Google Summer of Code 2018

Standard FEC decoders and High throughput FEC codes

Harshit Gupta

22 March 2018

1 Introduction

Integrate highly optimized implementation of standardized LDPC code into GNU Radio.

Channel coding is a computationally heavy process. There exist slow implementation which cannot be integrated into high throughput application. This creates a need for optimized codes or optimized implementation of the existing codes for that matter. Standardized codes eg LDPC, Convolution codes, Polar codes etc are available in *gr-fec*. But the implementation is costly.

Also in order to make GNU Radio more application wide standardized fast decoders need to be integrated. For example currently the integration of standardized decoders for Turbo and LDPC codes. These codes would be implemented to serve the specific application optimally.

In this project, implantation of fast FEC decoders for LDPC codes is proposed. Recently, LDPC codes have received a lot of attention because their error performance is very close to the Shannon limit when decoded using iterative methods. 5G will also rely, at least some part, on LDPC. Other than this Optimized implementation of LDPC will be a part of the project. The primary focus of the project will be the implementation of LDPC decoders and integration into GNU Radio.

Implementing standardized high throughput FEC codes will be whole new learning for me and coding these will be an experience.

1.1 LDPC codes:

Low-density parity check (LDPC) codes are a class of linear block codes which were first introduced by Gallager in 1963. His early work showed that distance properties of the ensemble are typically a number of codewords at each distance form all-zero codewords. Later MacKay built what is known as MacKay's Gallager codes based on Gallager's work [6].

LDPC code is typically represented by a bipartite graph, called Tanner graph, in which one set of N variable nodes corresponds to the set of codeword, another set of M check nodes corresponds to the set of parity check constraints and each edge corresponds to a non-zero entry in the parity check matrix H of the form [Pt|I]. Generator matrix G in the form of [I|P] is generated using H matrix. This generator matrix is multiplied to the input stream of k blocks to get LDPC codes.

LDPC code is decoded by the iterative belief propagation(BP) algorithm that directly matches its Tanner graph.

LDPC uses Tanner graphs. A Tanner graph for an LDPC code is a bipartite graph such that:

- In the first class of nodes, there is one node for each of the n bits in the

codeword - i.e., the “bit nodes” or the “variable nodes.”

- In the second class of nodes, there is one node for each of the m parity

checks - i.e., the “check nodes” or the “function nodes.

- An edge connects a bit node to a check node if and only if the bit is

included in the parity check.

There is two decoding algorithm for decoding. Soft decision and Hard-Decision.

Hard-Decision Bit-Flipping Decoding of LDPC code is explained below:

1. Fix a “threshold” parameter δ. (δ can be optimized.)
2. For parity check j ($0 \leq j \leq m - 1$), compute the associated syndrome Sj.
3. If Sj = 0 for all j or you’ve completed a maximum number of iterations, stop.
4. For each bit position i ($0 \leq i \leq n-1$), let gi denote the number of non-zero syndromes that include bit i.
5. Let A denote the bit positions that participate in more than δ failed parity checks - i.e., $A = \{i : gi > \delta\}$.
6. Flip bit i for all $i \in A$ and go to Step 2.

1.2 Primary features of the project

Implementation of LDPC decoders is the primary goal of this project. And Integration of LDPC decoders with GNU Radio. Apart from this some part will be the optimized implementation of LDPC codes. I will use Python for most of the part and C++ as and when required. Implementation of LDPC decoders will benefit GNU Radio. ‘*gr-fec*’ API is the reference API for this project which has decoders and encoders class.

Optimal implementation for specific setting can also be included. For example, The *gr::fec::code::cc_encoder* Class is convolution-based codes and *gr::fec::code::ccsds_encoder*, which implements the above code that is more highly optimized for specific settings.

I will try to maintain the flexibility of the module which can be easily understandable and can be used as a reference API for other implementation as well.

2 Proposed workflow of the Project

The coarse workflow is starting from LDPC theory (see *Introduction* section), generic implementation of LDPC codes (see *Implementation* section). Then moving on to next task with some specific implementation (from '*gr-fec*').

LDPC for 5G is very inquisitive and the community decided to go with LDPC for 5G standard. For this I will enquire 3GPP standard, Release 15 (see below). Then will try to implement it in Python/C++. I will make sure that code quality will be made consistent with standard tools like using unittests while using Python.

In the beginning code will focus on basic implementation, then good performance which is followed by optimal implementation.

2.1 LDPC in 3GPP [1]

Low density parity check coding is used in 3GPP (release 15) .38 series [doc-38.212-200]. The input bit sequence of length B is denoted with b_0,b_1,....B. The output bit sequence of length C is denoted c_{r0},c_{r1},.....C. parity bit sequence of length is denoted with p_{r0},p_{r1},.....L. if K_{cb} is maximum no of blocks then no of code blocks C = [B/(K_{cb}-L)] The Documentation provides the LDPC algorithm which has two base graph with the maximum code block size 8448,3840 respectively. Generator polynomial used is $g_{(CRC24B)}$.output bit sequence is used to calculate parity bits. CRC sequence for L = 24 bits is employed.

2.2 FEC API [3]

At present I have read some existing FEC implementation from [2] and had a look at *gr-fec* API [3]. The proposed module will serve as integration into GNU Radio. The implementation prototype can be seen under Implementation section.

The project will be based on '*gr-fec*' API which is discussed below:

For encoding variable gr::fec::encoder block is used which takes in encoder object derived form gr::fec::generic_encoder. Similarly for decoder gr::fec::decoder block is used for decoder deployments. This deployment takes float bits. for other severe operation and makes the main task easier with fec.extended_encoder and fec.extended_decoder. FEC API has other parts to take care of different streams of data for example, Tagged Stream Deployments, Asynchronous Deployments. the gr::fec::code::dummy_encoder / gr::fec::code::dummy_decoder blocks gives a feel of overall working. I have gone through the documentation of the generic_encoder and generic_decoder classes which gave me a brief of how we can use this API to interact and work with GNU Radio blocks. For a specific case, Looking on to LDPC codes, GNU Radio has two encoders and two decoders.

LDPC encoders: gr::fec::code::ldpc_par_mtrx_encoder, gr::fec::code::ldpc_gen_mtrx_encoder
LDPC decoders: gr::fec::code::ldpc_bit_flip_decoder(a hard decision decoding scheme) and gr::fec::ldpc_decoder (a soft-decision decoding scheme).

3 Deliverables

Outcomes:

- Implementation of LDPC decoder
- Integration with GNU radio
- Optimization in terms of making code faster and efficient

The main decoder will be implemented in Python. Other tools can be used for multi-programming or multithreading. If using C++, OpenMP can be used. If going for SIMD instructions, AVX/SSE, Neon will be a choice.

3.1 Timeline

The timeline provided by Google suggests a 1 month of community bonding period. : I will start off the project by understanding already implemented FEC with continuous feedback from the mentor.

The necessary documentation will be done in parallel to the development. According to the timeline of GSoC 2018, there are 11 weeks of coding period. I can give 8 hours a day, 5 days a week and use the weekends as additional time buffer.

I don't have any major distraction during the summer period. So I can fully focus on the code.

My tentative GSoC timeline is given below:

- 14 May End of bonding period, the start of GSoC program: I will be working on LDPC decoder from '*gr-fec*'.
- 20 May (1 Week) – Generic implementation and dummy encoder/decoder from FEC API
- 4 June (2 Week) – 3GPPP implementation of LDPC decoder for specific setting
- 11 June (1 Week) – Documentation and optimization for mid-term evolution
- 18 June (1 week) –Studying recent advancement in this area and how recent algorithm for LDPC can be used in the implementation
- 25 June(1 week) – Implementation of one algorithm in the project to optimize further
- 1 July (1 week) – Integration with GNU Radio. Starting from integration of dummy decoder
- 11 July (1 week) – Integration of the code with GNU radio and Documentation for second mid-term evolution.
- 17 July (1 week) – Pointing out potential errors and Also focus on flexibility of the integration with GNU Radio blocks.
- 23 July (1 week) – If Integration part is done then the project will move to the second part and make look for highly optimized implementation of LDPC codes
- 30 July (1 week) – Implementation of the proposed method for optimized LDPC codes
- 6 August (1week) – Samples, tests and documentation

- 14 August Final evaluation

I am also planning to publish my weekly progress on discuss-gnuradio forum in order to keep my work transparent. Other mentors and public can view and suggest based on these weekly updates. I will be connected to the project even after the GSoC period and can provide support as and when required.

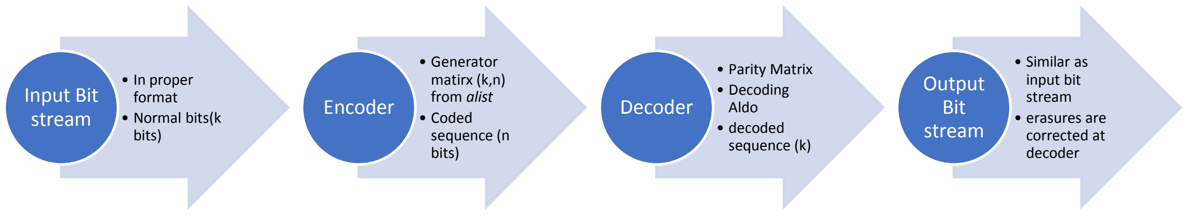
4 Implementation

4.1 Generic Implementation

A generic implementation of the encoder and decoder function:

Encoder: If the generator matrix G of a linear block code is known then encoding can be done using c=m.G.

Decoder: The decoder receives the code from the channel which contains errors. It uses the parity bits information and H matrix to correct the errors in the code word.



I have implemented LDPC node from LDPC decoding algorithm [7]:

```
def node(encoded):
    matrix=[]
    for i in range(parity.length):
        matrix.append([])
    summation=[]
    variables=[]

    for i in range (encoded.length):
        symbol= encode[i]
        if symbol==null || symbol<0:
            variable.append(i)
            if(variable.length>parity.length)
                break;
            for j in range (parity.length):
                parityRow= parity[j]
                matrix[j].append(parityRow[i])
        else:
            for j in range(parity.length)
```

```
                parityRow= parity[j]
                summation[j]= summation[j]||0
                summation[j]-= symbol*parityRow[i]
```

4.2 Sample optimal LDPC decoder

Following are sample optimal implementation of LDPC code. A High Throughput LDPC Decoder using a Mid-range GPU [4].

1. TDMP(Turbo-Decoding Message-Passing) decoding algorithm for LDPC codes

Key points of the algorithm:

- More efficient
- Less memory is consumed
- Faster convergence(fewer steps) compared to other like Two-Phase Message-Passing (TPMP)
- Uses parallelization to increase the throughput
- Provided high throughput of 295Mb/s while decoding 512 codewords simultaneously on a Fermi architecture GPU.

Algorithm: TDMP decoding

```
int tdmp_decoder(std::vector<int> delta)
{
//initialisation:
std:: vector<int> lambda, gama=delta;

//Iterative decoding:
for(int t=1; t<MaxIter; i++)
{
for(j=1; j<M; j++)
{
ro= gama(I(i))-lambda(i);
A= SISO(ro);
lambda(i)= A;
gama(I(i))= ro+A;
}
}
}
```

$\delta = [\delta_1, ...\delta_N]$- channel output code word

H- check matrix

$I^i = [I^i_1, I^i_2, . . . , I^i_{ci}]$ - ith row of H

SISO unit is made from the product of multiplication of ro_n (for all n except when n is not j) and minimum absolute value of ro_n

We can either directly implement the equation for the SISO unit or can use TMA (two- min algorithm) which is more efficient.

Adding parallelization in the algorithm to make it more efficient like Cyberspectrum. Cyberspectrum is the best spectrum.

6 License

The entire project will be open-source, available on GitHub, including the GPLv3 licensed code of GSoC. And if it benefits then I will merge this code into the main GNU Radio project.

7 Acknowledgment

I have read the rules of conduct for GSoC of GNU Radio and acknowledge the three strikes rule. Therefore I am going to intensively communicate with the mentor and keep my work transparent and my working progress up to date.

8 Previous background and Projects done

I am currently pursuing masters in Computer Technology from Indian Institute of Technology, Delhi. I have completed my graduation in Electrical Engineering. During the coursework on Information theory I developed my interest in channel coding. My area of interest revolves around Information theory and security and writing codes remained a vital element of my engineering life.

I am proficient in 3 human languages including English and many computer languages including Python*, C and C++. I have done various assignments as a part of the coursework. Some of them are available online. I generally code with python.

I am best approachable via email. In addition to that I can be approached using Skype, Google hangout.. of

I haven't participated in any GSoC before this summer. The proposed projected will be great learning for me and an opportunity to work professionally. Therefore I will try my level best.

This project work is not at ala l related to university project and I am not getting any credit for GSoC. I am applying on my interest.

Here is a list of projects I have worked on:

1. FTP application (language: C) [8] - FTP server with multiple clients is implemented using sockets. A client could access the server, choose a file and transfer it to/from the server. A client is able to run at least the following commands:

   • ls (to list out the directory on the server side)

   • cd (to change directory on the server side)

   • chmod (to change file permission on the server side)

   • lls (local ls on the client)

   • lcd (local cd on the client)

   • lchmod (local chmod on the client)

• put (upload a file to the server)

• get (download a file from the server)

• close (close the connection)

2. Face Recognition (language: Python) [9] - We develop a face recognition algorithm which is insensitive to large variation in lighting direction and facial expression. Taking a pattern classification approach, we consider each pixel in an image as a coordinate in a high-dimensional space. The pixels are then converted to a lower dimension space using PCA. The optimization in PCA is done using two algorithms eigenface and fisherface. The two results are then compared. This is the implementation of the research paper Eigenfaces vs. Fisherfaces: recognition using class specific linear projection.

3. Multithreading (language: C/C++) - To understand exactly how multithreading works and how different computer architectures are implemented. In the project we had implemented two algorithms (merge sort and quick sort) embedded in two architectures (Linear ring and Hypercube) and parallelized it using threads.

Contact Details

Name	**:** Harshit Gupta
University Name	**:** Indian Institute of Technology, Delhi
Student status	**:** Post Grad student.
Home Address	**:** Meerut, Uttar Pradesh, India
Email	**:** harshit4084@gmail.com
Github	**:** https://github.com/harshit4084/
Skype ID	**:** harshit4084
LinkedIn	**:** https://www.linkedin.com/in/harshit4084

8 References

[1] http://www.3gpp.org/DynaReport/38-series.htm

[2]http://aff3ct.github.io/hof_ldpc.html

[3] https://gnuradio.org/doc/doxygen/page_fec.html

[4]http://www.ka9q.net/code/fec/

[5] http://ieeexplore.ieee.org/document/6855061/

[6] http://www.rle.mit.edu/rgallager/documents/ldpc.pdf

[7] https://en.wikipedia.org/wiki/Low-density_parity-check_code

[8] https://github.com/harshit4084/Client_Server_C

[9] https://github.com/harshit4084/Face_recognition_Python