FPGA Acceleration of 3GPP Channel Model Emulator for 5G New Radio

IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Digital Object Identifier 10.1109/ACCESS.2017.DOI

FPGA Acceleration of 3GPP Channel Model Emulator for 5G New Radio

NASIR ALI SHAH[1], (Student Member, IEEE), LUCIANO LAVAGNO [1], (Senior Member, IEEE), MIHAI T. LAZARESCU [1], (Senior Member, IEEE), ROBERTO QUASSO [2], and SALVATORE SCARPINA

[1]Department of Electronics and Telecommunications, Politecnico di Torino, Corso Duca degli Abruzzi, 24, Torino 10129, Italy
[2]Innovation Department Telecom Italia

Corresponding author: Nasir Ali Shah (e-mail: nasir.shah@polito.it).

This work was supported in part by the Innovation Department of TIM S.p.A. under Grant S21AAPIA.

ABSTRACT The channel model is by far the most computing intensive part of the link level simulations of multiple-input and multiple-output (MIMO) fifth-generation new radio (5G NR) communication systems. Simulation effort further increases when using more realistic geometry-based channel models, such as the three-dimensional spatial channel model (3D-SCM). Channel emulation is used for functional and performance verification of such models in the network planning phase. These models use multiple finite impulse response (FIR) filters and have a very high degree of parallelism which can be exploited for accelerated execution on Field Programmable Gate Array (FPGA) and Graphics Processing Unit (GPU) platforms. This paper proposes an efficient re-configurable implementation of the 3rd generation partnership project (3GPP) 3D-SCM on FPGAs using a design flow based on high-level synthesis (HLS). It studies the effect of various HLS optimization techniques on the total latency and hardware resource utilization on Xilinx Alveo U280 and Intel Arria 10GX 1150 high-performance FPGAs, using in both cases the commercial HLS tools of the producer. The channel model accuracy is preserved using double precision floating point arithmetic. This work analyzes in detail the effort to target the FPGA platforms using HLS tools, both in terms of common parallelization effort (shared by both FPGAs), and in terms of platform-specific effort, different for Xilinx and Intel FPGAs. Compared to the baseline general-purpose central processing unit (CPU) implementation, the achieved speedups are **65X** and **95X** using the Xilinx UltraScale+ and Intel Arria FPGA platform respectively, when using a Double Data Rate (DDR) memory interface. The FPGA-based designs also achieved **~3X** better performance compared to a similar technology node NVIDIA GeForce GTX 1070 GPU, while consuming **~4X** less energy. The FPGA implementation speedup improves up to **173X** over the CPU baseline when using the Xilinx UltraRAM (URAM) and High-Bandwidth Memory (HBM) resources, also achieving **6X** lower latency and **12X** lower energy consumption than the GPU implementation.

INDEX TERMS Channel emulator, FPGA, fifth-generation new radio, hardware acceleration, high-level synthesis

I. INTRODUCTION

CHANNEL model simulation has become an essential part of mobile network planning. Every new cellular technology undergoes a critical simulation phase both before and during the physical deployment phase. It is thus essential to model the channel accurately for the design and evaluation of fifth-generation new radio (5G NR) and beyond wireless networks [1]. To obtain a realistic representation of the propagation effects, thousands of radio frequency parameters need to be adjusted. Parameter recalculation is needed even after the deployment, whenever the network configuration changes (i.e., the number or position of antennas change). Geometry-based stochastic model (GBSM) is a popular method for accurately characterizing channels in simulation environments [2]. Several channel models have been developed by different groups, such as 3rd generation partnership project (3GPP) [3], Wireless World Initiative New Radio II (WINNER II) [4], European Cooperation in Science and Technology (COST) 2100 [5], Mobile and wireless communications Enablers for the Twenty-twenty Information Society (METIS) [6], International Telecommunications Union Radio-communication Sector (ITU-R) [7], Millimeter-Wave Evolution for Backhaul

and Access (MiWEBA) [8] and NYU WIRELESS (NYUSIM) [9]. These channel models share many similarities and can be grouped into two main categories: 1) 3GPP/ITU based channel models for frequencies below 6 GHz, with modifications to accommodate up to 100 GHz, and 2) NYUSIM [1] based channel models for frequencies ranging from 0.5 GHz to 100 GHz and provide new features and enhancements, such as spatial consistency, mobility, and spherical wave propagation.

The 3GPP channel model [3] that we chose in this work 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.

COST 2100 [5] is a GBSM for frequency bands below 6 GHz. Cluster power, delays and angles in the COST 2100 model are drawn from fixed geometry locations. This model suffers from limited frequency range and lack of support for scenarios requiring dual mobility, such as device-to-device (D2D) and vehicular-to-vehicular (V2V) communication.

METIS [6] fulfills most of the requirements for fifth-generation (5G) channel modeling, such as blocking, specular reflection, diffraction and spherical wave propagation. It also adds support of spatial consistency with dual mobility. This model is based on ray-tracing and provides high accuracy at the cost of very high computational complexity.

Network simulators are used to model the routing protocol performance, traffic flows and evaluate the efficiency of the communication system using real-life parameters in a virtual environment [18]. Several channel simulators have been developed previously in the literature [9]–[16]. NYUSIM [9], [10] is a geometry-based channel simulator for the physical and link layers of 5G communication systems for frequencies from 0.5 GHz to 100 GHz. In [11] is proposed a geometry-based channel model for millimeter wave (mmWave) frequencies considering the effect of the ground reflection. A three-dimensional (3D) multi-cell channel model is reported [12] for predicting performance of an urban macro-cell setup with enhanced features such as 3D antenna patterns, 3D propagation time evolution, variable terminal speeds, and scenario transitions. A channel simulator for machine-to-machine (M2M) communication in indoor environments is presented in [13]. In [14] is presented a tutorial on an end-to-end simulation system for mmWave module in 5G communication systems. K-Simulator [15] is an open-source standard-compliant modular tool based on 3GPP roadmap for 5G. A stochastic channel model is proposed [16], which adds support for dual mobility and spatial correlation.

Accurate 5G channel model simulations require very high computational effort and incur very long execution time on general purpose processors. Hardware acceleration of such functions is an option to speed up the execution, hence to reduce the simulation time. Hardware accelerators based on Field Programmable Gate Arrays (FPGAs) improve the runtime performance and system energy efficiency of computationally intensive accurate channel simulators with respect to both general-purpose central processing units (CPUs) and Graphics Processing Units (GPUs) [28]. FPGAs can achieve better fine-grained parallelism by customizing the computing engines and memory hierarchy. E.g., an FPGAs implementing a distributed unit receiver can improve the performance even under varying computational load conditions, with optimized power consumption and less area [29]. Civerchia *et al.* [17] studied the optimization of Open Computing Language (OpenCL) designs implementing orthogonal frequency division multiplexing module in the 5G stack on FPGA platforms. Alimohammad *et al.* [19] proposed an implementation on FPGA of infinite impulse response models for Rayleigh fading channels.

Several GBSM emulators have been reported in the literature [20], [21], each with one or more application-specific target scenarios. Hofer *et al.* [20] proposed a parameterized GBSM 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 GBSM for fixed-to-mobile channels is presented in [21]. The channel emulator presented in [22] considers 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 [30] with support for dual mobility. The proposed technique relies on pre-computed ray coefficients, hence it introduces errors in the ray amplitudes. [23] 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 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 GBSM [3] 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 channel model, initially developed for execution on a general-purpose CPU, is adapted for the target Xilinx and Intel FPGA acceleration platforms, followed by the application of different FPGA optimization techniques. Hence, we analyze the effort required to use the different synthesis tools for these platforms. While the main goal of this effort is performance optimization compared to a channel model targeted for general-purpose CPUs, reduction of the energy per computation is also analyzed compared to implementations on CPU and GPU platforms. The results of in this paper indicate that the proposed techniques allow the creation of fast, efficient and accurate communication channel models.

In this article, we investigate the use of efficient high-performance FPGAs for accelerating the channel model in radio link simulators by means of various multi-objective op-

timizations. On the one hand, we focus on improving the code structure as well as the memory architecture of the 5G NR channel model on FPGA platforms to maximize the exposed parallelism and match memory access and computational capabilities by leveraging the analysis and synthesis capabilities of high-level synthesis (HLS) design environments. On the other hand, we analyze the performance of different HLS tools while following fairly similar optimization flows. The accelerated channel model is then integrated within a MATLAB-based simulation system (developed by the Innovation department of TIM S.p.A.) via a socket-based client/server architecture, in order to make it easier to use by several groups of researchers in a shared fashion. To analyze and compare the achievable performance on GPU platforms, the CPU implementation is ported to the Compute Unified Device Architecture (CUDA) framework and optimized for GPU targets. The performance achieved is reported for the NVIDIA GeForce GTX 1070 GPU platform, which is implemented using a similar technology node to the FPGAs that we used. Finally, we analyze the effort required when targeting the FPGA platforms using HLS tools.

The rest of the article is organized as follows. Section II introduces the 5G cellular technology and channel model used at the system and link levels. Section III discusses the flow and technologies used for hardware acceleration and the key benefits associated with them. Section IV explains the different optimization methodologies and techniques being adapted to make efficient use of FPGA-based acceleration platforms. Section V describes the overall channel emulation setup adopted in this work and the way different optimization are applied to the channel model. In Section VI, we discuss and evaluate the experimental results for the FPGA acceleration platforms and present a comparative analysis of the performance achieved for CPU, FPGA and GPU platforms. Section VII concludes the work performed in this research.

II. FIFTH-GENERATION MOBILE NETWORK

5G mobile networks promise important communication features, such as very low latency, very high data rates, and support for high density of devices and base stations. The new cellular network technology is expected to have a substantial impact and aid several sectors, including corporate networks, public networks and infrastructure. Transmission techniques using multi-antenna configurations and multiple-input and multiple-output channels are crucial for enhancing the reliability and spectral efficiency of a radio link. For assessing standardized technologies operating with a base stations equipped with horizontally arranged antennas, 3GPP has used two-dimensional spatial channel model (2D-SCM) on the horizontal cross-section of wireless channels [24]. These models capture poorly the characteristics of a real channel as they consider a two-dimensional (2D) plane and the transmission techniques for multiple-input and multiple-output (MIMO) systems (spatial multiplexing, beamforming and precoding, etc.) are limited to the azimuth dimension. A 3D channel model is required to evaluate communication techniques such as vertical sectorization. A narrow elevation beam is tailored to each vertical sector or user equipment (UE) specific elevation to efficiently adapt both the transmission elevation and azimuth for the UE [24].

A. THREE-DIMENSIONAL CHANNEL MODEL

A system-level simulation with many detailed scenarios, a large number of 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 small scale fading model predicts instead the effect of changes due to the Doppler or multipath effects on a wireless channel.

To model the correlation between the different antenna elements, researchers use spatial channel model. Unlike other traditional models, spatial channel model incorporates a random power delay profile and an angular profile and defines the large-scale parameters and the small-scale parameters of the channel model separately. Although the base stations antenna arrays generate 3D radio beams, it is sometimes modeled in 2D to simplify the calculations by ignoring the elevation angles [25]. 3GPP developed a 3D generic channel model for frequencies ranging from 0.5 GHz to 100 GHz for link-layer and the system-level simulations [3]. The proposed model is a GBSM that extends the ITU/WINNERII 2D channel models. It is also influenced by the WINNERII/WINNER+ expansion from the 2D channel model to the 3D channel model [4] and takes into account the elevation angles and the azimuth angle to model small-scale fading effects and correlation among the antenna elements. Fig. 1a and Fig. 1b show the different angles used in 2D and 3D spatial channel model.

In 3GPP GBSM, a cluster is composed of several rays that originate from same scatterers having similar characteristics such as arrival and departure angles. These clusters consist of multipath components having common propagation direction. Fig. 2 illustrates scattering of different sub-paths in GBSM. Several usage scenarios are defined in the 3GPP specification. For elevation beamforming, the urban micro street canyon and open area, 3D urban macro with outdoor next generation NodeBs, Backhaul, device-to-device, vehicle-to-vehicle, and outdoor to indoor are examples of some common usage scenarios. For each of these propagation scenarios, different parameters are defined to calculate path losses, microscopic and macroscopic fading. large-scale parameters are generated for each UE according to the propagation conditions at its location and geographical position. Delay spread, shadow fading, zenith, angle of arrival (AOA), angle of departure, azimuth angle of arrival (ZOA), and azimuth angle of departure (ZOD) are considered as large-scale parameters, while cluster powers,

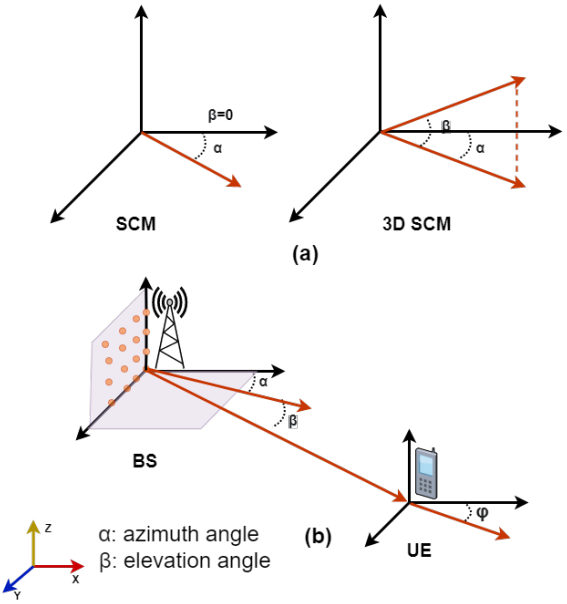

FIGURE 1. Two-dimensional spatial channel model to three-dimensional.

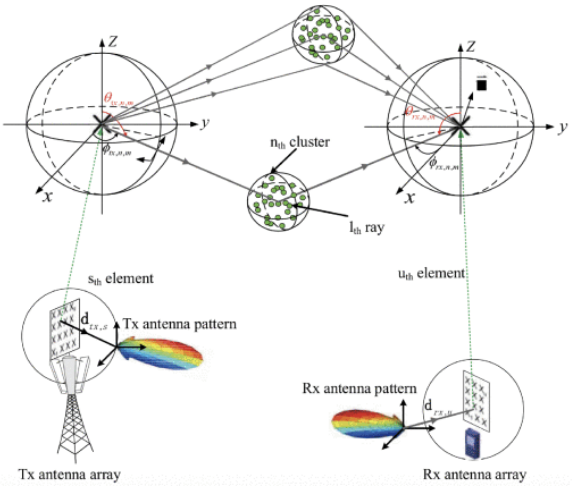

FIGURE 2. Cluster scattering in 3GPP channel model. (source [1]).

TABLE 1. Notations in the global coordinate system

$F_{\mathrm{rx},u,\theta}$	F[1] of receiving antenna element u in the direction of spherical basis vector $\hat{\theta}$
$F_{\mathrm{rx},u,\varphi}$	F of receiving antenna element u in the direction of spherical basis vector $\hat{\varphi}$
$F_{\mathrm{tx},s,\theta}$	F of transmitting antenna element s in the direction of spherical basis vector $\hat{\theta}$
$F_{\mathrm{tx},s,\varphi}$	F of transmitting antenna element s in the direction of spherical basis vector $\hat{\varphi}$
$\theta_{n,m,\mathrm{ZOD}}$	Azimuth angle of departure (ZOD) for ray m in cluster n
$\theta_{n,m,\mathrm{ZOA}}$	Azimuth angle of arrival (ZOA) for ray m in cluster n
$\phi_{n,m,\mathrm{AOD}}$	Angle of departure (AOD) for ray m in cluster n
$\phi_{n,m,\mathrm{AOA}}$	Angle of arrival (AOA) for ray m in cluster n
$\theta_{n,\mathrm{ZOD}}$	Azimuth angle of departure (ZOD) for cluster n
$\theta_{n,\mathrm{ZOA}}$	Azimuth angle of arrival (ZOA) for cluster n
$\phi_{n,\mathrm{AOD}}$	Angle of departure (AOD) for cluster n
$\phi_{n,\mathrm{AOA}}$	Angle of arrival (AOA) for cluster n
$\theta_{\mathrm{LOS,ZOD}}$	Line-of-sight (LOS) azimuth angle of departure (ZOD)
$\theta_{\mathrm{LOS,ZOA}}$	Line-of-sight (LOS) azimuth angle of arrival (ZOA)
$\phi_{\mathrm{LOS,AOD}}$	Line-of-sight (LOS) angle of departure (AOD)
$\phi_{\mathrm{LOS,AOA}}$	Line-of-sight (LOS) 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 (SUV) of rx element
$\hat{r}_{\mathrm{tx},n,m}$	Spherical unit vector (SUV) of tx element
$\bar{d}_{\mathrm{rx},u}$	Location vector (LV) of rx antenna element u
$\bar{d}_{\mathrm{tx},s}$	Location vector (LV) of tx antenna element s
λ_0	Wavelength of carrier frequency
$\bar{v}$	Velocity vector of user equipment (UE)
P_n	nth path power

[1]Field pattern (F)

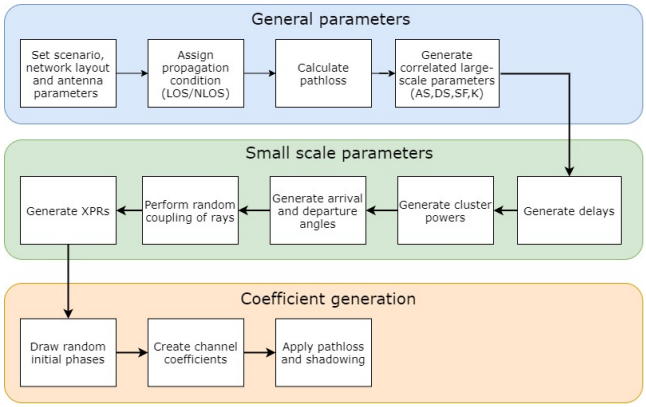

FIGURE 3. Channel coefficient generation in 3GPP 3D channel model.

delays, ZOA, ZOD, and elevation direction are considered as small-scale parameters, which change frequently. The parameters listed in Table 1 are used for the realization of the radio channel using the step-by-step method shown in Fig. 3. The first step of the channel modeling is the identification of the application environment. A simulation scenario is first chosen, then the corresponding network layout (number of base stations and UE), and the antenna parameters are specified. The final step for setting large-scale parameters is the calculation of path losses for the assigned propagation conditions.

B. FAST FADING CHANNEL MODEL

Fast fading coefficients model the fluctuating behaviour of the wireless channel due to changes in the UE movement or due to multipath [26]. The fast fading channel model calculates large-scale parameters to generate the channel coefficients. A down-link connection is assumed here for the notations, hence the arrival angles are defined at the UE side and the departure angles at the next generation NodeB side. For the up-link, the departure and the arrival parameters have to be swapped to obtain the respective realizations. At this stage, the azimuth spread angle of departure and azimuth spread angle of arrival and the AOA and angle of departure are generated, in addition to the zenith spread angle of departure and zenith spread angle of arrival, and the ZOA and ZOD. Random coupling among the arrival and departure angles is performed for different multiple path components of the composite channel. Finally, taking into account these parameters, the channel coefficients are generated. Considering N cluster scatterers with M resolvable paths each, the channel impulse response for ray m in cluster n,

UE antenna element u, and base stations antenna element s is

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

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 $\varphi_{n,m,\mathrm{AOA}}$. For cluster n and ray m within cluster n, the spherical unit vector is given by

$$\hat{r}_{\mathrm{rx},n,m} = \begin{bmatrix} \sin\theta_{n,m,\mathrm{ZOA}} \cos\phi_{n,m,\mathrm{AOA}} \\ \sin\theta_{n,m,\mathrm{ZOA}} \sin\phi_{n,m,\mathrm{AOA}} \\ \cos\theta_{n,m,\mathrm{ZOA}} \end{bmatrix} \quad (2)$$

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

$$\hat{r}_{\mathrm{tx},n,m} = \begin{bmatrix} \sin\theta_{n,m,\mathrm{ZOD}} \cos\phi_{n,m,\mathrm{AOD}} \\ \sin\theta_{n,m,\mathrm{ZOD}} \sin\phi_{n,m,\mathrm{AOD}} \\ \cos\theta_{n,m,\mathrm{ZOD}} \end{bmatrix} \quad (3)$$

The Doppler frequency component depends on the UE speed v with velocity vector $\bar{v}$, AOA, ZOA, travel elevation angle θ_v and azimuth angle ϕ_v

$$v_{n,m} = \frac{\hat{r}_{\mathrm{rx},n,m}^T \bar{v}}{\lambda_0} \quad (4)$$

where

$$\bar{v} = v.[\sin\theta_v \cos\phi_v \quad \sin\theta_v \sin\phi_v \quad \cos\theta_v]^T \quad (5)$$

In the conventional approach of beamforming, the array factor is applied to the field pattern of a single antenna element in a uniform array. In the 3D GBSM model, the array factor is applied to the coefficients of each channel, for each antenna element. Considering the delays and ray mappings listed in [3, Table 7.5-5], the final channel impulse response $H_{u,s}(\tau, t)$ are calculated by combining the partial coefficients for each transmitting and receiving antenna element in each cluster and scatter [27].

$$H_{u,s}(\tau,t) = \sum_{n=1}^{2} \sum_{i=1}^{3} \sum_{m \in R_i} H_{u,s,n,m}(t)\delta(\tau - \tau_{n,i}) + \sum_{m=3} H_{u,s,n}^{N}(t)\delta(\tau - \tau_n) \quad (6)$$

The channel can be either represented as a tapped delay line or a cluster delay line. For simplified evaluation, the tapped delay line model is defined as an impulse response, in which a radio channel is characterized by several delay taps while the cluster delay line model is characterized by the arrival and departure directions in the 3D space which allows better beamforming representation. The tapped delay line model defines the correlation between the antenna elements through a static correlation matrix, whereas the cluster delay line model depends on the geometry of the antenna elements and how the channel propagates. To obtain a tapped delay line model, a brick wall window is applied to the delay-scaled cluster delay line model followed by power normalization. The pseudo-code in Algorithm 1 shows the procedure for the channel coefficient generation. The channel coefficients in the GBSM are dependent on the location of the UE in the 3D space, hence they must be calculated dynamically. For *nTx* transmitting antennas, *nRx* receiving antennas, *nClust* number of clusters, *NSPS* oversampling factor, sampling frequency f and transmission time interval length *TTI* the number of partial coefficients calculated is

$$nCoeff = nRx \times nTx \times nClust \times NSPS \times f \times TTI \times 1000 \quad (7)$$

Thus, considering simulation parameters $nTx = 32$, $nRx = 2$, $nClust = 23$, $NSPS = 4$, $f = 122.88$ MHz and $TTI = 0.25$ ms, a total of 180 879 360 partial coefficients are generated.

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

Input: Input symbols
Output: channel impulse response

```
for u = 0 to nRxAntenna do
    for s = 0 to nTxAntenna do
        for n = 0 to nCluster do
            for m = 0 to nCDL do
                calculate for r̂_rx,n,m as in (2)
                calculate for r̂_tx,n,m as in (3)
            end for
        end for
        for l = 0 to nSymbol do
            for n = 0 to nCluster do
                H_u,s,n,m(t) as in (6)
            end for
        end for
    end for
end for
```

III. FPGA ACCELERATION USING OpenCL AND HLS

Several HLS tools have been introduced for rapid prototyping and hardware development using large FPGAs. These tools take as input programs written in C, C++, OpenCL [31]–[36], and other high-level languages [37]–[41] alongside some design constraints and pragmas, and translate them into lower level description such as register transfer level or hardware description language with equivalent functionality. This translation from high-level description into hardware description language is done by HLS toolchains. 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. 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 static RAM (SRAM) contents. OpenCL is a parallel programming language for multi-core and heterogeneous computing platforms [42]. OpenCL is developed as an open standard by the Khronos group, thus it has an edge over a similar framework, the Compute Unified Device Architecture, 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 [44]. An OpenCL application is comprised of one or more device or kernel functions, and host code. Device code is the part of the code which is highly data parallel and computationally intensive, and will be executed on the accelerator. Host code is the part which 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 and each may contain one ore more processing elements, depending on the platform and the designer implementation choices. OpenCL splits the computations in parallel threads called work-items which are then combined together in work-groups. This approach adds support for data-parallel computations and thus some "doall" loop iterations without inter-iteration dependencies (in particular those over work-groups), 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. Fig. 4 shows the main elements of an OpenCL design. It relies on a single instruction multiple data paradigm similar to GPUs to better exploit the hardware platforms. It enables the developers to generate efficient code that fits the architecture of target device by providing an abstract but non-uniform memory hierarchy.

OpenCL divides memory into different spaces namely global, local, private, and constant memory. Global and constant memories are shared among all the compute units 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 work-items in a work-group, has lower latency than global memory and is often mapped to on-chip SRAM. Each work-item finally has its private memory space, which is mapped to the register file and has the lowest latency.

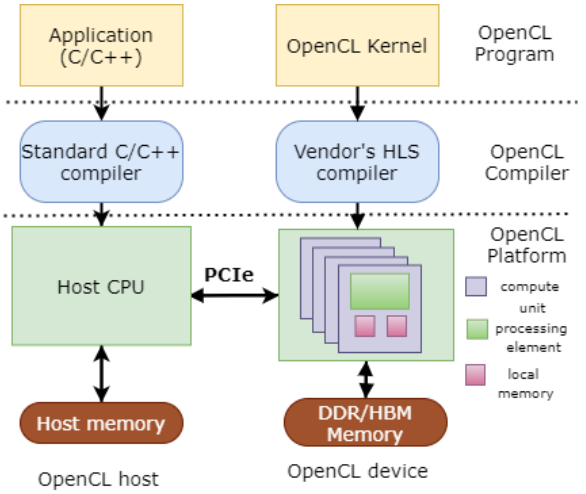

FIGURE 4. Open Computing Language (OpenCL) based hardware acceleration flow.

IV. 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 (6) 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, (6) offers a very high level of parallelism that can be exploited to significantly speed up the computation using a GPU or FPGA.

A. 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.

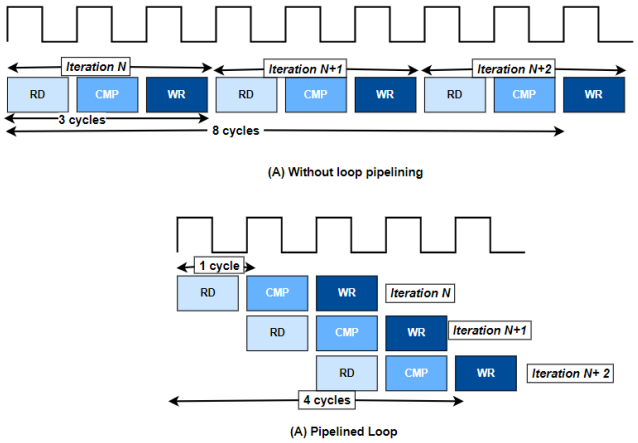

FIGURE 5. Loop pipelining.

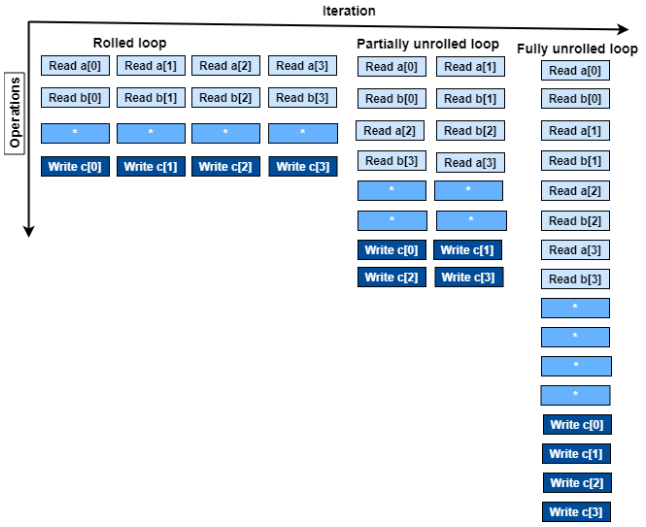

FIGURE 6. Loop unrolling/vectorization.

1) 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 an 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 initiation interval II and latency L that executes N iterations, the total execution time T when operating at frequency f can be described as in [45]

$$T = \frac{L + II \cdot (N-1)}{f}. \tag{8}$$

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. Fig. 5 shows execution of code in Listing 1 in sequential and pipelined manner.

Listing 1. Loop pipelning example

```
void func(m,n,0){
    for(i=0;i<=2;i++){
        Read_op;
        Compute_op;
        Write_op;
    }
}
```

Loop pipelining can be specified in OpenCL kernels with `__attribute__((xcl_pipeline_loop(N)))`, for `C/C++` kernels in Xilinx Vitis [46] with `#pragma HLS pipeline II=<N>`, or for Intel FPGA SDK for OpenCL [47] 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.

2) Loop unrolling

If there are no data dependencies among the iterations of the loop, the loop execution performance can be improved by executing multiple iteration 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 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. Fig. 6 shows the execution of code in Listing 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.

Listing 2. Loop unrolling example

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

3) Loop tiling

FPGAs have limited on-chip storage resources, which are often insufficient to store all the inputs and intermediate 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 [48]. For a better understanding of the tiling based optimization, Listing 3 shows an example of nested loops with a large tripcount and hence larger memory footprint.

Listing 3. Nested loops example

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

Loop tiling is applied as shown in Listing 4, resulting in a smaller memory footprint. This optimization can be used to add support for larger designs on platforms with limited memory resources.

Listing 4. Tiled loops example

```
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++){
                    ....
            }
        }
    }
}
```

4) 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 5 shows the coalesced structure of nested loops in Listing 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.

Listing 5. Coalesced loops

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

B. MEMORY 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 block RAM on FPGAs). Hence, to compute the channel impulse response 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 into on-chip buffers, which have low latency and thus reduce total access time [49].

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 processing element.

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 channels. Hence, 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 block RAM 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 Double Data Rate 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 Double Data Rate and HBM bandwidths.

V. EXPERIMENTAL SETUP

We used the setup shown in Fig. 7 for this experiment to enable access to remote accelerators for multiple MATLAB clients. The design environments and tools used are:

Baseline CPU Platform. Uses an Intel Xeon E5-2660 v4 @2.00GHz running MATLAB R2021a [50]. The setup consists of a co-simulation environment where the channel model is implemented in C++ and executed in MEX [51], 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 [52] 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 block RAM, 8 GB of HBM and 32 GB of Double Data Rate 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 development of the host and kernel code. We will refer to this platform as US+ hereafter.

FPGA Platform 2. Uses the Intel programmable acceleration card Arria 10GX 1150 [53]. This platform contains 8.2 MB on-chip embedded memory and 8 GB of Double Data Rate 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 [47], [54]. We will refer to this platform as Arria hereafter.

GPU Platform. Uses an NVIDIA GeForce GTX 1070 GPU which has 15 Streaming Multiprocessors, 1920 CUDA Cores, 1.4 MB of on-chip shared memory, core graphic clock of 1506 MHz, 8 GB of GDDR5 memory, and is implemented in a technology node similar to the FPGAs. The GPU platform is not a main focus of this work and is only used as another reference point to compare the results achieved via the FPGA acceleration of the channel model.

TABLE 2. Field Programmable Gate Array (FPGA) acceleration platforms and their main resources

	US+		Arria
Resource type	**Total**	**Per SLR**	
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	—

Both the FPGA Platform 1 and the GPU Platform used for this research are implemented in 16 nm whereas the FPGA Platform 2 is implemented in 20 nm.

Table 2 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.

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++ or OpenCL code parallelized for GPU. 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 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 work-groups and work-items. However, in this project we did not exploit this opportunity because we wanted to exercise finer control over the loop pipelining and unrolling, which is possible only with a single-work-item modeling style.

Fig. 7 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 enables to serve multiple remote clients avoiding to have 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

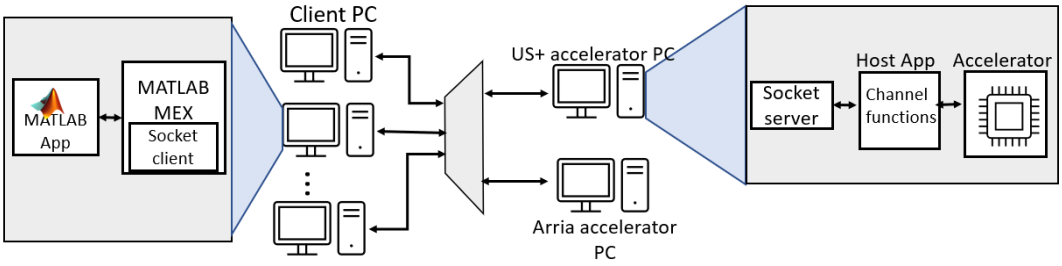

FIGURE 7. Emulation system setup

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 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 Section IV 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.

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.

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 finite impulse response coefficients are the result of intermediate computations (6). These partial results are kept in SRAM buffers with low latency, thus enabling to compute coefficients for all clusters in single instruction multiple data fashion, hence reducing the total latency.

Application-specific optimizations (ASO). This type 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, we replaced the iterative computation of an arithmetic sequence 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 accesses 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 reducing memory contention and bank conflicts. URAM is a special kind of memory that is wider and deeper than the block RAM and can be used to store large data structures. Since in OpenCL currently it is not possible to control these features, we ported the kernel to C++, which allows more control over these optimizations, while losing some of the 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.

VI. RESULTS AND ANALYSIS

We analyze here the performance of the accelerators before and after optimizations for the two target platforms. We used the reference parameter values from the 3GPP specification [3] for this phase. Table 3 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 to simplify the addressing logic.

TABLE 3. 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	$N_{TX} = 32$
Number of RX antennas	$N_{RX} = 2$
Oversampling factor	4
Number of clusters	23
Number of rays	20
User speed	120 km/h
Number of symbols per link	122 880
Transmission time interval	0.25 ms

TABLE 4. Kernel latency and speed up achieved compared to out-of-the-box (OOB) and general-purpose central processing unit (CPU) implementation, with application-specific optimizations (ASO), platform-specific optimizations (PSO), High-Bandwidth Memory (HBM), and UltraRAM (URAM)

Platform	Optimization	Latency (s)	Speedup **OOB** (X)	Speedup **CPU** (X)
CPU	Baseline	5.010	32.00	1.00
GPU	ASO	0.170	944.00	29.00
US+	OOB	160.400	1.00	0.03
	On-chip buffers	4.110	39.00	1.02
	Multi-port	23.860	7.00	0.21
	ASO	0.080	2083.00	65.00
	PSO (HBM)	0.033	4860.00	151.00
	PSO (HBM+URAM)	0.027	5531.00	173.00
Arria	OOB	655.000	1.00	0.01
	On-chip buffers	28.000	23.00	0.18
	Multi-port	6.390	103.00	0.78
	ASO	0.053	12 359.00	95.00

A. 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 4 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, we also report here 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

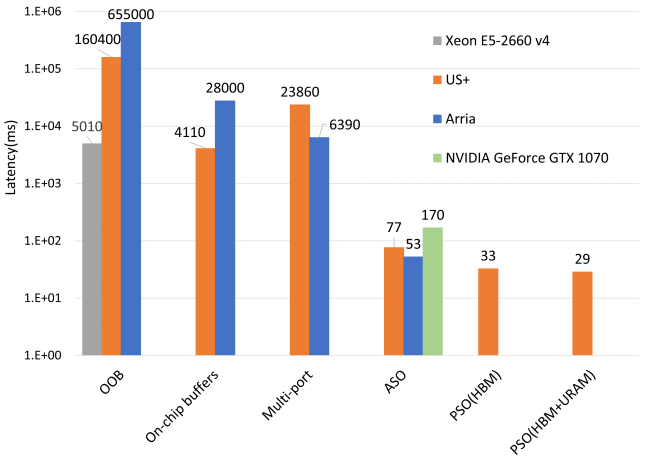

FIGURE 8. Average latency (log scale)

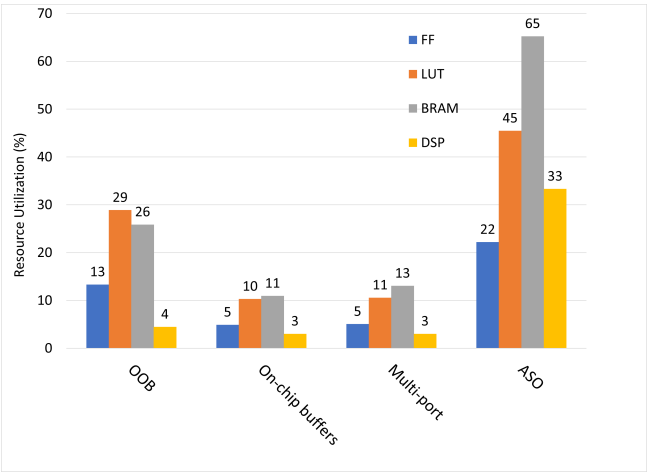

FIGURE 9. Resource utilization on Arria

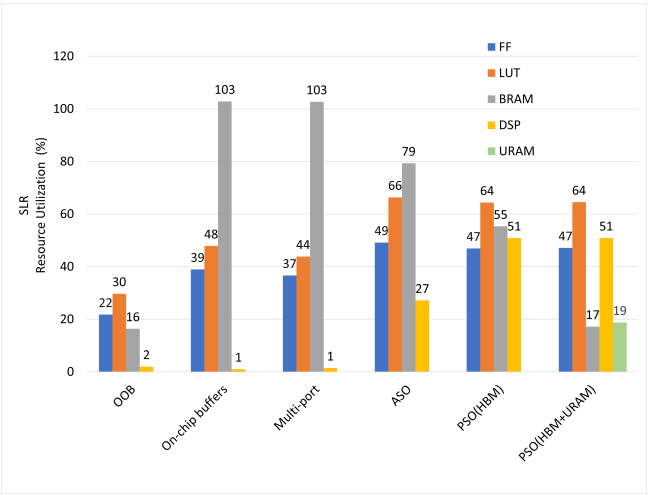

FIGURE 10. Single SLR resource utilization on US+

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 separate 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 utilization, however, is still poor due to the unaligned access patterns, which require a significant amount of multiplexing. To overcome this, ASO are applied (see Section V), yielding overall **95X** and **65X** speedups on the US+ and Arria platforms respectively compared to the baseline CPU implementation. The FPGA-based designs also achieved **~3X** better performance compared to that achieved on GPU platform. 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 (see Section V). 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. This leads to a speedup of **151X**. Performance is further improved by using URAM resources for some of the data structures, to better balance the resource utilization of the design and yield an overall **173X** and **6X** speedup compared to the baseline CPU and GPU based implementation respectively. Fig. 8 shows the latency of the design after various optimizations. As this study follows a step-by-step procedure, the new optimizations are added on the top of the ones applied in earlier steps.

B. 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 lists the resource usage for the various designs. The total available on-chip resources are listed in Table 2. 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 Fig. 9. 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. Fig. 10 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.

TABLE 5. Resource Utilization

		Utilized resources					SLR percent Utilization				
Platform	**Optimization**	**FF**	**LUT**	**BRAM**	**URAM**	**DSP**	**FF(%)**	**LUT(%)**	**BRAM(%)**	**URAM**	**DSP(%)**
US+	OOB	188 901	129 038	220	0	59	22	30	16	0	2
	On-chip buffers	338 349	207 833	1382	0	30	39	48	103	0	1
	Multi-port	318 186	190 510	1380	0	43	37	44	103	0	1
	ASO	427 206	288 237	1066	0	816	49	66	79	0	27
	PSO(HBM)	407 674	279 657	743	0	1530	47	64	55	0	51
	PSO(HBM+URAM)	409 292	280 272	231	60	1530	47	64	17	19	51
Arria	OOB	227 896	123 420	702	—	68	13	29	26	—	4
	On-chip buffers	83 891	44 059	297	—	46	5	10	11	—	3
	Multi-port	86 414	45 215	354	—	46	5	11	13	—	3
	ASO	379 312	194 295	1770	—	506	22	45	65	—	33

TABLE 6. Power and energy utilization of baseline and accelerated designs

Platform	**Optimization**	**Power (W)**			**Energy**
		Dynamic	**Static**	**Total**	**(J)**
CPU				105.00	526.05
GPU	ASO			55.6	9.45
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**

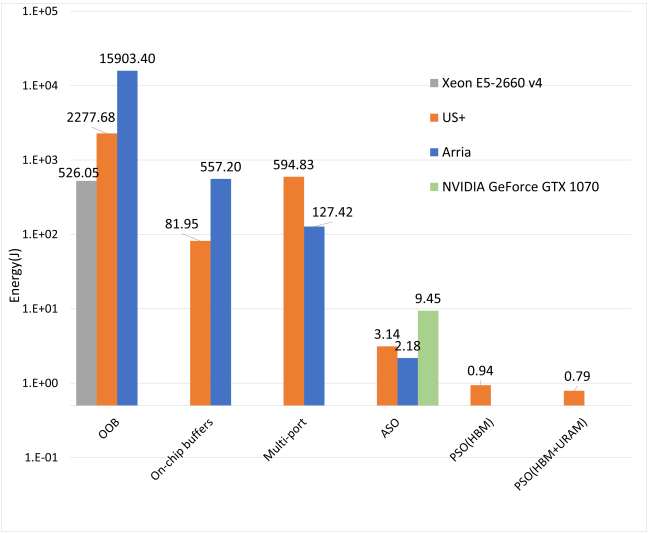

FIGURE 11. Energy utilization on target platforms (log scale)

C. POWER AND ENERGY

Improving the 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 6 reports the energy consumption of the design at various optimization levels. The CPU implementation has the highest energy consumption due its higher thermal design profile. 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 the energy consumption. Fig. 11 shows the energy consumption of implemented design at different optimization stages. The GPU platform implementation consumes more energy since the GPU that we used has less on-chip memory than the FPGAs.

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

VII. CONCLUSION

The simulation of the 3GPP three-dimensional spatial channel model channel model can be significantly accelerated using FPGA platforms from different vendors (we report for Xilinx and Intel) 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 to design 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 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 three-dimensional spatial channel model 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.

To compare our results with those achievable on another highly parallel acceleration platform, namely GPUs, the channel model was re-implemented and optimized using the Compute Unified Device Architecture (CUDA) framework and language for an NVIDIA GPU implemented on a comparable technology node to our FPGAs. One of the FPGA platforms exhibits both better performance, between 3X and 6X, and energy consumption, between 4X and 12X, than the GPU. This is thanks to the better memory access achievable for the memory-intensive channel model on the FPGAs, since they have a larger on-chip memory but comparably less computational resources than the GPU.

Over the timeline of this research work, 20% of the time was spent porting the CPU-based application to FPGA platforms, to make it synthesizable. At this stage, reports from HLS tools helped in analyzing bottlenecks of the design. 10% of the development time was spent for each one of the on-chip and multi-port memory optimizations. The application-specific optimizations required in-depth analysis of access pattern, computational flow and memory dependence, and took around 30% of the total development time. Platform-specific optimizations for HBM took about 20% of the total time and included memory access analysis and partitioning into separate HBM channels. Finally, exploiting URAM resources to balance the block RAM utilization required the final 10% of the development time.

The HLS tool for one of the FPGAs provides a user friendly graphical interface for rapid development and debugging, while there is no such Integrated Development Environment for the other 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 sub-optimal, due to the lack of an efficient on-chip memory architecture that is comparable to the cache in a CPU.

The HLS tool for one of the FPGAs 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. Lastly, the C++ based kernels for one of the 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 register transfer level implementations, it enables rapid design space exploration and thus ultimately can achieve respectable quality of results with a reasonable design optimization time.

ACKNOWLEDGMENT

The authors would like to thank the Innovation Department of TIM S.p.A. for providing the software and hardware resources and technical expertise to support this research.

REFERENCES

[1] S. Sun, T. S. Rappaport, M. Shafi, P. Tang, J. Zhang, and P. J. Smith, "Propagation models and performance evaluation for 5G millimeter-wave bands," IEEE Transactions on Vehicular Technology, vol. 67, no. 9, pp. 8422–8439, 2018.

[2] J. Yang, B. Ai, K. Guan, D. He, X. Lin, B. Hui, J. Kim, and A. Hrovat, "A geometry-based stochastic channel model for the millimeter-wave band in 3GPP high-speed train scenario," IEEE Transactions on Vehicular Technology, vol. 67, no. 5, pp. 3853–3865, 2018.

[3] G. R. A. N. W. Group et al., "Study on channel model for frequencies from 0.5 to 100 GHz (release 15)," 3GPP TR 38.901, Tech. Rep., 2018.

[4] Y. d. J. Bultitude and T. Rautiainen, "Ist-4-027756 WINNER II D1. 1.2 V1. 2 WINNER II channel models," EBITG, TUI, UOULU, CU/CRC, NOKIA, Tech. Rep, 2007.

[5] L. Liu, C. Oestges, J. Poutanen, K. Haneda, P. Vainikainen, F. Quitin, F. Tufvesson, and P. De Doncker, "The COST 2100 MIMO channel model," IEEE Wireless Communications, vol. 19, no. 6, pp. 92–99, 2012.

[6] V. Nurmela, A. Karttunen, A. Roivainen, L. Raschkowski, V. Hovinen, J. Y. EB, N. Omaki, K. Kusume, A. Hekkala, R. Weiler et al., "Deliverable d1. 4 metis channel models," Proc. Mobile Wireless Commun. Enablers Inf. Soc.(METIS), p. 1, 2015.

[7] M. Series, "Guidelines for evaluation of radio interface technologies for imt-2020," Report ITU, pp. 2412–0, 2017.

[8] R. J. Weiler, M. Peter, W. Keusgen, A. Maltsev, I. Karls, A. Pudeyev, I. Bolotin, I. Siaud, and A.-M. Ulmer-Moll, "Quasi-deterministic millimeter-wave channel models in MiWEBA," EURASIP Journal on Wireless Communications and Networking, vol. 2016, no. 1, pp. 1–16, 2016.

[9] S. Sun, G. R. MacCartney, and T. S. Rappaport, "A novel millimeter-wave channel simulator and applications for 5G wireless communications," in 2017 IEEE International Conference on Communications (ICC), 2017, pp. 1–7.

[10] S. Ju, O. Kanhere, Y. Xing, and T. S. Rappaport, "A millimeter-wave channel simulator nyusim with spatial consistency and human blockage," in 2019 IEEE Global Communications Conference (GLOBECOM), 2019, pp. 1–6.

[11] S. Jaeckel, L. Raschkowski, S. Wu, L. Thiele, and W. Keusgen, "An explicit ground reflection model for mm-wave channels," in 2017 IEEE Wireless Communications and Networking Conference Workshops (WCNCW), 2017, pp. 1–5.

[12] S. Jaeckel, L. Raschkowski, K. Börner, and L. Thiele, "Quadriga: A 3-D multi-cell channel model with time evolution for enabling virtual field trials," IEEE Transactions on Antennas and Propagation, vol. 62, no. 6, pp. 3242–3256, 2014.

[13] Y. Yu, Y. Liu, W.-J. Lu, and H.-B. Zhu, "Propagation model and channel simulator under indoor stair environment for machine-to-machine applications," in 2015 Asia-Pacific Microwave Conference (APMC), vol. 2, 2015, pp. 1–3.

[14] M. Mezzavilla, M. Zhang, M. Polese, R. Ford, S. Dutta, S. Rangan, and M. Zorzi, "End-to-end simulation of 5G mmwave networks," IEEE Communications Surveys Tutorials, vol. 20, no. 3, pp. 2237–2263, 2018.

[15] J. Baek, J. Bae, Y. Kim, J. Lim, E. Park, J. Lee, G. Lee, S. I. Han, C. Chu, and Y. Han, "5G K-simulator of flexible, open, modular (fom) structure and web-based 5G K-simplatform," in 2019 16th IEEE Annual Consumer Communications Networking Conference (CCNC), 2019, pp. 1–4.

[16] A. M. Pessoa, I. M. Guerreiro, C. F. M. E. Silva, T. F. Maciel, D. A. Sousa, D. C. Moreira, and F. R. P. Cavalcanti, "A stochastic channel model with dual mobility for 5G massive networks," IEEE Access, vol. 7, pp. 149 971–149 987, 2019.

[17] Civerchia, F., Pelcat, M., Maggiani, L., Kondepu, K., Castoldi, P. & Valcarenghi, L. "Is OpenCL Driven Reconfigurable Hardware Suitable for Virtualising 5G Infrastructure?," IEEE Transactions On Network And Service Management. **17**, 849-863 (2020)

[18] S. Bai and D. M. Nicol, "Acceleration of wireless channel simulation using GPUs," in 2010 European Wireless Conference (EW), 2010, pp. 841–848.

[19] A. Alimohammad, S. F. Fard, B. F. Cockburn, and C. Schlegel, "A compact single-FPGA fading-channel simulator," IEEE Transactions on Circuits and Systems II: Express Briefs, vol. 55, no. 1, pp. 84–88, 2008.

[20] M. Hofer, Z. Xu, D. Vlastaras, B. Schrenk, D. Löschenbrand, F. Tufvesson, and T. Zemen, "Real-time geometry-based wireless channel emulation," IEEE Transactions on Vehicular Technology, vol. 68, no. 2, pp. 1631–1645, 2019.

[21] J. Hua, J. Yang, W. Lu, L. Meng, and X. Yu, "Design of universal wireless channel generator accounting for the 3D scatter distribution and hardware output," IEEE Transactions on Instrumentation and Measurement, vol. 64, no. 1, pp. 2–13, 2015.

[22] P. Huang, M. J. Tonnemacher, Y. Du, D. Rajan, and J. Camp, "Towards massive MIMO channel emulation: Channel accuracy versus implementation resources," IEEE Transactions on Vehicular Technology, vol. 69, no. 5, pp. 4635–4651, 2020.

[23] Endovitskiy, E., Kureev, A. & Khorov, E. "Reducing Computational Complexity for the 3GPP TR 38.901 MIMO Channel Model," IEEE Wireless Communications Letters. **11**, 1133-1136 (2022)

[24] Y.-H. Nam, B. L. Ng, K. Sayana, Y. Li, J. Zhang, Y. Kim, and J. Lee, "Full-dimension MIMO (FD-MIMO) for next generation cellular technology," IEEE Communications Magazine, vol. 51, no. 6, pp. 172–179, 2013.

[25] Q.-U.-A. Nadeem, A. Kammoun, M. Debbah, and M.-S. Alouini, "Performance analysis of compact FD-MIMO antenna arrays in a correlated environment," IEEE Access, vol. 5, pp. 4163–4178, 2017.

[26] B. Mondal, T. A. Thomas, E. Visotsky, F. W. Vook, A. Ghosh, Y.-H. Nam, Y. Li, J. Zhang, M. Zhang, Q. Luo et al., "3D channel model in 3GPP," IEEE Communications Magazine, vol. 53, no. 3, pp. 16–23, 2015.

[27] F. Ademaj, M. Taranetz, and M. Rupp, "3GPP 3D MIMO channel model: A holistic implementation guideline for open source simulation tools," EURASIP Journal on Wireless Communications and Networking, vol. 2016, no. 1, pp. 1–14, 2016.

[28] Kundel, R., Eryigit, K., Markussen, J., Griwodz, C., Abboud, O., Hark, R. & Steinmetz, R. "Host Bypassing: Direct Data Piping from the Network to the Hardware Accelerator," 2021 IEEE 14th International Symposium On Embedded Multicore/Many-core Systems-on-Chip (MCSoC). pp. 23-30 (2021)

[29] Coutinho, F., Silva, H. & Oliveira, A. "FPGA-based Design and Optimization of a 5G-NR DU Receiver," 2021 Telecoms Conference (ConfTELE). pp. 1-6 (2021)

[30] Yang, Y., Tingpeng, L., Xiaomin, C., Manxi, W., Qiuming, Z., Ruirui, F., Fuqiao, D. & ZHANG, T. "Real-time ray-based channel generation and emulation for UAV communications," Chinese Journal Of Aeronautics. (2021)

[31] M. Sussmann and T. Hill, "Intel HLS Compiler: Fast Design, Coding, and Hardware," 2017, white paper.

[32] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, "High-level synthesis for FPGAs: From prototyping to deployment," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 30, no. 4, pp. 473–491, 2011.

[33] Z. Zhang, Y. Fan, W. Jiang, G. Han, C. Yang, and J. Cong, "AutoPilot: A platform-based ESL synthesis system," in High-Level Synthesis. Springer, 2008, pp. 99–112.

[34] C. Pilato and F. 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. IEEE, 2013, pp. 1–4.

[35] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. 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, 2011, pp. 33–36.

[36] "C++/SystemC Synthesis | Siemens Digital Industries Software," https://eda.sw.siemens.com/en-US/ic/catapult-high-level-synthesis/hls/c-cplus/, (Accessed on 09/29/2021).

[37] R. Nane, V.-M. Sima, B. Olivier, R. Meeuws, Y. Yankova, and K. Bertels, "DWARV 2.0: A CoSy-based C-to-VHDL hardware compiler," in 22nd International Conference on Field Programmable Logic and Applications (FPL). IEEE, 2012, pp. 619–622.

[38] T. S. Czajkowski, U. Aydonat, D. Denisenko, J. Freeman, M. Kinsner, D. Neto, J. Wong, P. Yiannacouras, and D. P. Singh, "From OpenCL to high-performance hardware on FPGAs," in 22nd international conference on field programmable logic and applications (FPL). IEEE, 2012, pp. 531–534.

[39] M. Gokhale, J. Stone, J. Arnold, and M. 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). IEEE, 2000, pp. 49–56.

[40] R. 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. IEEE, 2004, pp. 69–70.

[41] J. Hammarberg and S. Nadjm-Tehrani, "Development of safety-critical reconfigurable hardware with Esterel," Electronic Notes in Theoretical Computer Science, vol. 80, pp. 219–234, 2003.

[42] L. Struyf, S. De Beugher, D. H. Van Uytsel, F. Kanters, and T. Goedemé, "The battle of the giants-a case study of GPU vs FPGA optimisation for real-time image processing," in International Conference on Pervasive and Embedded Computing and Communication Systems, vol. 2. SCITEPRESS, 2014, pp. 112–119.

[43] Xiao, K. & Zhang, W. Systematic Study on Hardware Optimization of 5G Communication. 2021 5th International Conference On Computing Methodologies And Communication (ICCMC). pp. 92-96 (2021)

[44] F. B. Muslim, L. Ma, M. Roozmeh, and L. Lavagno, "Efficient FPGA implementation of OpenCL high-performance computing applications via high-level synthesis," IEEE Access, vol. 5, pp. 2747–2762, 2017.

[45] J. de Fine Licht, M. Besta, S. Meierhans, and T. Hoefler, "Transformations of high-level synthesis codes for high-performance computing," IEEE Transactions on Parallel and Distributed Systems, vol. 32, no. 5, pp. 1014–1029, 2021.

[46] "Vitis Platform," https://www.xilinx.com/products/design-tools/vitis/vitis-platform.html, (Accessed on 10/22/2021).

[47] "Intel FPGA SDK for OpenCL pro edition: Version 19.4 release notes," https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/rn/archives/rn_aocl-19-4.pdf, (Accessed on 10/25/2021).

[48] Y. Ma, Y. Cao, S. Vrudhula, and J.-s. 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, 2017, pp. 45–54.

[49] J. Wang, J. Lin, and Z. Wang, "Efficient hardware architectures for deep convolutional neural network," IEEE Transactions on Circuits and Systems I: Regular Papers, vol. 65, no. 6, pp. 1941–1953, 2018.

[50] MATLAB, version 9.10.0 (R2021a). Natick, Massachusetts: The MathWorks Inc., 2021.

[51] "Build MEX function or engine application - MATLAB mex - MathWorks Italia," https://it.mathworks.com/help/matlab/ref/mex.html, (Accessed on 10/17/2021).

[52] "Alveo U280 data center accelerator card," https://www.xilinx.com/products/boards-and-kits/alveo/u280.html, (Accessed on 10/25/2021).

[53] "Intel FPGAs - Intel Arria 10 FPGAs," https://www.intel.com/content/www/us/en/products/details/fpga/arria/10.html, (Accessed on 10/23/2021).

[54] "Intel® Quartus® Prime pro edition version 19.2 software and device support release notes," https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/rn/archives/rn-qts-pro-dev-support-19.2.pdf, (Accessed on 10/25/2021).

NASIR ALI SHAH has received M.S. degree in computer engineering with specialization in embedded from Politecnico di Torino, Italy. Currently he is pursuing his Ph.D. degree with the Department of Electronic and Telecommunications Engineering under the supervision of Prof. Luciano Lavagno and Prof. Mihai Teodor Lazarescu.

LUCIANO LAVAGNO received the Ph.D. degree in EECS from the University of California at Berkeley, Berkeley, CA, USA, in 1992. He was the architect of the POLIS HW/SW co-design tool. From 2003 to 2014, he was an Architect of the Cadence CtoSilicon high-level synthesis tool. Since 1993, he has been a Professor with the Politecnico di Torino, Italy. He co-authored four books and more than 200 scientific papers. His research interests include synthesis of asynchronous circuits, HW/SW co-design, high-level synthesis, and design tools for wireless sensor networks.

MIHAI T. LAZARESCU (CM'92) received the Ph.D. degree in Electronics and Communications from Politecnico di Torino, Italy, in 1998, where he serves now as Associate Professor. He was Senior Engineer at Cadence Design Systems and founded several startups. He co-authored more than 60 scientific publications, 4 books, and several international patents. His research interests include design tools for reusable WSN/IoT platforms, low-power sensing, neural networks, embedded design, indoor human localization, distributed IoT data processing, high-level HW/SW co-design and synthesis.

ROBERTO QUASSO received degree in Electronics Engineering from the Politecnico di Torino, Italy, in 1992. He serves in TIM as System Architect in Innovation Department, developing applications and solutions for the evolution of mobile radio access network. Currently he works at application of hardware acceleration to automated measurement systems for Active Antenna Systems and to fast simulation of mobile radio links.

SALVATORE SCARPINA received degree in Electronics Engineering from the Politecnico di Torino, Italy, in 2005. His field of applications includes project management, virtualization and cloud architecture, hardware acceleration and machine learning. He served in TIM as Network Engineer in Innovation Department; currently he serves in IVECO S.p.A. as Digital Project Manager for Cloud Applications.