How GPUs Can Outperform ASICs for Fast LDPC Decoding

Gabriel Falcão
Instituto de Telecomunicações
Electrical & Comp. Eng. Dep.
University of Coimbra,
Portugal
gff@co.it.pt

Vitor Silva
Instituto de Telecomunicações
Electrical & Comp. Eng. Dep.
University of Coimbra,
Portugal
vitor@co.it.pt

Leonel Sousa
INESC-ID/IST
Electrical & Comp. Eng. Dep.
Technical University of Lisbon,
Portugal
las@inesc-id.pt

ABSTRACT

Due to huge computational requirements, powerful Low-Density Parity-Check (LDPC) error correcting codes, discovered in the early 1960s, have only recently been adopted by emerging communication standards. LDPC decoders are supported by VLSI technology, which delivers good parallel computational power with excellent throughputs, but at the expense of significant costs.

In this work, we propose an alternative flexible LDPC decoder that exploits data-parallelism for simultaneous multi-codeword decoding, supported by multithreading on CUDA-based graphics processing units (GPUs). The ratio of arithmetic operations per memory access is low for the efficient min-sum LDPC decoding algorithm proposed, which causes a bottleneck due to memory latency and data collisions. We propose runtime data realignment to allow coalesced parallel memory accesses to be performed by distinct threads inside the same warp. The memory access patterns of LDPC codes are random, which does not admit the simultaneous use of coalescence in both read and write operations of the decoding process. To overcome this problem we have developed a data mapping transformation which allows new addresses to be contiguously accessed for one of the mentioned memory access types. Our implementation shows throughputs above 100Mbps and BER curves that compare well with ASIC solutions.

Categories and Subject Descriptors

I.3.1 [**Computer Graphics**]: Hardware Architecture—*Parallel processing*

General Terms

Algorithms, Design, Performance

Keywords

Parallel Processing, Multithreading, Memory Coalescence, Graphics Processing Units, Low-Density Parity-Check Codes

ICS'09, June 8–12, 2009, Yorktown Heights, New York, USA.

1. INTRODUCTION

In 1962 [1] Robert Gallager proposed low-density parity-check (LDPC) codes, which are powerful error correcting codes. They are usually represented by bipartite graphs formed by bit nodes (BNs) and check nodes (CNs), and linked by bidirectional edges also called *Tanner* graphs [2]. LDPCs are linear (N, K) block codes defined by parity-check sparse binary $\mathbf{H}$ matrices of dimension $M \times N$, with $M = N - K$. Due to their huge computational demands, only recently they have recaptured the attention of the scientific community. In fact, only after the appearance of turbo codes [3] in 1993, did Mackay and Neal rediscover LDPC codes [4], inspiring the scientific community to develop efficient LDPC coding solutions for emerging data communication systems. They have been recently adopted by the DVB-S2 [5], DVB-C2, DVB-T2, WiMAX (802.16e), Wifi (802.11n), 10Gbit Ethernet (802.3an), and other new standards for communication and storage applications.

LDPC decoders, in particular the min-sum algorithm, are based on the belief propagation of messages between the nodes of the *Tanner* graph, which requires very intensive computation. Therefore, dedicated application-specific integrated circuit (ASIC) solutions have been proposed for LDPC decoding in the last few years [6, 7, 8]. The need for hardware dedicated solutions is justified by the high throughputs required by the standards that incorporate such LDPC decoders; the DVB-S2 demands 90Mbps, while the WiMAX can require up until 75Mbps. VLSI technology usually implements the min-sum algorithm based on integer-arithmetic operations [9], where quantization naturally influences the coding gains of an LDPC code. To overcome these and other difficulties, a variety of solutions have been proposed that compare well with LDPC competitors, namely the turbo codes [3]. They report excellent throughputs for low power consumption solutions [10], or use efficient interconnection frameworks to achieve high-throughput LDPC decoders [11]. However, they typically involve considerable resources and require complex projects, with long development times and associated costs, to produce non-flexible and non-scalable solutions. Alternative programmable solutions such as software defined radio (SDR) hardware platforms [12], which produce acceptable throughputs, have already been proposed for LDPC decoding.

As more transistors are being placed on a die and, more specifically, as new trends in computer architectures show that the number of cores on a chip is increasing steadily every year at a considerable rate, new forms of parallelism are being exploited. Multicore architectures now provide

large single instruction multiple data (SIMD) units for vector processing [13], while at the same time support multithreading. Multithreading has been exposed to hide memory latency [14] and parallel coalesced memory accesses have been introduced in recent architectures, also to reduce time spent performing memory accesses. Generally accessible consumer products such as the GPUs from NVIDIA and AMD/ATI, or the Cell Broadband Engine (Cell/B.E.) Architecture from Sony-Toshiba-IBM, already incorporate a large number of cores and are supported by convenient languages and programming tools [15, 16] that allow to expose parallelism. A real possibility for data-parallel computing lies in the use of GPUs, fostered by continuous performance improvements for visualization technology in the games industry. Even a commodity personal computer now offers high quality graphics created by real-time computing, mainly due to the GPU included. Furthermore, the performance of GPUs is still improving significantly and presently offers remarkable floating-point computational power for applications which demand a huge workload.

The GPU's highly-parallel nature associated to a good memory bandwidth has encouraged high performance computing research to apply the GPU's computational power to tackle general purpose applications (GPGPU) [19, 20, 21]. To hide the complexity of having to control the GPU's hardware from the programmer, new programming interface tools [15] such as the Compute Unified Device Architecture (CUDA) from NVIDIA [16] have been developed. The CUDA, which provides a powerful environment for managing computations on the Tesla NVIDIA GPUs, was selected as the programming interface tool for this work.

Although an LDPC decoder that uses the sum-product algorithm (SPA) has been described in [16], in this paper we propose a novel approach for the more efficient min-sum algorithm, which delivers a 10x higher throughput. The authors have also proposed an LDPC decoder on the Cell/B.E. based on the min-sum that delivers a very good throughput [17]. Nevertheless, the new GPU-based approach described here is even more efficient, especially for codes of large dimensions. It exploits data-parallelism using multithread capabilities and parallel memory accesses based on coalescence on GPUs supported by CUDA that can produce performances comparable to dedicated ASIC LDPC decoders. The data structures that represent the *Tanner* graph are shared among different threads, and this allows the proposed 8-bit data precision solution to perform efficient and simultaneous multicodeword decoding using 128 bits per memory access. The impossibility of having all data contiguously aligned at compile time forced us to develop an automatic mapping procedure for data alignment at runtime, which allows the use of efficient parallel coalesced memory read accesses. Furthermore, the adopted 8-bit data precision representation is more favorable than the typical 5 to 6-bit precision approach used in VLSI, and a lower bit error rate (BER) can be expected regarding to hardware-dedicated LDPC decoders [9]. We also propose an analytical model that predicts well the execution time of the min-sum LDPC decoder on GPUs.

The parallel algorithms for LDPC decoding on GPUs supported by CUDA, the runtime automatic data realignment that allows the use of coalescence, the concept of multicodeword decoding, the proposed models that predict the performance of an LDPC decoder on GPUs, and the important notion that such a class of computationally demanding algorithms is no longer limited to be performed only by ASICs, are the main contributions of this paper.

These contributions are presented in the paper according to the following organization. The next section introduces the min-sum algorithm used in LDPC decoding and analyzes its computational/memory access complexity. The third section investigates how an LDPC decoder algorithm can exploit parallelism to be efficiently computed on a GPU. Section 4 analyzes parallelism on CUDA devices, namely multithreading and coalesced memory accesses. Section 5 reports experimental results and the last section contains some concluding remarks.

2. LDPC CODES

Graph theory has guided error correcting codes to performances near the Shannon limit [18]. If we consider a set of bits, or codeword, to be transmitted over a noisy channel and a bipartite *Tanner* graph with a large number of connecting nodes, or edges, the certainty of an information bit can be spread over several bits of the codeword which, under certain circumstances, allows erroneous bits on the decoder side of the algorithm to be corrected. Among several reasoning algorithms used to exploit probabilistic relationships between neighboring nodes imposed by parity-check node restrictions, is the min-sum algorithm. It is one of the most efficient algorithms used for LDPC decoding [23] and consists of a simplification of the well known SPA [1] [4] [23]. Like the SPA, the min-sum is based on the intensive belief propagation between nodes connected as indicated by the *Tanner* graph edges (see example in figure 1), but uses only comparison and addition operations. The workload produced is lower than the one in the SPA, but still quite significant. If the number of nodes is large (in the order of thousands) the workload can become extraordinarily heavy.

2.1 Min-Sum algorithm for LDPC decoding

The min-sum based LDPC decoder proposed here uses the λ-min algorithm, with $\lambda = 2$, and it does not include LUTs for correction [24].

Let us denote the log-likelihood ratio (LLR) of a random variable as $L(x) = ln(p(x = 0)/p(x = 1))$. Also, considering the message propagation from nodes CN_m to BN_n and vice-versa, the set of bits that participate in check equation m, with bit n excluded, is represented by $N(m)\backslash n$ and, similarly, the set of check equations in which bit n participates with check m excluded is $M(n)\backslash m$. LP_n designates the *a priori* LLR of BN_n, derived from the values received from the channel, and Lr_{mn} is the message that is sent from CN_m to BN_n, computed based on all received messages from BNs $N(m)\backslash n$. Also, Lq_{nm} is the LLR of BN_n, which is sent to CN_m and calculated based on all messages received from CNs $M(n)\backslash m$ and the channel information LP_n.

For each node pair (BN_n, CN_m) we initialize Lq_{mn} with the probabilistic information received from the channel, and then we proceed to the iterative body of the algorithm by performing:

1. **Horizontal Processing** – Calculating the LLR of messages sent from CN_m to BN_n:

$$Lr_{mn} = \prod_{n' \in N(m)\backslash n} \alpha_{n'm} \min_{n' \in N(m)\backslash n} \beta_{n'm}, \qquad (1)$$

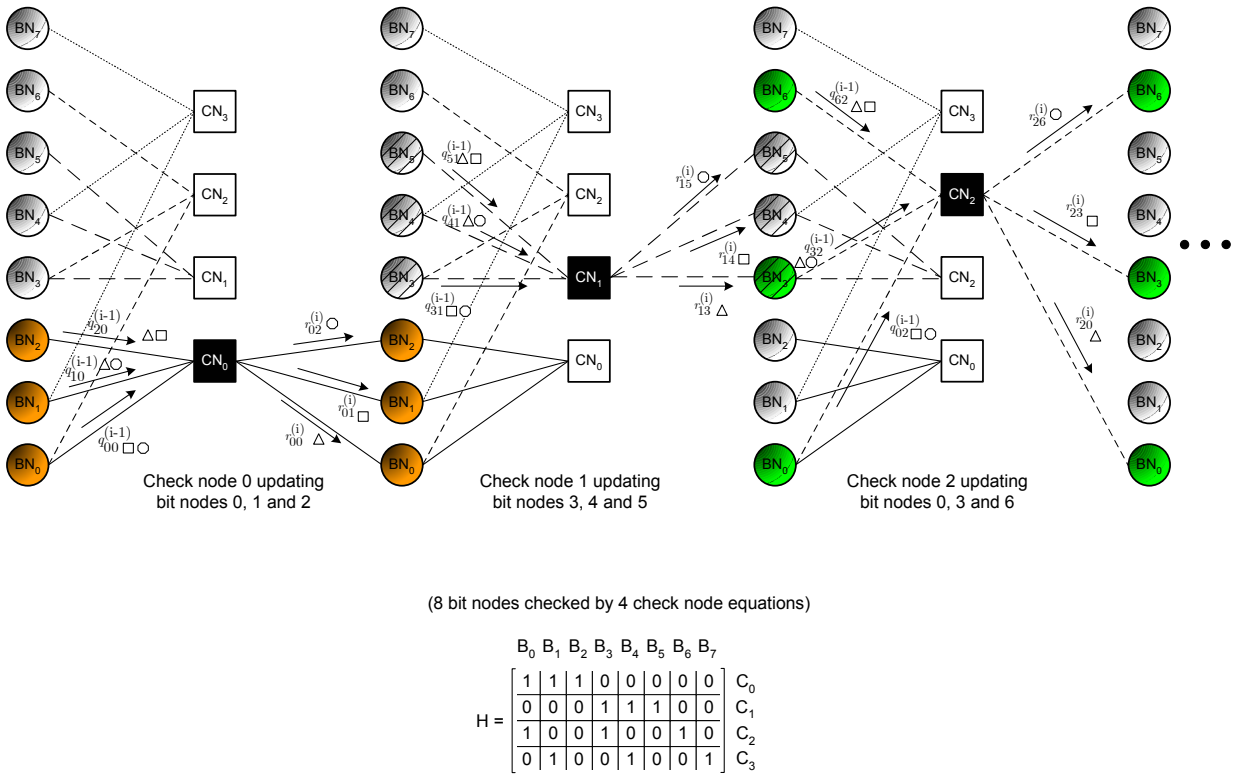

Figure 1: Tanner graph representation of a 4×8 H matrix, with a few messages being exchanged between CN_m and BN_n as indicated for the horizontal processing.

with

$$\alpha_{nm} \triangleq sign\left(Lq_{nm}\right), \tag{2}$$

and

$$\beta_{nm} \triangleq \left|Lq_{nm}\right|. \tag{3}$$

2. **Vertical Processing** – Calculating the LLR of messages sent from BN_n to CN_m:

$$Lq_{nm} = LP_n + \sum_{m' \in M(n) \setminus m} Lr_{m'n}. \tag{4}$$

3. Finally, we compute the *a posteriori* pseudo-probabilities and perform hard decoding:

$$LQ_n = LP_n + \sum_{m' \in M(n)} Lr_{m'n}. \tag{5}$$

2.2 Decoding complexity

Given an **H** matrix with M CNs (or rows) and N BNs (or columns), a mean row weight w_c and a mean column weight w_b of **H**, with $w_c \geq 2$ and $w_b \geq 2$, and depending on the size of the codeword and channel conditions (SNR), the minimal throughput necessary for an LDPC decoder to achieve a desired performance can require a substantial number of computations and memory access operations per second which typically are only achieved using VLSI technology. Table 1 presents the number of compare operations, and Abs (absolute value calculations), Min (the smallest of two given numbers), and Xor (for signal calculations) operations per iteration. It also shows the number of memory accesses per iteration necessary to update (1) to (5). They exhibit linear complexity as a function of w_c and w_b and can be useful for the investigation of new strategies for parallelizing LDPC decoding on platforms with manycores, where memory access conflicts are often the limiting factor. The ratio of arithmetic operations per memory access, here defined as arithmetic intensity, can be described by (6), for a regular code with rate $= \frac{1}{2}$, where $N = 2M$, $w_c = 2w_b$, $w_c \geq 2$ and $w_b \geq 2$. The arithmetic intensity $\alpha_{Min-Sum}$ for the min-sum-based algorithm is then:

$$\alpha_{Min-Sum} = \frac{(4w_c - 1)M + 2w_bN}{2w_cM + (2w_b + 1)N} \approx 1.5. \tag{6}$$

The value $\alpha_{Min-Sum}$ achieved is > 1 but not $\gg 1$, which shows that when applying the λ-min algorithm to manycore architectures, memory accesses have to be optimized in order to minimize the effects of latency and data conflicts.

3. EXPLOITING PARALLELISM TO COMPUTE LDPC DECODING ON GPUS

The GPU used in this work represents an evolution from previous GPU stream-based architectures. It is based on a computer unified architecture where geometry, pixel and vertex shaders share common resources. Figure 2 depicts the architecture of the NVIDIA 8800 GTX GPU, which has 8 stream processors (SP) per multiprocessor and 16 multiprocessors, for a total of 128 SP that can support a peak performance of 350 GFLOPS [16]. Each multiprocessor has 8192 available registers. The programming interface used is the CUDA, composed of its own API and runtime environment,

Table 1: Number of arithmetic and memory access operations per iteration for the optimized min-sum algorithm.

Min-Sum - Horizontal Processing	
	Number of operations
Comparisons	$w_c M$
Abs	$w_c M$
Min	$(w_c - 1)M$
Xor	$w_c M$
Memory Accesses	$2w_c M$
Min-Sum - Vertical Processing	
	Number of operations
Additions/Subtractions	$2w_b N$
Memory Accesses	$(2w_b + 1)N$

two mathematical libraries and a hardware driver that only supports the most recent GPUs from NVIDIA [16]. CUDA exploits data-parallelism by executing in parallel multiple threads. Multithreading is also a powerful technique used to hide latency. In fact, if the number of threads is large

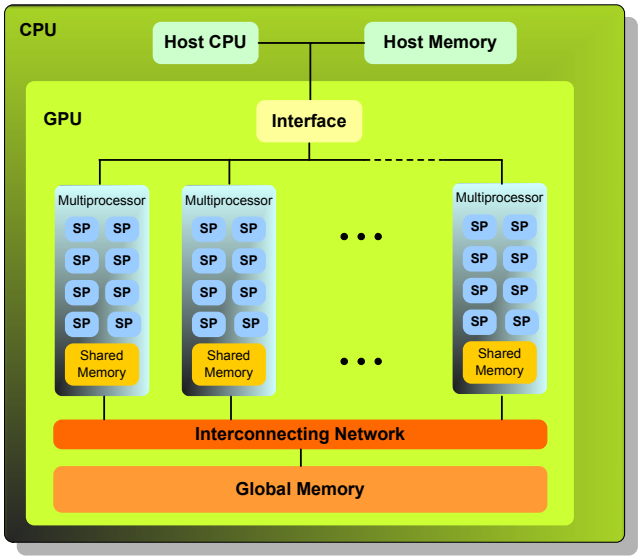

Figure 2: The compute unified 8800 GTX GPU architecture with a total of 128 SP and 8 SP per multiprocessor.

enough the idle time that each thread spends accessing memory no longer represents a significant penalty in the overall performance. It can be masked because the thread scheduler commutates and starts processing a different thread, while the other(s) finish(es) accessing data in memory. The GPU is formed by a grid of thread blocks of adjustable dimensions. The size of the block and the number of threads executing per block can be configured according to the algorithm. Each multiprocessor shown in figure 2 can control more than one block of threads, with a maximum of 512 threads per block. The specificities of the algorithm and the number of necessary registers per thread (code dependent) will mainly determine the best size of the grid. The warp size of the 8800 GTX GPU adopted here is 32 threads. Each of the 16 multiprocessors time-slices the 32 threads in a warp among its 8 stream processors. Instructions are executed synchronously in all the threads of a warp (using SIMD).

The GPU data-parallel computation is performed under the SIMD and the Single Program Multiple Data (SPMD) models. The SPMD model exploits workload balancing efficiency, as it attempts to assign an approximate workload to each core by selecting different execution paths based on conditions that check data dependencies. However, allowing the existence of inconvenient divergent threads inside a warp (only threads on the same path can be executed at the same time), or using a number of threads smaller than 32 can cause a wasting of resources or render them temporarily idle. Understanding how threads are grouped into warps is fundamental to choosing the best block size configuration. In this work, the use of divergent threads has been minimized by adopting regular LDPC codes. Irregular codes could also be used, but at the expense of workload unbalancing.

Under the proposed parallel approach, for minimizing memory accesses, each thread is responsible for updating all the BNs in the same row during the horizontal processing, while the equivalent happens for the vertical processing and all respective CNs associated with a column (see figure 5). The GPU is capable of reading 32-bit, 64-bit, or 128-bit words from global memory into registers. We chose to read 128-bit words. Using 8-bit precision to represent data, 16 data elements corresponding to 16 distinct codewords are accessed in a single operation and then all data words are unpacked inside the GPU before the processing starts. This is important to improving efficiency because it increases the arithmetic intensity of the program. Figure 3 shows the data alignment chosen. However, the GPU is not an autonomous

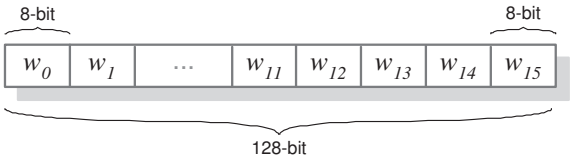

Figure 3: A 128-bit word read from memory in a single instruction holds 16 data elements corresponding to 16 distinct codewords, each having 8-bit precision.

processing system. One very important aspect concerns the need to transfer data between the host (CPU) and the GPU. The high bandwidth available in modern GPUs attenuates this problem. The LDPC decoder under test supports peak data transfers that range approximately from 1.2 to 1.3GByte/s experimentally measured between the host and the 8800 GTX device connected to a workstation through a PCIe bus. The main limitation here is imposed by the PCIe bus under test.

3.1 Data structures to represent H

We propose to represent the **H** matrix that defines the *Tanner* graph of an LDPC code, in two separate data structures, namely $\mathbf{H_{BN}}$ and $\mathbf{H_{CN}}$. This is because one iteration

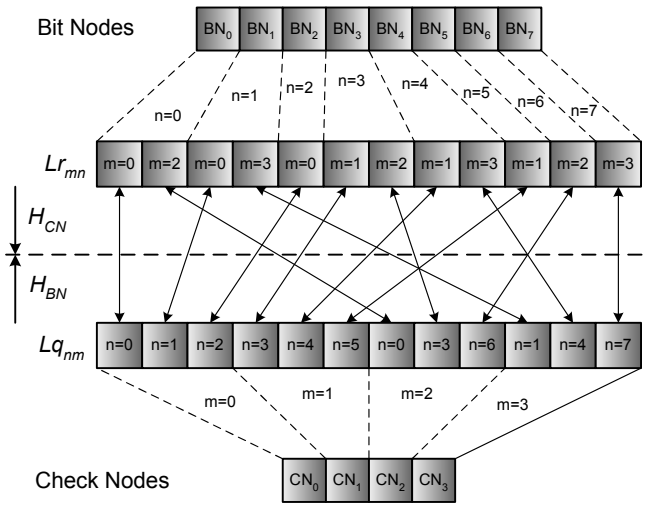

Figure 4: Data structures representing the edges associated to common BNs and CNs for the example in figure 1.

of the LDPC decoder can be decomposed into horizontal and vertical processing, two independent steps described previously in (1) and (4). $\mathbf{H_{BN}}$ defines the data structure used in the horizontal step. This data structure is generated by scanning the $\mathbf{H}$ matrix in a row-major order and by sequentially mapping only the BN edges associated with non-null elements in $\mathbf{H}$ for each CN equation (in the same row). Algorithm 1 describes this procedure in detail for a matrix having M rows and N columns. Steps 4 and 5 show that only edges representing non-null elements in a row associated with the same CN are collected and stored in consecutive positions inside $\mathbf{H_{BN}}$. The $\mathbf{H_{CN}}$ data structure is used in the vertical

Algorithm 1 Generating the compact $\mathbf{H_{BN}}$ structure from the original $\mathbf{H}$ matrix for regular codes

1: {Reading a binary $M \times N$ matrix $\mathbf{H}$}
   $\{idx = 0;\}$
2: **for all** CN_m (rows in $\mathbf{H}_{mn}$) : **do**
3: **for all** BN_n (columns in $\mathbf{H}_{mn}$) : **do**
4: **if** $\mathbf{H}_{mn} == 1$ **then**
5: $\mathbf{H_{BN}}[idx++] = mw_b + n;$
6: **end if**
7: **end for**
8: **end for**

processing step and it can be defined as a sequential representation of the edges associated with non-null elements in $\mathbf{H}$ connecting every BN to all its neighboring CNs (in the same column). It is generated by scanning the $\mathbf{H}$ matrix in a column-major order. Both data structures are depicted in figure 4, where the example shown in figure 1 is mapped into the corresponding edge connections that form such a *Tanner* graph.

The *Tanner* graph is common to all 16 words under decoding. Such an important property allows the simultaneous decoding of different codewords, which can be performed in parallel on the multiple threads of the GPU, as described next.

Algorithm 2 Generating the compact $\mathbf{H_{CN}}$ structure from the original $\mathbf{H}$ matrix for regular codes

1: {Reading a binary $M \times N$ matrix $\mathbf{H}$}
   $\{idx = 0;\}$
2: **for all** BN_n (columns in $\mathbf{H}_{mn}$) : **do**
3: **for all** CN_m (rows in $\mathbf{H}_{mn}$) : **do**
4: **if** $\mathbf{H}_{mn} == 1$ **then**
5: $\mathbf{H_{CN}}[idx++] = nw_c + m;$
6: **end if**
7: **end for**
8: **end for**

3.2 Data dependencies

The w_cM messages involved in the updating of the horizontal step and the w_bN messages used in the procedure for the vertical step show no data dependency constraints. These operations can be parallelized by adopting a suitable scheduling that supports the updating of different messages for different vertices, simultaneously, in distinct parts of the *Tanner* graph, while guaranteeing data consistency. In every iteration, all the messages used in the computation of a new message were obtained from the previous iteration. This principle is fundamental when developing a min-sum based LDPC decoder for parallel architectures. Analyzing the example illustrated in figure 1 for the horizontal processing, the update of the 3 BNs associated with the first CN equation (first row of $\mathbf{H}$ shown in figures 1 and 5) can be performed in parallel with the update of other CN equations, by different threads (represented in figure 5 by th_n), without any kind of conflict between nodes. The remaining messages on the *Tanner* graph show that the same principle applies to the other CN equations of the example. A similar conclusion can be drawn when analyzing the vertical processing. In spite of the irregular memory access pattern, in this case too it is possible to parallelize the processing of the w_bN new messages, as it is the case of $BN_0 \rightarrow CN_0$ and $BN_0 \rightarrow CN_2$, that exemplify the update of the two messages associated to BN0 (first column of $\mathbf{H}$ shown in figures 1 and 5) and that can be performed in simultaneous with other messages associated with different BNs.

It should be noted that if data elements are first aligned in a proper order, under certain circumstances the memory accesses can also be parallelized, allowing considerable gains. This is even more significant because not only are global memory access times reduced, but also far fewer collisions occur when accessing data in memory. This will be detailed in the next section.

4. PARALLEL COMPUTING MODEL FOR LDPC DECODING

Consider N_{it} LDPC decoding iterations being performed simultaneously on P codewords each of size N. Let us denote by Th the total number of threads running on the GPU, with MP being the number of multiprocessors and SP the equivalent number of stream processors per multiprocessor, with frequency of operation f_{op}. If we assume that each thread is expected to perform a total of Mop memory access operations with latency L per memory operation, and that the number of cycles per iteration for non-divergent instructions is represented by ND_{op}, while the opposite number of divergent cycles per iteration is D_{op}, we can predict the processing time per codeword T_{proc} for the LDPC decoder

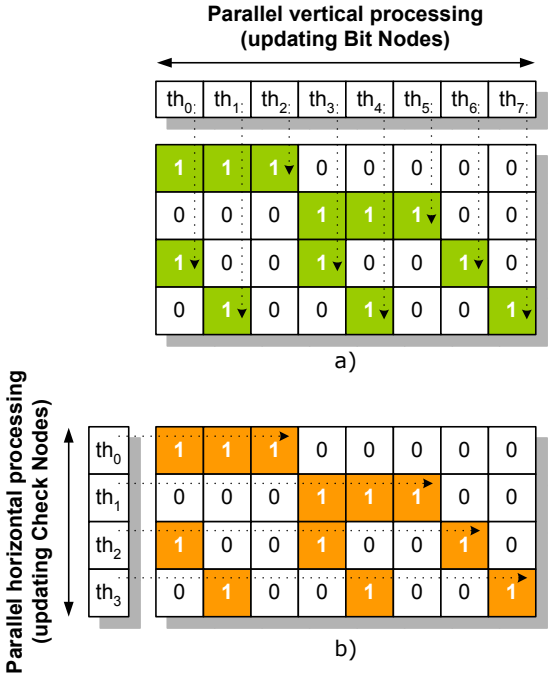

Figure 5: Multiple threads processing in parallel the a) vertical and b) horizontal kernels, for the example shown in figure 1.

on this parallel architecture as:

$$T_{proc} = N_{it}\frac{\frac{Th}{MP} \times (\frac{ND_{op}}{SP} + D_{op}) + Th \times Mop \times L}{f_{op} \times P}. \quad (7)$$

Then, the global decoding time can be obtained by including the time necessary to transfer data:

$$T_{total} = T_{host \to gpu} + T_{proc} + T_{gpu \to host}, \quad (8)$$

where $T_{host \to gpu}$ and $T_{gpu \to host}$ represent data transfer times between host and GPU.

Examination of (7) shows that a significant portion of the total processing time is expected to be spent on memory operations. If we consider that data collisions do occur, the problem becomes even more noted.

4.1 Multithread coalesced memory accesses

The global memory of the GPU is not cached and the latency accessing memory can range from 400 to 600 clock cycles. We have seen that the use of multithreading can minimize this problem on the GPU. Nevertheless, it is possible to be even more efficient. The amount of time spent on memory accesses defined in (7) can be reduced by a factor of 16 by using coalescence. The maximum fine-grain level of parallelism on the NVIDIA 8800 GTX GPU defines that, instead of performing 16 individual memory accesses, all the 16 threads within a half-warp can have a unique coalesced read or write access to global memory. This is only possible if data lies on a contiguous memory block, where the k^{th} thread accesses the k^{th} data element, and if data and addresses respectively obey specific size and alignment requirements. The activity of half-warps 0 and 1 is depicted in figure 6 where threads t_0 to t_{15} were captured reading data in a single coalesced memory transaction from the GPU's global memory. For coalesced accesses to take place in a single data transaction, data has to be previously realigned in a contiguous mode (from a thread perspective), and the alignment cannot be performed at compile time. The permutation we have to apply to data elements $\mathbf{H_{BN}}$, $\mathbf{H_{CN}}$, Lr_{mn} and Lq_{nm} can be described from (9) to (16). Data permutations necessary for the horizontal processing are:

$$newAddr = j * p + k \;\; div \;\; w_c + (k \;\; mod \;\; w_c) * 16, \quad (9)$$

with $p = w_c * 16$, $j = i \; div \; p$, $k = i \; mod \; p$, and $0 \leq i \leq Edges - 1$. Then,

$$\breve{\mathbf{Lr}}^{c}_{mn}(i) = \mathbf{Lr}_{mn}(newAddr). \quad (10)$$

The new permuted memory addresses become:

$$e = \mathbf{H}_{BN}(newAddr), \quad (11)$$

$$\breve{\mathbf{H}}^{c}_{BN}(i) = j * p + k \; div \; w_b + (k \; mod \; w_b) * 16, \quad (12)$$

where $p = w_b * 16$, $j = e \; div \; p$ and $k = e \; mod \; p$.
The permutations necessary for the vertical processing are:

$$newAddr = j * p + k \;\; div \;\; w_b + (k \;\; mod \;\; w_b) * 16, \quad (13)$$

where $p = w_b * 16$, $j = i \; div \; p$ and $k = i \; mod \; p$.

$$\breve{\mathbf{Lq}}^{c}_{nm}(i) = \mathbf{Lq}_{nm}(newAddr), \quad (14)$$

and the new memory addresses become:

$$e = \mathbf{H}_{CN}(newAddr), \quad (15)$$

$$\breve{\mathbf{H}}^{c}_{CN}(i) = j * p + k \; div \; w_c + (k \; mod \; w_c) * 16, \quad (16)$$

where $p = w_c * 16$, $j = e \; div \; p$, and $k = e \; mod \; p$.

With coalesced memory accesses, the model proposed in (7) can be even more detailed in order to incorporate the parallelism in memory accesses by a factor of 16. Also, considering we apply efficient programming techniques which avoid the use of branching, the number of divergent threads is very low, i.e., $D_{op} \approx 0$. Thus, the average number of cycles is similar in both the horizontal and vertical processing steps, allowing (7) to be rewritten as:

$$T_{proc} = \frac{2N_{it} \times \frac{Th}{MP} \times \frac{ND_{op}}{SP}}{f_{op} \times P} + \underbrace{\frac{Th \times Mop \times L}{f_{op} \times P}}_{T_{MA}}, \quad (17)$$

where T_{MA} defines all memory access operations and can be decomposed in memory accesses to CNs and BNs, respectively:

$$T_{MA} = T_{MA\_BNs} + T_{MA\_CNs}. \quad (18)$$

T_{MA_BNs} defines the time spent on memory accesses to update CNs (accessing BNs):

$$T_{MA\_BNs} = N_{it}\frac{M \times w_c \times L + M \times 2w_c \times L/16}{f_{op} \times P}, \quad (19)$$

and T_{MA_CNs} represents the time needed to update BNs (accessing CNs):

$$T_{MA\_CNs} = N_{it}\frac{N \times w_b \times L + N \times (2w_b + 1) \times L/16}{f_{op} \times P}. \quad (20)$$

The last partials in (19) and (20) show memory read operations that are divided by 16 to model the parallelism

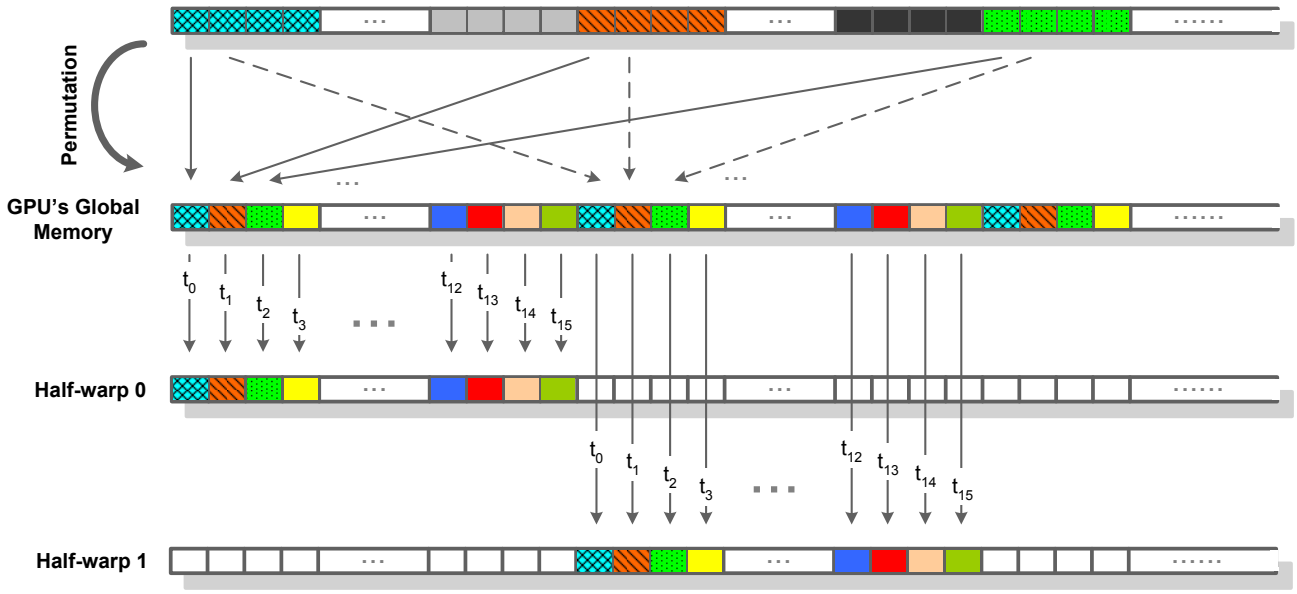

Figure 6: Coalesced memory read operations performed by the 16 threads of a half-warp in a single memory access.

Table 2: Parity-check matrices under test.

Matrix	Size ($M \times N$)	Total number of edges ($w_c \times M$ or $w_b \times N$)
A	512×1024	3072
B	4000×8000	24000
C	10000×20000	60000

obtained with the introduction of coalesced reading operations. The nature of the LDPC decoder algorithm allows either read or write accesses to be coalesced, but not both. If data is read on a contiguous mode, the store procedure has to be irregular and vice-versa. This is why only read accesses are coalesced in the implementation described in this paper.

The recent GT2xx GPUs from NVIDIA have less demanding requirements for coalescence than those just discussed for the G8x family. Nevertheless, code hand-tuned for coalesced memory accesses on the G8x series family scales well on a GT2xx device.

5. EXPERIMENTAL RESULTS

The experimental setup to evaluate the performance of the proposed parallel LDPC decoder on the GPU consists of an NVIDIA 8800 GTX GPU with 16 multiprocessors and 128 stream processors, each running at 1.35 GHz and with a total of 768 MByte of video RAM (VRAM) memory, and was programmed using the C language and the CUDA programming interface (version 2.0b). The characteristics of the matrices used to perform LDPC decoding are presented in table 2 and range from matrix A to matrix C: they all have 6 edges per row (w_c) and 3 edges per column (w_b).

Matrices A to C were run using a number of threads per block that ranges from 64 to 128. The variation in performance is minimal and in table 3 we show only the best results achieved. Matrix A uses 64 threads per block and a total of 16 blocks. Matrices B and C use 128 threads per block and 63 and 157 blocks, respectively. The maximum number of threads allowed per block is 128, because the number of registers per multiprocessor necessary for the algorithm to execute is high. Compiling with 256 threads per block produces code that will not execute. The number of blocks has to be chosen according to the number of threads desired per block, and it also depends on the length of the LDPC code. For small LDPC codes such as the one defined by matrix A, we achieve better results with fewer threads per block. But larger codes, such as B and C, are decoded faster if we use more threads per block. The decoding times reported in table 3 define global processing times, which include data transfers according to (8), to simultaneously decode 16 codewords with length depending on the matrix. Data elements were represented and processed using 8-bit precision, causing 16 data elements to be read in each 128-bit memory access. The GPU seems to perform better for larger codes. If we consider throughput, it is interesting to see that the GPU achieves a higher coded data rate for matrix C, which has 60000 edges to compute. But it is likely that beyond a certain dimension of an LDPC code the bottleneck may be in the data transfer between host and GPU. A very significant throughput of 109.1 Mbps is achieved for matrix C running 10 iterations. It decodes 16 codewords simultaneously, each of length 20000-bit, in 2.932 ms. Table 4 presents some comparisons with recently published work. The throughputs obtained compare well with those reported in recent publications [7, 12, 25, 26] that describe hardware solutions (ASIC and others) to implement LDPC decoders.

Figure 7 shows that the average number of iterations in the LDPC decoding process depends on the SNR (represented by E_b/N_0) of data present at the input of the de-

Table 3: LDPC decoding times on the GPU (ms) and corresponding throughputs (Mbps) for a parallelism degree of 16 and 8-bit data precision using the min-sum algorithm.

	Time (ms)				Throughput (Mbps)			
Number of iterations	10	20	30	50	10	20	30	50
Matrix A	0.393	0.597	0.765	1.102	41.7	27.4	21.4	14.9
Matrix B	1.331	1.589	1.812	2.261	96.2	80.6	70.6	56.6
Matrix C	2.932	3.176	3.425	3.841	109.1	100.8	93.4	83.3

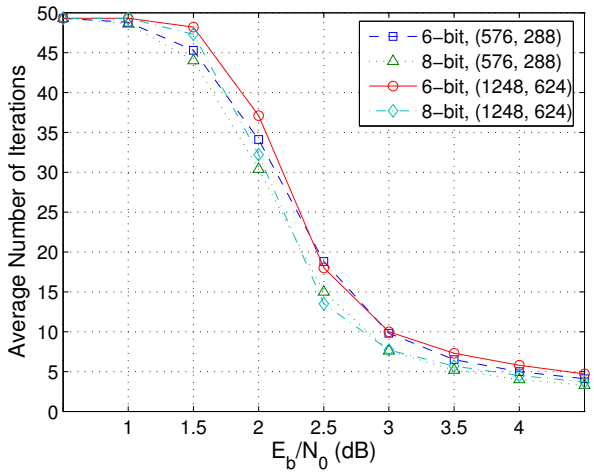

Figure 7: Average number of iterations per SNR, comparing 8-bit and 6-bit data precision representations for WiMAX LDPC codes (576,288) and (1248,624) running a maximum of 50 iterations.

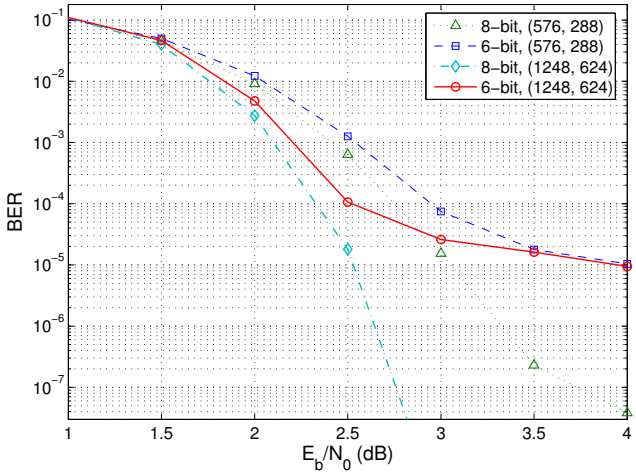

Figure 8: BER curves comparing 8-bit and 6-bit data precision for WiMAX LDPC codes (576,288) and (1248,624).

coder. The average number of iterations naturally influences the decoding time. For a low SNR the average number of iterations necessary for a codeword to converge is greater than for higher SNRs. As shown in figure 7, if we decode for example 10^8 bits using code (576, 288), for a maximum number of iterations equal to 50 and an SNR of 1 dB, we can observe that the average number of iterations nearly reaches its maximum (50 in this case). On the other hand, if the SNR rises to 4 dB, the average number of iterations drops below 4. Depending on the adopted code, LDPC decoders can achieve very good performances (low BER) even for low SNRs, at the cost of computational power. Also, it can be seen that the 8-bit precision adopted here compares favorably in terms of the number of iterations needed for the algorithm to converge, when we look at state-of-the-art ASIC LDPC decoders in the literature [7, 12, 25, 26] that use lower precision architectures (e.g. 5 to 6-bits). The 8-bit solution developed helps to cut the computational workload on the GPU, and thus the processing time.

The random nature of LDPC codes does not allow to simultaneously perform read and write operations in contiguous blocks of memory. One of them has to perform irregular accesses. By placing the data to be read in contiguous mode, it becomes clear that only coalesced memory read operations are performed. Table 5 shows that because we use coalescence to read data, the percentage it takes of the overall processing time remains approximately constant at around 20%. But non-coalesced memory write operations in memory can take up a very important portion of the processing time. For matrix B, which has 24000 edges, the amount of time spent on write accesses to memory is superior to 39%, and for matrix C, with 60000 edges, it is above 50%. This shows that using coalescence is extremely important not only because parallel accesses to memory can be performed by multiple threads concurrently, but also because the number of data collisions decreases.

An additional advantage of this programmable solution for LDPC decoding based on multicores is observed in the BER curves for typical ASIC solutions that use 5 to 6-bit precision to represent data. A minimal data precision representation is fundamental in ASIC solutions in order to achieve acceptable die area and power consumption. Figure 8 shows the BER curves we obtained for two different codes used in the WiMAX standard, namely (576,288) and (1248,624) using either 6-bit or 8-bit precision. Our implementation, based on an 8-bit data precision multicore architecture, achieves improved BER compared with ASIC solutions.

6. CLOSURE

Projects to implement LDPC decoders, usually based on ASIC due to their extremely intensive computational nature, can entail long development times and consume considerable resources. This paper proposes a multicodeword parallel LDPC decoder using a largely disseminated low-cost many-

Table 4: Comparing with state-of-the-art ASIC LDPC decoders using the min-sum algorithm.

	Rate	Code length	Precision (bit)	Iterations	Throughput (Mbps)
[12]	5/6	2304	8	10	30.4
[25]	1/2	2304	6	20	63
[26]	1/2	2304	8	8	60
Matrix B [proposed solution]	1/2	8000	8	10	96.2

Table 5: Percentage of thread execution times representing memory accesses and computation on GPU.

Operation	% of processing time		
Matrix	A	B	C
Reading data from memory	20.2	22.3	21.9
Reading indices from memory	3.1	6.8	7.3
Writing data to memory	24.3	39.2	50.6
Remaining processing	52.4	31.7	20.2

core architecture, a GPU running CUDA. This offers all the advantages associated with programmability and flexibility as opposed to hardware-dedicated non-scalable rigid solutions. As is well known, one of the most challenging problems associated with manycore architectures is that a performance bottleneck is often imposed by memory accesses. We also propose a solution to this problem (scalable to future devices) that allows parallel memory accesses to be performed simultaneously by multiple threads. It is based on the efficient use of coalescence, which reduces data collisions and, consequently, memory access times. To achieve this solution we propose a runtime mapping technique for automatic data realignment in order to compensate for data misalignment at compile time due to the irregular memory access pattern nature of LDPC codes. We also developed a model to predict processing times on the GPU. The experimental results report throughputs superior to 100 Mbps, which stand up well against many ASIC solutions. Using the platform under test as a reference, more recent GPUs (also supporting CUDA) have twice the number of cores available and they should produce even better throughputs. This will allow the future introduction of irregular LDPC programmable decoders, thereby guaranteeing throughputs high enough to support most applications. Furthermore, we show that a more convenient 8-bit data precision representation can be used at no extra cost to provide a much better BER, which is a fundamental metric in LDPC codes. Increasing data precision at the expense of throughput is still possible with this programmable solution, which could produce an even better BER.

7. ACKNOWLEDGMENTS

This work has been partially supported by the Portuguese Foundation for Science and Technology (FCT) under grant SFRH/BD/37495/2007. The authors would like to thank David Luebke at NVIDIA his kind support and useful comments during this research.

8. REFERENCES

[1] R. G. Gallager. Low-density parity-check codes. *IRE Transactions on Information Theory*, 8(1):21–28, January 1962.

[2] R. Tanner. A recursive approach to low complexity codes. *IEEE Transactions on Information Theory*, IT-27(5):533–547, September 1981.

[3] C. Berrou, A. Glavieux, and P. Thitimajshima. Near Shannon limit error-correcting coding and decoding: Turbo-codes (1) In *IEEE International Conference on Communications (ICC'93)*, pages 1064–1070, May 1993.

[4] D. J. C. Mackay and R. M. Neal. Near Shannon limit performance of low density parity check codes. *IEE Electronics Letters*, 32(18):1645–1646, August 1996.

[5] Digital video broadcasting (DVB); second generation framing structure, channel coding and modulation systems for broadcasting, interactive services, news gathering and other broad-band satellite applications. *EN 302 307 V1. 1.1, European Telecommunications Standards Institute (ETSI)*, 2005.

[6] T. Zhang and K. Parhi. Joint (3,k)-regular LDPC code and decoder/encoder design. *IEEE Transactions on Signal Processing*, 52(4):1065–1079, April 2004.

[7] F. Kienle, T. Brack, and N. Wehn. A Synthesizable IP Core for WIMAX 802.16E LDPC Code Decoding. In *IEEE 17th International Symposium on Personal, Indoor and Mobile Radio Communications*, pages 1–5, September, 2006.

[8] J. Dielissen, A. Hekstra, and V. Berg. Low cost LDPC decoder for DVB-S2. In *Design, Automation and Test in Europe: Designers' forum (DATE'06)*, pages 1–6, Munich, Germany, March 2006.

[9] Li Ping and W.K. Leung. Decoding Low Density Parity Check Codes with Finite Quantization Bits. *IEEE Communications Letters*, 4(2):62–64, February 2000.

[10] A. J. Blanksby and C. J. Howland. A 690-mW 1-Gb/s 1024-b, rate-1/2 low-density parity-check code decoder. *IEEE Journal of Solid-State Circuits*, 37(3):404–412, March 2002.

[11] F. Quaglio, F. Vacca, C. Castellano, A. Tarable, and G. Masera. Interconnection framework for high-throughput, flexible LDPC decoders. In *Proceedings of Design, Automation and Test in Europe, 2006 (DATE '06)*, pages 1–6, Munich, Germany, March 2006.

[12] Sangwon Seo, Trevor Mudge, Yuming Zhu, and Chaitali Chakrabarti. Design and Analysis of LDPC Decoders for Software Defined Radio. In *Proceedings*

of IEEE Workshop on Signal Processing Systems, pages 210–215, October 2007.

[13] A. Ghuloum, E. Sprangle, J. Fang, G. Wu, and X. Zhou. Ct: A Flexible Parallel Programming Model for Tera-scale Architectures. *Intel*, pages 1–21, 2007.

[14] Wilson W. L. Fung, Ivan Sham, George Yuan, and Tor M. Aamodt. Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'07)*, December 2007.

[15] M. McCool. Scalable Programming Models for Massively Multicore Processors. *Proceedings of the IEEE*, 96(5):816–831, May 2008.

[16] G. Falcão, L. Sousa, and V. Silva. Massive Parallel LDPC Decoding on GPU. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming (PPoPP'08)*, pages 83–90, Salt Lake City, Utah, USA, February 2008.

[17] G. Falcão, V. Silva, L. Sousa and J. Marinho. High coded data rate and multicodeword WiMAX LDPC decoding on Cell/BE. *IET Electronics Letters*, 44(24):1415–1417, November 2008.

[18] S. Chung, G. Forney, T. Richardson and R. Urbanke. On the Design of Low-Density Parity-Check Codes within 0.0045 dB of the Shannon Limit. *IEEE Communications Letters*, 5(2):58–60, 2001.

[19] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: stream computing on graphics hardware. *ACM Trans. Graph.*, 23(3):777–786, May 2004.

[20] N. Goodnight, R. Wang, and G. Humphreys. Computation on Programmable Graphics Hardware. *IEEE Computer Graphics and Applications*, 25(5):12–15, September/October 2005.

[21] Ka-Ling Fok, Tien-Tsin Wong, and Man-Leung Wong. Evolutionary Computing on Consumer Graphics Hardware. *IEEE Intelligent Systems*, 22(2):69–78, March/April 2007.

[22] J. Kruger and R. Westermann. Linear algebra operators for GPU implementation of numerical algorithms. *ACM Transactions on Graphics*, 22(3):908–916, 2003.

[23] S. Lin and D. J. Costello. *Error Control Coding*. Prentice Hall, second edition, 2004.

[24] F. Guilloud, E. Boutillon, and J.-L. Danger. λ-min decoding algorithm of regular and irregular LDPC codes. In *Proc. 3rd Int. Symp. Turbo Codes Relat. Topics*, pages 1–4, September 2003.

[25] C.-H. Liu, S.-W. Yen, C.-L. Chen, H.-C. Chang, C.-Y. Lee, Y.-S. Hsu, and S.-J. Jou. An LDPC Decoder Chip Based on Self-Routing Network for IEEE 802.16e Applications. *IEEE Journal of Solid-State Circuits*, 43(3):684–694, 2008.

[26] Xin-Yu Shih, Cheng-Zhou Zhan, Cheng-Hung Lin, and An-Yeu Wu. An 8.29 mm^2 52 mW Multi-Mode LDPC Decoder Design for Mobile WiMAX System in 0.13 μm CMOS Process. *IEEE Journal of Solid-State Circuits*, 43(3):672–683, 2008.