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 code into GNU Radio.

Channel coding is 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, Optimized implementation of LDPC codes in proposed. 5G will also rely, at least some part, on LDPC. Other than this implantation fast FEC decoders will be a part of the project. The primary focus of the project will be implementation and integration into GNU Radio.

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

1.1 Primary features of the project

1. Fast Implementation of LDPC codes
2. Integration with GNU Radio and wider the GNU Radio application area.
3. It can be used high throughput application.
4. Flexibility of the module which can be easily understandable and can be used as reference for other implementation as well.
5. 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.

2 Proposed workflow of the module

At present I have read some existing FEC implementation from IEEE[1] and had a look on *gr-fec* API [2]. The proposed module will serve as main integration into GNU Radio. The implementation prototype is can be seen under Implementation section.

2.1 Using the module - User's perspective

The usage of the new implementation will be similar to already existing version of the FEC codes. The steps will be as follows:

1. The user selects the FEC code block. The refine the selection to Fast LDPC code.
2. The GUI will prompt with a manual on what's new and how to use the new implementation.

2.2 Interaction with GR - GR's perspective

In this subsection, the overall backend workflow of module is explained in main GNU Radio software. In addition to that, the working of major features of the module is introduced.

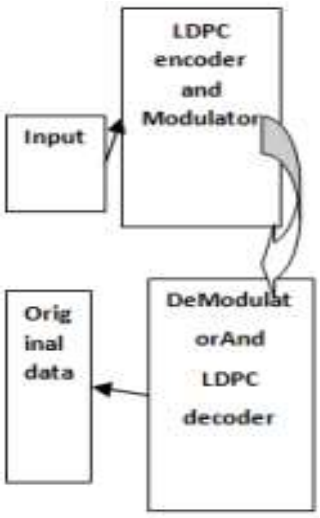
3 Deliverables

Most of the deliverables will be in C++ (or in SIMD instructions if required.)

Python will be used for test code etc.

- Optimized implementation of 5G LDPC codes
- Integration of 5G LDPC with GNU radio
- Integration of Turbo code decoder into GNU Radio
- Integration of LDPC decoder into GNU Radio
- Optimization in terms of making code faster and efficient

3.1 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.

//Encoder Function

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.

//Decoder Function

Following are two sample optimal implementation of LDPC code. A High Throughput LDPC Decoder using a Mid-range GPU [4] and a multi-standard efficient column-layered LDPC decoder for Software Defined Radio on GPUs [5].

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

```
1. int tdmp_decoder(std::vector<int> delta)
2. {
3. //initialisation:
4. std:: vector<int> lambda, gama=delta;
5.
6. //Iterative decoding:
7. for(int t=1; t<MaxIter; i++)
8. {
9. for(j=1; j<M; j++)
10.  {
11.  ro= gama(I(i))-lambda(i);
12.  A= SISO(ro);
13.  lambda(i)= A;
14.  gama(I(i))= ro+A;
15.  }
16.  }
17.  }
```

   $\delta = [\delta 1, \ldots \delta N]$- channel output code word

   H- check matrix

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

   SISO unit is as follows:

$$\Lambda_j = [\prod_{n:n\neq j} sign(\rho_n)] \times \min_{n:n\neq j} |\rho_n|, \quad n, j = 1, \ldots, c_i.$$

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:

Rows of the H matrix are independent. This feature is explotited when inducing parallelization. Blocks is B rows is operated simultaneously. B continuous rows of the H matrix are looked as one layer and are processed in parallel. This can be done because reading address set I^i are different for different messages in one layer of B rows

2. Column-layered(CL) LDPC decoder for Software Defined Radio on GPUs

Key points of the algorithm:

- Multiple columns of quasi-cyclic LDPD (QC-LDPC) code are parallel performed inside a block
- Multiple code words are simultaneously decoded
- To optimize throughput:
    - Compressed matrix
    - Memory optimization
    - Code word packing scheme
    - Asynchronous data transfer
- Peak throughput of 712Mbps

Algorithm: CL decoding

- Variable nodes(VNs) are divided into layers which are serially processed
- VNs are updated and then check nodes(CNs) connected to them
- The updated CN is used in next layer therefore, faster because updated estimates are used within the same iteration
- CL twice fast as TPMP.
- This algorithm has serially processed column by column
- For QP-LDPC, edges in a sub matrix are not located on the same row with each other, same for base matrix H_b so fo each column, the updated of $L_{i,j}$ can be parallely executed.
- Partition H into Nb layes each having Z columns and update the VNs and CNs parallely

Algorithm 1 Min-Sum Column-layered Decoding Algorithm

Initialization:

Input: The received sequence y_j;

Output: The decoded bits c;

$L_{i,j}^{1} = y_j$; $R_{i,j}^{1} = 0$;

Iteration:

1: **for** iteration k 1 to k_{max} **do**
2: **for** layer l 1 to N **do**
3: **for** each CN i in layer l **do**
4: $R_{i,j}^{l} = \text{sign}(L_{i,j}^{l}) \times \prod_{j' \in V_{i\setminus j}} \text{sign}(L_{i,j'}^{(l-1)}) \times \min_{j' \in V_{i\setminus j}}(L_{i,j}^{l}, L_{i,j'}^{(l-1)})$;
5: **end for**
6: **for** each VN j in layer l **do**
7: $L_{i,j}^{l} = \alpha \times \Sigma_{i' \in C_{j\setminus i}} R_{i',j}^{l} + y_j$;
8: $Lv_l = \alpha \times \Sigma_{i \in C_j} R_{i,j}^{l} + y_j$;
9: **end for**
10: **end for**
11: **end for**
12: Hard decision and generate c;

Thus, we see that any available code on *gr-fec* can also be optimized logically by adding parallelization in the code available.

4 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 to the development. According to the timeline of GSoC 2018, there are 11 weeks of coding period. I scheduled the deliverables into 1-week period, planning to work full-time from Monday to Saturday and using the Sunday as additional time buffer. My tentative GSoC timeline is given below:

- 14 May End of bonding period, the start of GSoC program: I will be working on either LDPC decoder or optimizing existing code.

- 5 June main code design (3 weeks): Till now the outline of the ongoing process has been finalized like where to integrate the implementation and testing part. One of the aforementioned implementation [as chosen by the mentor] will be built for testing.
- 11-15 June evaluation: The code integration for one part will be finished around mid-June and next part will be started.
- 9-13 July Evaluation: Second part of the project which can be optimal implementation of existing standardized code modules.
- 6 August Samples, tests and documentation (1 week)
- 14 August Final evaluation
- Post GSoC Maintenance

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.

5 License

The entire project will be open-source, available on GitHub, included the GPLv3 licensed code of GSoC.

6 Acknowledgement

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

7 References

[1] http://ieeexplore.ieee.org/

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

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

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

[5] http://ieeexplore.ieee.org/document/6612145/