POLITECNICO DI TORINO

Master Degree Course in Electronic Engineering

Master Thesis

Hardware Acceleration of 5G LDPC using datacenter-class FPGAs

Supervisor:
Prof. Luciano Lavagno

Candidate:
Luca Romani
255244

Internship Tutors
Telecom Italia:
Ing. Salvatore Scarpina
Ing. Roberto Quasso

Academic year 2019-2020

Acknowledgements

I have reached the end of this chapter of my life and I think that this goal could not be possible without the people I met in this journey.
First of all, I would like to thank my parents, grandparents and my uncle Cristian for giving me the emotional support and for having encouraged me to complete this path.

I am thankful to Professor Luciano Lavagno who gave me the opportunity to grow up professionally. I really appreciated to work with him and his team, all of them have been really helpful and kind with me for the whole thesis period.
I want also to thank Mr. Roberto Quasso and Mr. Salvatore Scarpina for the attention and feedback they gave me during my speeches.

The most important contribution to this achievements belongs to my friends. All of them have been important for the time they spent to listen to my complains and my endless discussions.
In particular, I am grateful to Chiara for the many lunches shared together and for all the coffee time we had in these years. I also thank Giuseppe for the funny and no sense moments we had, they have been essential.
Last but not surely least, Valentina has been and she still is fundamental. She restored the confidence in my self and most of the goals I achieve are due to her love. I will always owe you for that.
Thank you all again.

Abstract

Low density parity check codes, known also as LDPC, are error correcting codes discovered in 1963 by Robert Gallager and they have been forgotten for many years because of the computational requirements needed to achieve the theoretical performance. LDPC codes can work with different block lengths and high rates which are very close to the Shannon channel capacity, thus they can provide a good bit error rate in noisy channels with a high throughput.
Recently LDPC codes have been chosen for the new wireless networks 5G by 3GPP as error correcting codes for user data since they can satisfy the demands requested by 5G.
Despite the technology improvement since the 60s, the LDPC decoder still represents a compute intensive task for 5G today because of the iterative algorithm used for decoding. Therefore, the LDPC decoder has been chosen to be off-loaded onto a field programmable gate array (FPGA) device in order to be accelerated. Nowadays FPGA devices offer flexibility and low power consumption since they operate at low frequency, although they handle parallel operations to improve the throughput.

The aim of this work is to accelerate on FPGA the LDPC decoder for 5G wireless networks using a software solution provided by the OpenairInterface Software Alliance consortium (OAI). The code by OAI is explored and optimized inside the SDAccel development environment by Xilinx, then the corresponding bitstream is generated and uploaded on a Xilinx FPGA.

The optimization work has begun from a C code for Intel processors supporting the Advanced Vector eXtension 2 (AVX2) library. Later the AVX2 solution is discarded for two reasons: the first one is that AVX2 instructions are not synthesizable by definition because they are not written in C language. Secondly pointer casting is not a synthesizable operation, thus the casts in the code must be converted into synthesizable code and the time required for this action is not negligible due to the large number of occurrences.
As an alternative option, a CUDA code for GPU has been chosen to be deployed on FPGA. The CUDA code has been firstly imported in OpenCL language and then optimized using the SDAccel environment exploiting the design flow of High Level Synthesis (HLS).
HLS techniques are adopted to implement the decoder. Firstly DRAM memory, or off-chip global memory, accesses are fully optimized by means of reading and writing burst operations. In addition, widening of global memory ports is used. In order to improve the data transfer during computation, the on-chip memory is exploited instead of the off-chip one, which would have increased the latency of the application due to the large access time.

Array reshaping is used to force the on-chip memory to have the same port width of the off-chip memory, therefore the data transfer between off-chip and on-chip memory is performed with the maximum memory interface width. Secondly some loops in the code are merged in order to reduce the filling and draining of the pipeline stages. Moreover loop unrolling and pipelining are exploited to improve the throughput of the kernel.

Three different performance results are obtained: the first one is related to the C code using the AVX2 library running on a 3.20GHz i7-6900K Intel CPU (Intel 14 nm). The second one is the CUDA code tested on a Nvidia Quadro P2000 (TSMC 16 nm) and the latter one is the performance of the FPGA accelerator board which is a VirtexUltrascale+ (TSMC 16 nm) by Xilinx.
The final results show that the FPGA decoder is slower than the GPU implementation with a factor of 382x, in fact the GPU solution takes 107.589 μs, whilst the FPGA at 300MHz spends 41.152 ms. On the other hand the AVX2 solution takes 257.549 μs to complete. The main reason of the FPGA poor performance is caused by the code structure that has been discovered being a worst case scenario for FPGA applications. An optimum code to be run on an FPGA must have inner loops with fixed loop bounds whilst the outermost ones can be variable. If inner loops have a constant number of iterations then they can be easily unrolled and memories can be proportionally partitioned to get the maximum parallelism. The CUDA code, which has been ported in OpenCL, instead has the opposite scenario. The innermost loops have variable loop bounds meanwhile the outer ones have fixed bounds.
The synthesizer used in this work, namely Vivado HLS, is not able to partition memories that are accessed with a non-constant index inside those loop. The logic that is implemented to use the memory banks dynamically with the loop bound almost triplicates the latency of the application. Moreover the size of the on-chip memory is not suitable to partition them completely because of limited resources on the device.
The final decoder implementation has completely unrolled inner loops and no memory partitioning, work items pipelining is applied to reduce the latency due to the work items loop. Given the source code structure, the OpenCL decoder cannot reach the performance of the GPU one since the combination of unrolling and memory partitioning cannot be totally exploited to improve the parallelism of the FPGA. To have better performance one has to write the code from the beginning or to hardly change the code structure, otherwise it is not possible to use complete loop unrolling and memory partitioning together.

Contents

List of Tables

List of Figures

Listings

Abbreviations and Acronyms

3GPP Third Generation Partnership Project.

5G Fifth Generation.

AVX2 Advanced Vector Extension 2.

AWS Amazon Web Service.

AXI4 Advanced eXtensible Interface 4.

BER Bit Error Rate.

BG Base Graph.

BN Bit Node.

BRAM Block of Random Access Memory.

CN Check Node.

CRC Cyclic Redundancy Check.

CUDA Compute Unified Device Architecture.

DRAM Dynamic Random Access Memory.

eMBB Enhanced Mobile Broadband.

FPGA Field Programmable Gate Array.

GPU Graphics Processing Unit.

HDL Hardware Description Language.

HLS High Level Synthesis.

i.i.d. Indipendent and Identical Distributed.

II Initiation Interval.

IoT Internet of Things.

LLR Logarithm Likelihood Ratio.

LTE Long Term Evolution.

LUT Look Up Table.

mMTC Massive Machine Type Communication.

NR New Radio.

OAI OpenAirInterface.

OpenCL Open Computing Language.

PCIe Peripheral Component Interconnect Express.

RTL Register Transfer Level.

SISO Soft-In Soft-Out.

SNR Signal to Noise Ratio.

URLLC Ultra Reliable Low Latency Communication.

VN Variable Node.

XOCC Xilinx OpenCL Compiler.

Chapter 1

Introduction

The 5G NR is a new wireless system that will play an important role in our world, since it is common nowadays to have many electronic devices that exchange information, like a group of drones coordinating each other, or dozens of sensors in a building that are collecting some data and sending them to a computer which elaborates the collected samples. This world is the widely known Internet of Things, where several objects communicate using the internet. An example of an IoT environment is the industry of today, where devices are strongly involved in the production chain and each of them communicate using the internet. Given the high amount of devices which is involved in such activities [1] and considering that it will increase every year, very high data rates must be achieved. Also security, effectiveness and low latency are key aspects of the Industry 4.0 [2], which is the fourth industry revolution where artificial intelligence, automation and similar technologies will exploit the 5G network.
5G will not be used only for the IoT, or more generally in Massive Machine Type Communication, but there are two additional use cases: Ultra Reliable Low Latency Communication and Enhanced Mobile Broadband as reported in [3] [4]. Besides the mMTC case, the URLLC scenario can be applied for the autonomous vehicles or remote surgery, whilst the eMBB is the improved user experience that 5G is supposed to provide as an evolution of the current mobile broadband service. Even though these are the main use cases for 5G, there are also intermediate possibilities, for instance something between the mMTC and URLLC can be a possible application field for the new radio access technology. So mMTC URLLC and eMBB are the main vertical use cases for NR.

3GPP is a group of seven telecommunication organizations that defines the standards for cellular telecommunications technologies, like the 5G ones. To summarize the requirements for 5G specified by 3GPP in [4], the new wireless network must be able to ensure low latency, high throughput, reliability and safety. Regarding the message that one device sends to another, a variable code rate and block length are required by the 5G. It has been found out by 3GPP that for the user data side

of the channel coding, low density parity check (LDPC) codes can achieve those demands as they can work with different length for the message to transmit and receive. Additionally LDPC codes can support different code rate according to the performance one wants to achieve. It is easy to imagine a typical scenario in which a message is passed from a device to a radio station, for example consider a vehicle or a sensor working in a factory that have to request an emergency service to a radio station. In these scenarios a lot of noise might be present and it will affect in an undefined manner the information from the transmitter along the path to the receiver. If this problem is not taken into account the received message will be wrong and the result of the operation might lead the two cars to have a crash or the two sensors to raise the alarm in the factory because of the erroneous measurements. So, from the reliability and safety point of views, LDPC , explored by Robert Gallager [5], can also handle these common situations by providing a good error detection and correction of the message with an high throughput.

In the LTE wireless network turbocodes were used for channel coding, since they are able to provide a reliable communication, especially at low code rates,. The code rate is defined as the ratio of the message length over the coded message length. With the new requirements of 20Gb/s for downlink, 10 Gb/s for uplink and a latency lower than 1 ms, turbocodes have been substituted by LDPC codes. People [6] [7] [8] have compared the two code families in terms of complexity and performance as Signal to Noise Ratio (SNR) and Bit Error Rate (BER). The two codes have a similar iterative decoding technique based on message passing, in other words the computational units exchange information about the received message in order to remove the noise from it. Turbo decoder have a serial structure which is a first possible bottleneck related to the latency constraint, secondly turbo decoder does not have a stopping criterion. The lack of a stopping condition leads the decoder to run useless decoding iteration even if the correct message has been already found. Moreover, the serial structure increases further the decoding latency unless specific solutions are taken into account to make the structure parallel. Even in this case, LTE turbo decoder is not capable to satisfy the demand of 5G channel coding.
On the other hand, LDPC decoder supports a parallel execution, this feature can be easily exploited to achieve high performances, thus improving the throughput during the decoding. For critical situation in which a very long message is received it would be possible to divide the processing in small chunks, each one working on one part of the message, in order to speed up the correction phase. Therefore the throughput is increased as the number of parallel work items increases.
It is demonstrated that LDPC codes are better than LTE turbo codes in terms of BER and SNR. Some researchers [7] show that for different code rate LDPC codes guarantee a better BER with respect to turbo codes. In those works it is illustrated that for a threshold SNR the LDPC decoder does not increase the amount of operations required to retrieve the original message. LTE turbo decoder, instead,

requires more operations when the SNR drops. Latency speaking, LTE turbo codes alternative would not be suitable for that use case, although for lower code rates LDPC codes converges towards the LTE turbo codes behavior.
Ranging from all the possible scenarios in which the 5G is used, it is important to have different codes to support all the services required. LDPC is capable of encapsulating the codes in an unique structure, which is a matrix, instead of having a dedicated description for each code. In fact in 5G there are thousands of codes to be supported and having a dedicated structure for each of them is not always feasible. Thus the matrix adopted in 5G LDPC codes is flexible in terms of description of supported codes and in terms of variable code rates.

Although the solutions adopted to implement LDPC decoding exploit basic operations which are simple to deploy on hardware platforms, the structure of the algorithm for decoding has a complexity which grows as the code rate drops. To be more precise, the check and correction performed by the decoder during one iteration can slow down the application. The latency drop is because of the explosion of the matrix itself whose dimension is proportional with the code rate. So for the latency requirement low code rates can be an issue and LDPC must be properly optimized, otherwise the grown complexity would affect the single decoding iteration.
The optimization must aim to improve the processing of a single iteration. It is clear that if a single decoding lap takes less time to complete, the overall latency is reduced. If one is able to shorten the duration of one iteration, then the designer can choose between two possible strategies: the first one is to reduce the duration of one iteration in order to have more time to improve the BER, which means that an higher maximum number of iterations is achievable. The second one is to spent less time to correct the message and provide an acceptable result as fast as possible. The choice depends on the specific scenario. If the constraints ask for low latency, like the URLLC use case, the designer should maintain the same number of maximum iterations but the correction will not be effective. Instead, if the scenario requires an hundred percent correct message regardless of the latency, the designer should improve the decoding processing to achieve a greater number of maximum iterations.

Given that LDPC decoder has a complexity which increases with low code rates and given that the iterative behavior can be, by definition, a problem for latency constraints, it must be off-loaded on platforms which exploits parallel execution. Since LDPC decoder processing can be split in small compute units [9] [10] that work on small block of the same message concurrently, CPU applications are not suitable for this task. CPU execution is strictly serial and its strong feature is the high performance data path which can achieve high frequencies to complete even a complex operation. Nowadays a new trend in designing is explored, for compute intensive application like the LDPC decoder, GPUs and FPGAs are used as accelerator boards. FPGAs uses simple logic operations to perform computations,

additionally small memory elements are present on board. Thus the capability of connecting simple operations and small memory elements together makes FPGAs extremely flexible. On the other hand GPUs exploit input vectorization, namely, a single input is spread over all the available compute units in order to perform multiple data operations.
Ffrequency is the metrics for CPUs performance, for FPGAs it is the throughput, in other words it is the amount Bytes elaborated in a unit of time. In fact FPGAs work with a clock frequency in the MHz range, on the other hand CPUs have clock of the order of GHz, but exploiting concurrency allows an FPGA application to reach good performance. Another important feature for which FPGAs are used for acceleration purposes is also they low power consumption [11], since the lower the operating frequency the lower is the dissipated power, in fact a MHz application wastes less power than a GHz one.
FPGAs devices allow also resources sharing, which is useful to save free space but it is not trivial if the design is done at RTL. HDL languages as VHDL and Verilog makes harder to use the flexibility of FPGAs, since the design is performed at low level. On the other hand, programming languages like C or C++ work at higher level therefore it is easier to implement complex operations and to optimize the target application within that point of view. Hardware designers are trying a new approach which consists in describing the application using programming languages, mainly using C or C++, and other tools convert the code program into an HDL file for the FPGA. These tools are also known as High Level Synthesis (HLS) tools. The tool used in this work is Vivado HLS [12]. Using HLS, the designer can easily optimized the application using pragma directives to tell the synthesizer to implement a pointed section of the code in a desired manner; hence the optimization is performed directly on the source code considering the hardware aspect and not the software one. Vivado HLS provides also a C validation and C co-simulation to verify the correctness and the functionality of the source code.
To complete the design flow, HLS is used inside the SDAccel environment [13] by Xilinx, which exploits a compiler to transform the RTL code into a binary bitstream which is loaded on the FPGA. The web cloud computing service by Amazon, AWS, is used to test the LDPC decoder on an FPGA board.
In this thesis the LDPC decoder for 5G NR is optimized using HLS techniques to reduce the execution time of the application. The starting code is a software solution for GPUs provided by the Openairinterface consortium and has been imported from CUDA into OpenCL, which is a C/C++ based framework used to write codes that are executed across different platforms, like CPU and FPGA in this case. The SDAccel environment supports OpenCL and it is used to generate the bitstream for the target FPGA.
To reduce the execution time, global memory accesses have been improved, as well as on-chip memory of the FPGA has been extremely used to reduce the time

spent in off-chip memory acceses. To improve the throughput and degree of parallelism unrolling techniques, loop fusion and work items pipelining are exploited. The achieved performance are compared with the initial porting, with the original CUDA code and with the solution for Intel processors.

The thesis is organized as follows: the second chapter provides the theoretical knowledge to understand the LDPC processing, the encoder is briefly covered. The decoder is explored presenting shortly the work done by Gallagher and presenting some algorithms generally used for decoding.
The third chapter introduces the OAI consortium which provides the open source codes for the 5G LDPC module. The two available solutions will be analyzed, the AVX2 for Intel CPUs and the GPU code written in CUDA language.
The fourth chapter shows the design flow used to accelerate the LDPC decoder, firstly the porting from CUDA to OpenCL source code is analyzed. Then each optimized solutions are shown step by step, the code is explored in detail. In the final part of the chapter a comparison between all the solutions is presented.
A future work chapter describes additional theoretical improvements that can be explored and the last chapter summarizes the results obtained at the end of the thesis.

Chapter 2

Low density parity check codes

Low density parity check codes where invented by Robert Gallager in 1963 [5] and were proposed as a good solution for noisy transmission where the message going through a symmetric binary input channel is affected by noise with an arbitrary error probability P_{err}. Gallager in his work presents a decoding strategy based on a posteriori probability on the input message. The proposed decoding method is able to minimize the decoding error for the given length of the input message, the final performance are closed to the channel capacity C_{cap}. The channel model taken into account in [5] is a binary symmetric channel like the one depicted in figure 2.1, which is: time-discrete, memoryless (the current channel output does not depend on previous outputs or inputs) and it works with digit sequences of zeroes and ones.

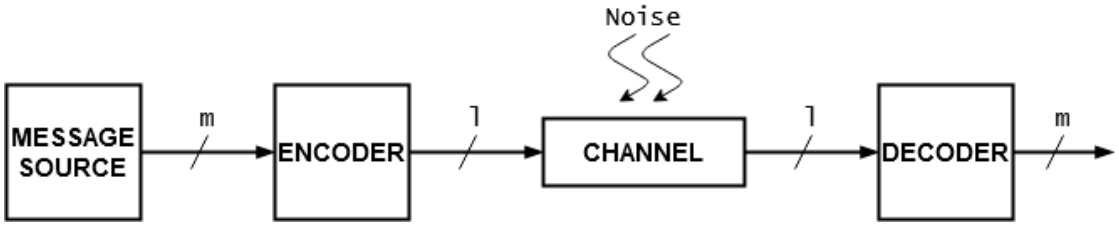

Figure 2.1. Scheme of a noisy channel with encoding and decoding modules

The channel capacity is defined as the rate of transmitting the information per unit of time throughout a channel within an arbitrary error probability that can be considered small. From Shannon coding theorem [14], there exists a channel working with an information rate $R_t < C_{cap}$ in which a m-message is coded within

l bits and it is decoded with an error probability P_{err} that is contained in the following interval [5]:

$$e^{-m \cdot X_1(R_t)} \leq P_{err} < e^{-m \cdot X_2(R_t)} \tag{2.1}$$

where X_1 and X_2 are functions that describe the behavior of the probability according to the channel they are applied to. As R_t grows X_1 and X_2 become smaller, on the other hand if R_t gets closer to zero the two functions rise, moreover, if R_t is high such that it is equal to C_{cap}, then X_1 and X_2 are zeroed, therefore the upper bound of P_{err} would be 1 and the lower one would be a value greater than 0 that depends on X_1. So, the higher the information rate R_t the higher P_{err}.
Surely the error probability cannot be null for any codes but it can be reduced or maintained low enough to have a good outcome. From the coding side Gallager proposes a redundant solution in which n redundant bits are appended to the information to protect it from the noise, the coded message is defined as codeword. The redundant bits are called parity bits since they represent the modulo two sum (or the exclusive-or boolean operation) of a set of message bits, i.e. if the number of bits is odd the result is 1 otherwise it is 0. The message bits and the parity bits involved in this operation form a parity check set.
Regarding the complexity, the encoder has to work with a large number of bits which is m+n, these bits are involved in the parity check equations each one containing a small set of bits. Despite the operation itself is trivial to implement, the number of sums is not that low, this can be a problem for applications in which the amount of resources to use for computation is limited, for instance in an FPGA application. Moreover the encoder must know which parity and message bits are involved in which parity check equation, in other words memory is required to hold these information and the memory cost is not low too.

Concerning the decoding of LDPC codes, Gallager explored an algorithm based on belief propagation of a bit through the parity check equations in which the bit is involved. In particular, the *a posteriori* probability for an input bit conditional on the received symbol is used to determine the belief for that bit. Using the *a posteriori* probability the decoder have more information than the hard decision solution in which the received symbol is set to 1 or 0 and it is eventually corrected using only the codeword set.
Instead, with *a posteriori* probability the decoder uses also the knowledge coming from the transmitted message with the information coming from its parity check equations that must be matched in order to be considered valid. The feature of the LDPC decoder of operating at fast speed on several small parity check equations providing a low bit error rate (BER) makes it a good candidate for application in which high speed and high reliability are required, like the 5G wireless system.

In this chapter LDPC codes are covered from a top-level view in order to provide the information needed to understand the work done in the thesis, for sake of clarity also the encoder is taken into account although it is not the main target

of this work. Furthermore, an important tool, namely the Tanner graph, is used to support the explanation of the decoder. In the decoder section also a brief explanation of the message passing algorithm based on minimum sum approximation is discussed and some optimized alternatives to the minimum sum algorithm are briefly introduced.

2.1 Encoding

As mentioned in the introduction of this chapter, low density parity check codes have two types of bits: the information or message bits and the parity bits. Both message and parity bits are involved in the so called parity check equation, this is one of the reason for the name parity check codes.
Parity check equations are modulo 2 sums of the bits involved, for example:

$$\begin{aligned} &1. \quad x_2 + x_3 + x_4 + x_5 = 0 \\ &2. \quad x_1 + x_3 + x_4 + x_6 = 0 \\ &3. \quad x_1 + x_2 + x_4 + x_7 = 0 \end{aligned} \tag{2.2}$$

where x_1, x_2, x_3 and x_4 are message bits, x_5, x_6 and x_7 are the parity bits which must be computed and appended to the message to protect it from the noise. An erasure is a phenomenon due to the noise which corrupts a transmitted bit such that it is not possible to determine at the receiver if the bit is 0 or 1. Since one parity bit protects one information bit of the equation in which it is involved, the parity bit is able to fix only one erasure. Instead, if in the linear system 2.2 two erasures occur, for instance on bits x_2 and x_4, the parity check equations 1. and 3. are not able to recover the information but the second equation has one unknown bit, therefore it is able to restore x_4. Once x_4 has been recovered it is possible to correct also x_2. This situation is easy to handle because message bits are involved in more parity check equations, otherwise if multiple erasures occur and each bit participates in only one equation the decoding becomes impossible.
By noting the linear system in 2.2, parity bits are not protected in the same manner of message bits, in fact in case of multiple erasure affecting also parity bits it is not possible to fix the corruption. To solve this problem related to the erasures, parity bits are protected in the same way of the message ones, therefore additional equations are required to have parity bits participating in more than one equation. Hence the linear system in 2.2 becomes:

$$\begin{aligned} &1. \quad x_1 + x_2 + x_5 + x_6 = 0 \\ &2. \quad x_4 + x_5 + x_6 = 0 \\ &3. \quad x_2 + x_3 + x_4 + x_6 + x_7 = 0 \\ &4. \quad x_1 + x_2 + x_5 + x_7 = 0 \end{aligned} \tag{2.3}$$

In this case, even if two parity bits and a message one are erased it is possible to recover the information because all the bits are protected equally.
The example reported above is valid for a 4 bits long message, for very long messages the complexity grows linearly because the number of equations to add increases as the number of message and parity bits raises. The problematic part of the encoder is related to the number of equations and to the number of parity bits to be evaluated, in fact the more the parity and message bits the more the number of equations to be solved. The growth behavior for this kind of codes is linear, in fact LDPC codes belongs to the family of binary linear block codes. By definition a linear binary block code has m message bits, they are encoded in l code bits, thus the code block is defined as a *(l,m)* block code with l-m parity bits [15]; for the code in 2.3 the code is a (7,5) linear block code.

The ensemble of the parity check equations of an LDPC code form the parity check matrix H, which is a big matrix made of ones and zeroes. To be more precise, the rows of a parity check matrix are the parity check equations, the columns of the matrix represent the bit of the codeword. The main feature of these kind of matrices in LDPC codes is that they are sparse matrices, since the number of ones is low if compared to the number of entries of H.
In order to derive the matrix H the i-th column of a row is set to 1 if that bit of the codeword is involved in that parity check equation, if it is not that entry is 0. For instance, if we refer to the equation 2.3 the corresponding parity check matrix is:

$$H = \begin{pmatrix} 1 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 & 1 & 0 & 1 \end{pmatrix} \tag{2.4}$$

The matrix reported above is not sparse but it has been presented just as an example, a sparse parity check matrix is similar to the following one:

$$H = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \end{pmatrix} \tag{2.5}$$

Hence, the name of low density parity check codes derives from the low number of ones with respect to the number of entries of the matrix H. It is important to point out that there is not only one parity check matrix H for all LDPC codes, but each code has its own matrix based on the use cases of that code. For instance, the parity check matrix of LDPC codes for digital video broadcast (DVB) is different with respect to the one used in 5G, whose matrix changes according to the block length of the message as will be described in the next chapter.
Finally, to encode a message of m bits into a codeword of length l with l-m parity

bits p, the following equation must hold:

$$Hc = 0 \tag{2.6}$$

where 0 is a null vector and c is the codeword column vector:

$$c = [m_1 \quad m_2 \quad ... \quad m_m \quad p_1 \quad p_2 \quad ... \quad p_{l-m}]$$

Since the ones in H are bits involved in a parity equation, we have that the equations to be solved are very fast because of the few number of bits, this structure makes LDPC encoder very fast and easy to implement, also from the decoder point of view.
Moreover, such structure is perfectly suitable to have a parallel execution rather than a sequential one. The main drawback of the encoder is the storage requirement because the entire matrix, which can be very large, must be stored in a memory (ROM or RAM) and must be read every time for the parity bits evaluation. For instance H can have a size of 17664 rows and 26112 columns, thus the memory requirement is high as well as the computational part, even if the operation itself is a simple one.
Because of the discussed drawback there is a lot of research activity [16] [17] to reduce the design complexity and resource usage.

2.2 Tanner Graph

Before proceeding with the explanation of the decoding of LDPC codes, a brief presentation of Tanner graphs is required since it is an important tool that helps to understand how the data set is organized and how it will be processed during the decoding.
Tanner graphs are bipartite graphs which are made of two kinds of nodes that are linked together through edges. It is important to point out that an edge starts from a node of a type and it goes inside a node of the other type, there are no edges connecting two nodes of the same family. The term bipartite comes from the disjointedness of the nodes into two classes of nodes.
For LDPC codes the Tanner graph is derived directly from the parity check matrix H. First of all the two class of nodes must be defined:
the first one is CN, they represent the parity check equations of H, the second one is VN or BN class, they are the columns of H, thus they corresponds to the bit of the codeword.
As described in the previous section, an entry corresponding to 1 in H means that the i-th bit of the codeword is involved in the j-th parity equation of the matrix. From the Tanner graph point of view, that entry corresponds to an edge between the i-th BN and the j-th CN.
Another property of Tanner graphs is the degree of the nodes, it is defined as the

number of edges going from/to a given node. For example, the degree of a check node corresponds to the number of bit nodes it is connected and viceversa for the bit node degree. The degree property of Tanner graphs distinguishes two kinds of LDPC codes: the regular and the irregular ones.
All the check nodes of a regular code have a C-degree, whilst the variable nodes have a V-degree, hence the Tanner graph has a regular structure that can be re-arranged to be a tree-like one. Such codes have an advantage and a drawback: the first one is that the processing is trivial to implement since the structure is regular. In fact, once the processing of the bit nodes connected to a check node is performed the computation for other BNs is done in the same manner. This kind of processing can be implemented in software as two nested loops, the inner one iterates over the check nodes and the outer one loops over the BNs. On the other hand, regular structure does not provide good performances as well as the irregular ones, moreover the designer has to choose properly the degree for both nodes, which might be not as trivial as the implementation of the processing part. The Tanner graph corresponding to the matrix in 2.4 is the one depicted below:

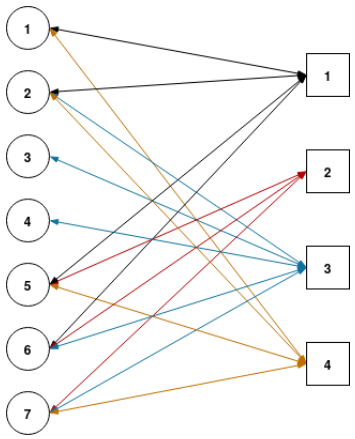

Figure 2.2. Tanner graph of a parity check matrix

The Tanner graph in figure 2.2 belongs to an irregular code since the check nodes (the squares) have three, four or five incoming edges and variable nodes 2, 6 and 7 (the circles) have a variable number of edges as well. Node 3 has three edges and the remain nodes have four edges, hence the code is irregular. The main advantage of creating irregular codes is that it is simpler than the regular ones since they are generated according to some error probability distribution. Therefore the design complexity discussed before is no more valid when designing irregular codes because it is enough to specify an error distribution according to the channel and the requirements of the code. On the other hand, the weak point of irregular codes is the processing organization which is not the same of the regular ones, since the

number of edges of the Tanner graph changes from one node to another. In order to make the processing regular, the designer can organize the nodes with the same degree into groups, which means partitioning the nodes into small set of nodes, and then use an additional loop, with respect to those discussed for regular codes, to move the processing from group to group.

2.3 Decoding

According to Gallager [5] to improve the performance of decoding LDPC codes it is better to use the *a posteriori* probability, namely the probability of a bit being equal to 0 or 1 conditional on the transmitted symbols, in this way the information that is used by the decoder is more explanatory because it takes into account also the starting condition of a symbol, which is the received value from the channel. Moreover, Gallager takes into account also the condition that the corresponding parity check equation is satisfied given that value of the bit. Instead, if a predefined value is applied on the input symbol then the performance would deteriorate because only the information related to the known codeword set is used to correct the input symbol without its history.
In his work, Gallager established the following theorem:

Theorem 1 *Let P_d be the probability that the transmitted digit in position d is a 1 conditional on the received digit in position d, and let P_{il} be the same probability for the l^{th} digit in the i^{th} parity check set in which d is involved. Let the digits be statistically independent of each other, and let S be the event that the transmitted digits satisfy the j parity check constraints in which d is involved. Then:*

$$\frac{Pr[x_d = 0|\{y\}, S]}{Pr[x_d = 1|\{y\}, S]} = \frac{1 - P_d}{P_d} \prod_{i=1}^{j} \left[\frac{1 + \prod_{l=1}^{k-1}(1 - 2P_{il})}{1 - \prod_{l=1}^{k-1}(1 - 2P_{il})} \right] \tag{2.7}$$

where $Pr[x_d|\{y\}, S]$ is the probability of the digit d being equal to 1 or 0, since the channel is a binary one, conditional on the received symbol y and on the event S that each transmitted digit in the same parity check equation j of d satisfy the equation itself.
The theorem is based on a notation similar to the Tanner graphs but a little bit different:

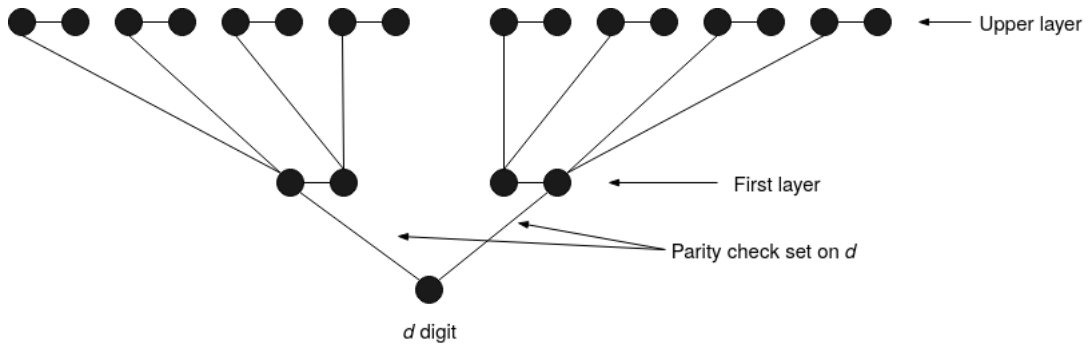

Figure 2.3. Parity check set tree according to Gallager notation

in this new notation, an edge connecting two nodes is a parity check equation involving the two digits in a specific position of the tree-like structure. The root of the tree is represented by the digit d whose probability must be computed considering the bits participating in its equations. Once the probabilities of digit d and its bits are evaluated, they can be used to compute the probability of other bits by re-arranging the equation 2.7. Then the theorem can be applied for each received digit. Once all probabilities have been computed, they can be used in the equation 2.7 again for a new iteration through the tree starting from the root. As the number of iteration increases, the probabilities will converge either to 0 or 1.

Basically, the decoder scheme proposed by Gallager is a probability-based decoder, in which the information that is used to correct and decode the received symbol is not the actual value received, known also as hard-input, but it is a belief on the received symbol, namely soft-in, moreover the output of the decoder is not a sequence of binary digits (hard-out) but it is a sequence of believes of the decoded message (soft-out). In fact, to translate the believes of the soft-in soft-out (SISO) system into actual values a hard decision must be applied on the output believes.

Conventionally, to represent the probability of a bit conditional on its received value for LDPC decoding algorithms people use the so called log-likelihood ratio (LLR) of the probabilities. Suppose that after the encoder in figure 2.1 a binary phase shift keying (BPSK) modulation is applied on each bit of the codeword [15]: binary value zero is transmitted as a +1 and the binary one is transmitted as -1. Furthermore the noise that affects the channel is an additive white Gaussian noise (AWGN) with variance σ^2 and a vector of independent and identical distributed variables (i.i.d.) $\mathrm{n} = [n_1 \quad n_2 \quad ... \quad n_l]$ applied to the codeword in the channel, such that:

$$y_i = x_i + n_i \tag{2.8}$$

where x_i is the i-th bit of the codeword at the output of the encoder, whilst y_i is the corresponding received bit by the decoder.
Starting from the conditional probability of a bit x_i being equal to +1 for a BPSK

channel given the received value y_i:

$$P_r(x_i = +1|y_i) = \frac{f_r(y_i|x = +1) \cdot P(x_i = +1)}{f_r(y_i)} \tag{2.9}$$

where the noise of the AWGN channel has a normal distribution while y_i has a uniform distribution $f_r(y_i)$. Because of a binary symmetric channel, the *a priori* probability of x_i $P(x_i = \{-1, +1\})$ can be considered one half. For simplicity, it is better to use the ratio of the two probabilities:

$$\begin{gathered} \frac{P_r(x_i = +1|y_i)}{P_r(x_i = -1|y_i)} = \frac{\frac{f_r(y_i|x_i=+1)\cdot P(x_i=+1)}{f_r(y_i)}}{\frac{f_r(y_i|x_i=-1)\cdot P(x_i=-1)}{f_r(y_i)}} \\ \frac{P_r(x_i = +1|y_i)}{P_r(x_i = -1|y_i)} = \frac{f_r(y_i|x_i = +1)}{f_r(y_i|x_i = -1)} \end{gathered} \tag{2.10}$$

The final result in equation 2.10 is called likelihood ratio and it is the ratio of the probability distribution of the received codeword conditional on the transmitted bit being +1 or -1. Thus, if y_i is the transmitted codeword plus noise in a AWGN channel with mean $\mu = 0$, it will be either +1 +N(0,σ^2) or -1 +N(0,σ^2). Hence the equation 2.10 becomes:

$$\frac{P_r(x_i = +1|y_i)}{P_r(x_i = -1|y_i)} = \frac{\frac{1}{\sqrt{2\pi}\sigma} e^{\frac{-(y_i-1)^2}{2\sigma^2}}}{\frac{1}{\sqrt{2\pi}\sigma} e^{\frac{-(y_i+1)^2}{2\sigma^2}}} = e^{\frac{2y_i}{\sigma^2}} \tag{2.11}$$

If the simple likelihood ratio is used, from the implementation point of view it would be amazingly expensive, because the decoder would have to carry big numbers, which means floating point data, and for FPGA applications such as accelerator this representation for the data is too complex. Hence, people use the logarithm of the likelihood ratio, therefore the quantity in equation 2.11 is transformed into:

$$log\left(\frac{P_r(x_i = +1|y_i)}{P_r(x_i = -1|y_i)}\right) = \frac{2y_i}{\sigma^2} \tag{2.12}$$

generally it is reported as:

$$LLR = log\left(\frac{P_r(x_i = +1|y_i)}{P_r(x_i = -1|y_i)}\right) \tag{2.13}$$

The final formula in 2.13 has the great advantage of having a small dynamic range, therefore it can be easily implemented using fixed point representation, which is extremely important if the LDPC decoder must be deployed on an FPGA for acceleration purpose. Finally, in the decoder algorithm that is presented in the next subsection, uses LLRs for data representation, the two drawbacks of this representation is that a conversion of channel samples into logarithmic probabilities ratio

is required, but the conversion can be simplified using the formula in 2.12. The second drawback is the precision loss during calculation.
The algorithm adopted in LDPC codes for 5G NR is the message passing algorithm, the computation of the probabilities according to Gallager's theorem in 2.7 is basically divided into two parts, the first one is the evaluation of the event S for the parity check equation to be satisfied for a bit x_i, the latter is the evaluation of the LLR of x_i. After the new LLR evaluation for each bit of the message x, the decoder starts a new iteration..

2.3.1 Message passing algorithm

The message passing algorithm is an iterative algorithm which exploits the Tanner graph representation, it is based on the belief propagation and it is divided in two steps: the check node processing and the bit node processing. At each step a message is sent from one family node to the other one, when both steps have been completed a new iteration can start, the type of the message that is sent back and forth changes according to the direction of transmission. Once a node receives all the messages from the nodes connected to it, it will compute the reply to those messages and it will send back to its nodes the new computed message.
First of all, the message to send to a node is made of the extrinsic information only, the intrinsic one instead is the information of the destination node, for example, a bit node A receives the message of check nodes B C and D and it uses them with the channel sample for the evaluation of the new message; when A sends the reply to B C and D it will send the information derived from all the check nodes message except the one of the destination node. The reason of omitting the information related to the destination node is that we are interested in computing a belief on the destination node using the believes of other nodes. If also the intrinsic information is used, the node evaluation would be an overestimation and thus it would be wrong. The intrinsic information can be seen as the local information of a node, namely the channel sample for a bit node or the belief of being satisfied for a check node, whilst the extrinsic one is the global part coming from the rest of the graph.
Regarding the message passed from one type of node to the other it depends on the algorithm used for the processing, the notation adopted is the one used in [18] and the algebraic part will be covered in the next section.
In the following the notation used in this work is shown:

- $\mathrm{q}_{i \to j}$ is the message passed from the i-th bit node to the j-th check node, it carries the channel output value of bit node i and the extrinsic information which is the believes coming from all the check nodes in which the variable node is involved except the j-th one, which is the node that will receive the new message. Simply speaking, the message transmitted from the V-node tells if the bit is 0 or 1 given the channel sample (which is the intrinsic information)

and the received messages from its check nodes $P(x_i = 0,1|y_i, r_{i\to J\setminus j})$, where J is the ensemble of the check nodes connected to the bit node.

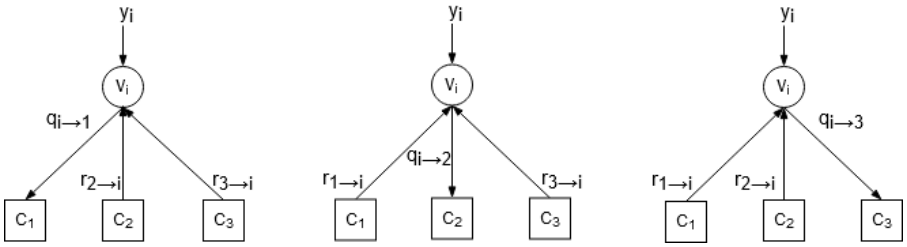

Figure 2.4. Bit node processing in one iteration

Then, the V-node V_i takes all the probabilities of the check nodes it is involved in and given the channel sample computes the probability of being 1 or 0. The final probability is sent through the edges to the check nodes but the destination C-node information is omitted in the message.

- $r_{j\to i}$ is the message from the check node j to the bit node i, it holds the information that the corresponding check equation is satisfied knowing that the V-node i has a certain belief and considering all the believes coming from other V-nodes involved in that parity check equation, except the V-node i.

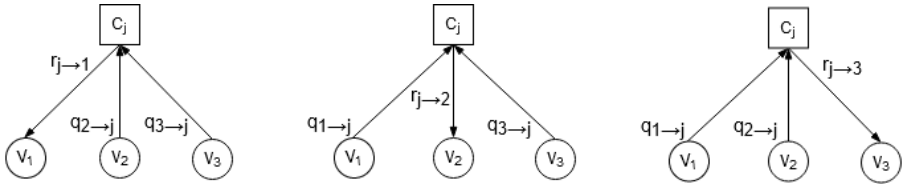

Figure 2.5. Check node processing in one iteration

Thus, in this part of the message passing the j-th check node computes the probability of itself being equal to zero that is $P(checkmet|q_{I\setminus i\to j})$ based on the probabilities of its bits.

The stopping criterion can be either a maximum iteration because of latency constraints or the matching of all parity check equation, in this case it means that decoder has separated the noise from the codeword and the original message has been found.

As reported by Gallager, the decoder has to compute the *a posteriori* probability for a digit x_i being to 1, or 0, conditional on the received symbol y and the event S_i that the j-th parity check equation is met, which means that the modulo two sum

of the parity check set must be 0, in other words the equation 2.6 must hold.
Thus the decoder has to verify that the result of each parity check equation is 0 using the conditional probability: P(x_i=$\{0,1\}|y$), to do so first the decoder derives the parity check matrix for the adopted LDPC code, then it gets the corresponding Tanner graph from the matrix H.

At the first iteration, the algorithm can start with the variable node processing or check node processing according to the designer choice; actually at the end of the first iteration the result will be the same, because initially the decoder has knowledge only on the received value since there is no history of previous iterations, hence the processing of the variable nodes relies only on that, thus the message passed from variable nodes to check nodes is the received value from the channel of each bit of the codeword. To be consistent with the design choice adopted in this work, the message passing algorithm starts with the check node processing, so the check nodes of the parity check matrix are fed with the received value of the symbol y whose bits are represented as LLRs. Hence, before proceeding with the first iteration, the hard input bits must be converted into soft input, which means from real value into LLRs. In case of BSPK modulation and AWGN channe with σ standard deviation, the initial LLRs are given by

$$LLR = 2\frac{y_i}{\sigma^2} = \begin{cases} \frac{2}{\sigma^2} & if \quad y_i = +1 \\ -\frac{2}{\sigma^2} & if \quad y_i = -1 \end{cases} \tag{2.14}$$

Then the initial values of the LLRs are fed into the check nodes, like if they are part of the $q_{i\to j}$ message. After the initialization, the check nodes will compute their extrinsic information to send to the bit nodes, given the LLRs, after that V-nodes will compute the probability of being 1 or 0 according to the received value from C-nodes. If all the parity check equations are satisfied, then the algorithm can stop, otherwise for the new iteration the LLRs must be updated with the results of the current iteration and a new message $q_{i\to j}$ is sent to the CNs . By noting that for each iteration the channel sample must be constant because it represents the starting condition for the codeword.

2.3.2 Minimum sum algorithm

By referring to the decoder scheme of Gallager, people develop different algorithms to compute the belief. A famous one is the sum product algorithm (SPA) investigated by Mackay and Neal [19] who provide results that demonstrate that SPA achieves good performance, although the cost in terms of complexity is high and from the implementation point of view it is expensive to have such operations in hardware.
As reported in [18], the message passing algorithm using SPA has the following four steps:

1. Initialization of LLRs and check node processing

2. Each check node j computes the message $r_{j\to i}$ using its bit node messages according to the formula:

$$r_{j\to i} = \left(\prod_{i'\in I\setminus i} sign(q_{i'\to j})\right) \cdot 2 \cdot tanh^{-1}\left(\prod_{i'\in I\setminus i} tanh\left(\frac{|q_{i'\to j}|}{2}\right)\right) \tag{2.15}$$

   With I being the ensemble of bit nodes that participate to the j-th parity check equation, on the other hand i' is a bit node used to compute the extrinsic information, since the i-th bit node belief must not be included in the computation because it is the destination node.

3. Each bit node i processes the received message from the check nodes connected to it to evaluate the new belief:

$$q_{i\to j} = y_i + \sum_{j'\in J\setminus j} r_{j'\to i}(x_i) \tag{2.16}$$

   with J being the ensemble of the check nodes connected to the bit node i whilst j is the destination node. y_i is the channel sample of the x_i bit whose belief must be computed.

4. The so called decision step occurs after the bit node processing and computes the final LLRs for that iteration. In this step a similar operation to the one in equation 2.16 is performed:

$$q_{i\to j} = y_i + \sum_{j\in J} r_{j\to i}(x_i) \tag{2.17}$$

   but in this step all the messages coming from the check nodes are taken into account because no message must be sent but the LLR for each bit of the codeword must be evaluate considering all the information from the Tanner graph. By referring to the equation 2.13, knowing that the LLRs are logarithmic functions, if the sign is negative it means that the denominator (thus the probability of that bit being a 1) is greater than the nominator, then the belief of x_i=1 is stronger than the x_i=0. Otherwise, if the LLR is positive, the hard decision step sets x_i equal to 0.

If the maximum number of iterations is reached, then the algorithm can stop, otherwise from step 4. a new iteration can start by updating the old LLRs with the new computed ones. In the equation 2.15 there are the two issues related to this algorithm, although the SPA provides good performance, the complexity introduced by the tanh and $\tanh^{-1}$ functions is not negligible, especially for FPGA applications in which floating point units are absent and in which the resources are limited.

Thus, people in [20] present an approximated version of sum-product algorithm, namely the minimum sum (min-sum) algorithm, which has the same operation structure, the only difference is related to the problematic equation 2.15. The solution they presented exploits the addition even for the check node processing, according to the authors the new solution is simple to implement both in software and hardware.
Thus, the new equation to evaluate the probabilities for the check nodes becomes 2.5:

$$r_{j\to i} = \left(\prod_{i' \in I \setminus i} sign(q_{i'\to j}) \right) \cdot \min_{i' \in I \setminus i} (|q_{i'\to j}|) \tag{2.18}$$

Hence, the $\tanh^{-1}(\prod tanh())$ operation is simplified with a minimum operation which reduces amazingly the complexity but the approximations leads to a loss of performance with respect to the SPA algorithm because of loss in computation accuracy.

In order to reduce the loss of performance, as said in [18], researchers work on another variant of min-sum which is the scaling min sum, where the equation 2.18 has a scalar quantity in front to compensate the loss. Since the scalar quantity is not constant but it might change from one iteration to another, or from run to run of decoding, it is required to recompute every time the scaling factor. In [18] people present a simplified version of the scaling min sum called simplified variable scaling min sum, where the scaling factor α is computed iteratively using an heuristic equation that adapts the quantity with the current iteration. In their paper, they show that the new algorithm has the best BER vs SNR behavior for different code rates if it is compared with the SPA, min-sum and scaling min-sum algorithms.
In the work presented in this thesis, the basic minimum sum algorithm is used but the simplified variable scaling minimum sum can be considered as a possible target for the optimization and acceleration of LDPC decoder for the 5G network.

Chapter 3

LDPC in OpenAirInterface

In this chapter the open source codes provided by the OpenAirInterface™ Software Alliance[1] (OSA or OAI) are described. OAI is a french non profit consortium which offers standard compliant solutions for 5G wireless network and more. Regarding the LDPC codes they provide a solution for Intel processors exploiting the Advanced Vector Extension2 (AVX2) to improve the performance using single input multiple data (SIMD) instructions. These instructions uses a vector data type made of 256 bits of integer, or 128 bits in some cases. The size of an integer is of 8 bits, therefore there are 32 elements in a vector, each of them is aligned in the memory. One part of the computation involves vectors whose elements are 16 bits wide, thus the total number of item in the vector is reduced to 16.
Recently also a CUDA (Compute Unified Device Architecture) code has been added to one branch of the OAI repository[2] tested on an nVidia Quadro P2000 GPU.
Firstly a description concerning the CPU code is provided, the OAI code is explored both for the decoder and for the testbench used to produce the input data set and to collect the results from the decoder. Also the issue that makes the AVX2 code not synthesizable is shortly discussed. Finally the CUDA code structure is shown and then converted into OpenCL (Open Computing Language) to be used in the SDAccel environment for FPGA deployment, as explored in the next chapter.

3.1 Testbench

One of the two available codes of OAI concerning LDPC is a C based simulation environment where both encoder and decoder are implemented and tested within a testbench. The encoder has several solution starting from the simplest one implementing the modulo two sums up to the optimized solution exploiting the AVX2

[1] `https://www.openairinterface.org`

[2] `https://gitlab.eurecom.fr/oai/openairinterface5g`

library from Intel. On the other hand the decoder has two implementations, one for Intel processors, the second one is for GPUs. The Intel solution is the one that has been explored first to be deployed on FPGA.

First of all, LDPC codes in 5G NR belong to the quasi-cyclic class, i.e. the parity check matrix H is built starting from a smaller matrix, called Base Graph (BG). According to the base graph entries, which are not binary values but natural numbers, and to a lifting factor Z_C the final matrix is obtained. People in [21] say that quasi-cyclic LDPC codes are used mainly for their low complexity for both encoding and decoding, in fact instead of storing the whole parity check matrix, which might be huge, a smaller one is stored and then it is modified according to the parameters of the LDPC module.
The 3GPP group states in the technical report [22] that 5G supports two base graphs, namely the base graph 1 (BG1) and the base graph 2 (BG2) are used for different payload size and code rate $R = \frac{message\,length}{codeword\,length}$:

Block size [bits]	3841∼8448			192∼3840		
Base graph	BG1 46x68			BG2 42x52		
Base graph	1			2		
Code rate R	1/3	2/3	8/9	1/5	1/3	2/3

Table 3.1. Block sizes and code rates supported by BG1 and BG2 in OAI implementations

where the block size is the size of the message. Actually, in the coding theory terminology the message to be send is called transport block. In NR the transport block is divided into segmentation blocks which can be up to 8448 bits, then the segment can be encoded as described in chapter two. In the table 3.1, each BG can support different code rates and different block sizes, BG1 is used mainly for an high payload since the length of the codeword can be close to the message length with a code rate of 8/9, this means that most of the codeword is dedicated to message bits. On the other hand BG2 is used for small block size but with a low code rate down to 1/5, which means that the codeword is 5 times bigger than the original message, thus the information is strongly protected and easily recoverable. Also the number of rows and columns for each base graph is reported.

In order to determine the size of the parity check matrix, the lifting factor Z_C, or lifting size, must be computed. By referring to the notation used in Fig.2.1:

$$Z_C = \min_{z \in Z} \left[z > \frac{m}{N_b} \right] \tag{3.1}$$

where N_b is the amount of bit to encode, z is a lifting factor of a set as reported in [22], where a discrete ensemble of z is given to reduce the complexity of the LDPC module. The list of the lifting factors available for 5G NR is reported in the

appendix A, table A.1.
After Z_C is obtained, the parity check matrix size is built, since each entry of the BG is replaced by a Z_CxZ_C identity matrix, circularly shifted to the right by a fixed amount. The shifting value is given by $H_{BG}[i][j] \mod Z_C$, where H_{BG}[i][j] is the entry of the base graph. If H_{BG}[i][j] is -1 then it is replaced by a Z_C x Z_C 0 matrix, if H_{BG}[i][j] is 0 then the entry is replaced by a non-shifted identity matrix, otherwise it is substituted by a circular shifted identity matrix. Therefore, if BG is MxN, the parity check matrix obtained with a Z_C lifting factor has M· Z_C x N· Z_C elements.
For instance let us consider the following base graph and a lifting factor $Z_C = 2$:

$$H_{BG} = \begin{pmatrix} 9 & 117 & 6 \\ 81 & 0 & -1 \end{pmatrix} \tag{3.2}$$

according to the method presented above, the corresponding parity check matrix is:

$$H = \begin{pmatrix} 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 \end{pmatrix} \tag{3.3}$$

thus, starting from a 2x3 matrix a new one with 4 rows and 6 columns is derived. As a reminder, the columns of the matrix H represent the bits of a codeword, the rows are the parity check equations in which the bits are involved and a 1 in H tells that the corresponding bit participates to that parity check equation.
Regarding the decoder, once the parity check matrix has been generated from the corresponding BG, the decoder knows exactly what is the relationship between the bit nodes and the check nodes of the corresponding Tanner graph.

In 5G NR standards, before transmission, rate matching is applied to improve the performance and to achieve high rates, these goals are obtained through puncturing and shortening of the bits.
For instance, at the encoder side after matrix expansion and encoding, the first $2Z_C$ bits for BG1 are always punctured to increase the code rate, therefore they are not transmitted but still used by the decoder. Thus the decoder has to recover the value of those bits because it does not know which value has been sent, in fact the LLR of punctured bits is always equal to 0 because they are treated as erasures. Further puncturing is performed for the parity bits according to the LDPC parameters (Z_C, block length, base graph choice, code rate), also in this case their LLR at the decoder is 0. Generally the parity bits to be punctured are the rightmost ones in the parity check matrix.
Shortening occurs only for message bits before encoding and it consists in adding a specific number of zeroes to the message to adapt the length. After shortening the encoding of the message occurs. Later, the bits corresponding to the additional zeroes are removed from the codeword and transmission takes place. This technique

is used to adapt the message length to the desired one, shortened bits are set to high value at the decoder because they are certainly zeroes.
Puncturing ruins the performance because the decoder assumes that those bits are erasures and hence they must be recovered during decoding but puncturing allows to achieve higher code rates during transmission. Shortening improves the performance because the removed bits are certainly zeroes, therefore their value can be set to an high value by the decoder and it can be used to decode the remain part of the information.

To simulate the LDPC module, OAI uses a testbench which receives from the command line the parameters to set up the simulation environment and to feed the decoder. In the following table the command line arguments are reported with their corresponding default value:

Command	**Description**	**Default value**
-q	set number of quantization bits	8
-r	set code rate nominator	1
-d	set code rate denominator	1
-l	set block length	8448
-G	enable CUDA flag	0
-n	set number of simulation run	1
-s	set SNR per simulation bit	-2
-S	set number of segment blocks	1
-t	set SNR simulation step	0.1
-i	set maximum iteration value	5
-u	set SNR per coded bit	0

Table 3.2. Command line arguments for LDPC testbench

According to the command line arguments, the testbench selects the base graph and Z_C to work with. If the block length is greater than 3840 then BG1 is chosen, otherwise BG2 is selected but the number of message bits in the base graph can change for different values of the block length, in this case shortening is applied.

The testbench body is a big for loop where the loop variable is the SNR and it ranges from -2(dB) up to 18 (dB), the iteration step is decided with the -t command line option. Inside the loop the *ldpctest* function is called to simulate both encoder and decoder of the LPDC module according to the function arguments. In the following the arguments are listed:

1. maximum decoder iterations
2. code rate nominator
3. code rate denominator

4. SNR of the current loop iteration
5. quantization bits
6. block length
7. number of simulation runs
8. number of block segments (maximum supported is 16)
9. pointer to segmentation block error counter
10. pointer to bit error counter
11. pointer to uncoded bit error counter
12. number of CRC misses (not explored here)
13. pointer to a decoder structure for profiling
14. pointer to an encoder structure for profiling
15. pointer to a decoder structure for iteration statistics
16. flag to run the decoder on a GPU

The *ldpctest* function returns the number of blocks that have not been decoded correctly.
Once the *ldpctest* function is fed, three data are of our interest:

- *test_input* contains the message that must be coded and decoded. It has 16 rows (the maximum number of segments supported by the testbench) and *block_length*/8 columns. It is an unsigned char vector.
- *channel_output_fixed* is the codeword affected by noise. It is a char 2D vector, with 16 rows and the same number of columns of the parity check matrix.
- *estimated_output* is the same type of variable of test_input since it is the decoder output, i.e. the decoded message.

The message to be transmitted is a vector of $\frac{block\_length}{8}$ unsigned chars and it is generated randomly using the *rand()* C function by filling the vector test_input. Then the message is encoded using one encoder solution proposed by OAI chosen according to the parameters of a structure (*encoder_implemparams_t*).
The new code word must be sent through a channel, therefore a simulation model of the channel is implemented: the code word is firstly modulated using BPSK modulation on the entire bit sequence except for the first $2Z_C$ bits which are always punctured. After modulation, the code word is quantized on n bits according to

the parameter pass to the testbench with the -q option. The function performing the quantization is the one reported below[3]:

```
signed char quantize(double D, double x, unsigned char B) {
    double qxd;
    short maxlev;
    qxd = floor(x / D);
    maxlev = 1 << (B - 1);
    if (qxd <= -maxlev)
      qxd = -maxlev;
    else if (qxd >= maxlev)
    qxd = maxlev - 1;
    return ((char) qxd);
}
```

Listing 3.1. Function for channel input quantization used in the testbench of OAI

where B is the number of digits that are used for quantization and it is set to 8 by default when the function is called. Positive and negative saturation is applied to the result of the floor division. D is $\sigma = \frac{1}{2\sqrt{SNR}}$, with SNR equals to the current simulation step, this allows the user to test the decoder in different noise conditions, namely with an high noise (low SNR) or low noise with respect to the signal power (high SNR). In this way the quantization changes with the SNR value.
The variable x is the modulated codeword which is contaminated with Gaussian noise with an uniform distribution, the function used to generate the Gaussian noise is reported in appendix B. Finally, the generation of the data set for the decoder using fixed point representation of the channel sample is [4]:

```
channel_output_fixed[j][i] = (char)quantize(sigma/4.0/4.0,
    modulated_input[j][i] + sigma*gaussdouble(0.0,1.0),qbits);
```

Listing 3.2. Channel output sample with modulation and noise

After quantization the decoder can be tested, one can choose to run either the implementation for Intel processors supporting AVX2 or the solution for GPUs, the choice is taken based on the flag *run_cuda* set with the -G command line argument.

[3] https://gitlab.eurecom.fr/oai/openairinterface5g/-/blob/develop/openair1/SIMULATION/NR_PHY/nr_unitary_defs.h

[4] https://gitlab.eurecom.fr/oai/openairinterface5g/-/blob/develop/openair1/PHY/CODING/TESTBENCH/ldpctest.c

3.2 Decoder for Intel processors

3.2.1 OpenairInterface implementation of LDPC decoder for Intel processors

OAI provides a decoder exploiting AVX2 instruction set for Intel processor to improve the throughput. The input LLRs are signed char variables and are byte aligned, in fact the vector variable channel_output_fixed is allocated using malloc16(x)s and memalign(32,x) functions (specified by a #define directive related to the ___AVX2___ flag) which align the memory blocks on 32 Bytes (256 bits).
Once the channel output sample is generated and represented using fixed point notation, the decoder must be set up by filling the parameters of the following structures[5]:

```
1 typedef struct nrLDPC_dec_params {
2     uint8_t BG; /**< Base graph */
3     uint16_t Z; /**< Lifting size */
4     uint8_t R; /**< Decoding rate: Format 15,13,... for code rates
      1/5, 1/3,... */
5     uint8_t numMaxIter; /**< Maximum number of iterations */
6     e_nrLDPC_outMode outMode; /**< Output format */
7 } t_nrLDPC_dec_params;
8
9 typedef enum nrLDPC_outMode {
10     nrLDPC_outMode_BIT, /**< 32 bits per uint32_t output */
11     nrLDPC_outMode_BITINT8, /**< 1 bit per int8_t output */
12     nrLDPC_outMode_LLRINT8 /**< Single LLR value per int8_t output
      */
13 } e_nrLDPC_outMode;
```

Listing 3.3. Data structures containing decoder parameters from AVX2 code of OAI

The t_nrLDPC_dec_params structure contains the base graph number (1 or 2), the lifting size Z that is chosen at run time when the code rate R is established, the maximum number of iterations, which is set by the user to stop the decoder, and the output format. By default, the output format is set to 32 bits per 32 unsigned output value which corresponds to the nrLDPC_outMode_BIT setting.
The top level function of the decoder receives pointers to input and output buffers (channel_output_fixed and estimated_output) and it returns the number of iterations reached by the decoder which might be either the maximum one set by the user or a lower value if the decoder managed to decode successfully the code word. In the following the list of all the parameters given to the decoder are shown:

[5] https://gitlab.eurecom.fr/oai/openairinterface5g/-/blob/develop/openair1/PHY/CODING/nrLDPC_decoder/nrLDPC_types.h

1. t_nrLDPC_dec_params* p_decParams
2. int8_t* p_llr
3. int8_t* p_out
4. t_nrLDPC_procBuf* p_procBuf
5. t_nrLDPC_time_stats* p_profiler

where 1. is the pointer to the decoder parameters structure, 2. and 3. are the pointers to channel_output_fixed and estimated_output vectors, 4. is a pointer to a structure of int8_t pointers to buffers, finally 5. is a pointer to a structure of time_stats_t variables for measuring the execution time of each decoder functions. The code for the structures 4. and 5. is reported in appendix B.

Concerning the internal structure of the decoder, it has several functions as shown below:

Function	Operation
llr2llrProcBuf	moves input LLRs into the LLR process buffer
llr2CnProcBuf	moves the content of LLR process buffer into CN process buffer
cnProc	performs CNs processing to evaluate the $r_{j\to i}$ message
cnProcPc	performs parity check to stop earlier the decoding
cn2bnProcBuf	moves $r_{j\to i}$s in the CNs output buffer into BNs input buffer
bnProcPc	performs the processing for the computation of the $q_{i\to j}$
bnProc	computes the new LLRs using all the $q_{i\to j}$
bn2cnProcBuf	moves the content of BNs output buffer into CNs input buffer
llrRes2llrOut	moves the content of BNs output buffer into LLRs output buffer
llr2bitPacked	hard decision on the final LLRs

Table 3.3. Functions of LDPC decoder in the code for Intel processors supporting AVX2

by noting the table 3.3, there are dedicated functions to move back and forth the intermediate results from one process to another one, the reason for this implementation is related to the AVX2 instructions which work with aligned and properly ordered data. Thus, from one side this solution spends some time in copying operations, on the other hand the AVX2 library provides fast SIMD instructions to speed up the computation. The speed up factor is due to the capability of those instructions to work with 32 elements at once.
As reference, OAI provides the execution time of the decoder functions obtained

by simulating the decoder module[6]. The measurements are obtained with OAI time_meas.h library, the simulation is run on a 2.9GHz processor and the values are averaged over 10 000 message blocks. From the available measurements the cn2bnProcBuf function is the slowest regarding the data movement, in fact it takes $38.7\mu s$ to copy the data into the bit node buffer, meanwhile the slowest computational function (cnProc) takes $89.8\mu s$ to complete its task. Thus the bottlenecks of this implementation are certainly the buffer management and the check node processing. For sake of completeness the duration time of each function shown in table 3.3 from OAI documentation is reported in appendix A.
Finally, to better understand the organization of the decoder algorithm in OAI for Intel processors, the following chart can be considered:

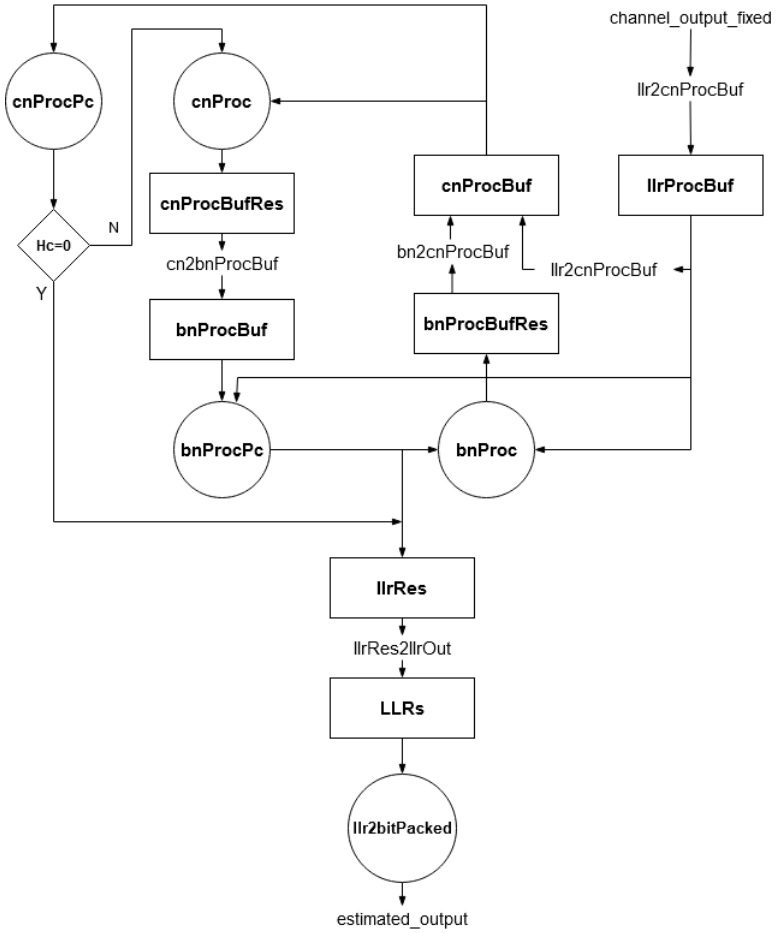

Figure 3.1. Decoder flow in the AVX2 implementation

[6]https://gitlab.eurecom.fr/oai/openairinterface5g/-/blob/develop/openair1/PHY/CODING/nrLDPC_decoder/doc/nrLDPC/nrLDPC.pdf

First of all, the LLRs coming from the channel are stored in the llrProcBuf buffer since they will be used by the bnProcPc and bnProc functions to compute the new LLR believes. Then the content of llrProcBuf buffer is copied into the cnProcBuf buffer using the llr2cnProcBuf function, this procedure occurs ONLY once in the first iteration because there is no previous estimation of the believes from the decoder. Then cnProc function is called to compute the believes of the check nodes using the data inside the cnProcBuf buffer, the result is stored into the cnProcBufRes buffer.
Later the first part of the bit node processing occurs but the content of cnProcBufRes must be copied into the bit node buffer bnProcBuf. At this point if the iteration counter corresponds to the maximum value, then the decoder can stop and the results are stored into llrRes buffer. Otherwise the results (which are the sum of all CNs believes and channel samples) are sent to the bnProc function which computes the extrinsic information to send to the check node processing. In other words the results of the bnProc function are cleaned by the message coming from the destination node.
The messages of the BNs must be sent to the check nodes, hence they are copied from bnProcBufRes to the cnProcBuf buffer using the bn2cnProcBuf function to rearrange the data. Before proceeding with the cnProc function, the new values stored in cnProcBuf buffer are used to verify the parity check equations, if only one parity check fails then this function is aborted and the processing can continue with the cnProc function, in that case a new iteration starts. Instead, if the cnProcPc functions returns with no parity check failure then the decoder has found the original message and the decoder loop can stop earlier.
When one of the two stopping conditions, namely the maximum iterations or parity checks success, are met the final LLRs are stored into llrRes buffer. From llrRes buffer the LLRs are moved into the output vector and then they are converted back to binary values using hard decision.

To summarize, the top level function of the decoder performs the following function calls for the first iteration only:
1a. llr2CnProcBuf_BG1 or llr2CnProcBuf_BG2 depending on the base graph used, 2a. cnProcBuf_BG1 or cnProcBuf_BG2, 3a. cn2bnProcBuf_BG1 or cn2bnProcBuf_BG2, 4a. bnProcPc, 5a. bnProc, 6a. bn2cnProcBuf_BG1 or bn2cnProcBuf_BG2.
For successive iterations the order is the following:
1b. cnProc_BG1 or cnProc_BG2, 2b. cn2bnProcBuf_BG1 or cn2bnProcBuf_BG2, 3b. bnProcPc, 4b. bnProc, 5b. bn2cnProcBuf_BG1 or bn2cnProcBuf_BG2, 6b. cnProcPc_BG1 or cnProcPc_BG2.
Then, if the parity check fails the decoder rolls back to the 1b function for a new iteration, otherwise it exits from the decoding loop and invokes llrRes2llrOut and llr2bitPacked functions to assert the message on the output buffer. To be noted that if the base graph changes then the data ordering inside the buffer is different,

therefore some functions are duplicated to support different arrangement.

3.2.2 Check node processing

As said previously, LDPC codes in 5G are irregular, thus check nodes have different degrees, or weights, as well as bit nodes have a different amount of connected check nodes. For this reason both check nodes and bit nodes are organized in groups. A bit node group is made of bit nodes with the same amount of check nodes connected. Vice versa for check nodes. The number of groups and nodes per group is reported in appendix A, tables A.2.
Since the nodes are organized in groups, the software implementation is almost straightforward. Both cnProc and cnProcPc execute sequentially the processing group by group. They start from the smallest group and by using an outer loop they select the i-th bit node to elaborate. An inner loop is used to process each check node of the group.

By denoting as I_j the ensemble of bit nodes connected to the j-th check node, the *cnProc* function performs the computation reported in equation 2.18:

$$r_{j\to i} = \left(\prod_{i' \in I_j \setminus i} sign(q_{i'\to j}) \right) \cdot \min_{i' \in I_j \setminus i} (|q_{i'\to j}|) \tag{3.4}$$

this operation is repeated for each bit node of the group of the j-th check node, in the following a snippet of the code is reported[7]:

```
1 const uint16_t lut_idxCnProcG4[4][3] = {{240,480,720}, {0,480,720}, {0,240,720},
      {0,240,480}};
2 if (lut_numCnInCnGroups[1] > 0)
3 {
4     // Number of groups of 32 CNs for parallel processing
5     // Ceil for values not divisible by 32
6     M = (lut_numCnInCnGroups[1]*Z + 31)>>5;
7     bitOffsetInGroup = (lut_numCnInCnGroups_BG2_R15[1]*NR_LDPC_ZMAX)>>5;
8     // Set pointers to start of group 4
9     p_cnProcBuf    = (__m256i*) &cnProcBuf    [lut_startAddrCnGroups[1]];
10    p_cnProcBufRes = (__m256i*) &cnProcBufRes[lut_startAddrCnGroups[1]];
11    // Loop over every BN
12    for (j=0; j<4; j++)
13    {
14        // Set of results pointer to correct BN address
15        p_cnProcBufResBit = p_cnProcBufRes + (j*bitOffsetInGroup);
16        // Loop over CNs
17        for (i=0; i<M; i++)
18        {
19            // Abs and sign of 32 CNs (first BN)
20            ymm0 = p_cnProcBuf[lut_idxCnProcG4[j][0] + i];
21            sgn  = _mm256_sign_epi8(*p_ones, ymm0);
22            min  = _mm256_abs_epi8(ymm0);
23            // Loop over BNs
24            for (k=1; k<3; k++)
25            {
26                ymm0 = p_cnProcBuf[lut_idxCnProcG4[j][k] + i];
27                min  = _mm256_min_epu8(min, _mm256_abs_epi8(ymm0));
28                sgn  = _mm256_sign_epi8(sgn, ymm0);
```

[7] https://gitlab.eurecom.fr/oai/openairinterface5g/-/blob/develop/openair1/PHY/CODING/nrLDPC_decoder/nrLDPC_cnProc.h

```
29              }
30              min = _mm256_min_epu8(min, *p_maxLLR);
31              *p_cnProcBufResBit = _mm256_sign_epi8(min, sgn);
32              p_cnProcBufResBit++;
33          }
34      }
35 }
```

Listing 3.4. Check node processing of CPU decoder for BG2

where the check node group involved is the fourth one which has 5 check nodes with 4 bit nodes connected to each of them for base graph 1, otherwise the CNs amount is 20 for the base graph 2. A variable containing the address of the bit nodes in the cnProcBuf buffer is used for each group, its name changes from group to group and it is of the type $lut\_idxCnProcG <>$, such variable is a 2D vector with (I_j-1) columns and one row for each check node of the group. Each row contains the address of all the bit nodes involved except the destination one.

In the code section, there are three loops: the outer loop ranges over the number of bit nodes of the group because each of them must receive a message from its CNs. In addition, the loop updates the address for the result buffer using an offset (bitOffsetInGroup) and the loop variable j.
The second loop iterates over the check nodes of the group which are taken as a block of 32 nodes in order to exploit the byte alignment of AVX2 instructions. In appendix C the list of all AVX2 instructions used by this solution are reported.
The inner most loop selects one bit of the I ensemble and performs the sign multiplication and finds the minimum of the LLRs. Thus the first bit node is always elaborated outside of this loop in order to initialize the minimum variable (*min* in listing 3.4) and the sign (*sgn*), *ymm0* is the variable related to value holded by 32 bit nodes and it is a ___m256i vector type.

After the completion of this group, the code moves to another group by updating the look up table variable and by executing a new code section with same structure discussed before, when all the groups have been processed the decoder top function moves the data from the CNs output buffer to the BNs buffer.

The remain decoder function involving check nodes is the parity check function, namely cnProcPc_BG1 and cnProcPc_BG2, which is an optional procedure enabled by the define directive #define NR_LDPC_ENABLE_PARITY_CHECK. Since the parity check function can abort as soon as a failure is detected it does not degrade the overall performance. Moreover the cnProcPc function can stop earlier the decoder once the parity check is met.
The code organization is the same of the cnProc function. The code begins from the first group of check nodes. First there is the loop iteration over all the CNs of the group, then there is the inner loop over the bit nodes of the group. Finally, if for the current group the parity check fails, the functions returns immediately with the number of parity check fails, otherwise the check continues till the last group. In the following the code fragment for the fourth group of base graph two is shown[7]:

```
1 // Process group with 4 BNs
2 if (lut_numCnInCnGroups[1] > 0)
3 {    // Reset results
4     pcResSum = 0;
5     M = lut_numCnInCnGroups[1]*Z;
6     // Remainder modulo 32
7     Mrem = M&31;
8     // Number of groups of 32 CNs for parallel processing
9     // Ceil for values not divisible by 32
10    M32 = (M + 31)>>5;
11    // Set pointers to start of group 4
12    p_cnProcBuf    = (__m256i*) &cnProcBuf   [lut_startAddrCnGroups[1]];
13    p_cnProcBufRes = (__m256i*) &cnProcBufRes[lut_startAddrCnGroups[1]];
14    // Loop over CNs
15    for (i=0; i<(M32-1); i++)
16    {
17        pcRes = 0;
18        // Loop over every BN
19        // Compute PC for 32 CNs at once
20        for (j=0; j<4; j++)
21        {
22            // BN offset is units of 20*384/32 = 240
23            ymm0 = p_cnProcBuf   [j*240 + i];
24            ymm1 = p_cnProcBufRes[j*240 + i];
25            // Add BN and input LLR, extract the sign bit
26            // and add in GF(2) (xor)
27            pcRes ^= _mm256_movemask_epi8(_mm256_adds_epi8(ymm0,ymm1));
28        }
29        // If no error pcRes should be 0
30        pcResSum |= pcRes;
31    }
32    // Last 32 CNs might not be full valid 32 depending on Z
33    pcRes = 0;
34    // Loop over every BN
35    // Compute PC for 32 CNs at once
36    for (j=0; j<4; j++)
37    {
38        // BN offset is units of 20*384/32 = 240
39        ymm0 = p_cnProcBuf   [j*240 + i];
40        ymm1 = p_cnProcBufRes[j*240 + i];
41        // Add BN and input LLR, extract the sign bit
42        // and add in GF(2) (xor)
43        pcRes ^= _mm256_movemask_epi8(_mm256_adds_epi8(ymm0,ymm1));
44    }
45    // If no error pcRes should be 0
46    // Only use valid CNs
47    pcResSum |= (pcRes&(0xFFFFFFFF>>(32-Mrem)));
48    // If PC failed we can stop here
49    if (pcResSum > 0)
50    {
51        return pcResSum;
52    }
53 }
```

Listing 3.5. Parity check for CPU decoder for BG2

For the current group the number of check nodes is derived from a look up table and stored into a local variable. Then the pointers to input and output buffers for check node processing are updated and widen to take 256 bits using pointer casting. The outmost loop ranges over 32 check nodes at once. The pcRes variable is used to store the intermediate parity check of 32 check nodes and it is initialized to zero when a new iteration starts. If the number of check nodes to elaborate concurrently is not a multiple of 32, the remainder of modulo 32 division is used to verify the parity check of last CNs. If the parity check is met, the pcRes variable is zero (which means that the equation 2.6 holds) and the processing will continue to next check node group, otherwise the function will return.

3.2.3 Bit node processing

The code structure is similar to the check node processing but the variable used during computation is larger to improve the accuracy.
The function _m256_cvtepi8_epi16 is exploited to increase the size of a ___m256i element from 8 bits to 16. After conversion the ___m256i vector will contain 16 integer of 16 bits instead of 32, therefore two ___m256i variables are used to elaborate 32 LLRs in one loop iteration. The variables are ymmRes0 and ymmRes1 in the code reported below.
As for the check nodes, also bit nodes are divided in groups, each one is made of bit nodes with the same amount of check nodes connected. The difference with respect to check node processing is that the number of groups is larger but the number of bit nodes per group is much smaller, except for the first group which has $42{\cdot}Z_C$ (BG1) or $38{\cdot}Z_C$ (BG2) bith nodes with only one check node.
Regarding the first bit node function, bnProcPc, a piece of its code is shown[8]:

```
1 if (lut_numBnInBnGroups[2] > 0)
2     {
3         // If elements in group move to next address
4         idxBnGroup++;
5         M = (lut_numBnInBnGroups[2]*Z + 31)>>5;
6         // Set the offset to each CN within a group in terms of 16 Byte
7         cnOffsetInGroup = (lut_numBnInBnGroups[2]*NR_LDPC_ZMAX)>>4;
8         // Set pointers to start of group 3
9         p_bnProcBuf  = (__m128i*) &bnProcBuf  [lut_startAddrBnGroups    [idxBnGroup]];
10        p_llrProcBuf = (__m128i*) &llrProcBuf [lut_startAddrBnGroupsLlr[idxBnGroup]];
11        p_llrRes     = (__m256i*) &llrRes     [lut_startAddrBnGroupsLlr[idxBnGroup]];
12        // Loop over BNs
13        for (i=0,j=0; i<M; i++,j+=2)
14        {
15            // First 16 LLRs of first CN
16            ymmRes0 = _mm256_cvtepi8_epi16(p_bnProcBuf[j]);
17            ymmRes1 = _mm256_cvtepi8_epi16(p_bnProcBuf[j+1]);
18            // Loop over CNs
19            for (k=1; k<3; k++)
20            {
21                ymm0 = _mm256_cvtepi8_epi16(p_bnProcBuf[k*cnOffsetInGroup + j]);
22                ymmRes0 = _mm256_adds_epi16(ymmRes0, ymm0);
23
24                ymm1 = _mm256_cvtepi8_epi16(p_bnProcBuf[k*cnOffsetInGroup + j+1]);
25                ymmRes1 = _mm256_adds_epi16(ymmRes1, ymm1);
26            }
27            // Add LLR from receiver input
28            ymm0    = _mm256_cvtepi8_epi16(p_llrProcBuf[j]);
29            ymmRes0 = _mm256_adds_epi16(ymmRes0, ymm0);
30            ymm1    = _mm256_cvtepi8_epi16(p_llrProcBuf[j+1]);
31            ymmRes1 = _mm256_adds_epi16(ymmRes1, ymm1);
32            ymm0 = _mm256_packs_epi16(ymmRes0, ymmRes1);
33            // ymm0     = [ymmRes1[255:128] ymmRes0[255:128] ymmRes1[127:0] ymmRes0[127:0]]
34            // p_llrRes = [ymmRes1[255:128] ymmRes1[127:0] ymmRes0[255:128] ymmRes0[127:0]]
35            *p_llrRes = _mm256_permute4x64_epi64(ymm0, 0xD8);
36            // Next result
37            p_llrRes++;
38        }
39    }
```

Listing 3.6. LLR estimation for CPU decoder

Also for the bit node processing the code is executed sequentially, the computation starts from the first group and it proceeds forward till the last group of nodes.

[8] https://gitlab.eurecom.fr/oai/openairinterface5g/-/blob/develop/openair1/PHY/CODING/nrLDPC_decoder/nrLDPC_bnProc.h

Firstly each group has fewer bit nodes, in the range of 1 to 4, hence the single group processing is much shorter if compared to the CNs one where a group could have even $18 \cdot Z_C$ check nodes.
Concerning the operation performed it is the one reported in equation 2.17. For each group there is an outer loop which select the 32 bit nodes to work with, whose total number is read from a look up table. An inner loop accumulates one LLR (for each bit node) per loop iteration. To be noted that from lines 21 to 25 of the code in 3.6 two adjacent accesses to the buffer bnProcBuf are performed, this is due to the widening of the variables. In fact the buffers bnProcBuf and llrProcBuf are accessed with a ___m128i pointer in order to convert 16 LLRs into a ___m256i vector of 16 bits per element.
As said before, for the bit node processing the variables are extended to 16 bits instead of 8 like in the check node processing. Thus, if 32 LLRs have to be processed in a single iteration of the innermost loop, two ___mm256i variables must be used with 16x16 bits each (ymm0 and ymm1 in the code). Then, the LLRs coming from the check nodes are added with saturation (___mm256_adds_epi16 AVX2 function). Once the innermost loop has reached the loop bound, the 512 LLR bits are converted back to 8 bits and stored in ymm0 (line 32 in listing 3.6).
Finally, the 256 bits are permuted in order to have the correct arrangement of the bits in the buffer. After the permutation the out most loop counter is updated. Basically the operation performed in this function is the following:

$$q_{i \to j} = y_i + \sum_{j \in J_i} r_{j' \to i} \tag{3.5}$$

The code reported above computes the final LLRs of one decoding iteration since it uses both the channel sample and the information coming from all the check nodes of a bit node. The second part of the bit node processing is related to the message passing equation 2.16 which can be written as:

$$q_{i \to j} = y_i + \sum_{j' \in J_i \setminus j} r_{j' \to i} \tag{3.6}$$

where J is the group of check nodes connected to the i-th bit node. Therefore, in the function bnProc the results obtained in bnProcPc routine are used and the message $q_{i \to j}$ is computed by removing the $r_{j \to i}$ part of the destination node j[8]:

```
1 if (lut_numBnInBnGroups[2] > 0)
2 {
3     // If elements in group move to next address
4     idxBnGroup++;
5     // Number of groups of 32 BNs for parallel processing
6     M = (lut_numBnInBnGroups[2]*Z + 31)>>5;
7     // Set the offset to each CN within a group in terms of 32 Byte
8     cnOffsetInGroup = (lut_numBnInBnGroups[2]*NR_LDPC_ZMAX)>>5;
9     // Set pointers to start of group 3
10     p_bnProcBuf    = (__m256i*) &bnProcBuf[lut_startAddrBnGroups[idxBnGroup]];
11    p_bnProcBufRes = (__m256i*) &bnProcBufRes[lut_startAddrBnGroups[idxBnGroup]];
12    // Loop over CNs
13    for (k=0; k<3; k++)
```

```
14      {
15          p_res = &p_bnProcBufRes[k*cnOffsetInGroup];
16          p_llrRes = (__m256i*) &llrRes[lut_startAddrBnGroupsLlr[idxBnGroup]];
17          // Loop over BNs
18          for (i=0; i<M; i++)
19          {
20              *p_res = _mm256_subs_epi8(*p_llrRes, p_bnProcBuf[k*cnOffsetInGroup + i]);
21              p_res++;
22              p_llrRes++;
23          }
24      }
25 }
```

Listing 3.7. Bit node processing for CPU decoder

In the code listing 3.7 the second part of the bit node processing is reported and it is related to the group with 3 check nodes per bit node. The order of the loops is reversed with respect to the other function, in fact the outer loop selects the destination check node, whilst the inner loop works on 32 bit nodes in one iteration. It is important to point out that for this function the computation is carried on 8 bits instead of 16. In each iteration of a loop the subtraction of the corresponding check node estimation is removed in order to transmit only the extrinsic information.

3.2.4 From LLR to bit

The conversion from LLRs to bit occurs when the decoder meets one of the two stopping condition. The function presented here[8] is llr2bitPacked because it is the one chosen when the nrLDPC_outMode structure in listing 3.3 is initialized with the value nrLPC_outMode_BIT. The function for hard decision receives the number of LLRs to convert (numLLR) and it uses a loop to convert 32 LLRs per iteration.

```
1 const uint8_t constShuffle_256_epi8[32] __attribute__ ((aligned(32))) =
      {7,6,5,4,3,2,1,0,15,14,13,12,11,10,9,8,
2      7,6,5,4,3,2,1,0,15,14,13,12,11,10,9,8};
3                 [...]
4 for (i=0; i<M; i++)
5 {
6     // Move LSB to MSB on 8 bits
7     inPerm = _mm256_shuffle_epi8(*p_llrOut,*p_shuffle);
8     // Hard decision
9     *p_bits++ = _mm256_movemask_epi8(inPerm);
10    p_llrOut++;
11 }
12 if (Mr > 0)
13 {
14    // Remaining LLRs that do not fit in multiples of 32 bytes
15    p_llrOut8 = (int8_t*) p_llrOut;
16    for (i=0; i<Mr; i++)
17    {
18        if (p_llrOut8[i] < 0)
19        {
20            bitsTmp |= (1<<((7-i) + (16*(i/8))));
21        }
22        else
23        {
24            bitsTmp |= (0<<((7-i) + (16*(i/8))));
25        }
26    }
27 }
```

Listing 3.8. Hard decision on LLRs for CPU decoder

The AVX2 instruction involved in the conversion is ___mm256_movemask_epi8 which creates a mask according to the inPerm variable whose value is set after the shuffling of the input LLRs. The shuffling is specified by 3GPP technical specification in [23] where the leftmost bit is the most significant bit and the right one is the least significant. The bit string must be read from left to right. If the number of LLRs is not a multiple of 32 then an additional loop is executed in order to convert the remaining LLRs. To convert one LLR to its binary value the sign is considered. If it is negative then the bit is 1 otherwise it is 0.

3.2.5 Buffer transfer

The data transfer from one buffer to another is extremely important for this decoder solution because, as it has been explained with the previous listings, the AVX2 instructions require to have the data aligned. The data transfer takes places when the data processing changes, namely when moving from check node to bit node processing (and viceversa) or when moving to the hard decision function. In these situation the variables to be elaborated are different (LLRs for bit nodes or estimations for check nodes for example), thus they are ordered in a different manner in the buffers to support AVX2 instructions properly.
To transfer data from buffer to buffer the memcpy function is exploited, moreover the copying can be circular rather than inverse circular. The code snippet is presented below[9]:

```
1 static inline void *nrLDPC_inv_circ_memcpy(int8_t *str1, const int8_t *str2, uint16_t Z,
      uint16_t cshift)
2 {
3     uint16_t rem = Z - cshift;
4     memcpy(str1+cshift, str2    , rem);
5     memcpy(str1       , str2+rem, cshift);
6
7     return(str1);
8 }
9 static inline void *nrLDPC_circ_memcpy(int8_t *str1, const int8_t *str2, uint16_t Z,
      uint16_t cshift)
10 {
11     uint16_t rem = Z - cshift;
12     memcpy(str1     , str2+cshift, rem);
13     memcpy(str1+rem , str2       , cshift);
14
15     return(str1);
16 }
```

Listing 3.9. Circular memory copy functions

In the listing, str1 is the destination buffer, cshift is the circular coefficient, Z instead is the size of data to be copied from buffer str2 and it corresponds to the lifting size. By calling the circular amount as chunk, it is equal to $Z - cshift$. The circular copy consists in replicating first the chunk of str2 and then the remainder. Instead, the inverse operation copies the remainder and then the circular amount.

[9] https://gitlab.eurecom.fr/oai/openairinterface5g/-/blob/develop/openair1/PHY/CODING/nrLDPC_decoder/nrLDPC_mPass.h

The circular coefficient cshift is determined by a set of look up tables and it depends on several parameters: code rate, kind of operation (check node or bit node processing) and base graph.
As a reference a portion of the llr2CnProcBuf_BG1 function code is reported[9] to illustrate how the data transfer is performed in these function:

```
1 for (j=0; j<3; j++)
2 {
3     p_cnProcBuf = &cnProcBuf[lut_startAddrCnGroups[0] + j*bitOffsetInGroup];
4     idxBn = lut_posBnInCnProcBuf_CNG3[j][0]*Z;
5     nrLDPC_circ_memcpy(p_cnProcBuf, &llr[idxBn], Z, lut_circShift_CNG3[j][0]);
6 }
```

Listing 3.10. Circular LLRs memory copy example

This code copies the LLRs from the input buffer of the decoder into the input buffer of the check node processing. A look up table (lut_startAddrCnGroups) is used to point to the correct location of the check node buffer as already explored for the processing functions, another table tells the source address of the LLR. To be more precise, the copy operation must arrange the LLRs in a specific manner for the group 3 of the base graph 1, the order is specified by the lut_circShift_CNG3 look up table. Since there are three bit nodes per check node in this group, the loop bound is set to three.
The same operation is performed for other data transfer functions.

3.3 Synthesizable AVX2 instructions

AVX2 instructions are custom code by Intel company which is not written in C language and therefore it is not synthesizable by definition. To have a synthesizable code and to feed properly Vivado HLS the AVX2 solution of LDPC decoder code must be first extracted from the repository in order to work as a stand alone module.

Once the LDPC decoder has been separated from the repository, some #ifdef directives are added to the original testbench in order to produce the input and output data sets. The test vectors are extremely important during the design phase since they allow the designer to detect when even a small change in the code can affect the functionality of the application.
The testbench with the #ifdef statements appears as depicted in the following picture:

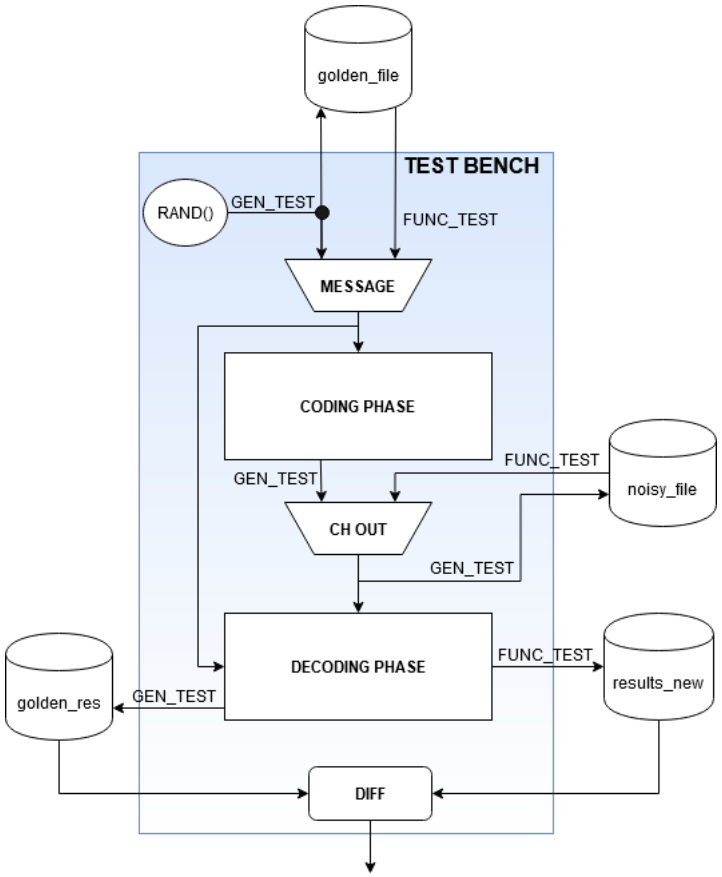

Figure 3.2. Diagram of the testbench that generates the reference data set and the simulation output vector for encoder (golden_file) and decoder (noisy_file, golden_res and results_new).

There are two additional defines in the testbench: GEN_TEST and FUNC_TEST. The first one makes the testbench to work in the normal way, so the test_input vector is initialized using the rand() function and then the codeword and decoded message are generated as described in the previous section. The feature provided by this define is that the content of test_input, channel_output_fixed and estimated_output is stored in three different files, which are golden_file, noisy_file and golden_res respectively. With these files one can test the code changes. An output file for the encoder is not implemented because the encoder testing was beyond the scope of this thesis.
On the other hand, the second define directive (FUNC_TEST) sets the testbench to read the content of the vectors from those files. The read values are used to run the simulation with known inputs. The content of noisy_file is used to feed the decoder, the content of golden_file instead is used to evaluate the BER, whilst golden_res is for comparison purpose. At the end of the simulation the decoded

message obtained with the files as input is compared with the golden_res content. To avoid additional lines of code the comparison is performed using the bash command *diff* with the -y and -s option. The -y asserts the content of the two files in two different columns. The -s option prints whether the files are identical or not. Moreover, the purpose of the golden_file for the test_input vector is to feed the testbench for C simulation and C co-simulation in Vivado HLS.
When both defines are not set, the testbench runs a simulation without using any external files to write or to read data. If both are set the priority is given to GEN_TEST define.
A brief comment regarding the generation of the values of *channel_output_fixed* is required otherwise the functional verification might fail.

```
1                           [...]
2
3 for (i = 2*Zc; i < (Kb+nrows-no_punctured_columns) * Zc-removed_bit; i++) {
4   #ifdef GEN_TEST
5   channel_output_fixed[j][i] = (char)quantize(sigma/4.0/4.0,modulated_input[j][i] + sigma*
      gaussdouble(0.0,1.0),qbits);
6
7                           [...]
8 }
```

Listing 3.11. Generation of decoder input

The code reported above shows how the input for the decoder is generated using the GEN_TEST directive. For the j-th segment of the transport block, the i columns are filled with the channel output. It is extremely important to point out that the first $2{\cdot}Z_C$ columns are punctured and thus they are considered as erasures by the decoder, moreover shortening can be applied (removed_bit in the listing). From the code point of view the first $2{\cdot}Z_C$ columns are not initialized and thus they assume a random value. If one simulation is run with the GEN_TEST and another one is run with the FUNC_TEST directive the decoder output might be different in the two scenarios. That situation occurs when the decoder does not recover the message (BER is not zero). If the BER is zero it means that for any values of the $2{\cdot}Z_C$ punctured columns the decoder is capable of recovering entirely the message. Instead, if the BER is not zero, the output is different, even for few bits, for each combination of the first $2{\cdot}Z_C$ columns. If this is not considered, the command *diff −s −y golden_res.txt results_new.txt* will assert that the two files are not identical.
To avoid this reasonable but not useful scenario, the code reported above can be used untouched but an additional loop in the testbench is used to store all the values, even the ones that are not transmitted:

```
1                           [...]
2#ifdef GEN_TEST
3 for(i=0; i < col*Zc; i++){
4   fprintf(noisy_file, "%hhd ",  channel_output_fixed[j][i]);
5 }
6 fprintf(noisy_file,"\n");
7#elif FUNC_TEST   // if functional verification read input from noisy_input.txt
8 for (i = 0; i < col*Zc; i++) {
9 if(fscanf(noisy_file, "%hhd", &channel_output_fixed[j][i])!=1)
```

```
10   exit(1);
11 }
12#endif
13
14                              [...]
15 }
```

Listing 3.12. Additional loops added to read an write the whole content of channel_output_fixed vector in order to not have random values for the 2·Zc punctured elements

The number of elements to be stored in channel_output_fixed corresponds to the codeword length, hence it is given by Z_C·col. The value of col depends on the chosen base graph and can be 68 or 52 if the base graph is 1 or 2 respectively. The output format to read/write from/to noisy_file is hhd, which is the format specifier for a signed char.
For sake of clarity, also the code sections related to test_input and estimated_output are reported:

```
1 for (j=0;j<MAX_NUM_DLSCH_SEGMENTS;j++) {
2     for (i=0; i<block_length/8; i++) {
3   #ifdef GEN_TEST
4    test_input[j][i]=(unsigned char) rand();  // generate input file
5    fprintf(golden_file, "%hhu ",  test_input[j][i]);
6   #elif FUNC_TEST
7    if(fscanf(golden_file, "%hhu",  &test_input[j][i])!=1)
8        exit(1);     // read inputs from file
9   #else
10   test_input[j][i]=(unsigned char) rand();  // normal operation
11  #endif
12    }
13  }
14 }
```

Listing 3.13. Generation of transport block for reference and verification purpose

```
1 for(j=0;j<n_segments;j++) {
2  #ifdef GEN_TEST
3     for(i=0; i<block_length/8; i++)
4     {
5     fprintf(golden_res, "%hhu ",  estimated_output[j][i]);
6     }
7    fprintf(golden_res, "\n");
8  #elif FUNC_TEST
9     for(i=0; i<block_length/8; i++)
10    {
11      fprintf(results_file,"%hhu ", estimated_output[j][i]);
12    }
13   fprintf(results_file, "\n");
14  #endif
15 }
```

Listing 3.14. Generation of decoder output for reference and verification purpose

From the code snippets shown above, hhu is the format specifier for unsigned char, block_length is the length of a segment of the transport block. Since the vectors are char type, block_length/8 elements must be randomly generated.

In order to have an automatic design and verification process a make file is created. The makefile has four rules:

1. *clean* to remove the executable and all .txt files (if exist) generated from previous compilation,

2. *generate* to set the GEN_TEST define and generate the input data set used for functional verification,

3. *compare* to set the FUNC_TEST define and run the functional verification. This rule also execute the diff -s -y golden_res.txt results_new.txt command to verify the functionality of the decoder.

4. *all* to compile only the code, regardless of the defines.

The makefile can also provide the testbench parameters by setting properly the LDPC variable. For instance, with *make generate LDPC="-r 1 -d 3 -t 5"* the testbench runs a simulation with a code rate equal to $\frac{1}{3}$ and an increment of the SNR of 5dB for the simulation step. The testbench is set to generate the data set.

Once the LDPC module has been separated and the testbench environment is modified the decoder code must be changed in order to be synthesizable. There are two steps to have the code synthesizable: the first one is to change all structures and vectors whose dimension is greater than 2 because Vivado HLS does not support it. The second one is to create a custom version of AVX2 functions written in C language whose parameters are passed by value.

3.3.1 Structures and vectors dimension reduction

To have a synthesizable C code all vectors whose dimension is greater than 2 must be reshaped, this is valid also for data structures where the access to the internal variables is done using a pointer. For example, the nrLDPC_types.h file contains the declaration of several structures, one of those is the look up table structure t_nrLDPC_lut[5]:

```
1 typedef struct nrLDPC_lut {
2     const uint32_t* startAddrCnGroups; /**< Start addresses for CN groups in CN processing
      buffer */
3     const uint8_t*  numCnInCnGroups; /**< Number of CNs in every CN group */
4     const uint8_t*  numBnInBnGroups; /**< Number of CNs in every BN group */
5     const uint32_t* startAddrBnGroups; /**< Start addresses for BN groups in BN processing
      buffer  */
6     const uint16_t* startAddrBnGroupsLlr; /**< Start addresses for BN groups in LLR
      processing buffer  */
7     const uint16_t** circShift[NR_LDPC_NUM_CN_GROUPS_BG1]; /**< LUT for circular shift
      values for all CN groups and Zs */
8     const uint32_t** startAddrBnProcBuf[NR_LDPC_NUM_CN_GROUPS_BG1]; /**< LUT of start
      addresses of CN groups in BN proc buffer */
9     const uint8_t**  bnPosBnProcBuf[NR_LDPC_NUM_CN_GROUPS_BG1]; /**< LUT of BN positions in
      BG for CN groups */
10    const uint16_t* llr2llrProcBufAddr; /**< LUT for transferring input LLRs to LLR
      processing buffer */
11    const uint8_t*  llr2llrProcBufBnPos; /**< LUT BN position in BG */
12    const uint8_t** posBnInCnProcBuf[NR_LDPC_NUM_CN_GROUPS_BG1]; /**< LUT for llr2cnProcBuf
      */
13 } t_nrLDPC_lut;
```

Listing 3.15. Look up table structures containing the pointers for each buffer used in the CPU code

The LUT structure contains information regarding the number of check nodes and bit nodes in a group and other variables used for data transfer and nodes processing. For instance when the code uses the *circShift* variable three pointers are used, namely one to access the structure element and two to point to the element of

interest. These operations are not accepted by HLS, thus the code is not synthesizable. To guarantee the synthesizability of a code like the one reported above the following change must be applied:

```
1 static     const uint32_t* startAddrCnGroups;
2 static     const uint8_t*  numCnInCnGroups;
3 static     const uint8_t*  numBnInBnGroups;
4 static     const uint32_t* startAddrBnGroups;
5 static     const uint16_t* startAddrBnGroupsLlr;
6 static     const uint16_t** circShift[NR_LDPC_NUM_CN_GROUPS_BG1];
7 static     const uint32_t** startAddrBnProcBuf[NR_LDPC_NUM_CN_GROUPS_BG1];
8 static     const uint8_t**  bnPosBnProcBuf[NR_LDPC_NUM_CN_GROUPS_BG1];
9 static     const uint16_t* llr2llrProcBufAddr;
10 static    const uint8_t*  llr2llrProcBufBnPos;
11 static    const uint8_t** posBnInCnProcBuf[NR_LDPC_NUM_CN_GROUPS_BG1];
```

Listing 3.16. Synthesizable version of the look up table structure

The structure declaration has been removed and each variable is a static one to ensure that the value of each variable is saved from one file scope to one another. In this way there are no more triple pointers in the code.

3.3.2 AVX2 in C language

AVX2 instructions are not standard C therefore they must be converted into C code.
When printing the content of a ___m256i vector it is shown as a long int vector with four entries. Thus to emulate the organization of the elements stored in the vector data type by Intel a new data type is created:

```
typedef struct mm256i {long int data[4] __attribute__ ((aligned(32))) ;} m256i;
typedef struct mm128i {long int data[2] __attribute__ ((aligned(16)))  ;} m128i;
```

Listing 3.17. New AVX2 data structures

Also a replica of the ___m128i vector is created. The attribute ___attribute___ ((aligned())) is necessary to have the same alignment of the Intel variable.

```
1 m256i mm256_adds_epi8(m256i a, m256i b){
2   int8_t* p_a = (int8_t *) a.data;
3   int8_t* p_b = (int8_t *) b.data;
4   m256i dest={{0,0,0,0}};
5   int8_t* p_dest = (int8_t *) dest.data;
6   int16_t adds; // to handle overflow
7   int i;
8   for(i=0; i<32; i++){
9     adds = *p_a + *p_b;
10    if(adds < -128)
11      *p_dest = -128;
12    else if(adds > 127)
13      *p_dest = 127;
14    else
15      *p_dest = (int8_t) adds;
16    p_a++;
17    p_b++;
18    p_dest++;
19  }
20  return dest;
21  }
```

Listing 3.18. Synthesizable AVX2 function performing addition of bytes with saturation

The synthesizable version of AVX2 instruction has been developed both with parameters passing by value and by reference. Above the function implementing the addition with saturation is reported. The other functions used by the decoder are reported in appendix C with parameters passed by value.

To verify the functionality of the new AVX2 functions, a dummy testbench is used within the intrinsics file. In the testbench for each function there is an if statement that compares the 256 bits of the original function and the C one. Inside the statements a printf asserts a message saying whether the the new function matches the one from Intel or not:

```
mm256_abs_epi8 0 MATCHED mm256_abs_epi8 1 MATCHED mm256_abs_epi8 2 MATCHED mm256_abs_epi8 3
    MATCHED
```

Listing 3.19. Custom AVX2 functions verification

Since ___m256i vector is made of 4x64 bits, the check is done by comparing 64 bits at once. The output message reports also which element of the vector has been verified.

3.3.3 Results of the adaption of the AVX2 decoder code

Theoretically, the code should be synthesizable and accepted by HLS. Thus a synthesis is run using the version 2018.2 of Vivado HLS but the simulation aborts providing the following output:

```
1 ERROR: [SYNCHK 200-61] ./nrLDPC_bnProc.c:2808: unsupported memory access on variable 'out'
    which is (or contains) an array with unknown size at compile time.
2 ERROR: [SYNCHK 200-41] intrinHLS.c:94: unsupported pointer reinterpretation from type '
    mm256i' to type 'i8*' on variable 'a.data'.
3 ERROR: [SYNCHK 200-11] ./nrLDPC_cnProc.c:46: Argument 'p_procBuf.cnProcBuf' of function '
    nrLDPC_decoder' (nrLDPC_decoder.c:43) has an unsynthesizable type (possible cause(s):
    pointer to pointer or global pointer).
4 ERROR: [SYNCHK 200-22] nrLDPC_decoder.c:90: memory copy is not supported unless used on bus
    interface possible cause(s): non-static/non-constant local array with initialization).
5 ERROR: [SYNCHK 200-43] nrLDPC_decoder.c:386: use or assignment of a non-static pointer '
    llrOut' (this pointer may refer to different memory locations).
6 ERROR: [SYNCHK 200-11] ./nrLDPC_cnProc.c:438: Variable 'ymm0' has an unsynthesizable type '
    m256i' (possible cause(s): structure variable cannot be decomposed due to (1)
    unsupported type conversion; (2) memory copy operation; (3) function pointer used in
    struct; (4) unsupported pointer comparison).
```

Listing 3.20. Vivado HLS synthesis errors for custom AVX2 in C language

The error reported in the first line is related to the variable out which is passed as int8_t and then it is used as an int32_t variable inside the llr2bitPacked. This error can be solved by declaring that variable as int32_t in the testbench and then passing it without casting.
In line 2 of the listing there is another pointer casting in the new AVX2 functions, in this case the pointer is an 8 bit integer and the pointed element is a 32 bit integer. To turn around the problem a mask to work only on a specific byte can do the job. By exploring further the other errors the work on LDPC for Intel processors supporting the AVX2 library has been dropped. The effort required to solve all the pointer castings which are widely spread in the code and the adaption of the new structure m256i is time consuming and prone to errors. A lot of time would be

spent in trying the new code and then debugging it. Therefore the code described up to now has been quit and the focus has been moved to the GPU version of LDPC.

3.4 LDPC decoder for GPUs

Few weeks before quitting the AVX2 code, Professor Terng-Yin Hsu and his group of the National Chiao Tung University released a CUDA code of LDPC decoder in a branch of OAI repository. The code has been tested on a Quadro P5000 Nvidia GPU. This code has been immediately selected as a clever alternative to the AVX2 code since the CUDA language can be easily imported into the OpenCL language, which is synthesizable. The code is briefly explored in the following to have an idea of how it is organized.

3.4.1 LDPC in CUDA language

The code can be divided in two parts. The first one is run on an host machine typically a CPU. The second has four functions that are executed on a GPU. These functions are called kernels.

The host code is responsible for memory allocation on the GPU and for the data transfer from the memory which is on the GPU (on-chip memory) and the one on the host machine. The received parameters from the testbench are the following:

1. *t_nrLDPC_dec_params* which is the same of the AVX2 decoder.
2. int8_t *p_llr* which is a pointer to *channel_output_fixed* input vector.
3. int8_t *p_out* which points to *estimated_output* output vector.
4. int *block_length*.
5. structure for timing statistics.

In t_nrLDPC_dec_params the information related to the BG, the lifting factor the output mode and the maximum number of iterations are present. From the values stored in the structure the base graph matrix is chosen. Then the parity check matrix is derived and it is divided into two small matrices, namely h_compact1 and h_compact2. The reason of using smaller matrices instead of the big parity check matrix is that there are some entries which are zero and therefore they are not used in the computation. The read_BG function scan the chosen base graph and removes those elements from the matrix in order to have only non null entries. To detect the zero entry, the original base graph is scanned and if the highlighted entry is -1 it is not stored in the compact matrix. It is important to remember that a -1 entry in the base graph corresponds to a $Z_C \cdot Z_C$ null matrix. The drawback of

this solution is that the base graph must be scanned twice to fill first h_compact1 and h_compact2 later. The value stored in these matrices is not a binary one but still a natural number, therefore for each iteration of the decoder the modulo operation to retrieve the circular shift is performed.
The data type of h_compact1 and h_compact2 is a data structure called h_element, which contains the x and y coordinates in the matrix (char type) and the value stored in that entry (short type), which is used to derive the circular shift coefficient for the identity matrix.
The first matrix is used for check node processing, it is a 1 dimension matrix with 46·19 elements, hence it has the size of the largest BG row (46 for BG1). In other words, each rows has 19 elements, in this way the matrix has been compressed and the processing would be improved. To know exactly how many columns one row has, there are constant vectors called h_ele_row_bg_count with 46 or 42 elements reporting the precise amount.
Similarly, for variable node (or bit node) processing the h_compact2 vector is used with 68·30 elements. The number of rows for each column is reported in the constant vector h_ele_col_bg_count which has 52 or 68 elements depending on the base graph.
After the initialization of the matrices for processing, the host part of the code allocates buffers and copy the content of h_compact1 and h_compact2 in the corresponding vectors on the device memory which are dev_h_compact1 and dev_h_compact2. Also the content of the channel_output_fixed from the testbench is copied in two different device buffer: dev_const_llr and dev_llr. The first one is a constant one since it will store the channel samples and it will not be modified. The second buffer is a temporary buffer storing the intermediate LLRs value, it is filled with the channel samples for the first iteration and it is updated by the bit node processing kernel. The size of dev_llr and dev_const_llr is Z_C·n_col·sizeof(char).
Regarding the size of dev_h_compact1 and dev_h_compact2 they have the same size of h_compact1 and h_compact2 respectively.
The dev_dt is the last buffer allocated and has the size of the parity check matrix: n_row·n_col·Z_C·sizeof(char). It is used for data transfer between check node and bit node kernels.

Finally, after buffer allocation the data are copied from the CPU side to the GPU side, then the host body loop is executed[10]:

```
1 for(int ii = 0; ii < MAX_ITERATION; ii++){
2     if(ii == 0){              // first kernel
3       ldpc_cnp_kernel_1st_iter
4       <<<dimGridKernel1, dimBlockKernel1>>>
5       (dev_llr, dev_dt, BG, row, col, Zc);
```

[10] https://gitlab.eurecom.fr/oai/openairinterface5g/-/tree/develop/openair1/PHY/CODING/nrLDPC_decoder_LYC/nrLDPC_decoder_LYC.cu

```
6     }else{                    // second kernel
7       ldpc_cnp_kernel
8       <<<dimGridKernel1, dimBlockKernel1>>>
9       (dev_llr, dev_dt, BG, row, col, Zc);
10    }
11      ldpc_vnp_kernel_normal
12      <<<dimGridKernel2, dimBlockKernel2>>>
13      (dev_llr, dev_dt, dev_const_llr, BG, row, col, Zc);
14 }
15 int pack = (block_length/128)+1;
16 dim3 pack_block(pack, MC, 1);
17 pack_decoded_bit<<<pack_block,128>>>(dev_llr, dev_tmp, col, Zc);
```

Listing 3.21. GPU host body loop which launches the kernels on the device

In the loop body of the host code the four kernels are launched and executed on the GPU. The for loop reported in listing 3.21 in each iteration launches two kernels, the first one is one of the two check node kernels and the second one is the bit node kernel. The choice of the check node kernel depends on the iteration counter since in the first iteration of the loop the decoder uses the channel samples for the check node processing. Every time the host code launches a kernel it has to set the kernel arguments and has to specify the dimension of the execution units. In CUDA environment the kernels are executed by threads. A set of thread forms one block. Multiple blocks are grouped in a grid. More blocks can be run simultaneously in order to improve parallelism and speed up the kernel execution. In this case, the block dimension (i.e. the number of thread per block) is equal to the lifting factor accepted as parameters of the decoder. The grid size (the number of block per grid) depends on the kernel, it is equal to the number of base graph rows for the check node kernels and it corresponds to the number of columns for the bit node kernel. When the maximum number of iteration is reached, the last kernel is executed, it is the packing kernel (corresponding to the llr2bitPacked of the AVX2 code). It has 128 threads per block and (block_length/128)+1 blocks. After the conversion to bits the content of dev_llr buffer is copied from device memory to the host memory side.

Regarding the kernels part, the check node kernels are named as ldpc_cnp_kernel and ldpc_cnp_kernel_1st_iter. The bit node one is the ldpc_vnp_normal and the packing kernel is called pack_decoded_bit.
The ldpc_cnp_kernel_1st_iter does not use the dev_dt buffer for computations and use the channel samples to evaluate if the check nodes are satisfied given the current LLRs. The results are stored in dev_dt. The algorithm used is the same of the AVX2 code but it is splitted in two separated loop: the first one is related to the believes evaluation whilst in the second one the results are stored in the global memory buffer dev_dt. The ldpc_cnp_kernel kernel is identical to the one previously mentioned, except that in the first part the dev_dt buffer is read and used for computation.
The bit node kernel reads the believes of the check nodes from the dev_dt buffer and computes the new LLR estimation (intrinsic and extrinsic), the results is written in dev_llr buffer. In the next iteration ldpc_cnp_kernel will read the new believes from the bit nodes and will remove the intrinsic information from the value read in

dev_llr.
The pack_decoded_bit kernel has only one for loop which is iterated 8 times. First the kernel performs the hard decision on the final LLRs then, after threads synchronization, the loop stores the results in the global memory buffer dev_tmp, whose size are Z_C·n_col·sizeof(char).

Since the CUDA code is used as reference model and is converted in OpenCL in order to be used in the SDAccel environment, the code is simulated. The simulation is run on a P2000 Quadro GPU, and the simulation parameters are such that the biggest amount of data set is used by the decoder. Namely, the code rate is $\frac{1}{3}$, the block length is 8448 with a single segment, therefore the base graph used is the first one and the lifting factor Z_C is 384. Then the decoder has to work with 26112 bits. The maximum number of decoder iteration is 5 and the simulation step is 1dB (SNR). The execution time of the decoder to produce the final result when the SNR is 4dB is equal to 107.589 μs.

Chapter 4

Acceleration of LDPC application on Xilinx FPGA

In order to accelerate LDPC decoder the CUDA code provided by OAI is not compatible with Vivado HLS and it cannot be directly deployed on an FPGA, thus the code must be ported from CUDA language to C, C++ or OpenCL C. Since the porting from CUDA to OpenCL is almost straighforward this is the chosen way.
Once the porting is completed, the critical sections of the decoder code are explored and optimized, furthermore bad part of the code due to the porting are modified because they affect performance of the application. In the following sections the OpenCL code is shown and explained, as well as all the solutions are explored step by step.

4.1 SDAccel environment

SDAccel is a development environment used for heterogeneous computing, namely the application is divided in two parts: one runs on a CPU, the other one runs on a secondary device. The secondary device is responsible of performing high computing task, in this case it is an FPGA board. CPU and FPGA communicate using a PCIe bus.
The CPU runs the so called host code. The host code set the parameters to send to FPGA for execution, it has the input data set to feed the application on FPGA and it reads the final results for a comparison check. Moreover, the host code has the task of allocating buffers inside a Double Data Rate DRAM memory which is place between CPU and the FPGA. Since access to this memory is latency limited, it is not suggested to use it for intermediate results but just for reading and writing the input and output data. Four DRAM banks are present with AXI4 memory interfaces, the data width of the memory can be up to 512 bits. Since the memory is shared between CPU and FPGA, only one of the two devices can access it while

the other one waits its turn to use the memory. In other words, the host code first sets the execution environment for the FPGA and load into the DRAM memory the input data, in this interval the FPGA cannot use the memory. Then the host launches the kernels on the FPGA and loses the control of the memory, which belongs to the FPGA until it has completed the computation. Once the FPGA kernels return, the control is given back to the host code which can read the results from DRAM.
The FPGA runs the kernel code which is a compute intensive application that must be accelerated. Speaking of terminology, a kernel is a function which is executed on the FPGA, more kernels can be executed on a single FPGA (concurrently or not). To improve parallelism, more copy of the same kernel can be executed in parallel, the single execution of a kernel is defined as work item, a set of work items is named work group. The size of a work group is decided by the host code. One work item is identified by the id of the group (group_id) and by the work item id in the group (local_id). To identify one work items among all the work items of the application the global_id must be used. Kernels execution are put inside a command queue which can execute the kernels in order or out of order, the choice is taken by the host code. More queues can be used to improve parallelism but synchronization between them is required if kernels are sharing variables.
The command queues can be controlled by the host code with proper callback functions, but the synchronization between work items and work groups is not under the user control and can be done in any order. In OpenCL work items and work groups are described in a 3 dimensional matrix called NDRange, whose sizes are the global size (related to work items) and the local size (related to work groups). This 3D space is used to execute the kernels according to the host code specifications. Hence, if the number of work items specified for the kernel is greater than one, the kernel code is inserted into a loop which is related to the NDRange. This outmost loop will iterate through the work items of a group.

The work item and work group organization is important since it is related to the memory hierarchy used by OpenCL. Considering the reference guide of the API calls by the Khronos group [24], in OpenCL four memory address spaces are available on the FPGA:

1. Global which is shared by all work items and work groups. Read and Write memory.

2. Local which is shared by the work items of a work group. Read and Write memory.

3. Private which is accessible by one work items only. Read and Write memory.

4. Constant memory which is shared by all work items and work groups. Read only memory.

Regarding the off chip memory, the DRAM memory, has both a read write address space and a read only one. The off chip memory has the biggest storage but has the largest latency. The global (on-chip) memory is implemented as SRAM in small blocks (BRAM), also the local memory is implemented as BRAM memory. Local memory is smaller than on chip memory, thus the access time is much smaller, the drawback is the accessibility by work groups. The fastest memory of all is the private one but it is also the one with the smallest storage.

From the design point of view, SDAccel offers three build targets: the software emulation, the hardware emulation and the system one. The designer use software emulation to check the syntax of the kernel code and verify the functionality, it is the fastest build and it is executed on the x86 machine.
Hardware emulation uses the generated RTL by Vivado HLS to verify that the generated hardware works as expected. This emulation runs on a dedicated environment which is very slow and is very time consuming, thus a small data set is suggested to run and complete the emulation.
Finally the system target generates the bitstream that will be deployed on the FPGA using the RTL code from Vivado HLS. This target is run directly on the FPGA and provides the actual results of the generated hardware.
The design flow adopted for this work is the following: software emulation is used to verify the functionality of the code after code modifications. The hardware emulation of the target shows no data transfer between host and the device because of corrupted platform file that is used to communicate with the FPGA. Hence this emulation step is skipped. After software emulation the system target is generated and the bitstream is deployed on the target device xcvu9p-flgb2104-2-i belonging to the Xilinx VirtexUltrascale+ family. The FPGA has been provided by AWS, which is a cloud computing service by Amazon.

Regarding the profiling and optimization part of the design, SDAccel provides different tools to find the weak points of the application, also suggestions of possible improvement are given. The tools used in this work are the timeline trace, the HLS reports, the system estimate report and the profile summary. These reports are not always generated since the emulation builds do not provide all the information to SDAccel to generate completely the reports [25].
The timeline trace is a graphical reports showing the data transfer, kernel execution and OpenCL API calls on a timeline axis. This report is generated automatically for the hardware emulation. For hardware build the timeline trace generation must be enabled at compile time. If proper options are given to the compiler the stalls, execution times and data transfer from kernel to DRAM are shown.
The HLS report is generated by Vivado HLS, it is an estimation of the synthesis, therefore the results reported are not actual results. It contains information about the resource usage, latency of the loops in the design, the estimated clock period and details regarding the logic implementation.
The system estimate report is generated automatically for hardware emulation and

system builds, it provides high level details of the kernels.
The profile summary contains comments and accurate details of the kernels. It is divided in sections and provides details regarding the API calls, the data transfer from/to host to/from the FPGA. Also a section with comments regarding possible improvements is shown. It is generated for all the builds after kernel execution. For software emulation data transfer is omitted.
In the design flow used in this thesis the software emulation is used for functionality verification whilst the hardware reports are used for optimizations for the next version of the code.

4.2 Porting of CUDA code in OpenCL

In order to use the GPU code inside the SDAccel environment the code is ported from CUDA language in OpenCL C.
First of all, the part of the CUDA code related to the data transfer from host to the device and that allocate buffers is moved into the OpenCL host code. The porting is performed to be similar as much as possible to the original code. For the straightforward part of the porting, the following table can be used as reference:

CUDA	**OpenCL**
___global___	___kernel
___device___ ___constant___	constant
blockIdx.[x,y,z]	get_group_id([0,1,2])
threadIdx.[x,y,z]	get_local_id([0,1,2])
___shared___	___local
___synchthreads()	barrier(CLK_GLOBAL_MEM_FENCE)
cudaMalloc	clCreateBuffer()
cudaMemCpy()	clEnqueueReadBuffer/clEnqueueWriteBuffer()
dim3	size_t
kernerl_name <<<>>>	clEnqueueNDRangeKernel()
cudaFree	clReleaseMemObj()
Block size	Local size
Thread number	Work items number

Table 4.1. Code conversion table from CUDA language into OpenCL C. Trivial code mapping is shown.

The four kernels of the CUDA code are kept as kernels also in the OpenCL code. Namely, the kernels are called ldpc_cnp_kernel_1st_iter, ldpc_cnp_kernel, ldpc_vnp_kernel_normal and pack_decoded_bit in the code.
Besides the modifications reported in table 4.2, other changes are applied to the

original code. First of all, two global variables h_compact1 and h_compact2 are removed because they are redundant. In fact Read_BG function generates the compact base graph and stores it in h_compact1 and h_compact2, which reside in the host side part of the code, then they are copied in the device memory inside dev_compact1 and dev_compact2 memories on the device. To remove this memory copy operation from CPU to FPGA the base graph initialization is moved into the kernel code where read_BG function fills dev_compact1 and dev_compact2 vectors. These two vectors are stored in the DRAM memory.
The warmup() function used to warm the GPU for time measurements is removed.

The OpenCL host code part is organized in three sections: the first one is the environment initialization, then there is the kernel execution and finally the memory read and comparison of the results.
One function, namely fpga_init, allocates the OpenCL objects. One command queue is used to execute sequentially and in order the 4 kernels. The fpga_init function creates also the context in which the FPGA platform, the command queue, the binary bit stream are inserted. In this part of the host code the platform connected to CPU via PCIe bus is identified.
Four buffer objects are allocated in off-chip memory:

1. dev_llr is for intermediate LLRs value. It is write only for the host and read/write for the kernel. The size is equal to 68·384·sizeof(char)

2. dev_const_llr contains the channel samples. This buffer is read only for the kernel and write only for the host. The size is equal to 68·384·sizeof(char)

3. dev_tmp is for the final results. This buffer is read only for the host and read write for the kernel. The size is equal to 68·384·sizeof(unsigned char)

4. dev_dt is used for intermediate results exchanged between kernels. Read and write buffer. Its size corresponds to the biggest supported parity check matrix size which is 46·68·384·sizeof(char).

dev_llr and dev_const_llr are filled initially with the input data generated by OAI testbench. The input vector is generated using the GEN_TEST define in the testbench, whilst the output one is generated from the AVX2 decoder. Since the testbench generates an input vector for different SNR values, only the vectors corresponding to the last value of SNR is taken as reference. For all the solutions the parameters used to run the decoder are Zc=384, block_length=8448, BG1 with 5 iterations.

The host body loop of CUDA code reported in listing 3.21 is moved in the host part of OpenCL code. In the host loop each kernel is launched in the following way:

```
1 clSetKernelArg(kernel_name, 0, sizeof(variable_type), arg[0]));
2 clSetKernelArg(kernel_name, 0, sizeof(variable_type), arg[1]));
3
4                                    [...]
5
6 clEnqueueNDRangeKernel(commands, kernel_name, NDRange_dimension, NULL, number_of_workitems,
      number_of_groups, 0, NULL, 0));
7 clFinish(commands)
```

Listing 4.1. OpenCL host body loop for kernels execution using clEnqueueNDRangeKernel.

where the function clSetKernelArg sends the parameters for the kernel execution, the parameters are passed by position as they are declared in the kernel code. The clEnqueueNDRangeKernel is used to put the kernel in the command queue *commands*. The NDrange dimension is specified in order to tell the scheduler how the work items are organized in the NDRange matrix, in fact the number of work groups and work items allocated for each kernel are specified. For the check node kernels there are Zc work items in each group, the group number is equal to the number of row of the chosen base graph. Similarly for bit node processing the number of work group is equal to the number of columns of the base graph, whilst there are Zc work items per group. The pack_decoded_bit kernel instead has 128 work items for each group and (block_length-1)/128 groups.
clFinish(commands) is an OpenCL API which stops all commands previously issued until they have completed. It is a strong way to synchronize the execution of the FPGA and the host code.

Once all the four kernels return then the CPU can read from the DRAM the results produced during the computation. If the generated results match completely the output test vector obtained by OAI testbench then the functionality is met and the generated RTL works as expected. The results are read using the clEnqueueWriteBuffer.
After the comparison all the memory object of OpenCL must be release to allow the SDAccel tools to collect all the profiling information. The object to be released are the kernels, the binary program, the buffers, the device and context objects.

4.2.1 Software emulation build

In order to verify that the porting is correct, golden input and output vectors are fed to the host code. Those vectors are obtained by a simulation of AVX2 decoder inside the OAI repository environment. The testing parameters are: lifting factor equals 384, base graph 1, block length equals 8448 bits, code rate one third and an SNR of 4 dB, which is the last simulated SNR value of the AVX2 decoder that provides a 0 BER. This testing condition shows what is the slowest execution time of the decoder when the biggest amount of bits and the biggest base graph are used, therefore also the resource utilization is maximized in this case.
The comparison of the golden results with the ones produced during the software emulation was completely successful.

Before proceeding with the hardware build, the execution time of the software emulation is measured in order to have a software evaluation of the application. The software emulation is run on a 3.2GHz Intel processor, the measured execution time of the whole decoder is: 267.653ms.

4.2.2 Hardware build results

After the bit stream building process, the report estimates is generated and it can be analyzed to see which kernel is slowing down the application. In the following figure the sections regarding timing and latency information are depicted:

Timing Information (MHz)

Compute Unit	Kernel Name	Module Name	Target Frequency	Estimated Frequency
ldpc_cnp_kernel_1st_iter_1	ldpc_cnp_kernel_1st_iter	read_BG2_1	250	342.465759
ldpc_cnp_kernel_1st_iter_1	ldpc_cnp_kernel_1st_iter	read_BG1_1	250	342.465759
ldpc_cnp_kernel_1st_iter_1	ldpc_cnp_kernel_1st_iter	ldpc_cnp_kernel_1st_iter	250	318.775879
ldpc_cnp_kernel_1	ldpc_cnp_kernel	ldpc_cnp_kernel	250	318.775879
ldpc_vnp_kernel_normal_1	ldpc_vnp_kernel_normal	ldpc_vnp_kernel_normal	250	342.465759
pack_decoded_bit_1	pack_decoded_bit	pack_decoded_bit	250	342.465759

Latency Information (clock cycles)

Compute Unit	Kernel Name	Module Name	Start Interval	Best Case	Avg Case	Worst Case
ldpc_cnp_kernel_1st_iter_1	ldpc_cnp_kernel_1st_iter	read_BG2_1	121840 ~ 451624	121840	287824	451624
ldpc_cnp_kernel_1st_iter_1	ldpc_cnp_kernel_1st_iter	read_BG1_1	231820 ~ 704148	231820	469548	704148
ldpc_cnp_kernel_1st_iter_1	ldpc_cnp_kernel_1st_iter	ldpc_cnp_kernel_1st_iter	394758 ~ 272823558	394757	130800389	272823557
ldpc_cnp_kernel_1	ldpc_cnp_kernel	ldpc_cnp_kernel	375951 ~ 2329743	375950	1474958	2329742
ldpc_vnp_kernel_normal_1	ldpc_vnp_kernel_normal	ldpc_vnp_kernel_normal	133260 ~ 1848204	133259	990731	1848203
pack_decoded_bit_1	pack_decoded_bit	pack_decoded_bit	9612 ~ 18956	9611	14347	18955

Figure 4.1. OpenCL porting: system estimates for hardware build is shown. Frequency and latency values are reported for all the kernels and the functions used by the kernels

By referring to the figure above, the function read_BG is duplicated and one dedicated function for the two base graphs is implemented. The reason of the duplication is due to HLS compatibility. In the CUDA version read_BG function initializes the compact matrices according to the base graph. The initialization is implemented using an if statement to choose the base graph within a switch statement to select the proper matrix according to Z_C. This is not supported by HLS, hence the new code has two different functions (read_BG1 and read_BG2) which are called properly by ldpc_cnp_kernel_1st_iter kernel and a switch statement selects the corresponding matrix of the base graph according to the lifting factor.
The body loop in the host code launches ldpc_cnp_kernel_1st_iter as the first kernel as shown in listing 3.21, then that kernel has to initialize the decoder at the cost of several clock cycles.
Since ldpc_cnp_kernel_1st_iter kernel has 384 workitems per workgroup, which are 46, it means that read_BG1 or read_BG2 is executed once for each work item.

At each call one of those functions repeats the same writing operation into the off-chip global memory, since dev_compact1 and dev_compact2 reside there. Thus the kernel is expected to lose lot of time doing the same operation and accessing to the off-chip memory. The performance loss is due to the bad porting, since in CUDA the memory accesses are done in parallel, therefore there are no redundant operations in that case.
On the other hand the pack_decoded_bit kernel is the fastest kernel with the lowest latency and the fastest frequency.
By referring to the HLS reports, in the loop section one can detect which are the most problematic loops:

Kernel	**Loop name**	**Latency**	**Iteration Latency**
ldpc_cnp_kernel_1st_iter	CNP1st_loop1	519∼3287	173
ldpc_cnp_kernel_1st_iter	CNP1st_loop2	459∼2907	153
ldpc_vnp_kernel_normal	VNP_loop	203∼4669	203
ldpc_cnp_kernel	CNP_loop1	510∼3230	170
ldpc_cnp_kernel	CNP_loop2	444∼2812	148
pack_decoded_bit	PACKDECODED_loop	9344	73

Table 4.2. OpenCL porting: kernels loop section with lantecy information.

In table 4.2.2 the critical loops are reported, none of them is pipelined yet. The table 4.2.2 shows that the iteration latency (in terms of clock cycles) and the worst case latency are bigger for the variable node kernel with respect to the check node ones.
Considering the memories used in the loops in table 4.2.2, check node kernels uses dev_h_compact1, dev_llr, dev_dt. On the other hand variable node kernel uses dev_h_compact2, dev_llr , dev_const_llr and dev_dt. Therefore, the more the memory objects used in the kernels, the more are the DRAM accesses, thus the latency of the loops increases. The pack_decoded_bit loop has the highest latency because of a read and accumulate operation of DRAM which is performed inside the loop.

After the preliminary analysis of HLS and system estimate report, the application is run on the FPGA provided within the AWS cloud. Once the application completes two reports are generated, namely the timeline trace and the profile summary of the application. The execution time of the kernels is taken from the profile summary after the run on the AWS platform:

Kernel	Number of calls	Total Time (ms)	Average Time (ms)	Maximum Time (ms)
ldpc_cnp_kernel_1st_iter	1	36461.900	36461.900	36461.900
ldpc_cnp_kernel	4	1131.940	282.986	282.997
ldpc_vnp_kernel_normal	5	955.080	191.016	191.027
pack_decoded_bit	1	5.935	5.935	5.935

Table 4.3. OpenCL porting: kernels execution time on AWS platform from profile summary report.

As expected from the HLS and system estimates reports, ldpc_cnp_kernel_1st_iter is extremely slow if compared to other kernels, it takes approximately 36 seconds to complete because of the read_BG functions. Nevertheless ldpc_cnp_kernel is slower but comparable with respect to ldpc_vnp_kernel_normal kernel.
Both of them are more or less one order of magnitude smaller with respect to ldpc_cnp_kernel_1st_iter kernel. pack_decoded_bit kernel is the fastest one even if the loop latency is the highest one. The reason is due to the number of DRAM buffers used in the code, which is only two, also the global size of work items is smaller with respect to the other kernels. The total execution time of the decoder written in OpenCL is 38.554895 seconds, which is very high.
In order to determine the bottlenecks of the application one can take a look to the data transfer section between kernels and off-chip memory of the profile summary, whose relevant sections are shown in the following table:

Kernels	Op	Number Of Transfers	Transfer Rate (MB/s)	Avg Size (KB)	Avg Latency (ns)
pack_decoded_bit	R	8448	6.321	0.004	324.529
pack_decoded_bit	W	1056	0.198	0.001	226.303
ldpc_vnp_kernel_normal	R	1950720	8.204	0.004	318.812
ldpc_vnp_kernel_normal	W	130560	0.137	0.001	217.169
ldpc_cnp_kernel	R	2426880	8.592	0.004	331.217
ldpc_cnp_kernel	W	485376	0.430	0.001	221.173
ldpc_cnp_kernel_1st_iter	R	485376	1.718	0.004	329.030
ldpc_cnp_kernel_1st_iter	W	125412096	221.875	0.002	104.064

Table 4.4. OpenCL porting: data transfer between kernels and off-chip memory. The results are generated after the execution on AWS platform.

Since ldpc_cnp_kernel_1st_iter kernel is initializing dev_h_compact1 and dev_h_compact2 matrices a lot of writing operations are needed as shown in the table above. In fact the number of transfers for the writing operations is several

order of magnitude bigger with respect to the other kernels. It is important to remember that off-chip memory is used also for storing and reading intermediate results, hence an high number of transfer is expected also for the kernels which are executed more than once, namely ldpc_vnp_kernel_normal and ldpc_cnp_kernel kernels. From the table it is evident that the DRAM maximum width is not fully exploited, in fact the average data size is always smaller than 1KB. Another important parameter is the transfer efficiency. Given that four banks can be used, when they are completely used the transfer efficiency increases of 25% per bank, instead the current transfer efficiency is less than 0.1% for all the kernels. The transfer efficiency is defined as the ratio of the average Bytes per transfer over the minimum of 4KB and $\frac{memory\ bit\ width \cdot 256}{8}$. The transfer rate is defined as the total data transfer of a kernel over the compute unit total time.
Considering the amount of off-chip memory transfers, ldpc_cnp_kernel_1st_iter has the highest transfer rate, but it is also the kernel that spends lot of time in memory accesses. The average latency column shows the time required to access the off-chip memory, so given the number of transfer one can approximate the total time spent in memory access by multiplying the number of transfer for the average time. For the worst kernel it is 13.05 seconds for writing and 0.16 seconds for reading.

The timeline trace provides a graphical view of the kernels situation. One can easily notice which is the performance critical kernel and it is possible to verify if concurrency between memory operations and execution is possible. In the following a snapshot of the timeline generated after AWS execution is shown:

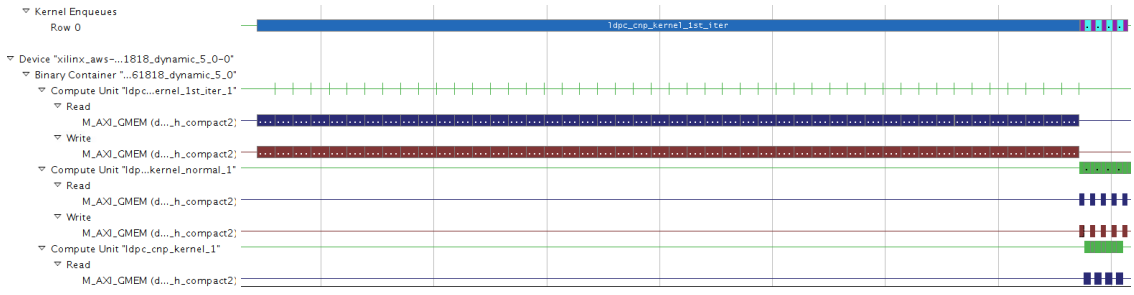

Figure 4.2. OpenCL porting: timeline trace of the decoder after execution on AWS using coarse profiling option.

By referring to the figure Fig.4.2, in the kernel enqueues row the execution of the four kernels is depicted. The blue rectangle is ldpc_cnp_kernel_1st_iter kernel, the purple one is the call of ldpc_vnp_kernel kernel, the teal one is ldpc_cnp_kernel kernel. The pack_decoded_bit kernel is not visible with the shown resolution.
Only one bank is used and for each port reading and writing operations are shown.

It can be verified that during the kernels execution most of the time is spent to perform DRAM accesses. Regarding ldpc_cnp_kernel and ldpc_vnp_kernel kernels they have comparable duration times as discussed before, moreover also for them memory operations consumes most of the execution time.

In order to have a reference metrics, a simulation with the same input data set and parameters is run on a 3.2GHz Intel i7-6900K processor with the AVX2 solution. The execution time measured for the AVX2 implementation is 257.549 μs, which is very low. If one compares the execution of the AVX2 with the OpenCL one the acceleration factor of AVX2 (execution of OpenCL over AVX2) is 149699x, whilst the acceleration factor for the GPU is 358353x. Hence, the OpenCL solution for the moment is very far from the other two implementations in terms of performance, on the other hand the acceleration of the GPU code with respect to the AVX2 one is 2.39x.

To summarize the results obtained so far, the four kernels do not have a uniform time occupation, in particular one of the kernels is taking large time because of the initialization of the matrices used for intermediate computation. Additionally, most of the decoder time is spent in DRAM interactions. One of the kernels (pack_decoded_bit) can be ignored during the optimization flow since it achieves the best performance (if compared with other kernels).
The first optimization step is to have a uniform time occupation for the three critical kernels, especially the first one. Therefore DRAM accesses must be reduced and repeated operations must be eliminated in order to avoid limitations due to initialization tasks.

4.3 Memory architecture optimization

The figure Fig.4.2 shows that all the kernels exchange lot of data with the off-chip global memory, therefore the first goal is to reduce the number of accesses to DRAM, in order to not pay every time the latency of the DRAM controller. The reference card of OpenCL [24] states that an on-chip global memory is available. To reduce the off-chip memory accesses on-chip global memory is chosen as storage block for intermediate results. Unfortunately the Xilinx OpenCL Compiler (XOCC) compiler used by SDAccel to generate the bitstream reports that the access to non-pipe global variable is not supported. Therefore the on-chip global memory is not used to store intermediate results. As alternative the local memory is used. Moreover, DRAM ports widening and bursts accesses are implemented to maximize the transfer efficiency for the critical kernels.

4.3.1 Local memory implementation

Local memory is smaller than the on-chip global memory as well as the access time. The code changes are not many, since the processing of the decoder is such that

work groups do not interfere between them. It is important to remember the work item and work group organization: check node kernels have 46 work groups with 384 work items each. As described in chapter 3, LDPC codes in OAI are cyclic, hence each entry of the base graph is replaced by a shifted identity matrix which is Z_CxZ_C. The number of rows of the base graph is multiplied by Z_C, thus in the OpenCL implementation each work group has a number of Z_C work items (384 in this case) to process the shifted identity matrix corresponding to an entry of the base graph. Therefore each work group do not access to the variable used by another one since each one has its own row to work on.
On the other hand bit node processing kernel has 68 work groups with Z_C equals 384 work items to elaborate the parity check matrix columns. Also in this case the groups do not use shared variables between them.
Thus local memory can be used to store the results that check nodes and bit nodes exchange, improving the access time to use the data.

Another limitation carried by local memory is that it is not shared between kernels, hence each kernel has its dedicated local memories, therefore some modification in the host code and in the kernel code must be applied. For this reason, starting from the current new solution, the ldpc_cnp_kernel_1st_iter, ldpc_cnp_kernel, ldpc_vnp_kernel_normal kernels are transformed into functions and merged into a single kernel called nrLDPC_decoder kernel. The pack_decoded_bit is kept as standalone kernel.
From the host code side, the loop to launch the kernels shown in 3.21 has been changed since now only two kernels must be executed. Check node and bit node kernels are now merged into a single one, hence they are executed according to a support variable, namely index, received by the host code. Thus the host loop now has the following structure:

```
1 for(int ii = 0; ii < numMaxIter; ii++){
2     if(ii==0){
3       index=1;
4       arg=0;
5       OCL_CHECK(clSetKernelArg(nrLDPC_decoder, arg++, sizeof(int), &index));
6       OCL_CHECK(clSetKernelArg(nrLDPC_decoder, arg++, sizeof(cl_mem), &dev_const_llr));
7       OCL_CHECK(clSetKernelArg(nrLDPC_decoder, arg++, sizeof(cl_mem), &dev_llr));
8       OCL_CHECK(clSetKernelArg(nrLDPC_decoder, arg++, sizeof(int), &BG));
9       OCL_CHECK(clSetKernelArg(nrLDPC_decoder, arg++, sizeof(int), &row));
10      OCL_CHECK(clSetKernelArg(nrLDPC_decoder, arg++, sizeof(int), &col));
11      OCL_CHECK(clSetKernelArg(nrLDPC_decoder, arg++, sizeof(int), &Zc));
12      OCL_CHECK(clEnqueueNDRangeKernel(commands, nrLDPC_decoder, 3, NULL, dimGridKernel2,
      dimBlockKernel2, 0, NULL, 0));
13      OCL_CHECK(clFinish(commands));
14    }
15    else{
16      index=2;
17      arg=0;
18      OCL_CHECK(clSetKernelArg(nrLDPC_decoder, arg++, sizeof(int), &index));
19      OCL_CHECK(clSetKernelArg(nrLDPC_decoder, arg++, sizeof(cl_mem), &dev_const_llr));
20      OCL_CHECK(clSetKernelArg(nrLDPC_decoder, arg++, sizeof(cl_mem), &dev_llr));
21      OCL_CHECK(clSetKernelArg(nrLDPC_decoder, arg++, sizeof(int), &BG));
22      OCL_CHECK(clSetKernelArg(nrLDPC_decoder, arg++, sizeof(int), &row));
23      OCL_CHECK(clSetKernelArg(nrLDPC_decoder, arg++, sizeof(int), &col));
24      OCL_CHECK(clSetKernelArg(nrLDPC_decoder, arg++, sizeof(int), &Zc));
25      OCL_CHECK(clEnqueueNDRangeKernel(commands, nrLDPC_decoder, 3, NULL, dimGridKernel2,
      dimBlockKernel2, 0, NULL, 0));
26      OCL_CHECK(clFinish(commands));
27    }
28    if(ii+1 != numMaxIter){
```

```
29        index=3;
30        arg=0;
31        OCL_CHECK(clSetKernelArg(nrLDPC_decoder, arg++, sizeof(int), &index));
32        OCL_CHECK(clSetKernelArg(nrLDPC_decoder, arg++, sizeof(cl_mem), &dev_const_llr));
33        OCL_CHECK(clSetKernelArg(nrLDPC_decoder, arg++, sizeof(cl_mem), &dev_llr));
34        OCL_CHECK(clSetKernelArg(nrLDPC_decoder, arg++, sizeof(int), &BG));
35        OCL_CHECK(clSetKernelArg(nrLDPC_decoder, arg++, sizeof(int), &row));
36        OCL_CHECK(clSetKernelArg(nrLDPC_decoder, arg++, sizeof(int), &col));
37        OCL_CHECK(clSetKernelArg(nrLDPC_decoder, arg++, sizeof(int), &Zc));
38        OCL_CHECK(clEnqueueNDRangeKernel(commands, nrLDPC_decoder, 3, NULL, dimGridKernel2,
      dimBlockKernel2, 0, NULL, 0));
39        OCL_CHECK(clFinish(commands));
40      }
41      else
42      {
43        index=4;
44        arg=0;
45        OCL_CHECK(clSetKernelArg(nrLDPC_decoder, arg++, sizeof(int), &index));
46        OCL_CHECK(clSetKernelArg(nrLDPC_decoder, arg++, sizeof(cl_mem), &dev_const_llr));
47        OCL_CHECK(clSetKernelArg(nrLDPC_decoder, arg++, sizeof(cl_mem), &dev_llr));
48        OCL_CHECK(clSetKernelArg(nrLDPC_decoder, arg++, sizeof(int), &BG));
49        OCL_CHECK(clSetKernelArg(nrLDPC_decoder, arg++, sizeof(int), &row));
50        OCL_CHECK(clSetKernelArg(nrLDPC_decoder, arg++, sizeof(int), &col));
51        OCL_CHECK(clSetKernelArg(nrLDPC_decoder, arg++, sizeof(int), &Zc));
52        OCL_CHECK(clEnqueueNDRangeKernel(commands, nrLDPC_decoder, 3, NULL, dimGridKernel2,
      dimBlockKernel2, 0, NULL, 0));
53        OCL_CHECK(clFinish(commands));
54      }
55    }
56    size_t pack = (block_length/128)+1;
57    size_t pack_block[3]={(pack-1)*128, MC, 1};
58    size_t pack_local[3]={128, 1, 1};
59    arg = 0;
60    OCL_CHECK(clSetKernelArg(pack_decoded_bit, arg++, sizeof(cl_mem), &dev_llr));
61    OCL_CHECK(clSetKernelArg(pack_decoded_bit, arg++, sizeof(cl_mem), &dev_tmp));
62    OCL_CHECK(clSetKernelArg(pack_decoded_bit, arg++, sizeof(int), &col));
63    OCL_CHECK(clSetKernelArg(pack_decoded_bit, arg++, sizeof(int), &Zc));
64    OCL_CHECK(clEnqueueNDRangeKernel(commands, pack_decoded_bit, 3, NULL, pack_block,
       pack_local, 0, NULL, 0));
65    OCL_CHECK(clFinish(commands));
```

Listing 4.2. New host code loop after kernels merge, index variable is used to execute the proper function in the kernel code.

The new kernel, nrLDPC_decoder, receives the decoder parameters like the kernels of the previous implementation.
The memory objects dev_h_compact1, dev_h_compact2 and dev_dt are no more set as kernel arguments since they are implemented as local memory in the kernel code side. The additional parameter is the index variable which tells the kernel code which function must be invoked.
When the variable *index* is set to one the ldpc_cnp_kernel_1st_iter is executed. If it is 2 ldpc_cnp_kernel is called. The values 3 and 4 are used to tell the kernel code to invoke the ldpc_vnp_kernel_normal function, two values are set in order to specify when to write by bursts.
In the previous solution, check node and bit node kernels had different global size but the local size was the same. Because of the merge, the work group size of the new kernel is still equal to Z_C (384) and the global size is equal to the maximum global size of the previous kernels, which corresponds to the number of columns multiplied by Z_C. For the given parameters the total number of work items is 26112 which corresponds to the number of bits of the codeword. Since in the previous solution ldpc_cnp_kernel_1st_iter and ldpc_cnp_kernel kernels have less work items with respect to ldpc_vnp_kernel_normal, an if statement in the kernel code is required to specify how many work items must be executed when those

functions are called. The pack_decoded_bit is not modified in this solution.

4.3.2 Burst accesses and memory ports widening

The kernel now stores the intermediate results in the BRAM on the chip, this means that it must read only once the input LLRs and write the final LLRs estimation once. From now on dev_llr is the local BRAM whilst llr is the memory object in the DRAM, similarly dev_const_llr is the constant channel sample LLRs in BRAM and const_llr is the data in DRAM.
In order to increase the transfer efficiency of the AXI4 interface with the DRAM, the reading and writing ports are widened, thus, instead of reading only 8 bits for each port (which is the size of the data), 512 bits are read or written through the memory ports. The SDAccel guide provide some tips to avoid manual code modifications to adapt the data to 512 bits. Thus some pragmas are suggested, namely *reqd_work_group_size* and *vec_type_hint*. The first one specifies which is the group size of the kernel. That pragma helps the runtime library to schedule efficiently the work items in the NDRange space. The second one tells XOCC compiler which is the main data used for computation and can help the synthesis tool to perform better optimizations.
To hide the latency due to the DRAM controller bursts are inferred, both for writing and reading operations from the off-chip global memory. Bursts consist in repeated memory accesses to sequential addresses, hence the latency due to the controller is paid only once. Also in this case the SDAccel manual [13] reports a specific way to implement burst operations:

```
1    __attribute__((xcl_pipeline_loop(1)))
2    read_loop: for(i=0; i<size_of_reading; i++)
3        dev_llr[i] = llr[i];
```

Listing 4.3. Example of suggest loop structure for easily implementation of burst memory accesses in SDAccel

To help xocc to implement the burst, reading writing and compute operations must be separated. To be noted that the *__attribute__((xcl_pipeline_loop(1)))* pragma is used for OpenCL kernels only. It asks HLS tool to pipeline the read_loop with an Initiation Interval (II) equal to 1. The initiation interval is defined as the number of clock cycles to wait for a new loop iteration.
Given the attributes and the rules to follow, Vivado HLS does not implement correctly the burst for dev_llr variable, instead for dev_const_llr the bursts are implemented correctly. In fact the read and write operations for dev_llr local memory are implemented using both ports of the BRAMs with a width of 64 for each port, hence only 128 bits are read per memory access. To workaround this HLS issue, the *__attribute__((xcl_array_reshape(cyclic, 32, 1)))* is used. Array reshaping

breaks the target array into small blocks, then the blocks are concatenated in order to widen the memory ports. The reshaping type can be cyclic, block or complete.
A cyclic reshaping concatenates the memory chunks in a serial way, in other words the memory is filled starting from the first entry of the the first chunk up to the first entry of the last chunk, then it goes back to the second entry of the first chunk. The memory is then filled cyclically.
Block reshaping fills the memory by blocks, so it starts to fill the first chunk of the array and then it moves to the next one.
A complete reshaping decomposes the memory into the single entries of the memory, this results into a widening of the memory equals to the number of the elements of the memory, which means registers.

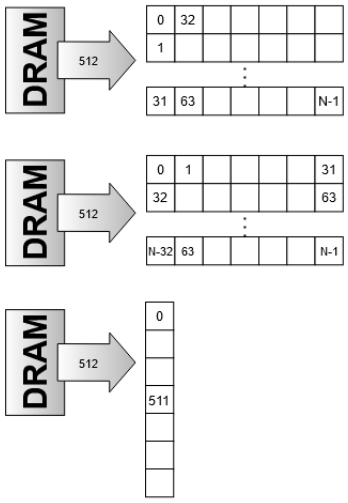

Figure 4.3. Array reshaping in action. The top most figure shows the cyclic reshaping. The central picture corresponds to the block reshape and the last one is the complete reshape

After reshape is applied to the dev_llr memory the operations involving the off-chip global memory and dev_llr are burst operations. In fact, both local memory ports are used and both have a width of 32·8 bits, thus the width of the ports is maximized to read and write 512 bits.

Burst read and write do not take place at each kernel call, they must be performed only for the first and last call respectively. Thus an if statement is necessary in the kernel code to execute the read and write only once. In particular, dev_llr and dev_const_llr reading operations are performed only for the first work item of the first work group, that has a global_id equals to (0,0,0) in the NDRange space. Moreover, the two read operations occur when the index variable is 1 that

is when ldpc_cnp_kernel_1st_iter function is called in the kernel code. Thus first the inputs are read and the function is later executed by all the work items of the group.
Similarly, the write operation occurs for the last work item of the NDRange, which has a (col-1,0,0) group_id and (Z_C-1,0,0) local_id. Also in this case an if statement is required to determine the work item that must write the results in the off-chip memory.

As discussed in the previous section, during the first execution, the kernel has to initialize the dev_h_compact1 and dev_h_compact2 matrices. Because of the porting the initialization was consuming most of the time of the kernel execution (ldpc_cnp_kernel_1st_iter kernel in the previous solution). The problem is related to the number of times that the initialization of the matrices is done. In a GPU, the memory accesses are parallel, in an FPGA they are sequential. Therefore in the FPGA each work item will access to these memories and the same operation is repeated many times. In order to avoid this code redundancy, an if statement is inserted in the ldpc_cnp_kernel_1st_iter function to initialize the matrices only once. The reasoning is the same of the burst reading, thus only the first work item of the NDRange space initialize the matrices.

4.3.3 Hardware build results

To verify the correctness of the code after those modifications, software emulation is run. The results of software emulation shows that the results produced by the kernel match the reference ones, hence the functionality is met.
With burst accesses and memory ports widening one expects to have short time spent in exchanging data with the DRAM, namely just at the beginning and at the last call of nrLDPC_decoder kernel. On the other hand, storing intermediate results in local memory reduces the amount of time spent in data transfer.
By running the hardware build one can have a look to the report estimates:

```
Timing Information (MHz)
Compute Unit        Kernel Name       Module Name       Target Frequency  Estimated Frequency
------------------  ----------------  ----------------  ----------------  -------------------
pack_decoded_bit_1  pack_decoded_bit  pack_decoded_bit  250               342.465759
nrLDPC_decoder_1    nrLDPC_decoder    read_BG2_1        250               429.737885
nrLDPC_decoder_1    nrLDPC_decoder    read_BG1_1        250               429.737885
nrLDPC_decoder_1    nrLDPC_decoder    nrLDPC_decoder    250               318.775879

Latency Information (clock cycles)
Compute Unit        Kernel Name       Module Name       Start Interval  Best Case  Avg Case  Worst Case
------------------  ----------------  ----------------  --------------  ---------  --------  ----------
pack_decoded_bit_1  pack_decoded_bit  pack_decoded_bit  795             794        794       794
nrLDPC_decoder_1    nrLDPC_decoder    read_BG2_1        11734           11734      11734     11734
nrLDPC_decoder_1    nrLDPC_decoder    read_BG1_1        18368           18368      18368     18368
nrLDPC_decoder_1    nrLDPC_decoder    nrLDPC_decoder    783 ~ 12392463  782        1170830   12392462
```

Figure 4.4. Improving data transfer: system estimates report. Off-chip memory is accessed by inferring bursts operations and on-chip BRAM is used to store intermediate results.

With respect to the report estimates in Fig.4.1, the read_BG functions can work with an higher frequency. The latency worst case for both functions has been reduced by one order of magnitude thanks to the local BRAM exploitation for the arrays dev_h_compact1 and dev_h_compact2.
The new kernel nrLDPC_decoder works with the minimum frequency of the three kernels that have been merged, which is the one of ldpc_cnp_kernel_1st_iter.

Regarding the loops in nrLDPC_decoder kernel, the HLS report shows the following values:

Loop Name	**Latency**	**Iteration Latency**	**Initiation Interval**
CNP1st_loop1	66	30	2
CNP1st_loop2	58	41	1
CNP_loop1	64	28	2
CNP_loop2	58	41	1
VNP_loop	118	60	2
llr_burst_read	409	3	1
const_burst_read	409	3	1
llr_burst_write	409	3	1

Table 4.5. Improving data transfer: oop section of HLS report of nrLDPC_decoder kernel. Kernel loops are pipelined.

Starting from this version of the code, the loop are automatically pipelined by HLS to achieve the lowest II. The corresponding II for each loop is listed in table 4.5. Given the HLS report in table 4.5, the next step is to have an II equals 1 for all the loops in order to avoid additional latency due to carried loop dependencies.
Using BRAMs and pipelining the overall latency of the loop is reduced, this can be noted if tables 4.5 and 4.2.2 are compared. In fact the latency shown in the system estimates report of the new solution is much lower than the previous implementation. To have an idea of the latency improvement, one can sum the average latency of the three merged kernels in figure 4.1 and compare the result with the average case latency of nrLDPC_decoder. In the first solution the average case latency for three kernels is 133266078, for the second one it is 1170830, thus the improvement factor is 114x.

As mentioned before local memory is implemented as BRAM, in fact five memory objects have been moved from off-chip memory into the on-chip one, one expects to increase the resource usage regarding the BRAMs. From HLS report, one can compare the resources usage of the two implementations:

	FF	LUT	DSP	BRAM
Total	2364480	1182240	6840	4320
Porting	1%	3%	2.4%	3%
Memory opt	2%	6%	5%	17%

Table 4.6. Improving data transfer: resource usage comparison before and after memory optimization. Values obtained from HLS reports.

Moving all the data transfer into on-chip memory makes the BRAM utilization percentage to rise from 3% up to 17%. Some more logic is required to store and compute BRAMs addresses, thus also other resources percentage increase.

After the preliminary analysis on system estimates and HLS reports, the host code can be executed on AWS platform to gather the profile summary and the timeline trace of the application.
The profile summary shows the following execution times:

Kernel	Number of calls	Total Time (ms)	Average Time (ms)	Maximum Time (ms)
nrLDPC_decoder	10	96.432	9.643	10.472
pack_decoded_bit	1	2.417	2.417	2.417

Table 4.7. Improving data transfer: execution time from the profile summary report of the hardware build. The average time and maximum execution time for both kernels are reported as well as the number of calls.

The execution time is highly reduced, pack_decoded_bit kernel halved its execution thanks to the pipelining of its loop. On the other hand nrLDPC_decoder kernel execution time is reduced by a factor of 400x. Also for the current implementation one can have a look to the data transfer section with the off-chip global memory. Since the code regarding pack_decoded_bit kernel is not modified, only the part related to nrLDPC_decoder kernel is reported:

Op	Number of transfers	Transfer Rate (MB/s)	Average Size (KB)	Average Latency (ns)
R	52	0.662	1.004	934.462
W	26	0.331	1.004	289.385

Table 4.8. Data transfer of nrLDPC_decoder kernel with off-chip memory when bursts are inferred and BRAM is used for intermediate operations.

By referring to the table 4.8 the average size of data transfer is increased, this corresponds to a strong reduction in the number of transfer. Moreover the number of data transfer with off-chip global memory drops because the kernel now reads the input and write the final result only.
Profile summary also reports that the current transfer efficiency is improved up to 24.519%, this means that DRAM ports widening and bursts are properly implemented. Hence the transfer efficiency of nrLDPC_decoder kernel is maximized because only one bank is used for data transfer and the efficiency is closed to 25%, given that four banks are available.

The timeline trace can be used to verify the bursts and to see how the functions ldpc_cnp_kernel, ldpc_cnp_kernel_1st_iter, ldpc_vnp_kernel_normal are now distributed over the whole kernel execution. Also a visual comparison between the execution time and data transfer with DRAM can be performed.

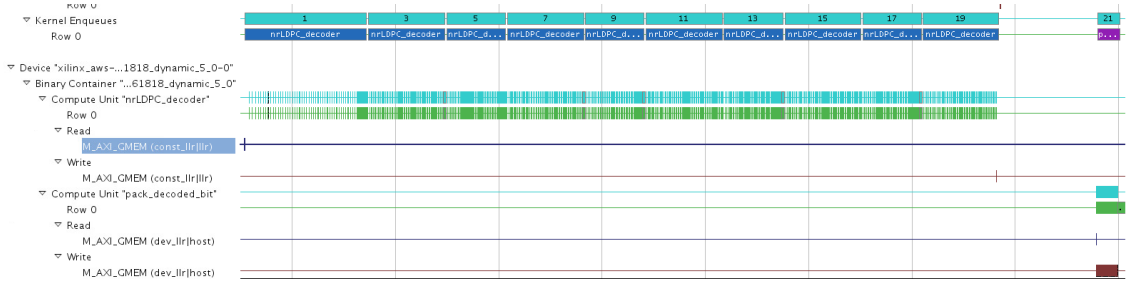

Figure 4.5. Improving data transfer: timeline trace after the execution on AWS. Data transfer and kernels execution are depicted.

As depicted in the timeline in Fig.4.5 the pack_decoded_bit kernel is still much faster than nrLDPC_decoder kernel, it is almost four times faster than a single execution of the other kernel.
By looking at the reading and writing transfers line of nrLDPC_decoder kernel one can notice that only two memory operations are performed. Namely two consecutive readings take place (blue line) spending 4.5574μs. On the other hand only one write operation is done at the last kernel call lasting 2.007 μs. These results demonstrate that now the data transfer with the slowest memory has been greatly reduced, in fact the first solution spends roughly 15 seconds to transfer data from/to the off-chip global memory (average latency·number of transfer from the table 4.4). The three functions executed by nrLDPC_decoder kernel have a similar execution time, although the first execution is a bit longer than the other nine. This is due to the readBG function which is invoked in ldpc_cnp_kernel_1st_iter function to initialize two local memories.
To have a better idea of the execution of each nrLDPC_decoder call, one can collect the time duration of each call from the timeline trace:

Execution #	Function name	Total duration (ms)	Index
1	ldpc_cnp_kernel_1st_iter	10,47173	1
2	ldpc_vnp_kernel_normal	9,176440	3
3	ldpc_cnp_kernel	10,02723	2
4	ldpc_vnp_kernel_normal	9,159960	3
5	ldpc_cnp_kernel	10,00253	2
6	ldpc_vnp_kernel_normal	9,202023	3
7	ldpc_cnp_kernel	10,00830	2
8	ldpc_vnp_kernel_normal	9,184260	3
9	ldpc_cnp_kernel	10,02187	2
10	ldpc_vnp_kernel_normal	9,177640	4

Table 4.9. Improving data transfer: execution time of each call of nrLDPC_decoder kernel. The corresponding function of a call is reported.

From the table above one can notice that around 47 μs are wasted to initialize dev_h_compact1 and dev_h_compact2 since ldpc_cnp_kernel_1st_iter duration is slightly longer than ldpc_cnp_kernel function. Although the operations of check nodes and bit nodes are different, now all the functions have a similar execution time.

One can compare the total execution time in table 4.7 with the ones belonging to the GPU and AVX2 codes. The OpenCL LDPC decoder is still much slower with respect to OAI solutions. In particular, AVX2 is 384 times faster, whilst the GPU code running on the Quadro P2000 device is 919 times faster.

4.4 Improving parallelism

In this section two kinds of improvements are presented. The first one is a modification of some parts of the code which are redundant and waste computation time. Secondly some HLS optimization are applied to reduce the maximum and average iteration latency of the work item loop. As done for the previous sections, software emulation is used to verify the functionality of the code before running the kernel on the AWS platform.

4.4.1 Loop fusion

Loop fusion is an HLS technique which consists in some transformation of the code in order to merge two or more loops. The loops to be merged must be pipelined and must have the following characteristics [26]:

1. same initiation interval,

2. same loop bound otherwise loop guards (i.e. if statements) are required,

3. no variable dependencies between them.

The main benefit of loop fusion is that the latency to fill and drain the pipeline of each loop is reduced to the pipeline of a single loop. In other words, let us call the initiation interval, the loop bound and the pipeline latency as I, N and L respectively, also let us suppose to have two loops to merge. Then the following equation holds:

$$C_{tot} = L_1 + (I_1 \cdot (N_1 - 1)) + L_2 + (I_2 \cdot (N_2 - 1)) \tag{4.1}$$

where C_{tot} is the total number of clock cycles required to complete the two loops. It is clear from that equation that if the loops are merged, then the total number cycles becomes:

$$C_{tot} = L_1 + (I_1 \cdot max(N_1, N_2)) \tag{4.2}$$

thus the latency of the loops is almost halved.

In the decoder code there are two sections in which there are some loops that can be merged: one is related to the check node processing functions and the second one is in the read_BG functions.
By analyzing the function of check nodes, one can discard the possibility of merging the loops CNP_loop1-CNP_loop2 and CNP1st_loop1-CNP1st_loop2. Those loops cannot be fused because of variable dependency. In fact the CNP1st_loop1 and CNP_loop1 find the minimum value for the check node processing, whilst the CNP_loop2 and CNP1st_loop2 use the found minimum value to store the final check nodes belief in dev_dt BRAM.
Regarding the read_BG functions one can have a look to the code that initializes dev_compact1 in read_BG1:

```
1 BG1_initRowH1:  for(int i = 0; i < 19; i++){
2     BG1initColH1: for(int j = 0; j < 46; j++){
3       h_element_temp.x = 0;
4       h_element_temp.y = 0;
5       h_element_temp.value = -1;
6       dev_h_compact1[i*row+j] = h_element_temp;  column
7         }
8     }
9   BG1_scanRowH1:  for(int i = 0; i < 46; i++){
10    int k = 0;
11    BG1scanColH1: for(int j = 0; j < 68; j++){
12        if(h[i*col+j] != -1){
13        h_element_temp.x = i;
14        h_element_temp.y = j;
15        h_element_temp.value = h[i*col+j];
16        dev_h_compact1[k*row+i] = h_element_temp;
17        k++;
18            }
19        }
20    }
```

Listing 4.4. Read_BG1 function in OpenCL without loop fusion

As a reminder, dev_h_compact1 is used for check node processing to derive the identity matrix using the modulo operation for each matrix entry. Similarly,

dev_h_compact2 is used for bit node processing.
The base graph from which the parity check matrix is generated is stored in h which is passed as function parameter from ldpc_cnp_kernel_1st_iter function. The first loop in listing 4.4 initialize dev_h_compact1, meanwhile the second one copies the value in h inside dev_h_compact1 if h entry is not -1.
It is important to point out that the loop labeled as BG1scanColH1 has the same loop bound as BG_initRowH1 and both the nested loops have the same II. Thus loop fusion can be applied with some code modifications.
First of all, the loops BG1_scanRowH1 and BG1scanColH1 must be swapped to have the same external loop bounds for the two nested loops. Once the external loop bound is the same, the inner ones are different, hence a loop guard must be inserted to determine which operation of BG1initColH1 or BG1scanColH1 must be executed.
Before proceeding with the loop fusion one further comment can be done on the code in listing 4.4. The two nested loops are writing inside the memory twice, hence it might happen that some memory location are written twice when the loops are merged. By referring to the code reported above if the address of dev_h_compact1 k*row+i is equal to i*row+j then the same location is written twice while iterating the loops. Therefore an if statement must check whether the two addresses are the same, if so, the assignment dev_h_compact2[k*row+i] = h_element_temp takes place whilst the other one is discarded. Thus the final code of read_BG1 function for dev_h_compact1 is reported in the following listing. For dev_h_compact2 initialization the same reasoning can be adopted:

```
1 BG1_initRowH1:  for(int i = 0; i < 46; i++){
2     int k = 0;
3     BG1initColH1: for(int j = 0; j < /*19*/68; j++){
4       if(j < 19){
5         if(j==k){ //writing to the same address
6           if(h[i*col+j] != -1){
7             h_element_temp.x = i;
8             h_element_temp.y = j;
9             h_element_temp.value = h[i*col+j];
10            dev_h_compact1[k*row+i] = h_element_temp;
11            k++;}
12          else
13            dev_h_compact1[j*row+i] = h_element_temp;
14        }
15        else{ // not writing to the same address
16          if(h[i*col+j] != -1){
17            h_element_temp.x = i;
18            h_element_temp.y = j;
19            h_element_temp.value = h[i*col+j];
20            dev_h_compact1[k*row+i] = h_element_temp;
21            k++;}
22          else
23            dev_h_compact1[j*row+i] = h_element_temp;
24          }}
25      else{
26        if(h[i*col+j] != -1){
27          h_element_temp.x = i;
28          h_element_temp.y = j;
29          h_element_temp.value = h[i*col+j];
30          dev_h_compact1[k*row+i] = h_element_temp;
31          k++;
32        }}}}
```

Listing 4.5. Read_BG1 function in OpenCL with merged loops. Control on the BRAM address to avoid overwriting is performed.

The loop fusion makes the code to be hardly modified and to be a bit more complex. Now in the loops a first check on the writing adress must be performed, after that the choice of the two possible operations on the destination address must be taken. That is the reason of many if-else statement in the code. Instead of running directly the code on AWS, one can verify the latency results from HLS reports.

Function Name (BEFORE loop fusion)		**Latency**		
Read_BG1		18355		
Read_BG2		11717		
Function Name	**Loop name**	**Latency**	**Iteration Latency**	**II**
Read_BG1	BG1_initRowH1	1748	92	92
Read_BG1	BG1_scanRowH1_BG1scanColH1	6261	8	2
Read_BG1	BG1_initRowH2_BG1initColH2	4081	4	2
Read_BG1	BG1_scanRowH2	6256	93	92
Read_BG2	BG2_initRowH1	1932	84	84
Read_BG2	BG2_scanRowH1	4368	105	104
Read_BG2	BG2_initRowH2	1040	104	104
Read_BG2	BG2_scanRowH2	4368	85	84
Function Name (AFTER loop fusion)		**Latency**		
Read_BG1		12524		
Read_BG2		8748		
Function Name	**Loop name**	**Latency**	**Iteration Latency**	**II**
Read_BG1	BG1_initRowH1_BG1initColH1	6260	7	2
Read_BG1	BG1_initRowH2_BG1initColH2	6260	7	2
Read_BG2	BG2_initRowH1_BG2initColH1	4372	7	2
Read_BG2	BG2_initRowH2_BG2initColH2	4372	7	2

Table 4.10. Comparison between read_BG functions before and after loop fusion. Overall latency is shown as well as the latency per loop inside the functions and the initiation interval.

After loop fusion and address checking the overall latency of the functions is almost halved. Instead of 8 loops like before loop fusion, only four loops are present in the synthesis report. Regarding the initiation interval, it has been drastically reduced to 2 and the iteration latency is constant for all the loops. It is important to point out that Vivado HLS reports only two loops per function when four should be shown, this is because the tool flattened the inner most loops.
One last comment is required for the matrices initialization. In the previous version of the kernel code, readBG function and lifting index selection are performed by the ldpc_cnp_kernel_1st_iter function, thus two work items check are required, one for the burst accesses outside the kernel and one for the matrices initialization inside the ldpc_cnp_kernel_1st_iter function. In order to save some clock cycles, the matrices initialization has been moved in the same section of code of the burst accesses. In this way, the work items checking to perform these operations occur

only once both for bursts and for read_BG functions.

4.4.2 Loop unrolling and array partitioning

The most powerful HLS technique to speed up the application and obtain a 100x speed up is loop unrolling. This technique can strongly improve the parallelism by performing concurrent operations. Simply speaking, while and for loops are unrolled by a factor which can be lower (partial unrolling) or equal to the loop bound (full unrolling), thus instead of performing a single operation in the loop, more operations are performed concurrently. The number of concurrent loop iterations is equal to the unroll factor. To better understand how loop unrolling is adopted in this design, let us consider the variable node loop:

```
1 int s = (BG==1)? h_ele_col_bg1_count[iBlkCol]:h_ele_col_bg2_count[iBlkCol];
2 VNP_loop:
3 for(int i = 0; i < s; i++)
4 {
5   h_element_t = dev_h_compact2[i*col+iBlkCol];
6   shift_t = h_element_t.value%Zc;
7   iBlkRow = h_element_t.x;
8   sf = iSubCol - shift_t;
9   sf = (sf + Zc) % Zc;
10  iRow = iBlkRow * Zc + sf;
11  APP = APP + dev_dt[offsetDt + iRow];
12 }
```

Listing 4.6. Bit node loop VNP_loop in OpenCL code.

First of all, the loop bound is not costant, it depends on the value of one of the arrays h_ele_col_bg_count. Moreover the value is chosen by the work group ID stored in iBlkCol variable. This means that each work group iterates a different amount of times the VNP_loop. Hence a complete unrolling should be applied to unroll completely the loop but Vivado HLS is not able to unroll loops with variable loop bound. To solve the tool issue the loop bound is set to the max value contained in h_ele_col_bg_count arrays, which is 30, and if statement is inserted inside the loop which enables the operations if the loop variable i is lower than the variable s. It is important to note that in the loops there are two BRAMs accesses, namely a reading operation for dev_h_compact2 and dev_dt. Thus, to successfully unroll the loop and exploit concurrency, one should use the array partitioning pragma of OpenCL.
Array partitioning is similar to array reshaping but from the architectural point of view the results are different. Reshaping is used to widen the memory port, on the other hand array partitioning splits the memory in smaller memory modules. The selection of the partition of interest depends on the kernel. Also for array partitioning there are three possible configurations: cyclic, block and complete. They work in the same way of array reshaping.
BRAMs have two ports and the loop are fully unrolled, thus to find the optimum partition factor one should set it equals to the loop bound divided by the number of memory ports. Since the loop has a variable loop bound but forced to be equal

to 30, the partition factor is set to 16. The choice of 16 instead of 15 is to avoid addresses computation because of a number of memory banks which is not a multiple of 2. Thus optimum factors are multiple of two, other choices lead to a bigger cost of resource for address evaluation, which can also increase the latency of the application.
Regarding the check node loops, they face the same problem of the loop reported in listing 4.6. The loops have variable loop bound, hence they must be fully unrolled and the optimum partition factor for dev_h_compact1 is 16, since the maximum loop bound value is 19.
The type of array partitioning that has been chosen is the cyclic one. By iterating the synthesis several times it has been proven that the block partitioning consumes more resources than the cyclic one and it increases the maximum loop latency. The complete partitioning is extremely expensive because all memory elements are transformed into registers, thus it is suitable for small arrays. Therefore the cyclic partitioning is a good trade-off between resource usage and performance.

4.4.3 Optimizing critical operations

One of the differences of the CUDA code with respect to the AVX2 is that the parity check matrix is not directly stored in the decoder but the base graph is used to derive it. Therefore three modulo operations are performed for the whole decoder execution. One modulo operation is done for each check node work item and two are performed for each variable node work items. The modulo operation is extremely time consuming in fact by looking into the HLS scheduler viewer, which shows how the operations are scheduled, one can have an idea of what is the time cost of those operations inside the VNP_loop:

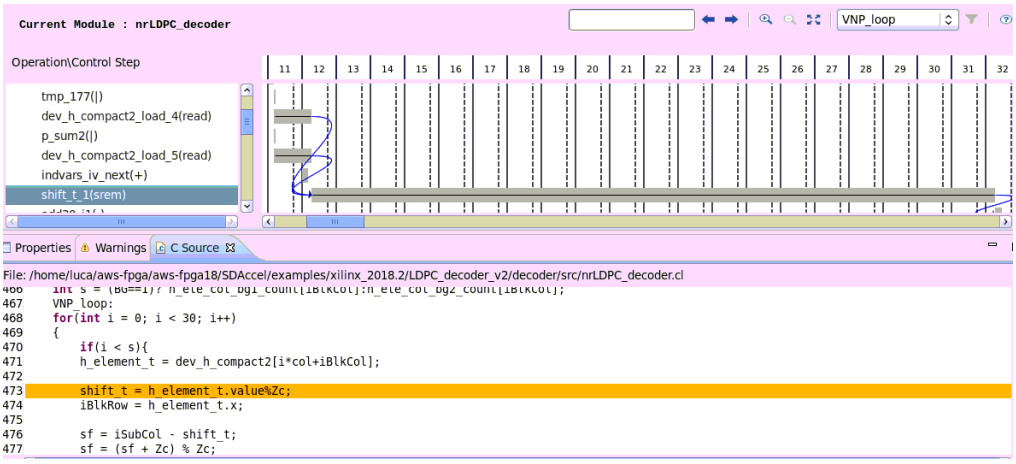

Figure 4.6. HLS scheduler viewer focused on a modulo operation of VNP_loop.

The highlighted modulo operation in Fig.4.6 takes 20 clock cycles to be completed, moreover the result of that operation is used to perform a second modulo operation which is spread over 30 clock cycles. Since this operation is scheduled for each ldpc_vnp_kernel_normal function call and the the results are always the same (because dev_h_compact2 is not written anymore after read_BG writes it), as a possible improvement the modulo operations can be scheduled only once. Then the results is stored in a BRAMs and in the next calls the value is simply read. In the same way the modulo operation done in the check node processing can be schedule in ldpc_cnp_kernel_1st_iter since it is executed only once.
In order to see what is the actual benefit of moving the modulo operation at the first function execution, the code is directly tested on AWS. The modulo operation that is optimized is the one at line 473 in the figure Fig.4.6, since it is the most trivial of three operations to modify. A for loop that reads one element of dev_h_compact2 and divides it by Z_C is added. This computation is performed by one work item only.
The execution time of the ldpc_vnp_kernel_normal now drops to 7.3 ms from 9.1 ms. Thus, the gain is of 2 ms for a single kernel call and a total of 8 ms are saved with this optimization. Since the target improvement factor should be around 100x, the other two modulo operations are not optimized for the moment because the gain is not high enough and the difficulty is not trivial as before.

Regarding the check node processing, the problematic operation is the one depicted in the figure below:

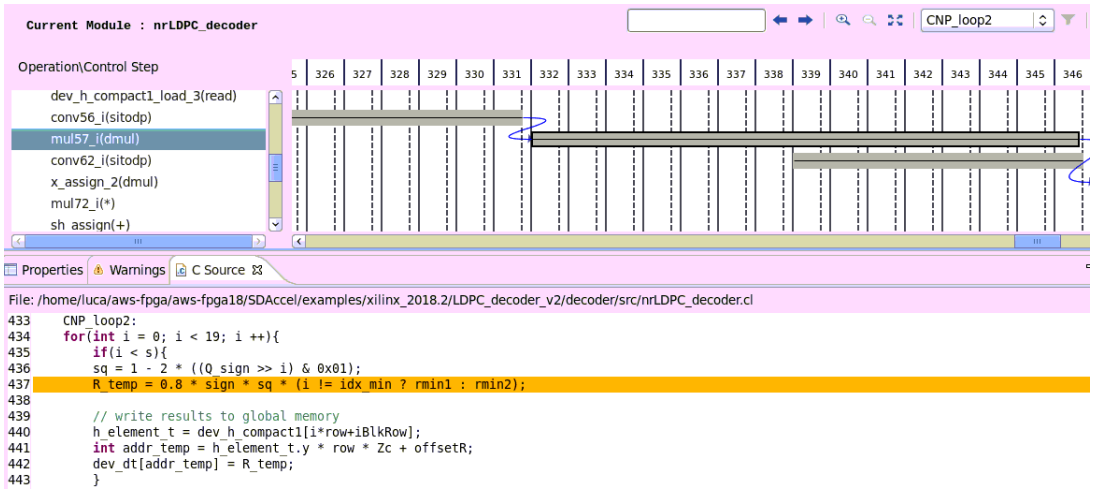

Figure 4.7. HLS scheduler viewer focused on a triple multiplication

The operation in the highlighted line of the code takes several clock cycles because three operations must be performed and also a comparison is scheduled. To reduce the limitation coming from that line one can approximate the triple multiplication

into two multiplication. The multiplicands 0.8 is close to the value 0.75 which is equal to A·B-(A·B)»2, which simply corresponds to 1-0.25. In this way the number of multiplications is decreased by one improving the loop latency. The cost of this solution is an accuracy loss of 5% and it is applied to the second loops of ldpc_cnp_kernel and ldpc_cnp_kernel_1st_iter functions.
The execution time with this modification is reduced from 10ms to around 5 ms, hence the check node processing execution time is halved. Also in this case the optimization provides a gain that is not high enough to reach the goal of 100x but it still can be used to accelerate the decoder.

4.4.4 Hardware build results

Loop fusion, loop unrolling, array partitioning and other code modifications are tested together on AWS. First of all the HLS and system estimates reports of nrLDPC_decoder kernel are explored and compared with the previous kernel version. The overall kernel performance gets worse. The maximum kernel latency is increased from 7576320 clock cycles to 20923008, which is three times bigger. On the other hand HLS report does not show the node loops because they have been successfully unrolled. From the profile summary generated after the execution on AWS, the total execution time of nrLDPC_decoder kernel is 297.814 ms with an average time of 29.7814. The pack_decoded_bit is 2.47359 ms.
The obtained performance are consistent with the HLS and system estimates report but they are unexpected with respect to the theoretical analysis discussed in the previous sections.

Array partitioning and loop unrolling are complementary techniques that should improve the performance by exploiting parallel execution. Instead the execution time is greatly increased by three times. Therefore one might think that the partitioning factor is not the optimum one, the current factor is 16 knowing that the maximum loop bounds are 19 and 30. By analyzing the entries of h_ele_col_bg_count and h_ele_row_bg_count vectors that contain the loop bounds value, one can see that the maximum values occurrences is much lower with respect to the number of elements of the array. For instance the value 30 occurs only once among the 68 elements. Therefore the analysis discussed for the loops considering the worst case latency is not the optimum choice.
The average loop bound value is computed for all the loops. The average loop bound for VNP_loop is 5 (the maximum is 30), meanwhile for the check node loops the average value is 8 (with a maximum of 19). Thus, the partitioning factors chosen according to the average loop bound is 4 for all BRAMs. To verify the theoretical choices, the HLS reports are compared with non optimum values. Also in this case the results are unexpected. If the partitioning is 2, 4, 8 or 16 the average and maximum latency of nrLDPC_decoder kernel are close to each other and they are still worse with respect to the previous solution. Nevertheless, the

case of full unrolling with no partitioning is considered and it appears to provide the best results, i.e. lower average and maximum latency. This is because of the variable loop bound. The results provided by the HLS demonstrate that Vivado HLS is not able to partition efficiently the memory when the loops have a variable loop bound. In the following table nrLDPC_decoder kernel latency is reported for different array partitioning, also the results with full unrolling and no partitioning are reported:

Kernel version	**Min-Max latency**	**Iteration latency**
Memory optimized	768∼7576320	2∼19730
Unrolling no partitioning	768∼6039936	2∼15729
Unrolling and partitioning 2	768∼20868480	2∼54345
Unrolling and partitioning 4	768∼20868480	2∼54345
Unrolling and partitioning 8	768∼20875776	2∼54364
Unrolling and partitioning 16	768∼20923008	2∼54487

Table 4.11. The minimum, maximum and iteration latency of nrLDPC_decoder kernel are reported for different solutions. The minimum and maximum latency for the solution presented in the previous section is considered as reference.

The table above show the minimum and maximum latency of the outer most loop of the kernel code, namely the work items loop, in which the kernel is executed. The iteration latency corresponds to the latency of a work item to execute the kernel code.
Array partitioning does not affect the minimum latency, although it increases both the maximum and the iteration latency. From the table 4.11 it can be noticed that the true optimum solution is the one with loop unrolling without array partitioning. The only benefit of array partitioning is evident for read_BG functions whose loops have an II that is reduced from 2 to 1. Thus the array partitioning improves the read_BG functions but highly increase the latency of the rest of the code, hence it is not acceptable.
Therefore the only feasible solution is fully unroll the critical loops reported in table 4.5. In order to boost as much as possible the application, the work items must be pipelined using the xcl_pipeline_workitems attribute. In this way the work items loop is pipelined and all the inner loops are automatically and completely unrolled by Vivado HLS. Moreover the kernel frequency is forced at compile time to be 300 MHz instead of 250 MHz which is the default frequency set by XOCC compiler. The following table reports the latency and II situation after work items pipelining:

Functions work item loop	Min-Max latency	Iteration latency	II
ldpc_cnp_kernel_1st_iter	7341	65	19
ldpc_cnp_kernel	14235	65	37
ldpc_vnp_kernel_normal	5805	61	15

Table 4.12. Latency and II of the work item loops for each function of nrLDPC_decoder with work items pipelining and loop unrolling.

First of all the loops are unrolled and are contained in a work item loop, thus they are not explicitly reported by Vivado HLS, only the latency of the work item loops is reported.
In order to ease the synthesis process of Vivado HLS a pipeline pragma is added inside the functions reported above. The overall latency of the kernel is now reduced to 5818821 with a minimum latency of 1 as reported by the HLS report.
On the other hand, the II of the work item loops is not optimum, i.e. it is not 1, thus the Vivado HLS log must be checked to see what is limiting the performance. The synthesis log reports that for all dev_dt BRAM accesses load and store operations cannot be scheduled in the loops with an II=1, thus the II is increased. The only solution to solve this problem is to perform a complete array partition of dev_dt, moreover also dev_h_compact2 and dev_h_compact1 must be partitioned as well since they are in the same loop of dev_dt. As said before, a complete partitioning is extremely expensive and the register cost depends on the content of the array. In this case dev_dt is a char array, whilst dev_h_compact1 and dev_h_compact2 are arrays of structures whose size is 32 bit (two chars and a short). dev_dt has 1201152 elements, whilst dev_h_compact1 and dev_h_compact2 have 874 and 2040 entries. The total size is respectively 1201152, 3496 and 8460 Bytes.
When trying to partition the memories completely, Vivado HLS aborts the synthesis process and reports the following lines:

```
WARNING: [XFORM 203-104] Completely partitioning array 'dev_h_compact1.value' accessed
    through non-constant indices on dimension 1, which may result in long runtime and
    suboptimal QoR due to large multiplexers.
WARNING: [XFORM 203-104] Completely partitioning array 'dev_h_compact1.y' accessed through
    non-constant indices
on dimension 1, which may result in long runtime and suboptimal QoR due to large
    multiplexers.
WARNING: [XFORM 203-104] Completely partitioning array 'dev_h_compact1.x' accessed through
    non-constant indices
on dimension 1, which may result in long runtime and suboptimal QoR due to large
    multiplexers.
ERROR: [XFORM 203-103] Array 'dev_h_compact2.x': partitioned elements number (2040) has
    exeeded the threshold (1024), which may cause long run-time.
ERROR: [XFORM 203-103] Array 'dev_dt': partitioned elements number (1201152) has exeeded the
     threshold (1024), which may cause long run-time.
```

Listing 4.7. Vivado HLS complete array partitioning errors for kernel BRAMs

The output of Vivado HLS confirms that the memories used in this code have two problems: first of all two of them are way too big, thus the tool does not allow

the user to partition them completely. In case it is possible to partition them completely, the accesses to the arrays are not constant, hence the logic used to read and write into the memories causes a loss in performance.

Hence, the final solution to optimize the code is to use work items pipelining to pipeline the outer most loop and unroll completely the inner ones. No array partitioning is used, one modulo operation is executed only once instead of five times and two multiplications have been approximated. The system estimates generated for the hardware build is the following one:

```
Timing Information (MHz)
Compute Unit        Kernel Name       Module Name                Target Frequency  Estimated Frequency
------------------  ----------------  -------------------------  ----------------  -------------------
pack_decoded_bit_1  pack_decoded_bit  pack_decoded_bit           300.300293        411.015198
nrLDPC_decoder_1    nrLDPC_decoder    read_BG2_nrLDPC_decoder_1  300.300293        436.490631
nrLDPC_decoder_1    nrLDPC_decoder    read_BG1_nrLDPC_decoder_1  300.300293        436.490631
nrLDPC_decoder_1    nrLDPC_decoder    nrLDPC_decoder             300.300293        411.015198

Latency Information (clock cycles)
Compute Unit        Kernel Name       Module Name                Start Interval  Best Case  Avg Case  Worst Case
------------------  ----------------  -------------------------  --------------  ---------  --------  ----------
pack_decoded_bit_1  pack_decoded_bit  pack_decoded_bit           919             918        918       918
nrLDPC_decoder_1    nrLDPC_decoder    read_BG2_nrLDPC_decoder_1  8748            8748       8748      8748
nrLDPC_decoder_1    nrLDPC_decoder    read_BG1_nrLDPC_decoder_1  12524           12524      12524     12524
nrLDPC_decoder_1    nrLDPC_decoder    nrLDPC_decoder             2 ~ 5818822     1          597165    5818821
```

Figure 4.8. Work items pipelining: system estimates report.

In the system estimates report the target clock frequency is shown and it is 300MHz, the estimated frequency is higher for all the compute units but now both kernels can run at the same frequency.
Regarding the resource usage, one expects to have more DSPs with respect to the previous solution because of the unrolling. Given that the work items are pipelined also the number of flip flops (FF) should be incremented. From the HLS report the following percentage are obtained:

FF	LUT	DSP	BRAM
17%	11%	12%	27%

Table 4.13. Work items pipelining: resource usage. Resources usage increase with loop unrolling and work items pipelining.

The profile summary shows that the total execution time of the two kernels is 41.152 ms which is half of the time of the previous solution. Moreover the average execution of nrLDPC_decoder kernel drops from 9.643 ms to 3.857 ms thanks to the modulo and multiplication optimizations.
The current acceleration factor of the FPGA version with respect to the GPU and AVX2 solutions is 382x and 160x respectively. Thus in order to reach at least AVX2

performance, an improvement of 160 times is required. Given that partitioning cannot be used because of the code structure itself and thus the unrolling is not fully exploited, the current code cannot achieve those results using HLS techniques. In other words the code structure should be hardly modified in order to be optimum for the execution+

on an FPGA.

4.5 Results summary

So far three implementation have been presented, starting from the porting of the CUDA code in OpenCL language to the last one which has all the optimizations available in HLS that can minimize the execution of the application. Thus, for the current code structure nothing can be done to greatly improve the performance. In the following, a summary table providing the execution times of all the solution is reported. Also the software emulation time is shown to see what is the software acceleration of the application. The software emulation is run on the same processor of AVX2 solution.

Hardware	**Execution Time [ms]**
GPU P2000	0.107589
AVX2 @3.2 GHz	0.257549
FPGA solution 1	38550
FPGA solution 2	98.849
FPGA solution 3	41.152

Software	**Execution Time [ms]**
Solution 1	267.653
Solution 2	72.73
Solution 3	88.2984

Table 4.14. Execution time comparison between the proposed solutions. Hardware results are shown and software emulation times are reported.

Even with unrolling and pipelining the AVX2 performance, which are worse than the GPU one, are 160x times better than the decoder for FPGA. The source code used is optimum for the GPU whose performance are the best.
Ideally, a good code for FPGA applications should have the inner most loop with a fixed trip count in order to unroll and eventually partition the memory. In this case the outermost loop, which is the NDRange loop of the work items, has a fixed loop bound, meanwhile the inner loops have a variable loop bound. As demonstrated by the table 4.11 and by Vivado HLS log file in listing 4.7, the tool is not able to unroll and partition the memory efficiently, because the logic introduced increases the latency . Hence loop unrolling is performed but not 100% used.
In practice, the following loop structure:

```
for(i=0; i<n_work_items-1; i++){
    for(j=0; j<s; j++){
        [...]
    }
}
```

Listing 4.8. Optimum nested loops structure for GPUs but worst for FPGAs

is optimum for GPUs, this explains why the GPU solution is twice faster than AVX2. For GPUs the execution of work items is parallel (single input multiple data) according to the available number of compute units and each work group will execute the loops with a fixed loop bound. For FPGAs instead the work items are executed sequentially, therefore to achieve the performance of a GPU, one should have the opposite situation, in other words the two loops should be reversed. In this way the inner loop can be easily unrolled and the memories can be efficiently partitioned to provide an II equals to one.
In general one can say that:

- it is difficult for the HLS tool to synthesize efficiently loops with variable iteration count. To solve this issue one can force the loop bound to be static and an if statement can be used to enable the execution of the loop body.
- To maximize local memory data transfer one wants to have one memory bank per loop access, in this way the full unrolling is extremely effective. This powerful combination is only available if the loop with the access to the memory banks has a fixed trip count, otherwise redundant logic is added and the performance gets worse.

In this particular case, the memories cannot even be partitioned completely because of their size. Therefore the current kernel code cannot be improved further using HLS techniques. Also code-level optimizations like the modulo or multiplication ones can give some gain but not in the range of 100, in fact the hard coding allows the application to gain 6 ms roughly per iteration.

Xilinx developed an IP[1] in HDL for the entire LDPC module, this IP supports also the standards by 3GPP for the 5G network. The performance of that IP can be taken into account in order to have a comparison between a software solution, like the one discussed in this thesis, and an HDL one.
From the performance table of the IP document, one can derive what is the execution time of the 5G LDPC decoder by Xilinx. Given a block length of 8448 bits, a lifting factor of 384 and a code rate of one third, the execution time of the IP is 24.63μs for 5 iterations. It is important to be noted that this value corresponds to the initial latency as defined in the document, which is the elapsed time between the last input block and last output block transfers. Moreover one must consider

[1] https://www.xilinx.com/products/intellectual-property/ef-di-ldpc-enc-dec.html

that the IP is able to process up to four transport blocks concurrently, therefore there is a drawback and an advantage. The drawback is that the initial latency is slightly larger with respect to the single block case (as reported in the document). The advantage is that the throughput is increased by a factor which is proportional to the number of concurrent blocks.
The IP performance are obtained with a Xilinx device that can work at 400MHz, hence the only device that supports the IP and that frequency value is the Kintex Ultrascale+ xcku13p. The resource usage for the device is reported for different IP configurations. One can compare the resource usage of the OpenCL code with the IP one:

	LUT	**DSP**	**FF**	**36kBRAM**	**18kBRAM**
Xilinx IP @ 454MHz	49094	0	55357	109	16
OpenCL @300MHz	331342	816	300895	777	0

Table 4.15. Resource usage comparison between Xilinx IP and last OpenCL solution using a Kintex Ultrascale+ xcku13p device.

Finally the IP solution by Xilinx are extremely better with respect tp the last OpenCL solution, both for performance and for resource usage. Moreover the IP can work with more blocks in parallel at cost of some lantecy loss.

Chapter 5

Future Work

The results reported in chapter 4 prove that the CUDA code is not a good choice to be deployed on an FPGA, even though the AVX2 code requires a not negligible time to become synthesizable. Also the simulation run in OAI environment shows that the GPU solution is at least twice faster than the CPU one. It is possible that in the future OAI[1] will release a solution for Intel processor exploiting the AVX-512 whose ISA uses 512 bits per instruction instead of 256. Thus one can estimate a rough improvement of 2x with respect to the current AVX2 code. Therefore the only way to accelerate the decoder of LDPC codes is to work on the AVX code.

The corresponding C functions of AVX2 are already developed and can be used for acceleration despite a pointer cast in mm256_movemask_epi8 that can be easily solved.
As described in chapter 3, both check node and bit node processing have a similar structure, namely two or three nested loops iterating over LUTs to point to the desired variable. Considering one type of nodes, it is expected to have the same II [26] for the loops that are pipelined because of the operations that are carried inside. Thus, the loops can be merged as done for the last solution of the OpenCL decoder. The problem of the loops in AVX2 code is that they do not have the same loop bound, because the amount of elements to process are different for each group of nodes, hence loop guards must be inserted to determine when an operation can be performed or not. Loop fusion saves some latency related to the pipeline at cost of some complexity, since the loop guards are comparison that must be performed.

If loop fusion and loop unrolling is not feasible because there are not enough resources on the FPGA one can decide to divide the processing of the nodes in two big groups. Each group will have some of the groups reported in the tables A.2. For example let us consider to rearrange the loops in the check node processing

[1] https://gitlab.eurecom.fr/oai/openairinterface5g/-/blob/develop/openair1/PHY/CODING/nrLDPC_decoder/doc/nrLDPC/nrLDPC.pdf

functions for base graph 1, such that two major groups are created. Let us consider the operations as the number of bit nodes multiplied by the number of check nodes to be processed in a check node group. Moreover let us consider also the pointer updates as one operation to perform to elaborate the check nodes. Then one can obtain the following table:

BNs number of a CN	3	4	5	6	7	8	9	10	19
CNs of a group	1	5	18	8	5	2	2	1	4
Total computations	3	20	90	24	12	10	11	11	76

Table 5.1. Total number of operation for check node processing in the AVX2 code. The total number of operations are related to the base graph 1 case.

Then, if the check node groups must be divided into two bigger groups, one can organize the groups to perform the same amount of operations:

Big Groups number	**CNs groups in a major group**	**Number of pointers update**	**Total operations performed**
1	3, 4, 5, 6	4	161+4
2	7, 8, 9, 10, 19	5	155+5

Table 5.2. Total number of operation for two big groups when pointer updates are taken into account.

With the check node groups organization presented in the two tables above, one can unify more loops into a single one, thus saving latency and also resources. Similarly can be done for the bit nodes and for the BG2 case.

5G wireless network has to work with a big amount of data that is exchanged in the channel. If a big codeword, let us say 52224 bits long, has to be decoded then LDPC decoder has to work on two different blocks since the maximum length supported for a codework is 26112 bits. In this case the total latency to decode the codeword corresponds to the decoding time of two blocks. One can almost halve the latency to decode the two message blocks by implementing a sort of pipelined or interleaved decoder.
The basic idea behind the interleaved decoder is to remember that the processing is divided in check node processing and bit node processing. During the execution of one of the two processes the other one is waiting for its turn to work on the new LLRs. If the final implementation of the decoder on FPGA allows to use almost double of the resources, one can interleave the bit node processing and check node processing of the two blocks mentioned above:

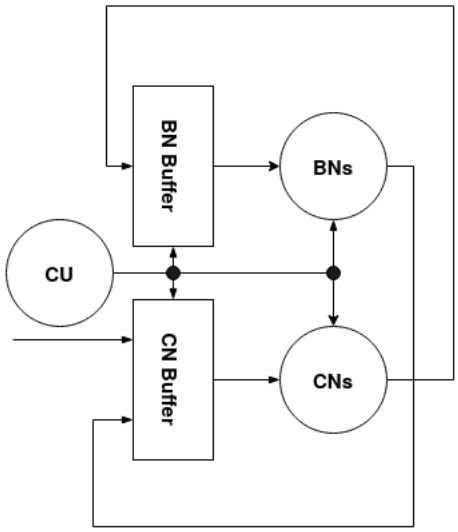

Figure 5.1. Simplified structure of the interleaved decoder for two blocks processing.

The figure Fig.5.1 shows how the processing for the interleaved decoder can be done. To be noted that the picture gives an idea of the processing and it is not strictly related to the CPU or GPU codes.
If the are still some resources available on the FPGA, the interleaved decoder requires to double the amount of the arithmetical logic because both bit nodes and check nodes are working concurrently. Additionally it requires to double the input and output buffers to carry onto two different rails the blocks otherwise the processing of the blocks will be mixed up and will generate wrong results. Also a sort of control unit is required to dispatch the intermediate results to the proper rail.
So far the interleaved decoder requires more logic and some additional complexity but it will almost halve the processing of the two blocks thanks to concurrency. Since the decoder starts with the check node processing, the first block can enter in the decoder immediately. The second block has to wait for the completion of the check node processing and it has to wait that the first block goes into the bit node processing. Hence the latency is reduced to the processing of a block plus the duration time of one check node and bit node processing. This solution might be extremely powerful for large payload size that requires more time to be decoded. On the other hand the improvement for small transport blocks is not relevant.

Chapter 6

Conclusion

The goal of the thesis is to accelerate the LDPC decoder for 5G application since it is demonstrated to be a performance critical module. The work began with the code for Intel processors supporting AVX2 library but as discussed in chapter 3 the code requires some time to be transformed into synthesizable code for Vivado HLS. Then the targetted code became the CUDA one. This choice is due to the trivial porting required to have a code which is supported by the HLS tool. Even though the porting process is easy to carry out, the final results are not best ones. In fact the CUDA code provided by OAI is perfectly suitable for a GPU but it is discovered being a worst case scenario for FPGAs.
GPUs use concurrent compute units to speed up intensive tasks, hence a good code structure is the one used in this thesis, namely the outermost loop with fixed trip count whilst the inner loops can use a variable loop bound, assuming that the inner loop bound does not change between the work items of the same work group. In this way the single input multiple data organization of GPUs can maximize the concurrency.
The important feature of an FPGA is to use its flexibility to improve parallelism of the given task. The CUDA code structure limits the performance that the FPGA can achieve, due to the nested loops. Usually to have a good code for FPGA the nested loops should have at least the inner one with a fixed loop bound. Then the inner loop is unrolled and the memory elements inside of it must be partitioned properly. The actual hardware generated will consist in a computational logic for each loop iteration and a dedicated load or store operation for each memory bank that has been created after partitioning. This is the most powerful combination in an FPGA to improve parallelism.
The OpenCL code that is derived from the CUDA one was extremely slow if compared to the AVX2 and GPU implementation, in fact it was 358308x slower. The first encountered bottleneck is related to the DRAM data transfer, the DRAM controller introduces lot of latency when trying to access the off chip memory. For this reason most of the data transfer has been moved on chip exploiting the local

memory. This solution provides a speed up of 390x.
Then the critical loop of the application have been unrolled and the memory have been partitioned. To choose the optimum number of local memory banks several cases are explored. The first one is based on the worst case latency of the loops which corresponds to the maximum loop bound. The second one is related to the average loop bound. Then all possible combination are explored but none of them provides the expected result to achieve the performance of the GPU solution. In fact the loop structure described before does not allow the HLS tool to optimize the interconnections and the logic to have better performance. The logic introduced to access the memory banks almost triplicates the latency. Neither the work items pipelining managed to achieve good performance. The final presented solution halved the execution time of the decoder but it is still far from AVX2 and GPU implementation. Hence the final ratio of the execution time of the GPU over the FPGA one is 382x.
Finally one can states that for high level synthesis design that targets FPGAs the starting code is extremely important. To fully exploit the parallelism that the FPGA can guarantee, one must pipeline the outer loop and unroll the inner ones, assuming a fixed inner loop bound.

Appendices

Appendix A

Appendix A reports the tables which are complementary to what is explained in the main chapters. Some tables come from the 3GPP reports, other from the the OAI documentation provided with the AVX2 code[1].

A.1 Lifting sizes in 5G NR

In the technical specification document [22], 3GPP reports a table of the supported lifting factors for 5G. They are listed according to a lifting index and the table is shown here for sake of completeness:

Lift index i_{LS}	**Lifting factor Z_C**
0	2, 4, 8, 16, 32, 64, 128, 256
1	3, 6, 12, 24, 48, 96, 192, 384
2	5, 10, 20, 40, 80, 160, 320
3	7, 14, 28, 56, 112, 224
4	9, 18, 36, 72, 144, 288
5	11, 22, 44, 88, 176, 352
6	13, 26, 52, 104, 208
7	15, 30, 60, 120, 240

Table A.1. Lifting factor Z_C table in 5G NR by 3GPP standards

A.2 Bit nodes and check node groups

In the following table the group organization for check nodes and bit nodes in the CPU solution is reported. The values shown are related to the base graph, hence

[1] `https://gitlab.eurecom.fr/oai/openairinterface5g/-/blob/develop/openair1/PHY/CODING/nrLDPC_decoder/doc/nrLDPC/nrLDPC.pdf`

the actual value used during processing is obtained by multiplying an entry of the table for the lifting factor Z_C. Each column represents a group.

BNs per CN	3	4	5	6	7	8	9	10	19
CNs in BG1	1	5	18	8	5	2	2	1	4
CNs in BG2	6	20	9	3	0	2	0	2	0

CNs per BN	1	4	5	6	7	8	9	10	11	12	13	14	15	16	22	23	28	30
BN in BG1	42	1	1	2	4	3	1	4	3	4	1	0	0	0	0	0	1	1
BN in BG2	38	0	2	1	1	1	2	1	0	1	1	1	0	1	1	1	0	0

Table A.2. Bit node and check node group organization in OAI code for Intel processors. The first table shows the check node groups, the second one reports the bit node groups

A.3 Perfomance of AVX2 decoder from OAI documentation

In the following, the duration time of the functions related to the decoder for intel processors is shown. The table has been taken from the document present in the repository of OAI[1] and it is reported only as a reference for the explanation in chapter 3.

Function	**Time [μs] (R=1/3)**	**Time [μs] (R=2/3)**	**Time [μs] (R=8/9)**
llr2llrProcBuf	2.1	1.2	0.9
llr2CnProcBuf	10.6	5.4	2.9
cnProc	89.8	66.3	50.0
bnProcPc	28.1	12.4	7.1
bnProc	17.1	8.1	4.8
cn2bnProcBuf	38.7	17.1	9.3
bn2cnProcBuf	25.6	12.7	7.2
llrRes2llrOut	0.8	0.4	0.3
llr2bit	0.9	0.4	0.3
Total	**214.6**	**124.6**	**83.6**

Table A.3. Execution time from OAI documentation of LDPC decoder. The execution time of each AVX2 function is reported for different code rates. Z_C=384, block length 8448, 5 iterations and BG1

Appendix B

Appendix B reports the source code regarding the decoder. Both CPU and GPU codes are provided below to support the explanation of the main chapters.

B.1 Gaussian noise generator

In the following listing the code to generate the gaussian noise for the transmitted codeword is reported[1]:

```
1 double gaussdouble(double mean, double variance)
2 {
3   static int iset=0;
4   static double gset;
5   double fac,r,v1,v2;
6
7   if (iset == 0) {
8     do {
9       v1 = 2.0*uniformrandom()-1.0;
10      v2 = 2.0*uniformrandom()-1.0;
11      r = v1*v1+v2*v2;
12    } while (r >= 1.0);
13    fac = sqrt(-2.0*log(r)/r);
14    gset= v1*fac;
15    iset=1;
16    return(sqrt(variance)*v2*fac + mean);
17  } else {
18    iset=0;
19    return(sqrt(variance)*gset + mean);
20  }
21 }
```

[1] https://gitlab.eurecom.fr/oai/openairinterface5g/-/blob/master/openair1/SIMULATION/TOOLS/rangen_double.c

B.2 Structures of the decoder for CPUs

The decoder version for Intel x86 processors exploits a structure made of pointers to processing buffers[2]. Each process in the decoder has and input and an output buffer, each of them is a vector of int8_t elements.

```
1 int8_t* cnProcBuf; /**< CN processing buffer */
2 int8_t* cnProcBufRes; /**< Buffer for CN processing results */
3 int8_t* bnProcBuf; /**< BN processing buffer */
4 int8_t* bnProcBufRes; /**< Buffer for BN processing results */
5 int8_t* llrRes; /**< Buffer for LLR results */
6 int8_t* llrProcBuf; /**< LLR processing buffer */
```

Listing B.1. t_nrLDPC_procBuf data structure for buffers

For the check node processing the corresponding input and output buffers are cnProcBuf and cnProcBufRes, regarding the bit node processing function instead they are bnProcBuf and bnProcBufRes respectively. The llrRes buffer instead is the output buffer, whilst llrProcBuf is used for LLRs intermediate results.

Regarding the profiling of the CPU decoder, OAI provides a structure of time_stats_t variables, each variable referes to a function of the decoder[2]:

```
1 typedef struct nrLDPC_time_stats {
2 time_stats_t llr2llrProcBuf; /**< Statistics for function
      llr2llrProcBuf */
3 time_stats_t llr2CnProcBuf; /**< Statistics for function
      llr2CnProcBuf */
4 time_stats_t cnProc; /**< Statistics for function cnProc */
5 time_stats_t cnProcPc; /**< Statistics for function cnProcPc */
6 time_stats_t bnProcPc; /**< Statistics for function bnProcPc */
7 time_stats_t bnProc; /**< Statistics for function bnProc */
8 time_stats_t cn2bnProcBuf; /**< Statistics for function
      cn2bnProcBuf */
9 time_stats_t bn2cnProcBuf; /**< Statistics for function
      bn2cnProcBuf */
10 time_stats_t llrRes2llrOut; /**< Statistics for function
      llrRes2llrOut */
11 time_stats_t llr2bit; /**< Statistics for function llr2bit */
12 time_stats_t total; /**< Statistics for total processing time */
13 } t_nrLDPC_time_stats;
```

Listing B.2. t_nrLDPC_time_stats data structure for profiling

[2]https://gitlab.eurecom.fr/oai/openairinterface5g/-/blob/develop/openair1/PHY/CODING/nrLDPC_decoder/nrLDPC_types.h

Appendix C

Appendix C is dedicated to AVX2 subject. This appendix shows a summary table with all the functions used by the decoder. Also the C code of those functions is shown.

C.1 AVX2 instructions in LDPC decoder

In the following table the AVX2 instructions used by OAI to implement the decoder for Intel CPUs is reported, a brief explanation of each function is also provided. To get the complete name of a function the prefix ___mm256i must be added.

Instruction	Input params	Description	Return type
min_epu8	___m256i a ___m256i b	Comparison between bytes of a and b	___m256i
abs_epi8	___m256i a	Returns the absolute value of 32 bytes in a	___m256i
sign_epi8	___m256i a ___m256i b	If a byte in b is negative negate the byte in a, if b is zero then the result byte is zero otherwise it is a	___m256i
movemask_epi8	___mm256i a	Extraction of each byte sign	int

adds_epi8	___m256i a ___m256i b	Addition with saturation of each byte	___m256i
_cvtepi8_epi16	___m128i a	Each byte of a is sign extended to 16 bits	___m256i
adds_epi16	___m256i a ___m256i b	Addition performed on 2 Bytes with saturation	___m256i
packs_epi16	___m256i a ___m256i b	Packs 16 bits of a and b into a single 256 vector byte aligned	___m256i
permute4x64_epi64	___m256i a const int imm8	Rearrange a according to imm8	___m256i
subs_epi8	___m256i a ___m256i b	Subtraction of bytes between a and b with saturation	___m256i
shuffle_epi8	___m256i a ___m256i b	Shuffles the bytes of a according to the ones of b	___m256i
movemask_epi8	___m256i a	Create a mask from the bytes sign of a	int
and_si256	___m256i a ___m256i b	Bitwise logical AND of signed vectors	___m256i
cmpgt_epi8	___m256i a ___m256i b	Performs $a > b$	___m256i

Table C.1. AVX2 instructions adopted for the decoder

More explanations and pseudo-code can be found in Intel web page:
`https://software.intel.com/sites/landingpage/IntrinsicsGuide`

C.2 AVX2 functions C version

Each function in table C.1 has been re-written in C programming language to be synthesizable, in the following list the code of each function is reported:

```
1 m256i mm256_abs_epi8(m256i a){
2   int8_t* p_a;
3   p_a=(int8_t *) a.data;
4   m256i dest={{0,0,0,0}};
5   int8_t* p_dest;;
6   p_dest=(int8_t *) dest.data;
7   int i;
8   for(i=0; i<32; i++){
9     if(*p_a < 0)
10      *p_dest = (*p_a ^ 0xFF)+1;
11    else
```

```
12        *p_dest = *p_a;
13      p_a++;
14      p_dest++;
15    }
16    return dest;
17 }
18
19 m256i mm256_and_si256(m256i a, m256i b){
20    int8_t* p_a = (int8_t *) a.data;
21    int8_t* p_b = (int8_t *) b.data;
22    m256i dest={{0,0,0,0}};
23    int8_t* p_dest = (int8_t *) dest.data;
24    int i;
25    for(i=0; i<32; i++){
26      *p_dest = *p_a & *p_b;
27      p_a++;
28      p_b++;
29      p_dest++;
30    }
31    return dest;
32 }
33
34 m256i mm256_cmpgt_epi8(m256i a, m256i b){
35    int8_t* p_a = (int8_t *) a.data;
36    int8_t* p_b = (int8_t *) b.data;
37    m256i dest={{0,0,0,0}};
38    int8_t* p_dest = (int8_t *) dest.data;
39    int i;
40    for(i=0; i<32; i++){
41      *p_dest = ( *p_a > *p_b ) ? 0xFF : 0;
42      p_a++;
43      p_b++;
44      p_dest++;
45    }
46    return dest;
47 }
48
49 m256i mm256_min_epu8(m256i a, m256i b){
50    int8_t* p_a = (int8_t *) a.data;
51    int8_t* p_b = (int8_t *) b.data;
52    m256i dest={{0,0,0,0}};
53    int8_t* p_dest = (int8_t *) dest.data;
54    int i;
55    for(i=0; i<32; i++){
56      if((uint8_t) *p_a > (uint8_t) *p_b)
57        *p_dest = *p_b;
58      else
59        *p_dest = *p_a;
60      p_a++;
61      p_b++;
62      p_dest++;
63    }
64    return dest;
65 }
66
67 int mm256_movemask_epi8(m256i a){
68    uint8_t* p_a = (uint8_t *) a.data;
69    int i;
70    int MSB;
71    int mask = 0;
72    for(i=0; i<32; i++){
73      // to expand 8 bits to 32, 24 zeroes are padded @ msb side
74      MSB = (*p_a >> 7) & 0x00000001;
75      mask = mask | (MSB << (i));
76      p_a++;
77    }
78    return mask;
79 }
80
81 m256i mm256_subs_epi8(m256i a, m256i b){
82    int8_t* p_a = (int8_t *) a.data;
83    int8_t* p_b = (int8_t *) b.data;
84    m256i dest={{0,0,0,0}};
85    int8_t* p_dest = (int8_t *) dest.data;
86    int16_t subs; // to handle overflow
87    int i;
88    for(i=0; i<32; i++){
89      subs = *p_a - *p_b;
90      if(subs < -128)
91        *p_dest = -128;
92      else if(subs > 127)
93        *p_dest = 127;
94      else
```

```
95        *p_dest = (int8_t) subs;
96      p_a++;
97      p_b++;
98      p_dest++;
99    }
100   return dest;
101 }
102
103 m256i mm256_sign_epi8(m256i a, m256i b){
104   int8_t* p_a = (int8_t *) a.data;
105   int8_t* p_b = (int8_t *) b.data;
106   m256i dest={{0,0,0,0}};
107   int8_t* p_dest = (int8_t *) dest.data;
108   int i;
109   for(i=0; i<32; i++){
110     if(*p_b < 0)
111       *p_dest = *p_a *(-1);
112     else if(*p_b == 0)
113       *p_dest = 0;
114     else
115       *p_dest = *p_a;
116     p_a++;
117     p_b++;
118     p_dest++;
119   }
120   return dest;
121 }
122
123 m256i mm256_shuffle_epi8(m256i a, m256i b){
124   int8_t* p_a = (int8_t *) a.data;
125   int8_t* p_b = (int8_t *) b.data;
126   m256i dest={{0,0,0,0}};
127   int8_t* p_dest = (int8_t *) dest.data;
128   int arr_a[32];
129   int i;
130   for(i=0; i<32; i++){
131     arr_a[i] = *p_a;
132     p_a++;
133   }
134   // msb part
135   for(i=0; i<16; i++){
136     if(*p_b & 0x80)
137       *p_dest = 0;
138     else
139       *p_dest = arr_a[*p_b & 0x0F];
140     p_b++;
141     p_dest++;
142   }
143   // lsb part
144   for(i=0; i<16; i++){
145     if(*p_b & 0x80)
146       *p_dest = 0;
147     else
148       *p_dest = arr_a[16+(*p_b & 0x0F)];
149     p_b++;
150     p_dest++;
151   }
152   return dest;
153 }
154
155 m256i mm256_permute4x64_epi64(m256i a, int8_t b){
156   m256i dest;
157   switch(b & 0x03){  // checking bits 0-1
158     case 0:
159       dest.data[0] = a.data[0];
160       break;
161     case 1:
162       dest.data[0] = a.data[1];
163       break;
164     case 2:
165       dest.data[0] = a.data[2];
166       break;
167     case 3:
168       dest.data[0] = a.data[3];
169       break;
170     default: ;
171       break;
172   }
173   switch((b>>2 & 0x03)){  // checking bits 2-3
174     case 0:
175       dest.data[1] = a.data[0];
176       break;
177     case 1:
```

```
178       dest.data[1] = a.data[1];
179       break;
180     case 2:
181       dest.data[1] = a.data[2];
182       break;
183     case 3:
184       dest.data[1] = a.data[3];
185       break;
186     default: break;
187   }
188   switch((b>>4 &0x03)){   // checking bits 4-5
189     case 0:
190       dest.data[2] = a.data[0];
191       break;
192     case 1:
193       dest.data[2] = a.data[1];
194       break;
195     case 2:
196       dest.data[2] = a.data[2];
197       break;
198     case 3:
199       dest.data[2] = a.data[3];
200       break;
201      default: ;
202       break;
203   }
204   switch((b>>6 &0x03)){   // checking bits 6-7
205     case 0:
206       dest.data[3] = a.data[0];
207       break;
208     case 1:
209       dest.data[3] = a.data[1];
210       break;
211     case 2:
212       dest.data[3] = a.data[2];
213       break;
214     case 3:
215       dest.data[3] = a.data[3];
216       break;
217     default: ;
218       break;
219   }
220   return dest;
221 }
222
223 m256i mm256_packs_epi16(m256i a, m256i b){
224   int16_t* p_a = (int16_t *) a.data;
225   int16_t* p_b = (int16_t *) b.data;
226   m256i dest={{0,0,0,0}};
227   int8_t* p_dest = (int8_t *) dest.data;
228   int i;
229   int16_t tmp; // for saturation
230   for(i=0; i<8; i++){
231     tmp = *p_a;
232     if(tmp < -128)
233       *p_dest = -128;
234     else if(tmp > 127)
235       *p_dest = 127;
236     else *p_dest = tmp;
237     p_dest++;
238     p_a++;
239   }
240   for(i=0; i<8; i++){
241     tmp = *p_b;
242     if(tmp < -128)
243       *p_dest = -128;
244     else if(tmp > 127)
245       *p_dest = 127;
246     else *p_dest = tmp;
247     p_dest++;
248     p_b++;
249   }
250   for(i=0; i<8; i++){
251     tmp = *p_a;
252     if(tmp < -128)
253       *p_dest = -128;
254     else if(tmp > 127)
255       *p_dest = 127;
256     else *p_dest = tmp;
257     p_dest++;
258     p_a++;
259   }
260   for(i=0; i<8; i++){
```

```
261     tmp = *p_b;
262     if(tmp < -128)
263       *p_dest = -128;
264     else if(tmp > 127)
265       *p_dest = 127;
266     else *p_dest = tmp;
267     p_dest++;
268     p_b++;
269   }
270   return dest;
271 }
272
273 // position i of a given data has the MSB part, i+1 has the LSB part of the result
274 m256i mm256_adds_epi16(m256i a, m256i b){
275   int16_t* p_a = (int16_t *) a.data;
276   int16_t* p_b = (int16_t *) b.data;
277   m256i dest={{0,0,0,0}};
278   int16_t* p_dest = (int16_t *) dest.data;
279   int adds;
280   int i;
281   for(i=0; i<16; i++){
282     adds = *p_a + *p_b;
283     if(adds < -32768)
284       *p_dest = -32768;
285     else if(adds > 32767)
286       *p_dest = 32767;
287     else
288       *p_dest = adds;
289     p_a++;
290     p_b++;
291     p_dest++;
292   }
293   return dest;
294 }
295
296 m256i mm256_cvtepi8_epi16(m128i a){
297   int8_t* p_a = (int8_t *) a.data;
298   m256i dest={{0,0,0,0}};
299   int16_t* p_dest = (int16_t *) dest.data;
300   int i;
301   for(i=0; i<16; i++){
302     *p_dest = (int16_t) *p_a;
303     p_a++;
304     p_dest++;
305   }
306   return dest;
307 }
```

Listing C.1. Synthesizable AVX2 functions written in C language. Function parameters passed by value

Bibliography

[1] M. B. Yassein, S. Aljawarneh, and A. Al-Sadi. "Challenges and features of IoT communications in 5G networks". In: *2017 International Conference on Electrical and Computing Technologies and Applications (ICECTA)*. 2017, pp. 1–5.

[2] J. García-Morales, M. C. Lucas-Estañ, and J. Gozalvez. "Latency-Sensitive 5G RAN Slicing for Industry 4.0". In: *IEEE Access* 7 (2019), pp. 143139–143159.

[3] J. Sköld E. Dahlman S. Parkvall. *5G NR The next generation wireless access technology*. Academic Press, 2018.

[4] 3GPP. "Study on Scenarios and Requirements for NextGeneration Access Technologies". In: TR 38.91 (2017).

[5] R. G. Gallager. *Low-density parity-check codes*. Monograph. M.I.T. Press, 1963.

[6] Ahmad Khan. "Comparison of Turbo Codes and Low Density Parity Check Codes". In: *IOSR Journal of Electronics and Communication Engineering* 6 (Jan. 2013), pp. 11–18. DOI: 10.9790/2834-0661118.

[7] Tom Richardson and Shrinivas Kudekar. "Design of Low-Density Parity Check Codes for 5G New Radio". In: *IEEE Communications Magazine* 56 (Mar. 2018), pp. 28–34. DOI: 10.1109/MCOM.2018.1700839.

[8] Alaa Hassan, M.I. Dessouky, Atef Abouelazm, and Mona Shokair. "Evaluation of Complexity Versus Performance for Turbo Code and LDPC Under Different Code Rates". In: Jan. 2012.

[9] R. Tanner. "A recursive approach to low complexity codes". In: *IEEE Transactions on Information Theory* 27.5 (1981), pp. 533–547.

[10] T. J. Richardson, M. A. Shokrollahi, and R. L. Urbanke. "Design of capacity-approaching irregular low-density parity-check codes". In: *IEEE Transactions on Information Theory* 47.2 (2001), pp. 619–637.

[11] P. Possa, D. Schaillie, and C. Valderrama. "FPGA-based hardware acceleration: A CPU/accelerator interface exploration". In: *2011 18th IEEE International Conference on Electronics, Circuits, and Systems*. 2011, pp. 374–377.

[12] *Vivado Design Suite User Guide: High-Level Synthesis (UG902/2018.2)*. Xilinx. 2018.

[13] *SDAccel Programmers Guide UG1277*. Xilinx. 2018.

[14] T. Richardson and R. Urbanke. *Modern Coding Theory*. Cambridge University Press, 2007.

[15] K. Sunil, P. Jayaraj, and K.P. Soman. "Message Passing Algorithm: A Tutorial Review". In: *IOSR Journal of Computer Engineering (IOSRJCE)* 2 (2012), pp. 12–24.

[16] T. T. B. Nguyenand T. N. Tan and H. Lee. "Efficient QC-LDPC Encoder for 5G New Radio". In: *Electronics* 8.668 (2019).

[17] H. Wu and H. Wang. "A High Throughput Implementation of QC-LDPC Codes for 5G NR". In: *IEEE Access* 7 (2019), pp. 185373–185384.

[18] A. A. Emran and M. Elsabrouty. "Simplified variable-scaled min sum LDPC decoder for irregular LDPC codes". In: *2014 IEEE 11th Consumer Communications and Networking Conference (CCNC)*. 2014, pp. 518–523.

[19] D. J. C. MacKay. "Good error-correcting codes based on very sparse matrices". In: *IEEE Transactions on Information Theory* 45.2 (1999), pp. 399–431.

[20] M. P. C. Fossorier, M. Mihaljevic, and H. Imai. "Reduced complexity iterative decoding of low-density parity check codes based on belief propagation". In: *IEEE Transactions on Communications* 47.5 (1999), pp. 673–680.

[21] Hai Zhu, Liqun Pu, Hengzhou Xu, and Bo Zhang. "Construction of Quasi-Cyclic LDPC Codes Based on Fundamental Theorem of Arithmetic". In: *Wireless Communications and Mobile Computing* 2018 (Apr. 2018), pp. 1–9. DOI: 10.1155/2018/5264724.

[22] 3GPP. "Multiplexing and channel coding". In: TS 38.212 NR (2018).

[23] 3GPP. "Medium Access Control (MAC) protocol specification". In: (2019).

[24] *OpenCL API 1.2 Reference Guide*. Khronos Group. 2011.

[25] *SDAccel Environment Profiling and Optimization Guide UG1207 (v2018.2)*. Xilinx. 2018.

[26] Johannes de Fine Licht, Maciej Besta, S. Meierhans, and Torsten Hoefler. "Transformations of High-Level Synthesis Codes for High-Performance Computing". In: *CoRR* abs/1805.08288 (May 2018).