CHAPTER

Parallel LDPC Decoding

38

Gabriel Falcao, Vitor Silva, Leonel Sousa

In this chapter we present a pair of kernels to decode a class of powerful error-correcting codes, known as low-density parity-check (LDPC) codes, on graphics processing units (GPU). The proposed parallel implementations adopt a compact data structures representation to access data, while the processing on the GPU grid efficiently exploits a balanced distribution of the computational workload. Moreover, we have analyzed different levels of thread coarsening and propose an efficient one, which balances computation and memory accesses. This case study shows that by adopting these practical techniques, it is possible to use the GPU's computational power to perform this type of processing and achieve significant throughputs, which until recently could be obtained only by developing very large scale integration (VLSI) dedicated microelectronics systems.

38.1 INTRODUCTION, PROBLEM STATEMENT, AND CONTEXT

In the past few years, LDPC codes have been adopted by several data storage and communication standards, such as DVB-S2 [1], WiMAX (802.16e) [2], Wifi (802.11n), or 10Gbit Ethernet (802.3an). As they present powerful error-correcting capabilities and also because the patent has expired, their successful exploitation is expected to continue. LDPC codes [3] decoding is a computationally intensive and iterative application that requires high-performance signal processing for achieving real-time decoding within a targeted throughput. Recovered back from the 1960s when they were invented by Gallager [3] at MIT, they recaptured the attention of academia and industry after the invention of Turbo codes in 1993 by Berrou *et al.* [4]. LDPC codes allow working close to the Shannon limit [5] and achieving excellent bit error rate (BER) performances.

Conventional approaches for LDPC decoding were, until recently, exclusively based on VLSI systems. Although they achieve excellent throughput [1, 2], they also represent nonflexible solutions with high nonrecurring engineering (NRE). Also, they apply low precision to represent data and use fixed-point arithmetic, which imposes quantization effects that limit coding gains, BER, and error floors. Thus, flexible and reprogrammable LDPC decoders can introduce important advantages, not only for real-time processing but also for researching new and more efficient LDPC codes. By developing suitable data structures and exploiting data parallelism with appropriate precision, it is possible to efficiently decode massive amounts of data by using multithreaded GPUs supported by CUDA.

GPU Computing Gems

38.2 CORE TECHNOLOGY

Belief propagation, also known as sum-product algorithm (SPA), is an iterative algorithm [5] for the computation of joint probabilities on graphs commonly used in information theory (e.g., channel coding), artificial intelligence, and computer vision (e.g., stereo vision). It has proved to be an efficient algorithm for inference calculation, and it is used in numerous applications, including LDPC codes [3], Turbo codes [4], stereo vision applied to robotics [6], or in Bayesian networks. LDPCs are linear (N, K) block codes defined by sparse binary parity-check $\mathbf{H}$ matrices of dimension $M \times N$, with $M = N - K$ and rate $= K/N$. They code messages of length K into sequences of length N and are usually represented by bipartite graphs formed by bit nodes (BNs) and check nodes (CNs) linked by bidirectional edges [5, 8] also called Tanner graphs [7]. The decoding process is based on the belief propagation of messages between connected nodes of the graph (as illustrated in Figure 38.1), which demands very intensive processing and memory accesses running the SPA [5].

The SPA applied to LDPC decoding is illustrated in Algorithm 1 and is mainly described by two (horizontal and vertical) intensive processing blocks [8] defined, respectively, by Eqs. 38.1 to 38.2 and Eqs. 38.3 to 38.4. Initially, the LDPC decoder receives the p_n channel probabilities, which represent the input data of Algorithm 1 (in the first iteration, q_{nm} values are initialized with p_n). Kernel 1 computes Eqs. 38.1 and 38.2, calculating the message update from CN_m to BN_n, which indicates the probability of BN_n being 0 or 1. In Eqs. 38.1 and 38.2 q_{nm} values are read and r_{mn} values are updated, as defined by the Tanner graph illustrated in Figure 38.1. Similarly, Eqs. 38.3 and 38.4 compute messages sent from BN_n to CN_m. In this case, r_{mn} values are read and q_{nm} values are updated. Finally, Eqs. 38.5 and 38.6 compute the a posteriori pseudo-probabilities and Eq. 38.7 performs the hard decoding. The iterative procedure is stopped if the decoded word $\hat{\mathbf{c}}$ verifies all parity-check equations of the code ($\hat{\mathbf{c}}\mathbf{H}^T = \mathbf{0}$), or if the maximum number of iterations (I) is reached.

Different levels of thread coarsening and data parallelism can be exploited, namely, by following an edge, node, or codeword processing approach. Although memory accesses in LDPC decoding are not regular, we propose a compact data representation of the Tanner graph suitable for stream computing. This type of processing can be efficiently handled by GPUs.

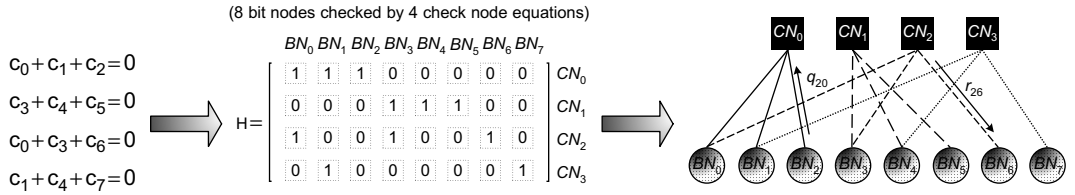

FIGURE 38.1

Linear block code [4, 8] example: parity-check equations, corresponding $\mathbf{H}$ matrix, and equivalent Tanner graph [7] representing an LDPC code.

Algorithm 1: Sum-product algorithm — SPA.

1: {Initialization}
$pn = p(yi = 1); q_{mn}^{(0)}(0) = 1 - p_n; q_{mn}^{(0)}(1) = p_n;$
2: **while** ($\mathbf{H}\hat{\mathbf{c}}^T \neq \mathbf{0} \wedge i < I$) {$\hat{\mathbf{c}}$-decoded word; I-Max. no. of iterations.} **do**
3: {For all pairs (BN_n, CN_m), corresponding to $\mathbf{H_{mn}} = \mathbf{1}$ in parity-check matrix $\mathbf{H}$ of the code **do**:}
4: {Compute messages sent from CN_m to BN_n, that indicate the probability of BN_n being 0 or 1:}

(Kernel 1 – Horizontal Processing)

$$r_{min}^{(i)}(0) = \frac{1}{2} + \frac{1}{2} \prod_{n' \in \mathcal{N}(m) \backslash n} \left(1 - 2q_{n'm}^{(i-1)}(1)\right) \tag{38.1}$$

$$r_{min}^{(i)}(1) = 1 - r_{mn}^{(i)}(0) \tag{38.2}$$

{where $\mathcal{N}(m)\backslash n$ represents BNs connected to CN_m excluding BN_n.}
5: {Compute message from BN_n to CN_m:}

(Kernel 2 – Vertical Processing)

$$q_{nm}^{(i)}(0) = k_{nm}(1 - p_n) \prod_{m' \in \mathcal{M}(n) \backslash m} r_{m'n}^{(i)}(0) \tag{38.3}$$

$$q_{nm}^{(i)}(1) = k_{nm} p_n \prod_{m' \in \mathcal{M}(n) \backslash m} r_{m'n}^{(i)}(1) \tag{38.4}$$

{where k_{nm} are chosen to ensure $q_{nm}^{(i)}(0) + q_{nm}^{(i)}(1) = 1$, and $M(n)\backslash m$ is the set of CNs connected to BN_n excluding CN_m.}
6: {Compute the *a posteriori* pseudo-probabilities:}

$$Q_n^{(i)}(0) = k_n(1 - p_n) \prod_{m \in \mathcal{M}(n)} r_{mn}^{(i)}(0) \tag{38.5}$$

$$Q_n^{(i)}(1) = k_n p_n \prod_{m \in \mathcal{M}(n)} r_{mn}^{(i)}(1) \tag{38.6}$$

{where k_n are chosen to guarantee $Q_n^{(i)}(0) + Q_n^{(i)}(1) = 1$.}
7: {Perform hard decoding:}
$\forall n$,

$$\hat{c}_n^{(i)} = \begin{cases} 1 \leftarrow Q_n^{(i)}(1) > 0.5 \\ 0 \leftarrow Q_n^{(i)}(1) < 0.5 \end{cases} \tag{38.7}$$

8: **end while**

38.3 ALGORITHMS, IMPLEMENTATIONS, AND EVALUATIONS

To obtain efficient parallel algorithms for programmable LDPC decoders, we have developed compact data structures to represent the sparse binary parity-check **H** matrices that describe the Tanner graph. They adequately represent data as streams to suit the parallel processing of the SPA in GPU stream-based architectures, which represents a different solution from the conventional compress row storage (CRS) and compress column storage (CCS) formats used to represent sparse matrices in a compact format. The edges of the Tanner graph are represented using the addressing mechanism depicted in Figure 38.2, which facilitates the simultaneous access to different data elements required by the SPA. The compact data structures herein proposed also suit parallel architectures that can impose restrictions owing to the size and alignment of data. Associated to these compact data structures, we adopted an adequate thread-coarsening level capable of supporting the GPU execution of both horizontal and vertical kernels described in Algorithm 1.

38.3.1 Data Structures and Memory Accesses

The first challenge when developing a parallel LDPC decoder on a GPU is related with the need of holding the addresses of the Tanner graph's edges in memory (like the one in Figure 38.1). In order to update r_{mn} and q_{nm} values as described from Eq. 38.1 to Eq. 38.2, we developed dedicated data structures to organize these addresses in memory, namely, $\mathbf{H}_{CN}$ and $\mathbf{H}_{BN}$, as illustrated in Figure 38.2 for the case of Figure 38.1. Before computation starts, the addresses are read from a file to initialize the data structures, which are then copied from host memory to the device. As Figure 38.2 indicates, $\mathbf{H}_{BN}$ and $\mathbf{H}_{CN}$ structures are used to write r_{mn} and q_{nm} data elements in kernels 1 and 2, respectively,

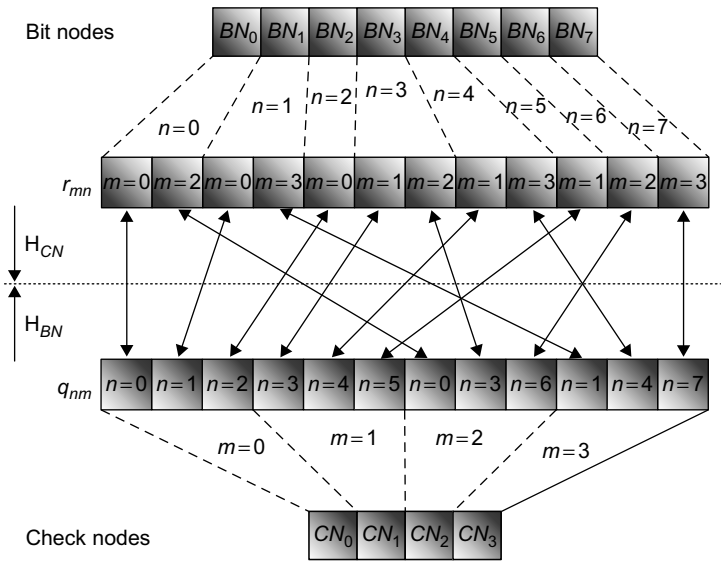

FIGURE 38.2

Data structures $\mathbf{H}_{CN}$ and $\mathbf{H}_{BN}$ that hold the addresses of **H**'s edges, illustrating the example in Figure 38.1.

while reading operations are performed sequentially. According to Algorithm 1, q_{nm} data are read and processed to update r_{mn} values, and the reciprocal applies for updating q_{nm} values. Because the Tanner graph can be shared among distinct codewords, for simultaneously decoding multiple codewords in parallel, the data structures $\mathbf{H}_{CN}$ and $\mathbf{H}_{BN}$ can be shared, as Figure 38.3 illustrates. The parallel algorithm developed exploits parallelism $P = 6 \times 16$ by decoding six clusters of data, with 16 packed codewords per cluster. Data are represented with 8-bit precision, which allows 16 elements to be fetched/stored from/into memory on a single 128-bit access.

38.3.2 Coarse versus Fine-Grained Thread Parallelism

The second challenge is to balance computation with memory access communications in order to avoid idle periods. This leads to the development of a thread-per-BN and thread-per-CN approach based on the well-known forward-and-backward method [5] to minimize computation and memory accesses in LDPC decoding.

Figure 38.3 illustrates this approach, by showing each thread th_n processing a complete row or column of $\mathbf{H}$. The adoption of the forward-and-backward optimization becomes even more important in the context of parallel computing architectures, because it minimizes memory accesses that often constrain the GPU performance. The proposed parallelization is more efficient than having each thread updating a single edge of the Tanner graph, a situation where redundant communications would dominate. Figure 38.4 illustrates the procedure used in the thread-per-CN approach, with threads j, u, and v processing the example in Figure 38.1, where each thread updates a complete row of $\mathbf{H}$ (or r_{mn} data). Each block of threads of the GPU grid updates groups of nodes of $\mathbf{H}$.

38.3.3 An LDPC Decoding Kernel

Figure 38.5 shows the source code for the LDPC decoder on the host and depicts the execution of the kernels on the GPU. It initializes the GPU grid, programs the kernel, and fires execution. Each iteration of the decoder executes the kernel twice, with the *pF* flag indicating a request for horizontal (kernel 1) or vertical processing (kernel 2), respectively. For the example of LDPC code (8000, 4000) in Figure 38.5, the grid is programmed with 125 blocks and 64 threads per block. For the horizontal

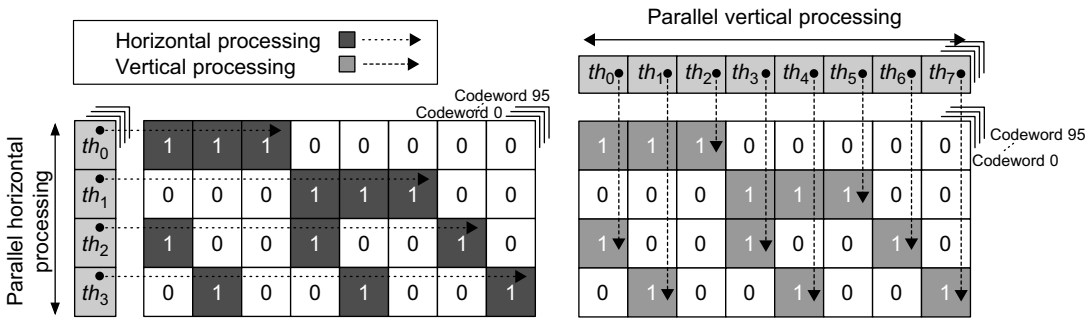

FIGURE 38.3

Task and data parallelism illustrating the example in Figure 38.1, where thread-per-CN (horizontal processing) and thread-per-BN (vertical processing) approaches are followed.

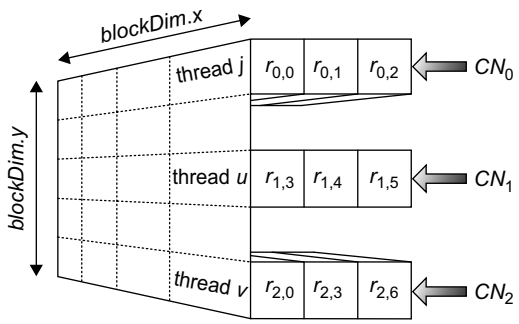

FIGURE 38.4

Block configuration of the GPU for the example in Figure 38.1. Each thread processes a complete row/column node of **H** according to Algorithm 1.

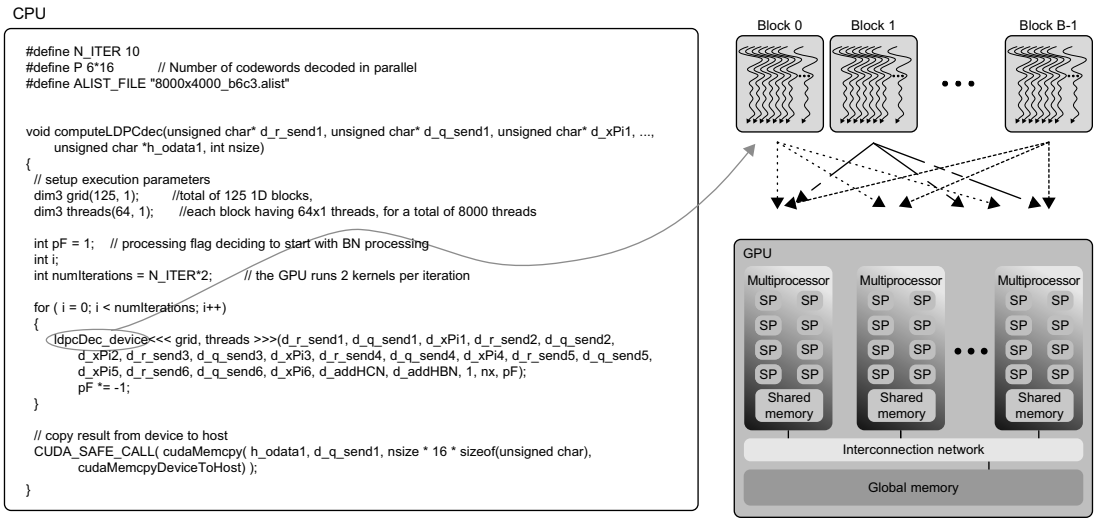

FIGURE 38.5

Host program launching parallel kernels on the device, where threads are grouped in blocks. The thread scheduler controls execution.

processing (kernel 1), the six input clusters of data are represented by elements *d_q_send1* to *d_q_send6*, while processed data is stored in *d_r_send1* to *d_r_send6*. The inverse applies for the vertical processing (kernel 2) where *d_xPi1* to *d_xPi6* represent the initial probabilities p_n in Algorithm 1 at the input of the decoder.

Listing 38.1 shows a segment of the device's kernel 1. The *update_rmn*() kernel in Listing 38.1 computes Eqs. 38.1 and 38.2 — the horizontal processing — and it updates and stores *d_r1* elements in noncontiguous memory addresses obtained from the data structure *d_HBN*. At this point, *d_q1* data is read sequentially, and the reciprocal applies for the *update_qnm*() kernel. Listing 38.1 shows the

```
update_rmn(uint4 *d_r1, uint4 *d_q1, ..., uint4 *d_r6, uint4 *d_q6,
                                short *d_HBN, int width, int height)
{
  unsigned int xIndex = blockDim.x * blockIdx.x + threadIdx.x;
  unsigned int yIndex = blockDim.y * blockIdx.y + threadIdx.y;
  unsigned int index_horiz = xIndex + width * yIndex;
  .
  .
  // processing first subword (reading d_q1 and producing d_r1) using the forward−and−backward method
  .
  .
  a2 = r−2.0 f*lei0; fin1 = r−2.0 f*lei1;
  fin2 = r−2.0 f*lei2; fin3 = r−2.0 f*lei3;
  fin4 = r−2.0 f*lei4; b5 = r−2.0 f*lei5;
  a3 = a2*fin1; b4 = b5*fin4;
  a4 = a3*fin2; b3 = b4*fin3;
  a5 = a4*fin3; b2 = b3*fin2;
  a6 = a5*fin4; b1 = b2*fin1;
  wm2 = a2*b2; wm3 = a3*b3;
  wm4 = a4*b4; wm5 = a5*b5;
  wm1_3 = b1*0.5 f+0.5 f; auxmul1_3 = wm2*0.5 f+0.5 f;
  auxmul2_3 = wm3*0.5 f+0.5 f; auxmul3_3 = wm4*0.5 f+0.5 f;
  auxmul4_3 = wm5*0.5 f+0.5 f; wm6_3 = a6*0.5 f+0.5 f;
  .
  .
  d_r1[dHBN[index0]].x = (((unsigned char)(wm1_0*SCALE))<<24)|(((unsigned char)(wm1_1*SCALE))
                <<16)|(((unsigned char)(wm12*SCALE))<<8)|((unsigned char)(wm1_3*SCALE));
  d_r1[dHBN[index1]].x = (((unsigned char)(auxmul1_0*SCALE))<<24)|(((unsigned char)(auxmul1_1*SCALE))
                <<16)|(((unsigned char)(auxmul1_2*SCALE))<<8)|((unsigned char)(auxmul1_3*SCALE));
  .
  .
   ... // processing next subword (reading d_q2 and producing d_r2), etc.
  .
  .
}

__global__ void ldpcDec_device( uint4 *d_r1, uint4 *d_q1, uint4 *d_xPi1, ..., uint4 *d_r6, uint4 *d_q6,
                        uint4 *d_xPi6, short *d_HCN, short *d_HBN, int width, int height, int procFlag )
{

  // procFlag decides between BN or CN processing
  if(procFlag>=0)                       //BN processing
  {     // Fire kernel 1
        update_rmn(     d_r1,    d_q1, ..., d_r6, d_q6, d_HBN, width, height );
  }
  else                                  // CN processing
  {     // Fire kernel 2
        update_qnm(     d_r1,    d_q1,    d_xPi1, ..., d_r6, d_q6, d_xPi6, d_HCN, width, height );
  }
}
```

Listing 38.1: Kernel code on the GPU.

level of parallelism $P = 96$ adopted. For each data element we allocate memory of type *unsigned char* (8 bits), while data buffers are of type *uint*4 (128 bits). In the beginning of processing, data are unpacked and normalized. After processing completes on the GPU, data elements are rescaled and repacked in groups of 8 bits in subword **x** (the type *uint4* disposes data into four 32-bit subwords **xyzw**). The same rule applies for the other codewords (**yzw** and also for data clusters 2, 3, 4, 5, and 6). This packing procedure optimizes data transfers between host and device, while at the same time increases the arithmetic intensity of the algorithm.

The same principle is followed for the computation of Eqs. 38.3 and 38.4 — vertical processing. Here, *d_q1* to *d_q6* data elements are processed and updated, after reading the corresponding *d_r1* to *d_r6* and *d_xPi1* to *d_xPi6* input elements.

38.4 FINAL EVALUATION

We adopt 8-bit data precision on the GPU, which contrasts with typical 5- to 6-bit representations used in VLSI, with advantages on BER. For example, for WiMAX code (1248, 624) [2] with an SNR = 2.5 dB, the BER performance with 8-bit precision is almost one order of magnitude better than by using 6-bit precision [10]. Furthermore, because it is programmable, the GPU allows using even higher precision to represent data (e.g., 16 or 32 bits) with superior BER results, at the cost of throughput performance.

The programming environments and platforms adopted for evaluating the proposed algorithm are the 8800 GTX GeForce GPU from NVIDIA with 128 stream processors (SP), each running at 1.35 GHz and with a total of 768 MB of video memory (VRAM), and the CUDA 2.0b programming environment. For comparison purposes, we have also programmed the LDPC decoder on an eight-core Intel Xeon Nehalem 2x-Quad E5530 multi-processor running at 2.4 GHz with 8 MB of L2 cache memory, by using OpenMP 3.0. The LDPC codes used represent regular codes with $w_c = 6$ edges per row and $w_b = 3$ edges per column. Table 38.1 presents the overall measured decoding throughputs running the SPA on the GPU and the speedup regarding the CPU [11, 12]. Experimental results in the table show that the GPU-based solution reports throughputs up to 35.1 Mbps for code (8000, 4000) (with 24,000

Table 38.1 LDPC decoding (SPA) throughput (Mbps) with an NVIDIA GPU using CUDA, and speedup regarding an Intel CPU (eight cores) programmed with OpenMP.

	LDPC code			
	(1024, 512)		(8000, 4000)	
No. of iterations	GPU	Speedup	GPU	Speedup
10	14.6	7.0	35.1	13.8
25	6.5	7.5	15.8	13.7
50	3.3	7.2	8.2	13.4

edges) running 10 iterations, while it achieves 2.5 Mbps for the same code decoded on the CPU using eight cores. The throughput values indicated for the GPU already consider computation time and data transfer time between host and device.

Algorithm 1 shows that the procedure to perform LDPC decoding is iterative and that each iteration executes two kernels. As illustrated in Figure 38.5 and Listing 38.1, each iteration implies calling the kernel *ldpcDec_device*() twice, with flag *pF* set to 1 or −1, respectively. This requires a number of communications between the host and the device that, under certain circumstances, may impact performance negatively, namely, if we are executing the application on a platform with multiple GPUs that share a bus with the host.

38.4.1 Conclusion

GPUs can be considered as reprogrammable and flexible alternatives to efficiently perform LDPC decoding. They report superior throughput performance when compared with multicore CPUs and also prove to be capable of competing with hardware-dedicated solutions [9] that typically involve high NRE costs. Also, they allow achieving BER performances that compare well against dedicated hardware systems because they permit higher precision to represent data.

All source code necessary to test the algorithm here proposed is available at *gpucomputing.net*, from where it can be downloaded.

38.5 FUTURE DIRECTIONS

The parallel algorithms here proposed have been validated under the context of CUDA. Nevertheless, if we run them on more recent GPU Tesla devices with a superior number of stream processors and larger global video memory, they expectedly will present higher throughputs, maintaining the BER performance. Also, research work can be done to improve the proposed algorithms in order to exploit the more powerful Fermi architectures.

Other improvements may still be achieved by using multi-GPU systems to exploit additional levels of data parallelism.

Moreover, the concept of belief propagation — namely, the SPA here described that holds the foundations of LDPC decoding — can be applied to other areas of computer science rather than error-correcting codes, such as stereo vision applied to robotics [6], or Bayesian networks among others. This chapter will also be of particular interest to researchers and engineers working on these topics.

References

[1] F. Kienle, T. Brack, N. Wehn, A synthesizable IP Core for DVB-S2 LDPC code decoding, in: Proceedings of the Design, Automation and Test in Europe (DATE'05), 7–11 March 2005, vol. 3, IEEE Computer Society, Munich, Germany, 2005, pp. 100–105.

[2] C.-H. Liu, S.-W. Yen, C.-L. Chen, H.-C. Chang, C.-Y. Lee, Y.-S. Hsu, S.-J. Jou, An LDPC decoder chip based on self-routing network for IEEE 802.16e applications, IEEE J. Solid-State Circuits 43 (3) (2008) 684–694.

[3] R.G. Gallager, Low-density parity-check codes, IRE Trans. Inf. Theory 8 (1) (1962) 21–28.
[4] C. Berrou, A. Glavieux, P. Thitimajshima, Near Shannon limit error-correcting coding and decoding: Turbo-codes (1), in: Proceedings of the IEEE International Conference on Communications (ICC '93), 23–26 May 1993, vol. 2, IEEE, Geneva, Switzerland, 1993, pp. 1064–1070.
[5] S.B. Wicker, S. Kim, Fundamentals of Codes, Graphs, and Iterative Decoding, Kluwer Academic Publishers, Norwell, MA 02061, 2003.
[6] A. Brunton, C. Shu, G. Roth, Belief propagation on the GPU for stereo vision, in: Proceedings of the 3rd Canadian Conference on Computer and Robot Vision (CRV'06), 7–9 June 2006, IEEE and IEEE Computer Society, pp. 76–76.
[7] R. Tanner, A recursive approach to low complexity codes, IEEE Trans. Inf. Theory 27 (5) (1981) 533–547.
[8] T.K. Moon, Error Correction Coding — Mathematical Methods and Algorithms, John Wiley & Sons, Inc., Hoboken, New Jersey, 2005.
[9] G. Falcao, V. Silva, L. Sousa, How GPUs can outperform ASICs for fast LDPC decoding, in: Proceedings of the 23rd International Conference on Supercomputing (ICS'09), ACM, New York, 2009, pp. 390–399.
[10] G. Falcao, V. Silva, L. Sousa, High coded data rate and multicodeword WiMAX LDPC decoding on Cell/BE, IET Electron. Lett. 44 (24) (2008) 1415–1417.
[11] G. Falcao, L. Sousa, V. Silva, Massive parallel LDPC decoding on GPU, in: S. Chatterjee, M.L. Scott (Eds.), Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'08), 20–23 February 2008, ACM, Salt Lake City, Utah, 2008, pp. 83–90.
[12] G. Falcao, L. Sousa, V. Silva, Massively LDPC decoding on multicore architectures, IEEE Trans. Parallel Distrib. Syst. (TPDS) (in press).