2014 IEEE International Conference on Acoustic, Speech and Signal Processing (ICASSP)

A High Throughput LDPC Decoder using a Mid-range GPU

Xie Wen[1], Jiao Xianjun[2], Pekka Jääskeläinen[3], Heikki Kultala[3], Chen Canfeng[2], Heikki Berg[2], Bie Zhisong[1]
Beijing University of Posts and Telecommunications[1], Nokia Research Center[2], Tampere University of Technology[3]
Beijing, China[1], Beijing, China[2], Espoo Finland[2], Tampere, Finland[3]
Email: zhisongbie@bupt.edu.cn[1], ryan.jiao@nokia.com[2], pekka.jaaskelainen@tut.fi[3]

***Abstract*—A standard-throughput-approaching LDPC decoder has been implemented on a mid-range GPU in this paper. Turbo-Decoding Message-Passing algorithm is applied to achieve high throughput. Different from traditional host managed multi-streams to hide host-device transfer delay, we use kernel maintained data transfer scheme to achieve implicit data transfer between host memory and device shared memory, which eliminates an intermediate stage of global memory. Data type optimization, memory accessing optimization, and low complexity Soft-In Soft-Out algorithm are also used to improve efficiency. Through these optimization methods, the 802.11n LDPC decoder on NVIDIA GTX480 GPU, which is released in 2010 with Fermi architecture, has achieved a high throughput of 295Mb/s when decoding 512 codewords simultaneously, which is close to highest bit rate 300Mb/s with 20MHz bandwidth in 802.11n standard. Decoding 1024 and 4096 codewords achieve 330 and 365Mb/s. A 802.16e LDPC decoder is also implemented, 374Mb/s (512 codewords), 435Mb/s (1024 codewords) and 507Mb/s (4096 codewords) throughputs have been achieved.**

I. Introduction

Low-Density Parity-Check (LDPC) codes are proposed in 1962 by Robert Gallager [1]. Due to the capacity-approaching performance (0.0045dB within Shannon Limit [2]) and the parallelism friendly decoding algorithm, LDPC has been adopted by many standards,such as DVB-S2, DVB-T2, 802.16e, 802.11n.

To get flexible and upgradable implementation of different standards, Software Defined Radio (SDR) is a promising scheme. Accelerating those computational intensive algorithm, such as FEC decoding, is an important topic in SDR area.

Parallel computing is one promising way to do acceleration. In November 2006, NVIDIA introduced Compute Unified Device Architecture (CUDA), a general purpose parallel computing platform which leverages the parallel compute engine on NVIDIA GPUs to solve many complex computational problems in a more efficient way than on a CPU [3].

There are some previous works on CUDA based LDPC decoder, such as [4]–[8]. Two-Phase Message-Passing (TPMP) decoding algorithm, which was proposed by Gallager [1], is used in those papers, and achieves significant speedup. Turbo-Decoding Message-Passing (TDMP) decoding algorithm is used in another CUDA based LDPC decoder work [9], and achieves impressive throughput 160Mb/s. In this paper, after some optimization methods are used on TDMP algorithm (NVIDIA GTX480 GPU, released in 2010, Fermi architecture), better throughput results are achieved compared to a very latest work [4] (GTX Titan GPU, released in 2013, Kepler architecture).

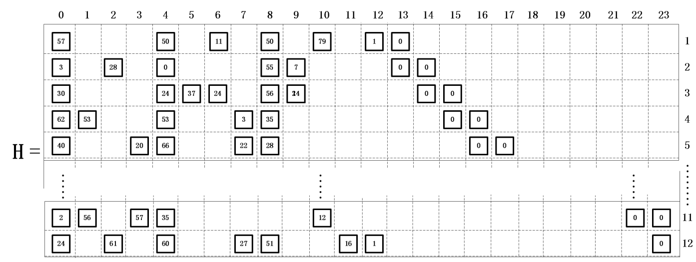

Fig. 1. An 802.11n (1944,972) LDPC check matrix.

II. LDPC Codes and Decoding Algorithm

Let $H_{M\times N}$ be the parity check matrix. There are N bits in the encoded codeword which is constrained by M parity check equations. When encoding, $N-M$ systematic bits are fed into encoder, M parity bits are generated and padded to construct a codeword.

LDPC decoding algorithm can be roughly divided into two categories: TPMP [1] and TDMP [10], [11]. Though TPMP has higher parallelism, TDMP is more efficent in terms of less memory consumed and faster convergence behavior (20%-50% fewer iterations than TPMP).

A. LDPC Codes in IEEE 802.11n Standard

Fig. 1 shows check matrix $H_{972\times 1944}$ which is composed by 12×14 submatrices (size 81×81, $B=81$). It represents a (1944,972) LDPC code with $M = 972$ parity bits in the $N = 1944$ bits codeword. Little squares labeled with a integer $\pi(\pi = 0,1,...,B-1)$ denotes the cyclic-shifted submatrices obtained from the $B\times B$ identity matrix by cyclically shifting the columns to the right by π elements. Vacant entries of H denote null (zeros) submatrices.There are $M_B \times N_B$ submatrices in H ($M_B = 12$, $N_B = 24$ in Fig. 1).

B. Turbo-Decoding Message-Passing (TDMP) Decoding Algorithm

TDMP decoding algorithm for LDPC codes is proposed by Mohammad M. Mansour et al. in [10]. It runs along H row by row from top to bottom sequentially. Posterior information of corresponding bits in codeword are updated in each row processing. It leads to faster convergence than TPMP because each row processing uses updated messages generated by previous processing.

TDMP algorithm is described in Algorithm 1.The channel output codeword values are denoted by vector $\delta = [\delta_1, ...\delta_N]$. The posterior information of codeword iteratively updated is denoted by vector $\gamma = [\gamma_1, \dots \gamma_N]$. For each row (row index $i = 1, \dots, M$) processing of check matrix H, extrinsic messages are stored in vector $\lambda^i = [\lambda^i_1, \lambda^i_2, \dots, \lambda^i_{c_i}]$. I.e., there are totally M vectors and c_i elements in the ith vector, where c_i indicates the number of '1's in the ith row of H. The number of '1's (c_i) is called the row weight of the ith row of H and locations of these '1's in the ith row of H are stored in the set $I^i = [I^i_1, I^i_2, \dots, I^i_{c_i}]$, which are used as reading address of γ in each row processing. Furthermore, the intermediate prior message is denoted by a vector $\rho = [\rho_1, \rho_2, \dots, \rho_{c_i}]$.

Algorithm 1 TDMP Decoding

$//Initialization:$
$\lambda^i \leftarrow 0; \gamma \leftarrow \delta.$
$//IterativeDecoding:$
for $t = 1$ **to** $MaxIter$ **do**
  for $i = 1$ **to** M **do**
    1) $\rho \leftarrow \gamma(I^i) - \lambda^i;$ $//Read \ and \ subtract$
    2) $\Lambda \leftarrow SISO(\rho);$ $//SISO\_unit$
    3) $\lambda^i \leftarrow \Lambda;$ $//Write \ back$
    4) $\gamma(I^i) \leftarrow \rho + \Lambda;$ $//Add \ and \ write \ back$
  end for
end for

Min-Sum (MS) [12] algorithm is used in this paper for its low complexity. The SISO_MS algorithm is as (1).

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

III. ALGORITHM MAPPING AND OPTIMIZATION IN GPU

In CUDA programming mode, a C function, so called kernel function, is executed on GPU. CUDA threads, which are generated from the kernel, are organized into specific hierarchy. Threads which need exchanging information with each other are grouped into a thread block. Each thread within the block can be indentified by a index which is accessible within the kernel through the built-in $threadIdx$ variable.

CUDA has several memory spaces. Shared memory, which is usually on chip and precious, can only be accessed by intra block threads. Global memory, which is usually off chip and big, can be accessed by all threads in a grid (grid is a group of thread blocks). The constant memory can also be accessed by all threads in a grid, it is faster than global memory because of the read-only character.

The NVIDIA GPU is composed by Streaming Multiprocessors (SMs). When a kernel is invoked, thread blocks are distributed to SMs which have spare execution resources. Blocks within a SM are split into warps of 32 threads during the execution. When some warps are stalled because of memory/transfer latency, SM can swap in other ready warps. Therefore, the more occupancy of warps per SM the program has, the higher data throughput it will get.

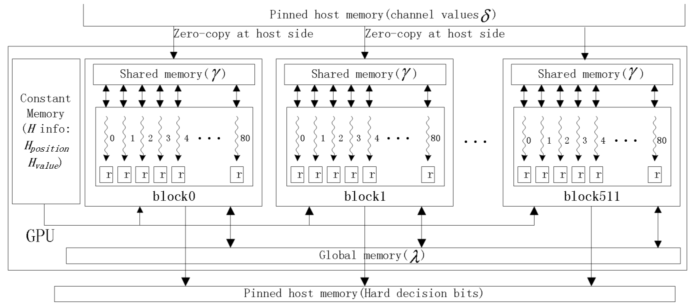

Fig. 2. Overall memory usage in decoding.

A. Algorithm Mapping

For the Algorithm 1, B continuous rows, which can be looked as one layer, in the matrix H can be processed in parallel, because reading address set I^i of the posterior messages γ are different (no conflicts) for each row of a layer. Thus, the inner loop of Algorithm 1 can be re-written in a parallel form in Algorithm 2.

Algorithm 2 Parallelization scheme of TDMP inner loop

for $layer\_idx = 1$ **to** M_B **do**
  // following loop is fully parallelized by B CUDA threads
  for $i = (layer\_idx - 1)B + 1$ **to** $layer\_idx \times B$ **do**
    1) $\rho \leftarrow \gamma(I^i) - \lambda^i;$ $//Read \ and \ subtract$
    2) $\Lambda \leftarrow SISO(\rho);$ $//SISO\_unit$
    3) $\lambda^i \leftarrow \Lambda;$ $//Write \ back$
    4) $\gamma(I^i) \leftarrow \rho + \Lambda;$ $//Add \ and \ write \ back$
  end for
end for

It means that γ are shared among all threads and all threads need to be synchronized after each layer's processing. So, one thread block which contains B threads is used to decode one codeword, because there are shared memory and synchronization mechanism inside one block. Multiple thread blocks are used to decode multiple codewords to get enough workload for delay hiding.

B. SISO_MS Unit

Instead of implementing Equation (1) as its direct form, Two-Min Algorithm (TMA) [13], [14] is chosen as a better alternative. TMA uses two sequential loops instead of two nested loops (direct form). Loop1 finds out minimum and secondary minimum values from $\rho_1 ... \rho_{c_i}$, loop2 calculates $\Lambda_1 ... \Lambda_{c_i}$ utilizing the values found by loop1. Refer to details in [4](Algorithm 1, page 2).

C. Device Memory Accessing Scheme

Fig. 2 shows the overall device memory allocation.

1) Using constant memory: As described in II-A and 1, check matrix H can be indexed by two 2-D arrays: $H_{position}[M_B][C_{max}]$, $H_{value}[M_B][C_{max}]$. They record the column positions of non-null sub matrices and corresponding shifting values. C_{max} is the maximum row weight, i.e., $max([c_1, c_2, ..., c_M])$. Reading address set I^i of γ for the tidth thread in the lth layer can be calculated as (2).

$$I_j^{i=(l-1)B+tid+1} = H_{position}[l-1][j-1] \times B + (tid + H_{value}[l-1][j-1]) \mod B. \quad (2)$$

The two arrays are stored in constant memory for fast accessing by all threads.

2) Using Shared Memory: Channel values δ, which are in pinned host memory, are buffered to shared memory γ (only once) by kernel using the device side pointer of host memory. This can be done in a coalesced form, because contents of these two memory are one-to-one mapped exactly.

For all threads accessing shared memory γ in each layer processing, as described in Algorithm 2 and Equation (2), the different threads will have different target addresses in a continuous address space. That implies half warp threads(16 threads) will fall in 16 distinct banks of shared memory. This bank conflict free character ensures accessing effectively.

3) Using Global memory: Shared memory is precious on-chip resources. Since γ has been allocated in shared memory, considering relative bigger size of extrinsic messages $\lambda^i, i = 1, ..., M$, λ is assigned in global memory. This will benefit occupancy promotion, which will be seen in later Section.

All λ^i can be stored in a 2-D array $\lambda[M][C_{max}]$. Considering parallel scheme in Section III-A, the contents are actually stored in the from of $\lambda[C_{max}][M]$, which ensures each time all threads accessing a memory block with continuous address. This coalesced accessing to global memory will have good efficiency.

D. Host-Device Data Transfer Optimization

During initialization phase of decoding, channel output values δ, which are in the host memory, need to be transfered into device shared memory γ. In the end of decoding, hard decision bits need to be transfered back from device to host. Considering host-device transfer throughput, pinned memory is used in host for device to access. Compared to pageable/normal host memory, the pinned memory improves host-to-device throughput from 1664MB/s to 2627MB/s and device-to-host throughput from 1702MB/s to 3275MB/s in our machine by CUDA bandwidth test.

By using "mapped memory" feature for pinned host memory, explicit copy operation in host, which can only copy data from host to device global memory, is avoided. Instead, a device side pointer is used by kernel to read δ in host memory to γ in device shared memory directly. This feature of zerocopy was added in CUDA Toolkit 2.2 in 2009. It allows that device schedule computation and data transfer as its demand, which implies data transfer latency may be hidden.

E. Early Termination Scheme

At the end of each iteration, Early Termination (ET) scheme is applied to avoid unnecessary computations. B checksums are calculated by B threads based on hard decisions and check equations (rows in H) layer by layer, and CUDA $\_\_syncthreads\_count()$ function is used to test if B checksums are all zeros. According to that, the decision, stopping current thread block or proceeding to next iteration, can be made in time.

F. Occupancy Promotion

Latency hiding (schedule other threads when some threads are stalled temporarily) is very important to achieve high throughput. More active warps per SM, more easy for SM to do latency hiding. Number of active warps is related to many factors. In the following analysis of occupancy, resources usage numbesrs are from $NVIDIA\ Visual\ Profiler$ software when decoding (1944,972) 802.11n codeword on GTX480 GPU.

According to the algorithm mapping scheme, one thread block can only have fixed $B = 81$((1944,972) code specific) threads. So, there are $N_{WPB} = \lceil \frac{ThreadsPerBlock}{32} \rceil = 3$ warps per thread block. The number of active blocks constrained by register resources is as (3).

$$N_{CBR} = \lfloor \frac{RegistersPerSM}{RegistersPerBlock} \rfloor = \lfloor \frac{32768}{2688} \rfloor = 12 \quad (3)$$

When float type is used for decoding, the number of active blocks constrained by shared memory resources is as (4):

$$N_{CBS} = \lfloor \frac{SharedMemoryPerSM}{SharedMemoryPerBlock} \rfloor = \lfloor \frac{49152}{7776} \rfloor = 6 \quad (4)$$

GTX480 SM can have up to 8 active blocks, the actual number of active blocks per SM is $N_B = \min(N_{CBR}, N_{CBS}, 8) = 6$. Then the number of active warps per SM is given:

$$N_{WPS} = N_{WPB} * N_B = 18 \quad (5)$$

Above analysis is based on float type. A short(16-bits) type version decoder is also tested without losing Bit Error Rate performance. TABLE. I gives the occupancy comparison between float and short type.

TABLE I. WARPS OCCUPANCY COMPARISON.

	float type	short type	Device Limitation
Threads/Block	81(fixed)	81(fixed)	1024
Registers/Block	2688	2688	32768
SharedMemory/Block	7776	3888	49152
ActiveBlocks/SM	6(by SharedMem)	8 (by Device)	8
ActiveWarps/SM	18	24	48
Occupancy of warps	37.5%	50%	100%

TABLE. I shows that occupancy is improved from 37.5% to 50% by converting float type to short type. After short type is used, the warps occupancy bottleneck becomes the fixed number of threads ($B = 81$) per block.

IV. EXPERIMENTAL RESULTS

Configurations of the experimental platform are: Intel 3GHz E8400 CPU with 4GB DDR memory; PCI-e 2.0 x16 interface; NVIDIA GTX480 card with 1.5GB memory; CUDA Toolkits 5.5; 802.11n LDPC code (1944, 972); 802.16e LDPC code (2304, 1152).

Throughput is calculated by $num\_codewords \times num\_systematic\_bits/total\_latency$, $total\_latency$ is gotten by counting time difference between two events (use $cudaEventElapsedTime$), which are recorded before and after kernel execution. Before counting time difference, $cudaEventSynchronize$ is used to wait for all threads' exit. Because host-device data transfer has been included in

TABLE II. COMPARISON OF THROUGHPUT PERFORMANCE(SUPERSCRIPT a:512CW, b:1024CW, c:4096CW. CW---CODEWORDS)

paper	[5]	[9]	[15]	[4]		THIS PAPER			
GPU type	GTX480	9800GTX+	GTX470	GTX TITAN		GTX480			
Gflops(FMA)(cores)	1344.96(480)	705(128)	1088.64(448)	4500(2688)		1344.96(480)			
Standard	802.16e	802.16e	802.11n	802.11n	802.16e	802.11n		802.16e	
LDPC codes(N,M)	(2048,1723)	(1536,768)	(1944,972)	(1944,972)	(2304,1152)	(1944,972)		(2304,1152)	
Algorithm	TPMP_SP	TDMP_MS	TPMP_SP	TPMP_MS	TPMP_MS	TDMP_MS		TDMP_MS	
Data Type	32-bits float	8-bits char	32-bits float	32-bits float	32-bits float	16-bits short		16-bits short	
Early Termination	YES	YES	YES	NO	NO	YES	NO	YES	NO
MaxNumIterations	10	5	10	10	10	10	10	10	10
Throughput(Mb/s)	24.5-146.6 (2.7dB-5.5dB)	160 (512CW)	22.5-100.3 (1.5dB-5.0dB) (300CW)	236.70 (1280CW)	316.07 (1280CW)	85.3-294.6^a (1.0dB-5.5dB) 330.42^b 364.93^c	83.5^a 87.7^b 91.8^c	112.9-374.3^a (1.0dB-5.5dB) 435.15^b 507.16^c	114.6^a 120.9^b 126.7^c
AvgNumIter(@MaxThr.put)	NA	NA	NA	10	10	1.85	10	1.93	10
NormThroughput	NA	NA	NA	0.263 (1280CW)	0.351 (1280CW)	0.405^a 0.454^b 0.502^c	0.621^a 0.652^b 0.682^c	0.537^a 0.624^b 0.728^c	0.852^a 0.899^b 0.942^c

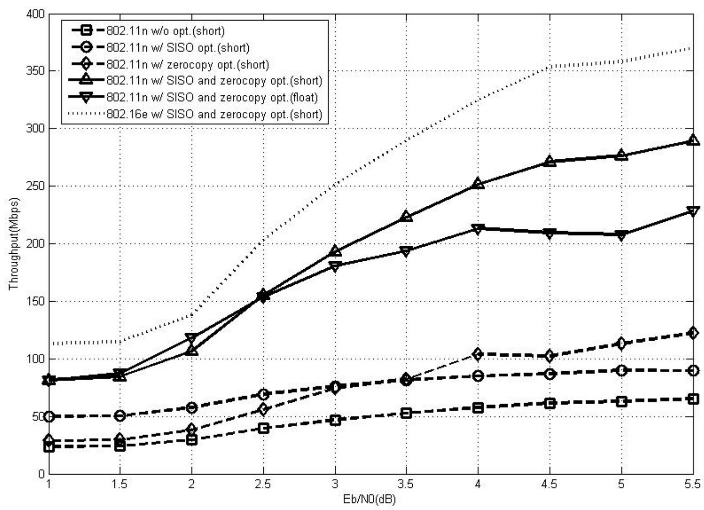

Fig. 3. Comparison of throughput under different optimization techs

kernel execution implicitly, the data transfer time has been included in the $total\_latency$. Notice that most of other GPGPU LDPC works use codeword (systematic+parity) bits throughput, while we use systematic bits throughput.

Fig. 3 shows the throughput improvements under different optimization methods. 512 codewords are decoded in parallel in this test. Without zerocopy optimization, the curve climbs slow when SNR goes high (actual number of iterations becomes small). That's because initialization phase occupies a significant and fixed portion of the whole decoding latency when zerocopy isn't used. SISO unit optimization and float-to-short conversion also contributes much. Maximum 295Mb/s (802.11n) and 374Mb/s (802.16e) throughput are achieved when average number of iterations is 2.

TABLE II compares throughput of different GPGPU LDPC decoder works (Note: "_SP" means Sum-Product algorithm is used for SISO unit). Normalized throughput is given by $MaxThroughput \times AvgNumIter/Gflops$. Because "codeword throughput" is used in [4], its throughput should be divided by 2 before normalization. Other works didn't give average number of iterations, so the normalized throughput isn't listed.

802.16e LDPC codes got higher throughput than 802.11n LDPC codes. That's because 802.16e codes has a more computing friendly check matrix structure. The number of rows/columns of sub matrices of check matrix is integer times of 32, while it can only be 27, 54 or 81 in 802.11n.

V. PREVIOUS WORK

[4], [9] are selected as counterparts. [9] also uses TDMP_MS algorithm, [4] is the latest work using the latest Nvida GPU. CUDA Multi-streamed execution is used to overlap data transfer and execution in [4], [9], while we use device side pointer of host memory to hide data transfer. In [4], [9], channel values are transfered from host memory to device global memory first and then shared memory, while in this paper kernel reads channel values from host memory to shared memory directly by device side pointer of host memory. [9] packs four 8-bits data into 32-bits in global memory and unpacks them in kernel calculation, while we don't do that because that bottleneck becomes insufficient number of threads per block after short type is used. In this paper extrinsic messages are organized to ensure coalesced global memory accessing, which isn't revealed in references. Other differences are listed in Table II.

VI. CONCLUSIONS

Mid-range GPU based 295$\sim$365Mb/s (802.11n) and 374$\sim$507Mb/s (802.16e) LDPC decoder are proposed. TDMP_MS algorithm is selected for its fast convergence and low complexity. Kernel managed data transfer is used through device side pointer of host memory to avoid multi-streams based latency hiding and intermediate global memory. 16-bits data type is used to save shared memory and improve occupancy. The achieved throughput makes the work close to the highest data rate of 802.11n 20MHz bandwidth configuration. Decoding latency, which is another key requirement, should be studied in the future.

Acknowledgements. The work has been financially supported by Academy of Finland (funding decision 253087).

REFERENCES

[1] R.G. Gallager, "Low-density parity-check codes," *Information Theory, IRE Transactions on*, vol. 8, no. 1, pp. 21–28, 1962.

[2] Sae-Young Chung, Jr. Forney, G.D., T.J. Richardson, and R. Urbanke, "On the design of low-density parity-check codes within 0.0045 db of the shannon limit," *Communications Letters, IEEE*, vol. 5, no. 2, pp. 58–60, 2001.

[3] NVidia Corporation , " CUDA C PROGRAMMING GUIDE ," http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf.

[4] Guohui Wang, Michael Wu, Bei Yin, and Joseph R. Cavallaro, "High throughput low latency LDPC decoding on GPU for SDR systems," in *Proc. IEEE Global Conference on Signal and Information Processing (GlobalSIP)*, Dec. 2013, to appear.

[5] Soonyoung Kang and Jaekyun Moon, "Parallel LDPC decoder implementation on GPU based on unbalanced memory coalescing," in *Communications (ICC), 2012 IEEE International Conference on*, 2012, pp. 3692–3697.

[6] Guohui Wang, M. Wu, Yang Sun, and J.R. Cavallaro, "A massively parallel implementation of QC-LDPC decoder on GPU," in *Application Specific Processors (SASP), 2011 IEEE 9th Symposium on*, 2011, pp. 82–85.

[7] G. Falcao, L. Sousa, and V. Silva, "Massively LDPC decoding on multicore architectures," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 22, no. 2, pp. 309–322, 2011.

[8] G. Falcao, J. Andrade, V. Silva, and L. Sousa, "GPU-based DVB-S2 LDPC decoder with high throughput and fast error floor detection," *Electronics Letters*, vol. 47, no. 9, pp. 542–543, April 2011.

[9] K.K. Abburi, "A scalable LDPC decoder on GPU," in *VLSI Design (VLSI Design), 2011 24th International Conference on*, 2011, pp. 183–188.

[10] M.M. Mansour and N.R. Shanbhag, "High-throughput LDPC decoders," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 11, no. 6, pp. 976–996, 2003.

[11] D.E. Hocevar, "A reduced complexity decoder architecture via layered decoding of LDPC codes," in *Signal Processing Systems, 2004. SIPS 2004. IEEE Workshop on*, 2004, pp. 107–112.

[12] E. Amador, V. Rezard, and R. Pacalet, "Energy efficiency of SISO algorithms for turbo-decoding message-passing LDPC decoders," in *Very Large Scale Integration (VLSI-SoC), 2009 17th IFIP International Conference on*, 2009, pp. 95–100.

[13] Kai Zhang, Xinming Huang, and Zhongfeng Wang, "High-throughput layered decoder implementation for quasi-cyclic LDPC codes," *Selected Areas in Communications, IEEE Journal on*, vol. 27, no. 6, pp. 985–994, 2009.

[14] G. Falcao, V. Silva, L. Sousa, and J. Andrade, "Portable LDPC decoding on multicores using OpenCL [applications corner]," *Signal Processing Magazine, IEEE*, vol. 29, no. 4, pp. 81–109, 2012.

[15] Guohui Wang, M. Wu, Yang Sun, and J.R. Cavallaro, "A massively parallel implementation of QC-LDPC decoder on GPU," in *Application Specific Processors (SASP), 2011 IEEE 9th Symposium on*, 2011, pp. 82–85.