Optimizations of the PGaB LDPC Decoder in CUDA

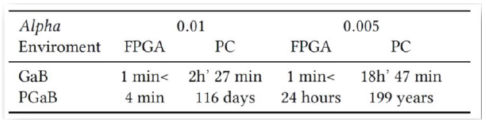
Sebastian Thiem, Matthew Johnson, and Jeremy Sears

***Abstract*—In order to analyze the performance of the PGaB LDPC decoding scheme one must sweep through a range of noise level until 100 frame errors are achieved. Achieving 100 frame errors becomes a exponentially harder problem as channel noise decreases. Previous work utilized the the CUDA development language to solve this problem and resulted in a throughput of 99KB/s. This work utilized shared memory, pinned memory, batching and streaming to achieve a maximum throughput of 5.77MB/s.**

***Index Terms*—CUDA, LDPC, PGaB, Shared Memory, Streams, Batching.**

I. Introduction

Running simulations to test the Probabilistic Gallager B (PGaB) algorithm is critical to validating and benchmarking the design. In order to consider an error rate as valid, a minimum of 100 frame errors must be reached. For low noise simulations, this becomes a near impossible task for CPUs since the number of processed frames needed to reach 100 frame errors increases exponentially. At 1% chance of bit flip a CPU implementation would take over 3 months (see Figure 1).



Alpha	0.01		0.005	
Enviroment	FPGA	PC	FPGA	PC
GaB	1 min<	2h' 27 min	1 min<	18h' 47 min
PGaB	4 min	116 days	24 hours	199 years

Fig. 1. CPU vs FPGA Timing

FPGA implementations have offered a significant improvement for these simulations, which has brought rise to this work. GPUs have a lot to offer in terms of ease of use and scalability, with a throughput that rivals the FPGA. This work sought to optimize the GPU implementation of PGaB in the hope that its performance could approach that of the FPGA.

II. Background

Low density parity check codes, or LDPCs, use parity encoding in a message to reconstruct messages over noisy channels. Extra parity bits are 'assigned' to predetermined parts of the message and are set to ensure an even or odd parity among the codeword. By using even/odd parities should a bit flip in transmission the receiver will know. By overlapping the codewords and assigning more than one parity bit to each message bit, the receiver can can use the parity information to reconstruct the message. There are many reconstruction algorithms, but the one of interest to this research is PGaB.

The PGaB is an adaptation of the original Gallager B algorithm developed by Robert Gallager in 1960 [1]. This adaptation introduces a probabilistic disturbance to the decoding process, that has been found to improve overall decoding accuracy and latency [2].

III. Related Work

A. *PGaB – Unal et al.*

In Unal's paper, the Gallager B decoding algorithm was augmented by randomly disturbing decisions made during the decoding process [2]. This probabilistic element was shown to lower the average number of iteration by up to 62%.

Using the Xilinx Virtex-6 FPGA, the PGaB algorithm was optimized and benchmarked. For 1296-bit codewords the following FER curve from figure 2 was generated. Of note here is the red curve for PGaB, this curve is what validates the design of the CPU and GPU implementations for PGaB.

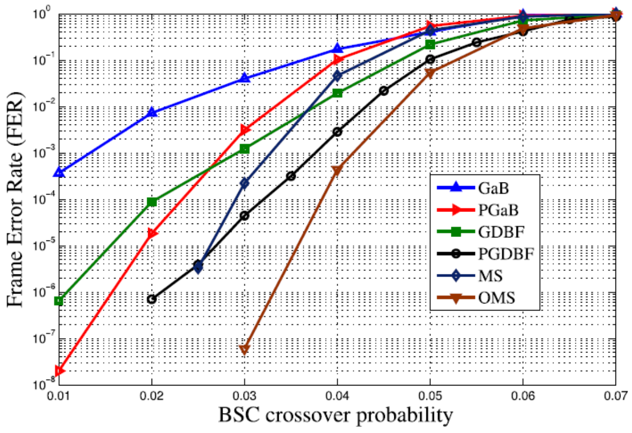


Fig. 2. FER vs Alpha Values

B. *Shared Memory LDPC – Wen et al.*

In 2014 a group of researchers used pinned and shared memory to improve the throughput of an LDPC decoder [3]. Continuing off of previous work in the following papers, [4][5][6][7][8], they moved the the host memory to pinned memory and utilized shared memory in the kernels. The pinned memory reduces the latency when transferring memory from the CPU to the GPU, and the shared memory has a lower latency than the standard global memory in GPUs.

After applying these improvements the team found boost in throughput from 146.6Mb/s to 374.3Mb/s. The algorithm used in this line of research is very similar to PGaB, so the same ideas should benefit PGaB's performance as well.

IV. Implementation Approach

A. Starting Point

Provided for this work was a C implementation of the PGaB algorithm, and a CUDA implementation of the algorithm. The CUDA version of PGaB was still in the early stages of development and had not yet begun going through optimizations available.

There were 6 kernels written in the previous work: three data pass kernels, one check pass, one kernel to apply the corrections to the data, and one to compute the syndrome. The first data pass kernel runs on iteration 0, the second on iterations 1-14, and the last runs for the remaining iterations. The algorithms being executed in each of the 6 kernels can be seen described very briefly in Algorithms 1 - 6.

Some variables of note are:

- idx – The index for the thread on the GPU
- varr – A Bernoulli random variable with probability 0.8
- I – The interleaver.
- W – The received word from the simulated channel.
- N – Length of each codeword.
- M – Parity bits per codeword.
- D – Decision array
- S – Syndrome

Algorithm 1 Data Pass 0

```
1: $idx \leftarrow [0, N)$
2: $stride = NbBranches/N$
3: for $s \in [0, 1, ..., stride]$ do
4:     $idx_{node} = I[idx * stride + s]$
5:     $VtoC[idx_{node}] = W[idx]$
6: end for
```

Algorithm 2 Data Pass 1

```
1: $idx \leftarrow [0, N)$
2: $global \leftarrow 1 - 2 * W[idx]$
3: $stride = NbBranches/N$
4: for $s \in [0, 1, ..., stride]$ do
5:     $idx_{node} = I[idx * stride + s]$
6:     $global+ = (-2) * CtoV[idx_{node}] + 1$
7: end for
8: for $s \in [0, 1, ..., stride]$ do
9:     $idx_{node} = I[idx * stride + s]$
10:    $buf = global - ((-2) * CtoV[idx_{node}] + 1)$
11:    if $buf < 0$ then
12:        $VtoC[idx_{node}] = 1$
13:    else if $buf > 0$ then
14:        $VtoC[idx_{node}] = 0$
15:    else
16:        $VtoC[idx_{node}] = W[idx]$
17:    end if
18: end for
```

Algorithm 3 Data Pass 2

```
1: $idx \leftarrow [0, N)$
2: $global \leftarrow 1 - 2 * (varr \oplus W[idx])$
3: $stride = NbBranches/N$
4: for $s \in [0, 1, ..., stride]$ do
5:     $idx_{node} = I[idx * stride + s]$
6:     $global+ = (-2) * CtoV[idx_{node}] + 1$
7: end for
8: for $s \in [0, 1, ..., stride]$ do
9:     $idx_{node} = I[idx * stride + s]$
10:    $buf = global - ((-2) * CtoV[idx_{node}] + 1)$
11:    if $buf < 0$ then
12:        $VtoC[idx_{node}] = 1$
13:    else if $buf > 0$ then
14:        $VtoC[idx_{node}] = 0$
15:    else
16:        $VtoC[idx_{node}] = W[idx]$
17:    end if
18: end for
```

Algorithm 4 Check Pass

```
1: $idx \leftarrow [0, M)$
2: $signe \leftarrow 0$
3: $stride = NbBranches/M$
4: for $s \in [0, 1, ..., stride]$ do
5:     $signe = signe \oplus VtoC[idx * stride + s]$
6: end for
7: for $s \in [0, 1, ..., stride]$ do
8:     $idx_{node} = idx * stride + s$
9:     $CtoV[idx_{node}] = signe \oplus VtoC[idx_{node}]$
10: end for
```

Algorithm 5 APP

```
1: $idx \leftarrow [0, N)$
2: $global \leftarrow 1 - 2 * W[idx]$
3: $stride = NbBranches/N$
4: for $s \in [0, 1, ..., stride]$ do
5:     $global+ = (-2) * CtoV[I[idx * stride + s]] + 1$
6: end for
7: if $global < 0$ then
8:     $D[idx] = 1$
9: else if $global > 0$ then
10:    $D[idx] = 0$
11: else
12:    $D[idx] = W[idx]$
13: end if
```

Algorithm 6 Compute Syndrome

```
1: $synd \leftarrow 0$
2: $idx \leftarrow [0, M)$
3: $stride = NbBranches/M$
4: for $s \in [0, 1, ..., stride]$ do
5:     $synd = synd \oplus D[idx * stride + s]$
6: end for
7: $S[idx] = synd$
```

The code provided was restructured and organized to allow for more modular changes to the algorithm. Examples of this restructuring are: breaking the code up into multiple files, removing redundant logic/computations/memory accesses, commenting, and abstracting parts of the main function into their own functions. During this process, timers from the Linux '$< sys/time.h >$' library were added to the code to monitor execution times in order to determine the CPU bottlenecks.

Two high-latency processes that were running on the CPU were discovered as a result of this work: a parity encoder and noise simulator.

B. Parity Encoder

The parity encoder is designed to read the message being sent and determine the state for the LDPC parity bits. The exact algorithm can be seen below in Algorithm 7. Due to the overlapping nature of the LDPC encoding scheme, each iteration is dependant on the previous iteration, and it would seem that it must be serialized, however that is not the case. These nested loops can be run as two separate loops, see Algorithm 8. The separate loops can be run in one kernel which brought the latency down from 1.279ms to 42us. This 30x speedup is critical for this application, as the parity encoder will run every iteration on every noise level.

Algorithm 7 Parity Encoding Algorithm

```
1: $Mat_G \leftarrow Matrix\ to\ describe\ node\ connectivity$
2: $rank \leftarrow Total\ number\ of\ rows\ in\ Mat_G$
3: $U \leftarrow Message\ sent$
4: for $k \in [rank-1, rank-2, ..., 0]$ do
5:     for $l \in [k+1, k+2, ..., N]$ do
6:         $U[k] = U[k] \oplus (Mat_G[k][l] * U[l])$
7:     end for
8: end for
```

Algorithm 8 Parity Encoding Algorithm GPU Kernel

```
1: $Mat_G \leftarrow Matrix\ to\ describe\ node\ connectivity$
2: $rank \leftarrow Total\ number\ of\ rows\ in\ Mat_G$
3: $U \leftarrow Message\ sent$
4: $idx \leftarrow GPU\ thread\ index$
5: for $i \in [rank-1, rank-2, ..., 0]$ do
6:     $u[idx] = u[idx] \oplus (MatG_D[idx * N + i] * u[i])$
7: end for
8: for $i \in [k+1, k+2, ..., N]$ do
9:     $u[idx] = u[idx] \oplus (MatG_D[idx * N + i] * u[i])$
10: end for
```

C. Channel Simulation

Similar to the parity encoder, the channel impairments are also re-simulated every iteration, and as such it is critical to minimize latency here. The algorithm, seen below in Algorithm 9, simply samples from a uniform distribution and decides whether or not to flip a bit based on the current probability of bit-flips α.

On the CPU this runs for 9.755ms per iteration. This was bottle-necking the performance and could be improved by unrolling the for-loop on the GPU. Once unrolled the device could then take advantage of the cuRAND library for efficient random number generation across the threads. Kernelizing this operation brought the latency down to 0.409ms, a 24x speedup.

Algorithm 9 Simulate Channel Noise

```
1: $\theta \sim [0, 1]$
2: $\alpha \leftarrow Probability\ of\ bit-flip$
3: for $n \in [0, 1, ..., N]$ do
4:     if $\theta < \alpha$ then
5:         $message[n] = 1 - message[n]$
6:     end if
7: end for
```

D. Pinned Host Memory & Shared Device Memory

From the paper, "A High Throughput LDPC Decoder using a Mid-range GPU"[3], there were two optimizations used that were replicated in this work: pinned host memory and shared device memory.

By default, host memory is pagable memory while the GPU uses pinned memory. During standard memory transfers the CPU memory must be converted to pinned memory before it can be transfered over to the GPU. This causes a delay in memory transfers, a delay which can be avoided by using pinned host memory. CUDA enables users to allocate pinned memory, rather than pagable memory, through the cudaMallocHost() function. This allocation has a longer delay than the standard C language malloc(), however the latency on device-host transfer speeds is reduced.

Since the PGaB algorithm allocates most of its memory only once at the beginning of execution and spends the rest of its run time moving the memory back and forth from device to host, pinned memory offered a large performance improvement for this application.

Once the data is on the device there is another type of memory transfer to consider, and that is the one from global memory to the thread's registers. All of the kernels seen in Algorithms 1 - 6 contain an abundance of memory accesses. In the case of data pass 1 & 2 as well as in check pass, the same blocks of memory are read from global in two different phases; this is where utilizing shared memory shines the most. By swapping out the global memory accesses in many of these kernels with shared memory accesses there was an improvement of up to 2x in overall throughput of PGaB.

E. Streaming & Batching

When dealing with large data structures one begins to see a bottleneck around the memory transfer latency. Pinned memory will lower this latency, however it does not address the fact that the GPU is sitting idle while waiting for the memory to transfer. This problem is addressed by cutting the data into smaller batches. These batches are streamed into the device one after another so that while the device computes one batch, the next batch is being moved into global memory. This will stack the batched kernels on top of one another allowing for a significant reduction in overall execution time.

This process is done by defining, at compile time, the number of streams and number of batches. The CUDA api requires streams to be allocated, so these are allocated as an array of cudaStream_t objects, and the data is broken up according to the number of batches. The memory can then be transfered asynchronously over to the device, while kernels execute (see Figure 3).

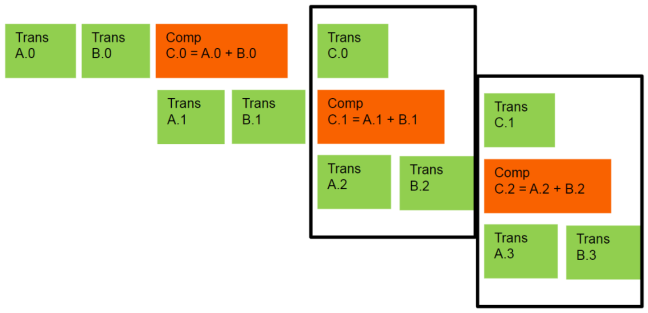


Fig. 3. Visualization of the streaming process

V. Results & Analysis

The algorithm was profiled on an Nvidia RTX 2080 and verified by referencing the expected FER (frame error rate) curve from the Burak's PGaB paper [2] (see Figure 2 and Table I).

TABLE I
FPGA FER

Alpha	0.06	0.05	0.04	0.03	0.02
FER	0.9	0.5	0.1	0.003	2e-05

This project began with a CPU and GPU implementation provided. Each version of the code was ran on a sweep from 0.06 to 0.02 alpha, the results may be seen in Table II and III respectively.

It can be seen that this GPU code did not produce the same FER curve as the original algorithm. This mistake was caused only analyzing latency without validating the algorithm itself. By running this validation and observing the throughput in bytes per second, one can see that this 1810x speedup invalidated the design and is actually a 50% cut in effective throughput.

The root of the problem, and why the simulation was now converging sooner than it should have been, was due to the removal of the P from P Gallager B. By looking at the Data Pass 2 kernel Algorithm 3, one can see that the varr variable is only a single Bernoulli random variable. The original PGaB algorithm utilized a full array of random variables to disturb the decoding process, and as such having only one removed the probabilistic element of the PGaB algorithm. The actual fix can be seen here in Algorithm 10

Algorithm 10 Data Pass 2 Fixed

```
1: $idx \leftarrow [0, N)$
2: $global \leftarrow 1 - 2 * (varr[idx] \oplus W[idx])$
3: $stride = NbBranches/N$
4: for $s \in [0, 1, ..., stride]$ do
5:     $idx_{node} = I[idx * stride + s]$
6:     $global+ = (-2) * CtoV[idx_{node}] + 1$
7: end for
8: for $s \in [0, 1, ..., stride]$ do
9:     $idx_{node} = I[idx * stride + s]$
10:    $buf = global - ((-2) * CtoV[idx_{node}] + 1)$
11:    if $buf < 0$ then
12:        $VtoC[idx_{node}] = 1$
13:    else if $buf > 0$ then
14:        $VtoC[idx_{node}] = 0$
15:    else
16:        $VtoC[idx_{node}] = W[idx]$
17:    end if
18: end for
```

TABLE II
CPU Simulation Results

Alpha	0.06	0.05	0.04	0.03	0.02
FER	0.934	0.559	0.077	0.003	1e-0.5
BER	0.304	0.179	0.025	0.001	3e-06
Iter.	107	179	1291	30476	5e+06
Lat.	1.07s	1.22s	3.25s	50.6s	83.8min
Thrpt.	16KB/s	24KB/s	64KB/s	98KB/s	161KB/s

TABLE III
Naive GPU Simulation Results

Alpha	0.06	0.05	0.04	0.03	0.02
FER	1.000	0.971	0.676	0.334	0.059
BER	0.334	0.324	0.221	0.108	0.019
Iter.	100	103	148	299	1692
Lat.	521ms	493ms	566ms	785ms	2.78s
Thrpt.	31KB/s	34KB/s	42KB/s	62KB/s	99KB/s

A. Pinned Host Memory & Shared Device Memory

After following the optimizations proposed in Wen's work [3] and including a kernel to handle the parity encoding, throughput increased from 99KB/s to 249KB/s.

By observing the Nvidia profiler results, as well as comparing them to the CPU timers there were two observations made.

The first being that the CPU was still bottlenecking this program. The channel simulation, BER, and convergence check were all 1-10ms each iteration. Any one of these could be kernelized to remove the workload from the CPU.

The second being that 56% + 19% of the overall execution time was spent transferring memory and calling kernels respectively. This suggested that a combination of batching and streaming would benefit this application.

Due to these observations, further work was done beyond the memory changes proposed in Wen's paper [3].

B. Streaming & Batching

By streaming batches for each of the kernels mentioned in Algorithms 1 - 8 throughput went up from 249KB/s to 555KB/s.

This process involved reverting back to global memory versions for all kernels except for the parity-encoding kernel and the compute syndrome kernel. This was due to the change in memory accesses needed to batch the data transfers. It would not be impossible to convert these kernels to shared memory again, but that will have to be done in future work.

C. Channel Simulation Kernel

This kernel resulted in the largest throughput increase of all the optimizations explored in this work.

The channel simulation in previous work ran for about 9.755ms every iteration, by bringing this down to 409us the throughput jumped from 555KB/s to 5.77MB/s, and is currently the best throughput achieved in this research (see Table IV).

Looking back at the profiler, we see that although our throughput is high and much of the kernel launches are being masked by streams, that the total kernel launches in the program are beginning to suggest merging kernels.

When running the simulation from alpha values 0.06 to 0.02 the three kernels: Check Pass, APP, and Compute Syndrome all run 560000+ times each. Adding in the other kernels running 100000+ times each results in 2.6 million kernels being launched in around 30s. Profiling will show that these kernel launches eat up around 34% of the execution time. The batching may have improved the throughput but as more kernels are added the kernel launch overhead is creeping back up.

Before continuing to kernelize the BER and convergence check, it is clear that work should be done to merge the kernels where possible.

TABLE IV
CURRENT GPU SIMULATION RESULTS

Alpha	**0.06**	**0.05**	**0.04**	**0.03**	**0.02**
FER	0.943	0.546	0.089	0.003	1e-05
BER	0.308	0.176	0.028	0.001	3e-06
Iter.	160	240	1200	32240	1e+06
Lat.	44.1ms	67.7ms	281ms	2.42s	28.1s
Thrpt.	588KB/s	574KB/s	692KB/s	2.16MB/s	5.77MB/s

D. Last Observations

Figure 2 shows that at an alpha of 0.01 it will take $(100) * 10^8$ iterations to achieve the 100 frame errors required. At a codeword length of 1296-bits the amount of data processed at 0.01 alpha is 1.62TB. With the current throughput of 5.77MB/s it would only take 78hrs to run this simulation.

VI. CONCLUSION

The PGaB algorithm started at with only a 161KB/s throughput on the CPU, which took over an hour to simulate an alpha of 0.02. By utilizing a GPU through CUDA that throughput went up a factor of 3584 to 5.77MB/s, letting the 0.02 alpha simulation run in less than 30 seconds.

78hrs at 0.01 alpha is certainly not beating the FPGA's 4min by any marks, but there is a lot of potential here. Considering how many large optimizations are still available for this code, and the convenience of a GPU based simulation, this implementation has a lot to offer.

If in future work the simulation could be brought down to $\leq$2hrs, which given the optimizations still available is more than possible, there could be a strong argument for using this GPU simulation rather than the FPGA sim in [2].

VII. FUTURE WORK

A. Compression

Currently every message bit is being stored in 32-bit integers. That means that for every one bit being simulated, 31bits are being filled in memory. This is a very poor usage of memory space and there are two options that should be explored:

The first and simplest optimization would be to replace the integer structures with shorts(16-bit) or even chars(8-bit). This would not change any of the computations involved and it would cut the memory usage in half or a quarter, respectively. While this will lower memory usage, in CUDA there is a cost from using data structures less than 32-bits. When the data is read from memory, if it is not already a 32-bit word it will be cast as a 32-bit word, and this will incur a latency.

The second and the most ideal solution would be to remain as 32-bit words, but utilize the full capacity of the word to store 32 bits. This would result in a compression ratio of 32:1 with none of the latency issues associated with the first optimization. For some operations, such as channel noise simulation, this will reduce computation by a factor of 32, as the bit flips can be done on 32-bits at a time.

B. More Shared Memory

During the batched-streaming portion of this work, many of the shared memory kernels were converted into global memory kernels due to the change in memory accesses. Returning to these kernels to utilize shared memory in them would show another boost in the simulation's throughput.

C. Further Kernelization

There still exist parts of the code that would benifit greatly from GPU kernels, one of them being the BER calculation. This process goes through each bit and compares the received message with the expected message to calculate the bit errors. This is an entirely data parallel process and it is currently being done on the CPU.

Another available kernel is the convergence check. This process goes through each element of the syndrome and raises a flag if any element is non-zero. On the CPU this is a slow process, however on the GPU this is a simple scan operation.

D. Field Expert Optimizations

All of the optimizations done in this work were purely ones done using GPU knowledge and did not affect the functionality of the PGaB algorithm. Discussing this work with a field expert will reveal any parts of this simulation that could be removed or improved by changing the functionality.

Acknowledgment

This work was supported by Dr. Akoglu, and his graduate student Burak Unal. Burak provided the CPU implementation of the PGaB that this work was based on.

An honorable mention goes out to Dr. Akoglu's previous students: Adithya Rao, Harshil Kakaiya, and Priyanka Goswami, who developed the initial CUDA implementation.

References

[1] R. Gallager, "Low-Density Parity-Check Codes," July 1963 MIT, Cambridge, Massachusetts.

[2] B. Unal, A. Akoglu, F Ghaffari, B. Vasić, "Hardware Implementation and Performance Analysis of Resource Efficient Probabilistic Hard Decision LDPC Decoders," Sept 2018 *IEEE Transactions on Circuits and SystemsI: Regular Papers*, vol. 65, no. 9, pp. 3074-3084. https://doi.org/10.1109/TCSI.2018.2815008

[3] X. Wen, J. Xianjun, P. Jskelinen, H Kultala, C. Canfeng, H. Berg, B. Zhisong, "A High Throughput LDPC Decoder using a Mid-range GPU," May 2014 *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 7565-7569. https://doi.org/10.1109/ICASSP.2014.6855061

[4] G. Wang, M. Wu, B. Yin, J. Cavallaro, "High throughput low latency LDPC decoding on GPU for SDR systems," Dec. 2013 *IEEE Global Conference on Signal and Information Processing (GlobalSIP)*, pp. 1258-1261. https://doi.org/10.1109/GlobalSIP.2013.6737137

[5] S. Kang and J. Moon, "Parallel LDPC decoder implementation on GPU based on unbalanced memory coalescing," June 2012 *IEEE International Conference on Communications (ICC)*, pp. 3692-3697. https://doi.org/10.1109/ICC.2012.6363991

[6] G. Wang, M. Wu, Y. Sun, J. Cavallaro, "A massively parallel implementation of QC-LDPC decoder on GPU," June 2011 *IEEE 9th Symposium on Application Specific Processors (SASP)*, pp. 82-85. https://doi.org/10.1109/SASP.2011.5941084

[7] G. Falcao, L. Sousa, V. Silva, "Massively LDPC decoding on multicore architectures," Feb 2011 *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, no. 2, pp. 309-322. https://doi.org/10.1109/TPDS.2010.66

[8] G. Falcao, J. Andrade, V. Silva, L. Sousa, "GPU-based DVB-S2 LDPC decoder with high throughput and fast error floor detection," April 2011 *Electronics Letters*, vol. 47, no. 9, pp. 542543.