOA distributions are modelled by a wrapped Gaussian distribution, the ZOD/ZOA distributions are modelled by a Laplacian distribution and the delay distribution is modelled by an exponential distribution.

NOTE 6: For UMa and frequencies below 6 GHz, use f_c = 6 when determining the values of the frequency-dependent LSP values

NOTE 7: For UMi and frequencies below 2 GHz, use f_c = 2 when determining the values of the frequency-dependent LSP values

Scenarios		RMa			Indoor-Office	
		LOS	NLOS	O2I	LOS	NLOS
Delay spread (DS) lgDS=$\log_{10}$(DS/1s)	μ_{lgDS}	-7.49	-7.43	-7.47	-0.01 $\log_{10}(1+f_c)$ - 7.692	-0.28 $\log_{10}(1+f_c)$ - 7.173
	σ_{lgDS}	0.55	0.48	0.24	0.18	0.10 $\log_{10}(1+f_c)$ + 0.055
AOD spread (ASD) lgASD=$\log_{10}$(ASD/1°)	μ_{lgASD}	0.90	0.95	0.67	1.60	1.62
	σ_{lgASD}	0.38	0.45	0.18	0.18	0.25
AOA spread (ASA) lgASA=$\log_{10}$(ASA/1°)	μ_{lgASA}	1.52	1.52	1.66	-0.19 $\log_{10}(1+f_c)$ + 1.781	-0.11 $\log_{10}(1+f_c)$ + 1.863
	σ_{lgASA}	0.24	0.13	0.21	0.12 $\log_{10}(1+f_c)$ + 0.119	0.12 $\log_{10}(1+f_c)$ + 0.059
ZOA spread (ZSA) lgZSA=$\log_{10}$(ZSA/1°)	μ_{lgZSA}	0.47	0.58	0.93	-0.26 $\log_{10}(1+f_c)$ + 1.44	-0.15 $\log_{10}(1+f_c)$ + 1.387
	σ_{lgZSA}	0.40	0.37	0.22	-0.04 $\log_{10}(1+f_c)$ + 0.264	-0.09 $\log_{10}(1+f_c)$ + 0.746
Shadow fading (SF) [dB]	σ_{SF}	See Table 7.4.1-1		8	See Table 7.4.1-1	
K-factor (*K*) [dB]	μ_K	7	N/A	N/A	7	N/A
	σ_K	4	N/A	N/A	4	N/A
Cross-Correlations	*ASD* vs *DS*	0	-0.4	0	0.6	0.4
	ASA vs *DS*	0	0	0	0.8	0
	ASA vs *SF*	0	0	0	–0.5	–0.4
	ASD vs *SF*	0	0.6	0	–0.4	0
	DS vs *SF*	-0.5	-0.5	0	–0.8	–0.5
	ASD vs *ASA*	0	0	-0.7	0.4	0
	ASD vs K	0	N/A	N/A	0	N/A
	ASA vs K	0	N/A	N/A	0	N/A
	DS vs K	0	N/A	N/A	-0.5	N/A
	SF vs K	0	N/A	N/A	0.5	N/A
Cross-Correlations [1)]	*ZSD* vs *SF*	0.01	-0.04	0	0.2	0
	ZSA vs *SF*	-0.17	-0.25	0	0.3	0
	ZSD vs *K*	0	N/A	N/A	0	N/A
	ZSA vs *K*	-0.02	N/A	N/A	0.1	N/A
	ZSD vs *DS*	-0.05	-0.10	0	0.1	-0.27
	ZSA vs *DS*	0.27	-0.40	0	0.2	-0.06
	ZSD vs *ASD*	0.73	0.42	0.66	0.5	0.35
	ZSA vs *ASD*	-0.14	-0.27	0.47	0	0.23
	ZSD vs *ASA*	-0.20	-0.18	-0.55	0	-0.08
	ZSA vs *ASA*	0.24	0.26	-0.22	0.5	0.43
	ZSD vs *ZSA*	-0.07	-0.27	0	0	0.42
Delay scaling parameter r_τ		3.8	1.7	1.7	3.6	3
XPR [dB]	μ_{XPR}	12	7	7	11	10
	σ_{XPR}	4	3	3	4	4
Number of clusters N		11	10	10	15	19
Number of rays per cluster M		20	20	20	20	20
Cluster *DS* (c_{DS}) in [ns]		N/A	N/A	N/A	N/A	N/A
Cluster *ASD* (c_{ASD}) in [deg]		2	2	2	5	5

Cluster *ASA* (c_{ASA}) in [deg]		3	3	3	8	11
Cluster *ZSA* (c_{ZSA}) in [deg]		3	3	3	9	9
Per cluster shadowing std ζ [dB]		3	3	3	6	3
Correlation distance in the horizontal plane [m]	*DS*	50	36	36	8	5
	ASD	25	30	30	7	3
	ASA	35	40	40	5	3
	SF	37	120	120	10	6
	K	40	N/A	N/A	4	N/A
	ZSA	15	50	50	4	4
	ZSD	15	50	50	4	4

f_c is carrier frequency in GHz; d_{2D} is BS-UT distance in km.

NOTE 1: *DS* = rms delay spread, *ASD* = rms azimuth spread of departure angles, *ASA* = rms azimuth spread of arrival angles, *ZSD* = rms zenith spread of departure angles, *ZSA* = rms zenith spread of arrival angles, *SF* = shadow fading, and *K* = Ricean K-factor.

NOTE 2: The sign of the shadow fading is defined so that positive SF means more received power at UT than predicted by the path loss model.

NOTE 3: The following notation for mean (μ_{lgX}=mean{$\log_{10}$(X) }) and standard deviation (σ_{lgX}=std{$\log_{10}$(X) }) is used for logarithmized parameters X.

NOTE 4: Void.

NOTE 5: For all considered scenarios the AOD/AOA distributions are modelled by a wrapped Gaussian distribution, the ZOD/ZOA distributions are modelled by a Laplacian distribution and the delay distribution is modelled by an exponential distribution.

NOTE 6: For InH and frequencies below 6 GHz, use f_c = 6 when determining the values of the frequency-dependent LSP values

Appendix B

FPGA HLS code

```
#include "constants.h"
#include <math.h>
#include <stdio.h>
#define BUFFER_SIZE (const_nSymbDataPerLink * const_NSPS)
const unsigned int dim_rx = const_nRxAntennas;
const unsigned int dim_OutBuffer = BUFFER_SIZE;
const unsigned int n_cluster = const_numCluster;
const unsigned int dim_circular_buffer_position = size_circular_buffer_position;

extern "C"
{

    // Channel Application kernel
    void kernSingleLink5gWithTdl(
        // singleLink5gChannelModelApplication method parameters
        const double *X_input_symb_I,
        const double *X_input_symb_Q,
        const unsigned int firstSubFrame,
        double *tdlChannelCoefOld_real,
        double *tdlChannelCoefOld_imag,
        const unsigned int circBuffIdx,
        double *Y_out_symb_real,
        double *Y_out_symb_imag,
        unsigned int *circular_buffer_position,
        unsigned int test,
        // tdlcompute method parameters
        const int los_info,
        const double cdlTimeVector,
        double *speedDesctiption,
        double *speedDesctiption_LOS,
        double *clusterDesctiptionReal,
        double *clusterDesctiptionImag,
        double *clusterDesctiption_LosReal,
        double *clusterDesctiption_LosImag,
        double *out_tdlChannelCoefReal,
        double *out_tdlChannelCoefImag,
        unsigned int applyChannel)
    {
// kernSingleLink5g axi master interface
#pragma HLS INTERFACE m_axi port = X_input_symb_I offset = slave bundle =
hbm0
#pragma HLS INTERFACE m_axi port = X_input_symb_Q offset = slave bundle =
hbm1
#pragma HLS INTERFACE m_axi port = tdlChannelCoefOld_real offset = slave bundle =
hbm2
#pragma HLS INTERFACE m_axi port = tdlChannelCoefOld_imag offset = slave bundle =
hbm3
#pragma HLS INTERFACE m_axi port = Y_out_symb_real offset = slave bundle =
hbm0
#pragma HLS INTERFACE m_axi port = Y_out_symb_imag offset = slave bundle = hbm1
#pragma HLS INTERFACE m_axi port = circular_buffer_position offset = slave bundle = hbm4
// tdlcompute axi master interface
#pragma HLS INTERFACE m_axi port = speedDesctiption offset = slave bundle =
hbm5
#pragma HLS INTERFACE m_axi port = speedDesctiption_LOS offset = slave bundle =
hbm6
#pragma HLS INTERFACE m_axi port = clusterDesctiptionReal offset = slave bundle =
hbm7
#pragma HLS INTERFACE m_axi port = clusterDesctiptionImag offset = slave bundle =
hbm8
#pragma HLS INTERFACE m_axi port = clusterDesctiption_LosReal offset = slave bundle =
hbm9
#pragma HLS INTERFACE m_axi port = clusterDesctiption_LosImag offset = slave bundle =
hbm10
#pragma HLS INTERFACE m_axi port = out_tdlChannelCoefReal offset = slave bundle =
hbm4
#pragma HLS INTERFACE m_axi port = out_tdlChannelCoefImag offset = slave bundle =
hbm5

// kernSingleLink5g axilite slave interface
#pragma HLS INTERFACE s_axilite port = firstSubFrame
#pragma HLS INTERFACE s_axilite port = circBuffIdx
#pragma HLS INTERFACE s_axilite port = test
// tdlcompute axilite slave interface
#pragma HLS INTERFACE s_axilite port = los_info
#pragma HLS INTERFACE s_axilite port = cdlTimeVector
#pragma HLS INTERFACE s_axilite port = applyChannel

        // kernSingleLink5g local buffers
        double mem_X_input_symb_I[const_nSymbDataPerLink * const_NSPS];
        double mem_X_input_symb_Q[const_nSymbDataPerLink * const_NSPS];
        double mem_tdlChannelCoefOld_real[size_tdlChannelCoefOld];
        double mem_tdlChannelCoefOld_imag[size_tdlChannelCoefOld];
        double mem_circular_bufferI[const_numCluster][size_circular_buffer];
        double mem_circular_bufferQ[const_numCluster][size_circular_buffer];
        double mem_Y_out_symb_real[BUFFER_SIZE];
        double mem_Y_out_symb_imag[BUFFER_SIZE];

```

```
76          double precomputed_tdlChannelCoefOld_real[const_numCluster];
77          double precomputed_tdlChannelCoefOld_imag[const_numCluster];
78          double precomputed_tdlChannelCoef_real[const_numCluster];
79          double precomputed_tdlChannelCoef_imag[const_numCluster];
80
81          // tdlcompute local buffers
82          const int col_size = const_nTxAntennas * const_nRxAntennas * const_numCluster;
83          double buff_TimeVector = cdlTimeVector;
84          double buff_speedDesctiption_LOS[const_nRxAntennas];
85          double buff_speedDesctiption[NRAYS];
86          double buff_clusterDesctiption_LosReal[const_nRxAntennas * const_nTxAntennas];
87          double buff_clusterDesctiption_LosImag[const_nRxAntennas * const_nTxAntennas];
88          double buff_tdlChannelCoef_Real[col_size];
89          double buff_tdlChannelCoef_Imag[col_size];
90          double buff_clusterDesctiptionReal[NRAYS];
91          double buff_clusterDesctiptionImag[NRAYS];
92
93  #pragma HLS ARRAY_PARTITION variable = buff_speedDesctiption_LOS complete
94  #pragma HLS ARRAY_PARTITION variable = buff_speedDesctiption complete
95  #pragma HLS ARRAY_PARTITION variable = buff_clusterDesctiption_LosReal complete
96  #pragma HLS ARRAY_PARTITION variable = buff_clusterDesctiption_LosImag complete
97  #pragma HLS ARRAY_PARTITION variable = buff_clusterDesctiptionReal complete
98  #pragma HLS ARRAY_PARTITION variable = buff_clusterDesctiptionImag complete
99
100 #pragma HLS bind_storage variable = mem_X_input_symb_I type = RAM_2P impl = URAM
101 #pragma HLS bind_storage variable = mem_X_input_symb_Q type = RAM_2P impl = URAM
102 #pragma HLS bind_storage variable = mem_Y_out_symb_real type = RAM_2P impl = URAM
103 #pragma HLS bind_storage variable = mem_Y_out_symb_imag type = RAM_2P impl = URAM
104 #pragma HLS bind_storage variable = mem_circular_bufferI type = RAM_2P impl = BRAM
105 #pragma HLS bind_storage variable = mem_circular_bufferQ type = RAM_2P impl = BRAM
106
107 #pragma HLS ARRAY_PARTITION variable = buff_tdlChannelCoef_Real cyclic factor = 4 dim = 1
108 #pragma HLS ARRAY_PARTITION variable = buff_tdlChannelCoef_Imag cyclic factor = 4 dim = 1
109 #pragma HLS ARRAY_PARTITION variable = mem_tdlChannelCoefOld_real cyclic factor = 4 dim
    = 1
110 #pragma HLS ARRAY_PARTITION variable = mem_tdlChannelCoefOld_imag cyclic factor = 4 dim
    = 1
111 #pragma HLS ARRAY_PARTITION variable = mem_circular_bufferI complete dim = 1
112 #pragma HLS ARRAY_PARTITION variable = mem_circular_bufferQ complete dim = 1
113 #pragma HLS ARRAY_PARTITION variable = mem_circular_bufferI cyclic factor = 2 dim = 2
114 #pragma HLS ARRAY_PARTITION variable = mem_circular_bufferQ cyclic factor = 2 dim = 2
115 #pragma HLS ARRAY_PARTITION variable = precomputed_tdlChannelCoef_real complete
116 #pragma HLS ARRAY_PARTITION variable = precomputed_tdlChannelCoef_imag complete
117 #pragma HLS ARRAY_PARTITION variable = precomputed_tdlChannelCoefOld_real complete
118 #pragma HLS ARRAY_PARTITION variable = precomputed_tdlChannelCoefOld_imag complete
119
120 #pragma HLS ARRAY_PARTITION variable = mem_X_input_symb_I cyclic factor = 8
121 #pragma HLS ARRAY_PARTITION variable = mem_X_input_symb_Q cyclic factor = 8
122 #pragma HLS ARRAY_PARTITION variable = mem_Y_out_symb_real cyclic factor = 8
123 #pragma HLS ARRAY_PARTITION variable = mem_Y_out_symb_imag cyclic factor = 8
124
125         unsigned int circBuf = 0;
126
127 #if SPREADING_FACTOR == 1
128         unsigned int mem_position[size_position] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1
        , 1, 1, 1, 1, 1, 1, 2, 2, 2, 2};
129 #elif SPREADING_FACTOR == 10
130         unsigned int mem_position[size_position] = {0, 1, 1, 1, 1, 1, 2, 2, 2, 2, 3, 3, 5
        , 6, 8, 9, 10, 14, 15, 18, 20, 21, 24};
131 #elif SPREADING_FACTOR == 100
132         unsigned int mem_position[size_position] = {0, 5, 11, 10, 14, 15, 18, 25, 18, 18,
         28, 26, 54, 63, 76, 88, 99, 139, 149, 178, 202, 210, 235};
133 #elif SPREADING_FACTOR == 30
134         unsigned int mem_position[size_position] = {0, 2, 3, 3, 4, 4, 6, 7, 5, 5, 8, 8,
        16, 19, 23, 26, 30, 42, 45, 53, 61, 63, 71};
135 #elif SPREADING_FACTOR == 300
136         unsigned int mem_position[size_position] = {0, 16, 32, 31, 42, 44, 55, 75, 54, 55
        , 84, 78, 163, 188, 228, 263, 297, 417, 446, 534, 606, 631, 705};
137 #endif
138 #pragma HLS ARRAY_PARTITION variable = mem_position complete
139
140         // FROM GLOBAL TO LOCAL MEMORY
141         if (applyChannel)
142         {
143         inTdl:
144             for (unsigned int i = 0; i < size_tdlChannelCoef; i++)
145             {
146                 mem_tdlChannelCoefOld_real[i] = tdlChannelCoefOld_real[i];
147                 mem_tdlChannelCoefOld_imag[i] = tdlChannelCoefOld_imag[i];
148             }
149         }
150
151     loopBuffLos:
152         for (unsigned int idxBuff = 0; idxBuff < const_nRxAntennas * const_nTxAntennas;
        ++idxBuff)
153         {
154             buff_clusterDesctiption_LosReal[idxBuff] = clusterDesctiption_LosReal[idxBuff
            ];
```

```
155             buff_clusterDesctiption_LosImag[idxBuff] = clusterDesctiption_LosImag[idxBuff
                ];
156         }
157
158     loopBuffSpeed:
159         for (unsigned int rxIdxBuff = 0; rxIdxBuff < const_nRxAntennas; ++rxIdxBuff)
160         {
161             buff_speedDesctiption_LOS[rxIdxBuff] = speedDesctiption_LOS[rxIdxBuff];
162         }
163
164         // KERNEL START
165
166     rxAnt:
167         for (unsigned int rxAntennaIndex = 0; rxAntennaIndex < const_nRxAntennas; ++
            rxAntennaIndex)
168         {
169 #pragma HLS LOOP_TRIPCOUNT min = dim_rx max = dim_rx
170         txAnt:
171             for (unsigned int txAntennaIndex = 0; txAntennaIndex < const_nTxAntennas; ++
                txAntennaIndex)
172             {
173 #pragma HLS DEPENDENCE false intra variable = mem_Y_out_symb_real
174 #pragma HLS DEPENDENCE false intra variable = mem_Y_out_symb_imag
175
176                 unsigned int jj_cb = const_singleCircularBufferDim * const_nTxAntennas *
                    rxAntennaIndex + const_singleCircularBufferDim * txAntennaIndex;
177                 unsigned int j = circBuffIdx;
178
179                 if (applyChannel)
180                 {
181                 cpyX:
182                     for (unsigned int idx = 0; idx < const_nsymb; ++idx)
183                     {
184 #pragma HLS UNROLL factor = 8
185                         mem_X_input_symb_I[idx] = X_input_symb_I[idx + txAntennaIndex *
                            const_nsymb];
186                         mem_X_input_symb_Q[idx] = X_input_symb_Q[idx + txAntennaIndex *
                            const_nsymb];
187                     }
188                 }
189                 double mem_deltaCh_real[size_deltaCh];
190                 double mem_deltaCh_imag[size_deltaCh];
191                 double mem_tapVectorI[size_tapVector];
192                 double mem_tapVectorQ[size_tapVector];
193 #pragma HLS ARRAY_PARTITION variable = mem_tapVectorI complete
194 #pragma HLS ARRAY_PARTITION variable = mem_tapVectorQ complete
195 #pragma HLS ARRAY_PARTITION variable = mem_deltaCh_real cyclic factor = 23
196 #pragma HLS ARRAY_PARTITION variable = mem_deltaCh_imag cyclic factor = 23
197
198             clusterIdx:
199                 for (unsigned int cdlInd = 0; cdlInd < const_numCluster; cdlInd++)
200                 {
201                     //! Index in writing the channel matrix (CDL)
202                     unsigned int chIdx = const_nRxAntennas * const_nTxAntennas * cdlInd +
                         const_nTxAntennas * rxAntennaIndex + txAntennaIndex;
203
204                     unsigned int idxWriteBuff = NRAYS * (const_nRxAntennas *
                        const_nTxAntennas * cdlInd + const_nTxAntennas * rxAntennaIndex +
                        txAntennaIndex);
205                     unsigned int idxWrite2Buff = NRAYS * (const_nRxAntennas * cdlInd +
                        rxAntennaIndex);
206
207                 loopBuffRays:
208                     for (unsigned int buffRayIdx = 0; buffRayIdx < NRAYS; buffRayIdx++)
209                     {
210                         buff_speedDesctiption[buffRayIdx] = speedDesctiption[
                            idxWrite2Buff + buffRayIdx];
211                         buff_clusterDesctiptionReal[buffRayIdx] = clusterDesctiptionReal[
                            idxWriteBuff + buffRayIdx];
212                         buff_clusterDesctiptionImag[buffRayIdx] = clusterDesctiptionImag[
                            idxWriteBuff + buffRayIdx];
213                     }
214
215                     double tmp_tdlChannelCoef_Real = 0.0;
216                     double tmp_tdlChannelCoef_Imag = 0.0;
217                     double tmp_tdlChannelCoef_Real_part[NRAYS_DIVISION];
218                     double tmp_tdlChannelCoef_Imag_part[NRAYS_DIVISION];
219 #pragma HLS ARRAY_PARTITION variable = tmp_tdlChannelCoef_Real_part complete
220 #pragma HLS ARRAY_PARTITION variable = tmp_tdlChannelCoef_Imag_part complete
221
222                 loopNRAYS:
223                     for (unsigned int rayIdx = 0; rayIdx < NRAYS; rayIdx++)
224                     {
225 #pragma HLS pipeline II = 6
226                         double j2pivt = buff_speedDesctiption[rayIdx] * cdlTimeVector;
227                         //! Phase value computation
228                         double temp_expReal = (double)cos((float)j2pivt);
229                         double temp_expImag = (double)sin((float)j2pivt);
```

```
230
231                         //! Power Computation
232                         tmp_tdlChannelCoef_Real += (buff_clusterDesctiptionReal[rayIdx] *
                             temp_expReal) - (buff_clusterDesctiptionImag[rayIdx] *
                            temp_expImag);
233                         tmp_tdlChannelCoef_Imag += (buff_clusterDesctiptionImag[rayIdx] *
                             temp_expReal) + (buff_clusterDesctiptionReal[rayIdx] *
                            temp_expImag);
234                     }
235                     // computation of LOS
236                     if ((los_info == 1) && (cdlInd == 0))
237                     {
238                         double j2pivtLos = buff_speedDesctiption_LOS[rxAntennaIndex] *
                            cdlTimeVector;
239                         double temp_expLosReal = (double)cos((float)j2pivtLos);
240                         double temp_expLosImag = (double)sin((float)j2pivtLos);
241                         unsigned int idxWrite = const_nTxAntennas * rxAntennaIndex +
                            txAntennaIndex;
242                         //! Power Compuitation
243                         tmp_tdlChannelCoef_Real += (buff_clusterDesctiption_LosReal[
                            idxWrite] * temp_expLosReal) - (buff_clusterDesctiption_LosImag[
                            idxWrite] * temp_expLosImag);
244                         tmp_tdlChannelCoef_Imag += (buff_clusterDesctiption_LosImag[
                            idxWrite] * temp_expLosReal) + (buff_clusterDesctiption_LosReal[
                            idxWrite] * temp_expLosImag);
245                     } // End if LOS
246
247                     buff_tdlChannelCoef_Real[chIdx] = tmp_tdlChannelCoef_Real;
248                     buff_tdlChannelCoef_Imag[chIdx] = tmp_tdlChannelCoef_Imag;
249
250                     if (applyChannel)
251                     {
252                         double tmp1_real = buff_tdlChannelCoef_Real[chIdx];
253                         double tmp1_imag = buff_tdlChannelCoef_Imag[chIdx];
254                         double tmp0_real = mem_tdlChannelCoefOld_real[chIdx];
255                         double tmp0_imag = mem_tdlChannelCoefOld_imag[chIdx];
256
257                         double N = const_nSymbDataPerLink * const_NSPS;
258                         mem_deltaCh_real[cdlInd] = (tmp1_real - tmp0_real) / N;
259                         mem_deltaCh_imag[cdlInd] = (tmp1_imag - tmp0_imag) / N;
260
261                         precomputed_tdlChannelCoefOld_real[cdlInd] = tmp0_real;
262                         precomputed_tdlChannelCoefOld_imag[cdlInd] = tmp0_imag;
263                         precomputed_tdlChannelCoef_real[cdlInd] = tmp1_real;
264                         precomputed_tdlChannelCoef_imag[cdlInd] = tmp1_imag;
265                     }
266                 }
267
268                 if (applyChannel)
269                 {
270                 nSymb:
271                     for (unsigned int i = 0; i < const_nSymbDataPerLink * const_NSPS; ++i
                        )
272                     {
273 #pragma HLS PIPELINE II = 1
274 #pragma HLS unroll factor = 2
275 #pragma HLS dependence variable = mem_tapVectorI inter false
276 #pragma HLS dependence variable = mem_tapVectorQ inter false
277 #pragma HLS dependence variable = precomputed_tdlChannelCoef_real inter false
278 #pragma HLS dependence variable = precomputed_tdlChannelCoef_imag inter false
279 #pragma HLS dependence variable = precomputed_tdlChannelCoefOld_real inter false
280 #pragma HLS dependence variable = precomputed_tdlChannelCoefOld_imag inter false
281 #pragma HLS dependence variable = precomputed_tdlChannelCoef_real intra false
282 #pragma HLS dependence variable = precomputed_tdlChannelCoef_imag intra false
283 #pragma HLS dependence variable = precomputed_tdlChannelCoefOld_real intra false
284 #pragma HLS dependence variable = precomputed_tdlChannelCoefOld_imag intra false
285
286                         double accI = 0.0;
287                         double accQ = 0.0;
288                         double symbI = mem_X_input_symb_I[i];
289                         double symbQ = mem_X_input_symb_Q[i];
290
291                         if (test)
292                         {
293                             double tdlI = buff_tdlChannelCoef_Real[const_nTxAntennas *
                                rxAntennaIndex + txAntennaIndex];
294                             double tdlQ = buff_tdlChannelCoef_Imag[const_nTxAntennas *
                                rxAntennaIndex + txAntennaIndex];
295                             accI = symbI * tdlI - symbQ * tdlQ;
296                             accQ = symbI * tdlQ + symbQ * tdlI;
297                         }
298                         else
299                         {
300                             double opdI[const_numCluster];
301                             double opdQ[const_numCluster];
302                             double reg_accI[const_numCluster];
303                             double reg_accQ[const_numCluster];
304
```

```
305                         nClust:
306                             for (unsigned int l = 0; l < const_numCluster; l++)
307                             {
308 #pragma HLS UNROLL factor = 23
309                                 mem_circular_bufferI[l][(jj_cb + j) & (
                                    const_circularBufferDim - 1)] = symbI;
310                                 mem_circular_bufferQ[l][(jj_cb + j) & (
                                    const_circularBufferDim - 1)] = symbQ;
311
312                                 unsigned int chIdx = const_nTxAntennas *
                                    const_nRxAntennas * l + const_nTxAntennas *
                                    rxAntennaIndex + txAntennaIndex;
313
314                                 double tapVI = firstSubFrame ?
                                    precomputed_tdlChannelCoef_real[l] : (((i + 1) *
                                    mem_deltaCh_real[l]) + precomputed_tdlChannelCoefOld_real
                                    [l]);
315                                 double tapVQ = firstSubFrame ?
                                    precomputed_tdlChannelCoef_imag[l] : (((i + 1) *
                                    mem_deltaCh_imag[l]) + precomputed_tdlChannelCoefOld_imag
                                    [l]);
316
317                                 if (i == const_nSymbDataPerLink * const_NSPS - 1)
318                                 {
319                                     mem_tapVectorI[l] = tapVI;
320                                     mem_tapVectorQ[l] = tapVQ;
321                                 }
322
323                                 unsigned int index;
324                                 if (j < mem_position[l])
325                                 {
326                                     index = (j + const_singleCircularBufferDim -
                                        mem_position[l]);
327                                 }
328                                 else
329                                 {
330                                     index = (j - mem_position[l]);
331                                 }
332                                 opdI[l] = mem_circular_bufferI[l][(jj_cb + index) & (
                                    const_circularBufferDim - 1)];
333                                 opdQ[l] = mem_circular_bufferQ[l][(jj_cb + index) & (
                                    const_circularBufferDim - 1)];
334                                 if (j == index)
335                                 {
336                                     reg_accI[l] = symbI * tapVI - symbQ * tapVQ;
337                                     reg_accQ[l] = symbI * tapVQ + symbQ * tapVI;
338                                 }
339                                 else
340                                 {
341                                     reg_accI[l] = opdI[l] * tapVI - opdQ[l] * tapVQ;
342                                     reg_accQ[l] = opdI[l] * tapVQ + opdQ[l] * tapVI;
343                                 }
344                             }
345
346                             for (unsigned int i = 0; i < const_numCluster; i++)
347                             {
348                                 accI += reg_accI[i];
349                                 accQ += reg_accQ[i];
350                             }
351                         }
352
353                         mem_Y_out_symb_real[i] *= (txAntennaIndex != 0);
354                         mem_Y_out_symb_imag[i] *= (txAntennaIndex != 0);
355                         mem_Y_out_symb_real[i] += accI;
356                         mem_Y_out_symb_imag[i] += accQ;
357
358                         j = (j + 1) & (const_singleCircularBufferDim - 1);
359                     }
360
361                     circBuf = j;
362
363                 saveTdl:
364                     for (int idx = 0; idx < const_numCluster; ++idx)
365                     {
366                         unsigned chIdx = const_nTxAntennas * const_nRxAntennas * idx +
                            const_nTxAntennas * rxAntennaIndex + txAntennaIndex;
367                         mem_tdlChannelCoefOld_real[chIdx] = mem_tapVectorI[idx];
368                         mem_tdlChannelCoefOld_imag[chIdx] = mem_tapVectorQ[idx];
369                     }
370                 }
371             }
372
373             if (applyChannel)
374             {
375             outYoutSymb:
376                 for (unsigned int i = BUFFER_SIZE * rxAntennaIndex; i < BUFFER_SIZE * (
                    rxAntennaIndex + 1); ++i)
377                 {
```

```
#pragma HLS LOOP_TRIPCOUNT min = dim_OutBuffer max = dim_OutBuffer
#pragma HLS UNROLL factor = 8
                    Y_out_symb_real[i] = mem_Y_out_symb_real[i % BUFFER_SIZE];
                    Y_out_symb_imag[i] = mem_Y_out_symb_imag[i % BUFFER_SIZE];
                }
            }
        }

    outTdl:
        for (unsigned int i = 0; i < col_size; ++i)
        {
            out_tdlChannelCoefReal[i] = buff_tdlChannelCoef_Real[i];
            out_tdlChannelCoefImag[i] = buff_tdlChannelCoef_Imag[i];
        }

        if (applyChannel)
        {
        outTdlOld:
            for (unsigned int i = 0; i < size_tdlChannelCoefOld; ++i)
            {
                tdlChannelCoefOld_real[i] = mem_tdlChannelCoefOld_real[i];
                tdlChannelCoefOld_imag[i] = mem_tdlChannelCoefOld_imag[i];
            }

            circular_buffer_position[0] = circBuf;
        }
    }
}
```