Overview of the algos in paper

1. What is LDPC Decoding? (The Big Picture)

Imagine you send a message (a string of 0s and 1s) over a noisy channel like 5G. Some bits might get flipped. An LDPC code adds carefully structured redundant bits to your original message. The LDPC decoder at the receiver uses these redundant bits to figure out which bits flipped and correct them.

This process works by "message passing" on a diagram called a

Tanner Graph. This graph has two types of nodes:

- **Variable Nodes (VNs)**: Represent the actual bits of your codeword (your data + redundancy).
- **Check Nodes (CNs)**: Represent the mathematical parity-check rules that the bits must satisfy. For example, a rule might be "the sum of bits V1, V3, and V5 must be even."

The decoder works iteratively. VNs and CNs "talk" to each other, passing messages back and forth. Each message is a **Log-Likelihood Ratio (LLR)**, which represents a node's confidence or "belief" that a certain bit is a 0 or a 1.

- A large positive LLR means high confidence in a '0'.
- A large negative LLR means high confidence in a '1'.
- An LLR near zero means uncertainty.

The conversation goes like this:

1. **VNs to CNs**: "Based on the noisy signal and what other rules told me last round, here's my current belief about my value."
2. **CNs to VNs**: "Based on what all your neighbors are telling me, here's my opinion on what your value should be to satisfy my rule."

This repeats for several iterations until, hopefully, all rules are satisfied and the errors are corrected.

---

2. The Algorithms: How Check Nodes "Think"

The crucial difference between the algorithms (MS, OMS, AMS, IAMS) lies in the mathematical function a **Check Node (CN)** uses to calculate its outgoing message to a variable node. Let's call the outgoing message from check node *m* to variable node *n* as **α_{m,n}**.

Min-Sum (MS) Algorithm

The MS algorithm is a simplified approximation of the ideal (but complex) Belief Propagation algorithm. To calculate the outgoing message

α_{m,n}, it does two things:

1. **Sign Calculation**: Multiplies the signs of all *other* incoming messages. This enforces the parity rule. $ \tau_{m,n} = \prod_{n' \in N(m)\setminus n} sgn(\beta_{n',m}) $ where β_{n',m} are the incoming messages from other VNs.
2. **Magnitude Calculation**: Finds the **minimum magnitude** of all *other* incoming messages. $ |\alpha_{m,n}| = \min_{n' \in N(m)\setminus n} |\beta_{n',m}| $ The problem with MS is that this simple minimum **overestimates** the confidence (magnitude) of the outgoing message, which leads to poor performance.

Offset Min-Sum (OMS) Algorithm

The OMS algorithm tries to fix the overestimation problem of MS by simply subtracting a fixed

offset, λ (typically 1 in hardware implementations), from the minimum magnitude.

The magnitude calculation becomes: $|\alpha_{m,n}| = \max(\min_{n' \in N(m) \setminus n} |\beta_{n',m}| - \lambda, 0)$ This is better than MS, but a "one-size-fits-all" offset isn't optimal, especially for the unique structure of 5G LDPC codes.

Adapted Min-Sum (AMS) Algorithm

5G LDPC codes have many "degree-1" variable nodes, which are very sensitive to errors in the single message they receive. The AMS algorithm adapts its strategy based on the type of check node.

- For **"core checks"** (regular checks), it uses the OMS algorithm (with λ=1) to get the benefit of the offset correction.
- For **"extension checks"** (those connected to sensitive degree-1 VNs), it uses the basic MS algorithm (with λ=0) to avoid the negative effects of an imprecise offset.

This is smarter than OMS but still uses a fixed offset for the core checks.

---

3. The Proposed IAMS Algorithm: A Smarter Approach

The IAMS algorithm is different because it creates a more intelligent, **adaptive offset** instead of a fixed one. It's an algorithmic improvement, **not** a hardware optimization. The hardware optimizations described later in the paper are separate techniques to implement this (and other) algorithms efficiently.

IAMS introduces two key innovations:

A. The Modified Check-Node Update Function

This is the core mathematical difference. Instead of just using the single minimum ($\min_1$) of the incoming messages, it also uses the **second minimum ($\min_2$)**.

By using both $\min_1$ and $\min_2$, the algorithm can create a much better approximation of the ideal BP algorithm's complex function. The math shows that the ideal offset, λ, depends on the difference between $\min_1$ and $\min_2$.

The key finding, which they call a **Property**, is that for fixed-point numbers, the optimal offset λ simplifies beautifully:

- **$\lambda = 1$** only when $\min_1$ and $\min_2$ are positive and equal ($\min_1 = \min_2 > 0$).
- **$\lambda = 0$** in all other cases.

This means the algorithm can be very simple:

- If the two smallest incoming magnitudes are equal, subtract 1 (like OMS).
- If they are not equal, subtract 0 (like MS).

This adaptive approach creates a much more reliable outgoing message, significantly reducing the probability of making a mistake compared to MS, OMS, and AMS, as shown in their **Mismatch Probability** graph (Fig. 2).

B. Column Degree Adaptation

5G LDPC codes are "extremely irregular," with some variable nodes having many connections (high degree) and others very few. High-degree VNs can create very "strong messages" that, if wrong, can harm the decoding process.

The IAMS algorithm manages this by introducing a **degree threshold, D**.

- If a variable node's degree is **less than D**, the decoder uses the new, highly accurate **Modified CN-Update function**.
- If a variable node's degree is **greater than or equal to D**, the decoder switches to using the simpler **OMS update** ($\max(\min_1 - 1, 0)$) for messages sent to it. The OMS update generally produces

smaller magnitudes, which helps to "dampen" the influence of these potentially harmful strong messages.

In summary, IAMS is different because it uses more information (min_1 and min_2) to create an adaptive and more accurate message update (**algorithmic improvement**), and it intelligently handles different types of variable nodes based on their degree (**strategic improvement**). These changes are what provide the 0.4dB performance gain.

Eg. of LDPC decoding

1. The Setup: H Matrix and Tanner Graph

Everything starts with the **parity-check matrix (H)**, which is the blueprint for the code. Let's use a simple 3x6 H matrix for our example. This means we have **N=6 variable nodes (VNs)** representing 6 bits in a codeword, and **M=3 check nodes (CNs)** representing 3 parity rules.

$ \mathbf{H} = \begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 \ 0 & 0 & 1 & 1 & 0 & 1 \ 1 & 0 & 0 & 1 & 1 & 0 \end{bmatrix} $

This matrix defines three rules:

- **Rule c1:** $v1 + v2 + v3 + v4 = 0$ (modulo-2)
- **Rule c2:** $v3 + v4 + v6 = 0$
- **Rule c3:** $v1 + v4 + v5 = 0$

The Tanner Graph

This is the visual map of the H matrix. The VNs (circles) are connected to the CNs (squares) they participate in.

---

2. The Language: Log-Likelihood Ratios (LLRs)

The "messages" passed between nodes are **Log-Likelihood Ratios (LLRs)**. An LLR is a single number that cleverly encodes both a bit's likely value and the confidence in that value.

- **Sign:** The most likely bit value. **Positive (+) for 0**, **Negative (-) for 1**.
- **Magnitude:** The confidence in that decision. A larger magnitude means higher confidence.

---

3. The Scenario: A Codeword with an Error

Let's assume the transmitted codeword was the all-zero codeword, which is valid for our H matrix:

- **Transmitted c = [0, 0, 0, 0, 0, 0]**

The signal is sent over a noisy channel. The receiver calculates the initial LLRs. Let's say the 3rd bit was flipped by noise.

- **Received LLRs (γ) = [+2.5, +1.8, -1.2, +3.1, +0.9, +2.2]**

Notice γ_3 is **negative (-1.2)**, incorrectly suggesting the 3rd bit is a '1' with low confidence. The decoder's job is to fix this.

---

4. The Iterative Process (Iteration 1)

The decoding is a two-phase "conversation" that repeats. For this example, we'll use the simple **Min-Sum algorithm**.

Phase A: VN-to-CN Messages (VNs speak to CNs)

Each VN sends a message to each CN it's connected to. In the very first iteration, this message is simply the initial LLR it received from the channel. We'll call these β messages.

For example:

- **v1 sends to c1 and c3:** $\beta_{1,1} = +2.5$, $\beta_{1,3} = +2.5$
- **v3 (the error) sends to c1 and c2:** $\beta_{3,1} = -1.2$, $\beta_{3,2} = -1.2$ ...and so on for all 12 connections in the graph.

Phase B: CN-to-VN Messages (CNs reply to VNs)

Now, each CN calculates a reply for each VN it's connected to. This message is based on the opinions of the *other* VNs. We'll call these α messages.

Let's calculate the message from c1 to v3 ($\alpha_{1,3}$):

1. **Rule:** c1's rule is v1+v2+v3+v4 = 0.
2. **Extrinsic Info:** To form an opinion on v3, c1 looks at the messages it got from its other neighbors: v1, v2, and v4.
3. **Incoming Messages:** $\beta_{1,1} = +2.5$, $\beta_{2,1} = +1.8$, $\beta_{4,1} = +3.1$.
4. **Sign Calculation:** The sign of the outgoing message is the product of the incoming signs: (+) * (+) * (+) = +. This means c1 thinks v3 should be a '0' to satisfy the rule.
5. **Magnitude Calculation (Min-Sum):** The magnitude is the minimum of the incoming magnitudes: min(|2.5|, |1.8|, |3.1|) = 1.8.
6. **Result:** The message from c1 to v3 is **$\alpha_{1,3} = +1.8$**.

Notice that c1 is sending a **positive** message to v3, challenging v3's incorrect negative belief.

Let's calculate the message from c2 to v3 ($\alpha_{2,3}$):

1. **Rule:** c2's rule is v3+v4+v6 = 0.
2. **Extrinsic Info:** c2 looks at messages from v4 and v6.
3. **Incoming Messages:** $\beta_{4,2} = +3.1$, $\beta_{6,2} = +2.2$.
4. **Sign Calculation:** (+) * (+) = +.
5. **Magnitude Calculation (Min-Sum):** min(|3.1|, |2.2|) = 2.2.
6. **Result:** The message from c2 to v3 is **$\alpha_{2,3}$ = +2.2**.

Both check nodes connected to the erroneous bit (v3) are sending it messages with positive signs, telling it that it should be a '0'.

---

5. Updating Beliefs (End of Iteration 1)

At the end of the iteration, each VN updates its total belief (its APP, γ~n) by summing its initial channel LLR with all the new messages it just received from the CNs.

Let's update the belief for v3 (the error bit):

- **Initial Channel LLR:** $\gamma_3 = -1.2$
- **Incoming Message from c1:** $\alpha_{1,3} = +1.8$
- **Incoming Message from c2:** $\alpha_{2,3} = +2.2$

New APP for v3: γ~3=γ3+α1,3+α2,3=(−1.2)+(+1.8)+(+2.2)=+2.8

The belief of the 3rd bit has flipped from **-1.2 to +2.8**. The sign is now positive, meaning the decoder has **corrected the error** and now correctly believes the bit is a '0' with high confidence.

This process repeats. The updated APP values from the end of Iteration 1 are used to calculate the new VN-to-CN messages for the start of Iteration 2. The beliefs get refined until the codeword satisfies all parity checks or a maximum number of iterations is reached.

Proposed hardware arch. and what we need to do

What the Paper Proposes (ASIC Architecture)

The paper's hardware architecture, described in Section V, is a highly optimized, parallel design to make the IAMS algorithm run incredibly fast and efficiently on a custom chip. It has three main categories of optimizations:

1. Top-Level Structure

The overall design (Fig. 11) is a **layered, parallel architecture**. It uses dedicated memory blocks for APP and CTV messages, and a set of parallel processing units (VNUs and a CNU) to handle the calculations for one full layer in a single clock cycle.

2. Memory and Clock Cycle Reduction

- **Layer Merging:** The designers noticed that some consecutive layers in the 5G matrix are "orthogonal," meaning they don't share any VNs. Their hardware processes two of these independent layers **simultaneously**, which reduces the total number of clock cycles needed for decoding by about 26-28%.
- **Split Storage:** To handle the extra data from layer merging without wasting memory, they use two memory banks. A main `CTV memory 1` stores most of the data, while a smaller `CTV memory 2` stores only the "overflow" from the wider, merged layers. This significantly reduces the total memory size.

3. Interconnection Network Optimizations

This is about reducing the complexity of the "wiring" between the memory and the processing units.

- **Selective-Shift Structure:** For the part of the data corresponding to the "extension bits," they use a clever shift-register-based memory. This ensures the required data set is always at a fixed output after a simple shift, drastically reducing the number of input connections to the processing network (e.g., from 52 down to 16 for one code).
- **Message Reordering:** They carefully re-mapped the data flow (as shown in Fig. 16) so that each processing unit only needs to choose from a small number of inputs (e.g., 2 or 3) instead of all of them. This simplifies the selection logic (multiplexers) and reduces signal delay.

Changes for HLS/FPGA and What You'll Write

Your job is to describe these architectural concepts in a high-level language (like C/C++) and then use HLS tools to synthesize them into a configuration for an FPGA. Here's what you'll need to write and how it differs from the paper's ASIC approach:

1. Data Types: Floating-Point to Fixed-Point (You Write This)

This is your first and most critical task. Your Python model uses high-precision `floats`. FPGAs work with fixed-point numbers to be efficient.

- **Your Task:** You'll need to convert all your variables (LLRs, messages) to fixed-point data types (like `ap_fixed` in Vitis HLS). You'll have to choose the bit widths yourself

(e.g., 4 bits for messages, 6 for APPs, as the paper suggests) and analyze the performance impact of this quantization.

2. Memory Architecture (You Describe This)

FPGAs don't have custom-designed memory; they have built-in

Block RAMs (BRAMs).

- **Your Task:** In your HLS code, you'll declare your APP and CTV memories as arrays. You will then use **HLS pragmas** (special comments that act as instructions for the HLS tool) to tell the synthesizer to map these arrays onto the FPGA's BRAMs.
- The **Split Storage** method is an architectural choice you *can* implement. You would declare two separate arrays in your C++ code for the CTV memory and write the logic to store/retrieve data from `memory 1` or `memory 2` based on the layer being processed.

3. Parallelism and Pipelining (You Control This)

The paper's design is heavily parallel to achieve high throughput.

- **Your Task:** You need to explicitly manage parallelism in HLS. You'll use pragmas like `#pragma HLS UNROLL` on loops to create parallel processing units (replicating the hardware for each element in the loop). You'll use `#pragma HLS PIPELINE` to ensure that a new set of data can enter the processing pipeline every clock cycle, which is how you'll achieve the "one layer per cycle" performance mentioned in the paper. The **Layer Merging** optimization would be implemented by writing a single loop iteration in your HLS code that fetches and processes data from two layers.

4. Interconnection Networks (You Write This Logic Explicitly)

This is the most challenging part to translate. HLS is not good at automatically inferring complex, custom routing like the paper describes. You have to write the logic yourself.

- **Your Task:**
    - For the **Selective-Shift** structure, you'll model it as an array in C++ and write explicit code to perform the circular shift operation when needed.
    - For the **Read/Write Networks** with message reordering, you will need to implement the mapping shown in Fig. 16 using C++ logic. This will likely be a large `switch` statement or a series of `if/else` blocks that explicitly route the correct input array to the correct output based on the current layer number. This is a manual, structural description of the wiring, not something HLS can figure out on its own.