Massive Parallel LDPC Decoding on GPU

Gabriel Falcão
Instituto de Telecomunicações
Dep. of Electrical and Computer Eng.
University of Coimbra
Portugal
gff@co.it.pt

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

Vitor Silva
Instituto de Telecomunicações
Dep. of Electrical and Computer Eng.
University of Coimbra
Portugal
vitor@co.it.pt

Abstract

Low-Density Parity-Check (LDPC) codes are powerful error correcting codes (ECC). They have recently been adopted by several data communication standards such as DVB-S2 and WiMax. LDPCs are represented by bipartite graphs, also called Tanner graphs, and their decoding demands very intensive computation. For that reason, VLSI dedicated architectures have been investigated and developed over the last few years. This paper proposes a new approach for LDPC decoding on graphics processing units (GPUs). Efficient data structures and an new algorithm are proposed to represent the Tanner graph and to perform LDPC decoding according to the stream-based computing model. GPUs were programmed to efficiently implement the proposed algorithms by applying data-parallel intensive computing. Experimental results show that GPUs perform LDPC decoding nearly three orders of magnitude faster than modern CPUs. Moreover, they lead to the conclusion that GPUs with their tremendous processing power can be considered as a consistent alternative to state-of-the-art hardware LDPC decoders.

Categories and Subject Descriptors I.3.1 [*Computer Graphics*]: Hardware Architecture—Parallel processing

General Terms Algorithms, Design, Performance

Keywords Parallel processing, Graphics Processing Unit, Low-Density Parity-Check codes, LDPC, Computer Unified Device Architecture, CUDA

1. Introduction

Low-Density Parity-Check (LDPC) codes are powerful error correcting codes (ECC) proposed by Robert Gallager in 1962 [1]. Rediscovered by Mackay and Neal in 1996 [2], they have inspired the scientific community to develop efficient LDPC coding solutions [3]–[7] for data communication systems. They have recently been adopted by the DVB-S2 standard [8], Wimax Mobile (802.16e), Wifi (802.11n), 10Gbit Ethernet (802.3an), and other new emerging standards for communication and storage applications. LDPC codes are usually represented by bipartite graphs formed by Bit Nodes (BNs) and Check Nodes (CNs), and linked by bidirectional edges, also called *Tanner* graphs [9]. LDPC decoders, and in particular the Sum Product Algorithm (SPA), require very intensive computation. For this reason, dedicated VLSI hardware solutions based on fixed-point arithmetic have been proposed over the last few years. Nevertheless, a software-based, flexible, scalable and low-cost solution to this computational intensive problem would be highly desirable.

Due to demands for visualization effects in the gaming industry, over the last decade graphics processing units (GPUs) have undergone increasing performance, nowadays achieving hundreds of GFLOPS [10]. With many cores driven by a considerable memory bandwidth, recent GPUs are targeted for compute-intensive, highly parallel computation. Moreover, researchers in high performance computing fields are applying their huge computational power to general purpose applications [11]–[17]. To achieve this goal we usually need to deal with very detailed code for hardware control. However, recent programming interface tools such as Caravela [22] or Computer Unified Device Architecture (CUDA) [19] hide this complex task from the programmer. The Caravela interface is a quite versatile and generalist tool that suits almost any GPU manufacturer and operating system. On the other hand, CUDA is a dedicated solution for NVIDIA GPUs starting on the 8000 series, that achieves higher computational performances. CUDA provides a new hardware and software architecture for managing computations on the GPU and was selected as the programming interface for this work.

By using data-parallel processing units of GPUs, a software LDPC decoder can be expected to achieve a performance some orders of magnitude higher than a modern CPU. Moreover, since recent GPUs are able to perform floating-point arithmetic operations, better accuracy and a lower Bit Error Rate (BER) can be achieved regarding to VLSI LDPC decoders [20]. This paper proposes a SPA algorithm oriented to LDPC decoding according to the stream-based computing model. A compact data representation and a new algorithm for LDPC decoding on GPUs are the main contributions of this paper.

This paper is organized in six sections. Section 2 describes the SPA in detail. Section 3 presents the main general characteristics of GPUs and the CUDA. Section 4 describes in detail the compact data structures and the algorithm specifically developed to suit a data-parallel stream-based LDPC decoder approach on GPUs. Section 5 reports experiments comparing speedups against a recent CPU. Finally, the last section concludes the paper.

2. SPA for LDPC Decoding

LDPCs are linear (n, k) block codes defined by sparse binary parity check $\mathbf{H}$ matrices of dimension $(n - k) \times n$. Usually they are represented by bipartite graphs also called *Tanner* graphs [9],

PPoPP'08, February 20–23, 2008, Salt Lake City, Utah, USA.

formed by Bit Nodes (BNs) and Check Nodes (CNs) linked by bidirectional edges. The SPA is a very efficient LDPC decoding algorithm [21]. It is based on the belief propagation between nodes connected as indicated by the *Tanner* graph edges (see example in figure 1). SPA, as proposed by Gallager, operates in the probabilistic domain [1] [2] [21].

Algorithm 1 SPA

1: {Initialization}
$p_n = p(y_i = 1)$; $q^{(0)}_{nm}(0) = 1 - p_n$; $q^{(0)}_{nm}(1) = p_n$;
2: **while** ($\hat{\mathbf{c}}\,\mathbf{H}^T \neq \mathbf{0} \wedge$ i < I) {c-decod. word;I-Max no. iterations.} **do**
3: {For each node pair (BN_n, CN_m), corresponding to $\mathbf{H_{mn}} = \mathbf{1}$ in the parity check matrix $\mathbf{H}$ of the code **do**:}
4: {Compute the message sent from CN_m to BN_n, that indicates the probability of BN_n being 0 or 1:}

(Kernel 1 - Horizontal Processing)

5: {Compute the message sent from BN_n to CN_m:}

(Kernel 2 - Vertical Processing)

6: {Compute the *a posteriori* pseudo-probabilities}
7: {Perform hard decoding}
8: **end while**

Given an (n, k) LDPC code, we assume BPSK modulation which maps a codeword $\mathbf{c} = (c_1, c_2, \cdots, c_n)$ onto the sequence $\mathbf{x} = (x_1, x_2, \cdots, x_n)$, according to $x_i = (-1)^{c_i}$. Then $\mathbf{x}$ is transmitted through an additive white Gaussian noise (AWGN) channel originating the received sequence $\mathbf{y} = (y_1, y_2, \cdots, y_n)$, with $y_i = x_i + n_i$, and n_i being a random variable with zero mean and variance, $\sigma^2 = N_0/2$.

SPA [21] is mainly described by two horizontal and vertical intensive processing blocks, respectively identified as steps 4 and 5 of Algorithm 1.

(Kernel 1 - Horizontal Processing)

$$r^{(i)}_{mn}(0) = \tfrac{1}{2} + \tfrac{1}{2} \prod_{n' \in N(m) \backslash n} \left(1 - 2q^{(i-1)}_{n'm}(1)\right) \quad (1)$$

$$r^{(i)}_{mn}(1) = 1 - r^{(i)}_{mn}(0) \quad (2)$$

Equations (1) and (2) define the horizontal processing block that updates the message from each CN_m to BN_n, accessing $\mathbf{H}$ in row major order (they indicate the probability of BN_n being 0 or 1). Figure 1 exemplifies, for a particular 4×8 $\mathbf{H}$ matrix, BN_0 being updated by CN_0 and BN_4 being updated by CN_1. It also shows BN_6 and BN_7 being updated, respectively, by CN_2 and CN_3. For each iteration, $r^{(i)}_{mn}$ values are updated according to equations (1) and (2), as defined by the *Tanner* graph. For the example in figure 1, $r_{00} = f(q_{10}, q_{20})$ and $r_{26} = f(q_{02}, q_{32})$.

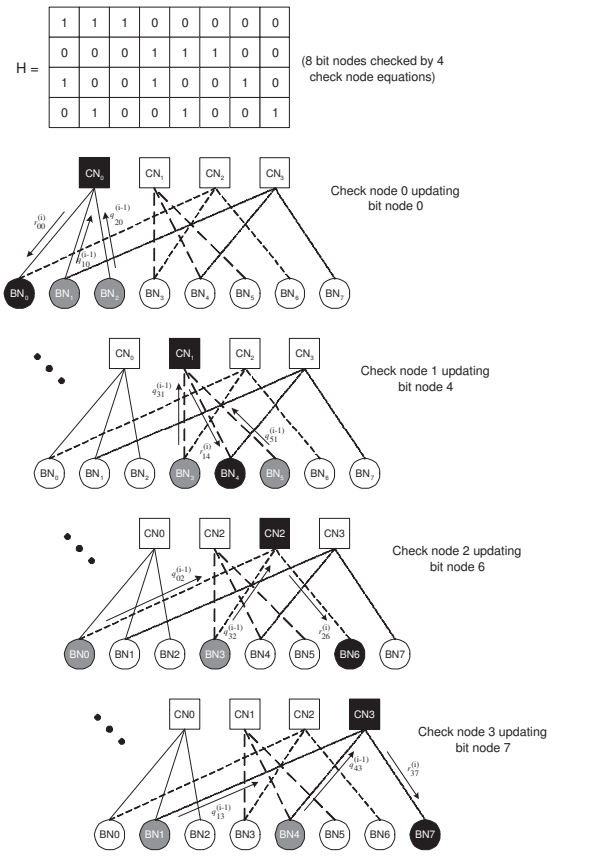

Figure 1. Tanner graph representation of a 4×8 $\mathbf{H}$ matrix and examples of some messages being exchanged between CN_m and BN_n (horizontal processing). A similar procedure applies for vertical processing.

(Kernel 2 - Vertical Processing)

$$q^{(i)}_{nm}(0) = k_{nm}(1 - p_n) \prod_{m' \in M(n) \backslash m} r^{(i)}_{m'n}(0) \quad (3)$$

$$q^{(i)}_{nm}(1) = k_{nm} p_n \prod_{m' \in M(n) \backslash m} r^{(i)}_{m'n}(1) \quad (4)$$

Similarly, the vertical processing block, described by equations (3) and (4), computes messages sent from BN_n to CN_m, assuming accesses to $\mathbf{H}$ in column major order. In this case, $q^{(i)}_{nm}$ values hold the updated information according to equations (3) and (4) for the edges connectivity indicated by the *Tanner* graph. The inspection of figure 1 also allows the conclusion that $q_{00} = f(r_{20})$ and $q_{41} = f(r_{34})$. Equations (1) to (4) can define very irregular memory access patterns.

$$\hat{c}^{(i)}_n = \begin{cases} 1 & \Leftarrow \quad Q^{(i)}_n(1) > 0.5 \\ 0 & \Leftarrow \quad Q^{(i)}_n(1) < 0.5 \end{cases} \quad (5)$$

Equation (5) indicates the final hard decoding performed at the end of an iteration. 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 a maximum number of iterations is reached.

The implemented decoder is based on a *flooding schedule* [21] version of the algorithm that controls the order in which BNs and CNs are updated: it guarantees that no BN update will interfere with any other BNs currently under updating. This principle was adopted to allow true parallel execution of the SPA for LDPC decoding on stream-based computing approaches.

3. Massively Parallel Computing on GPUs

Over the last decade, graphics processing units (GPUs) have evolved to performances surpassing the 500 GFLOPS. In fact, the pressure introduced by the gaming market, with increasingly complex demands for special visualization effects, has created processing machines capable of delivering unmatched performance.

3.1 GPU architecture

One of the reasons for such a tremendous processing power lies in the fact that GPUs dedicate more die area to processing units and less to cache and flow control. They were intentionally developed to perform highly parallel intensive computations according to the stream-based model, rather than the general purpose computing model with random access to the memory in space and time. Originally dedicating their potential to graphics rendering, GPUs nowadays also support non-graphics applications. The graphics pipeline of a traditional GPU mainly consists of vertex processors that compute perspective or resize the objects, calculating rotations and transformations of the coordinates, and pixel processors that compute colors applying texture data in the RGBA format [18]. Input data streams represent the pixel textures.

The GPU used in this work is based on an unified architecture, which represents an evolution from previous streaming architectures, where pixel and vertex programs share common stream processing resources. The GPU has 16 multiprocessors, each one consisting of 8 processors, which makes a total of 128 stream processors, with a global number of 8192 dedicated registers [19].

Moreover, the complexity involved in controlling all the GPU hardware resources has been masked by modern programming interfaces [19] [22]. Furthermore, the previous generations of GPUs experienced some limitations exploring their full potential, namely because they could gather data from any part of the memory but could only scatter data to some specific locations.

3.2 Programming model

Recently, the Computer Unified Device Architecture or CUDA [19] adopted in this work has eliminated some of the limitations referred above. The CUDA is composed of its own application programming interface (API) and its runtime environment, two mathematical libraries and a hardware driver that is currently available to program the more recent GPUs from NVidia [19]. The GPU platform used in this work is based on a unified architecture where geometry, vertex, and pixel programs share common stream processing resources and a fast shared memory mechanism also allows to efficiently exploit data parallelism. Data-parallel processing is exploited with the CUDA by executing in parallel multiple threads. If a task is executed several times, independently, over different data, it can be mapped into a kernel, downloaded to a GPU and executed in parallel on many different threads.

The execution of a kernel on a GPU is distributed according to a grid of thread blocks with adjustable dimensions. Figure 2 depicts in detail the organization of threads inside blocks and the access of each thread to each distinct type of memory. In this platform, each multiprocessor has several cores and can control more than one block of threads. Each block controls a maximum of 512 threads that execute the kernel at a very high speed, obeying to specific synchronization procedures. The number of threads per block has to be programmed according to the number of registers available on the GPU, in order to guarantee that enough physical registers are allocated to each thread, and spread across the grid at compile time.

All threads inside the same block can share data through a fast shared memory mechanism. They can also synchronize execution at specific synchronization barriers inside the kernel, where threads in a block are suspended until they all reach that point.

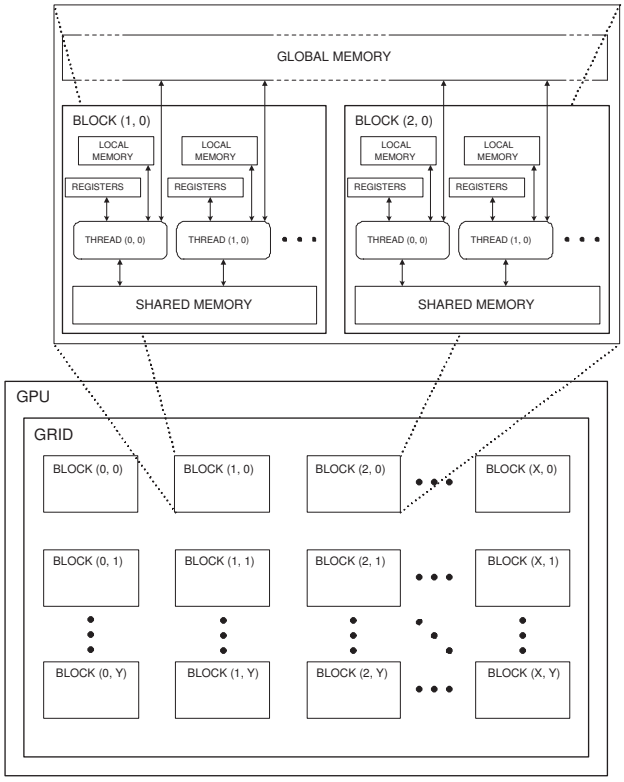

Figure 2. A grid partitioned into threaded blocks and the threads' memory scope using CUDA.

4. Parallel SPA LDPC Decoding on GPU

Operations (1), (2), (3) and (4) in Algorithm 1 presented in section 2 represent the most intensive processing in SPA. Hence, they should be processed in a high performance specific computing engine, or in a highly parallel programmable device. The latter approach is adopted in this paper, by considering a multicore system with shared memory and a programming model based on multithreads. This general model is the one supported by the GPU using CUDA (see subsection 3.2), which executes kernels in parallel on several multiprocessors, each composed by several cores that can dispatch multiple threads. This section proposes compact data structures to represent $\mathbf{H}$ and a suitable algorithm for the considered computational model.

4.1 Compact representation of the Tanner Graph

The $\mathbf{H}$ matrix of an LDPC defines the *Tanner* graph, whose edges represent the bidirectional flow of messages being exchanged between BNs and CNs. In medium to large matrices, a considerable

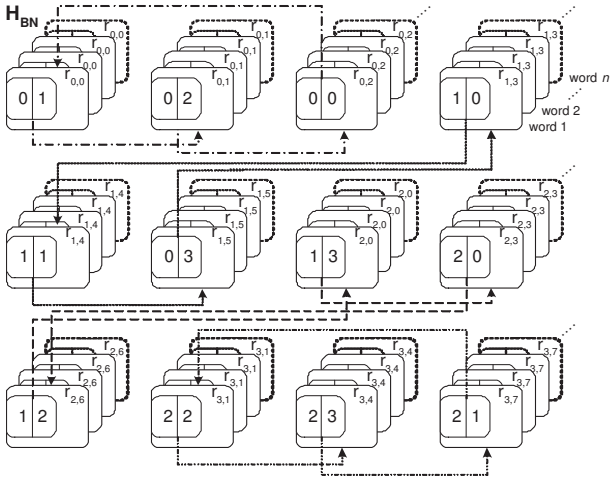

Figure 3. $\mathbf{H_{BN}}$ structure. A 2D texture representing Bit Node edges with circular addressing for the example in figure 1.

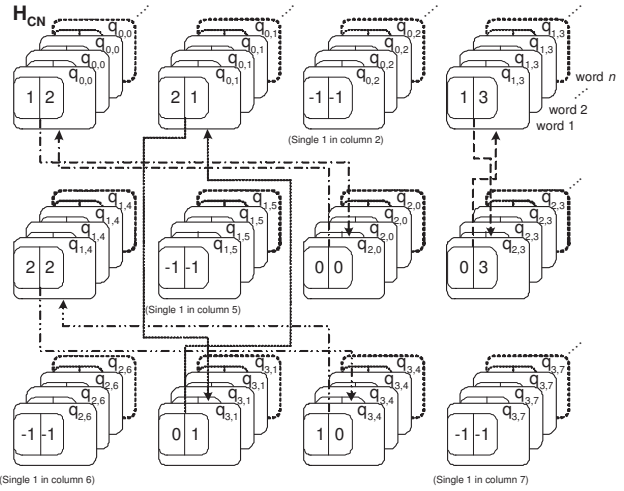

Figure 4. $\mathbf{H_{CN}}$ structure. A 2D texture representing Check Node edges with circular addressing for the example in figure 1.

amount of memory may be required to store such structure. Moreover, in a stream computing model this matrix itself has to be gathered into a data stream. Therefore, in this paper we propose to separately code the information about $\mathbf{H}$ in two independent data streams, $\mathbf{H_{BN}}$ and $\mathbf{H_{CN}}$, suitable for processing kernel 1 (horizontal processing) and kernel 2 (vertical processing) in SPA, respectively. These two data streams require significantly less memory to control the message updating procedure between BNs and CNs.

Algorithm 2 Generating compact $\mathbf{H_{BN}}$ from original $\mathbf{H}$ matrix

1: {Reading a binary $M \times N$ matrix $\mathbf{H}$}
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: $p_{next} = j : \mathbf{H}_{mj} == 1, with\ n+1 \leq j < (n+N) \bmod N$;
{Finding circularly the right neighbor on the current row}
6: $\mathbf{H_{BN}} = p_{next}$;
{Store p_{next} into the $\mathbf{H_{BN}}$ structure, using a texture of dimension $D \times D, with\ D = \left\lceil \sqrt{\sum_{m=1}^{M}\sum_{n=1}^{N} \mathbf{H}_{mn}} \right\rceil$}
7: **end if**
8: **end for**
9: **end for**

The example in figure 1 can be used to describe the transformation performed in $\mathbf{H}$ in order to produce the compact stream data structures $\mathbf{H_{BN}}$ and $\mathbf{H_{CN}}$. $\mathbf{H_{BN}}$ codes information about edge connections used in each parity check equation (horizontal processing). 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 $\mathbf{H}$ used by a single CN equation (in the same row). Algorithm 2 describes in detail this procedure 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 to the same CN are collected and stored in consecutive positions inside $\mathbf{H_{BN}}$. The access to the elements of each row of $\mathbf{H}$ is circular. Each element of the data structure, here represented by a pixel texture, records the address of the next entry pointer and the corresponding value of r_{mn}. The pixel element corresponding to the last non-null element of each row in $\mathbf{H}$ points to the first element of that row. By implementing this circular list it is possible to update all the BNs connected to a single CN, and the circular addressing allows a higher level of parallelism.

Algorithm 3 Generating compact $\mathbf{H_{CN}}$ from original $\mathbf{H}$ matrix and $\mathbf{H_{BN}}$

1: {Reading a binary $M \times N$ matrix $\mathbf{H}$}
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: $p_{tmp} = i : \mathbf{H}_{in} == 1, with\ m+1 \leq i < (m+M) \bmod M$;
{Finding circularly the neighbor below on the current column}
6: p_{next} =search($\mathbf{H_{BN}}, p_{tmp}, n$);
{Finding in $\mathbf{H_{BN}}$ the pixel with indices (p_{tmp}, n)}
7: $\mathbf{H_{CN}} = p_{next}$;
{Store p_{next} into the $\mathbf{H_{CN}}$ structure, with addresses compatible with $\mathbf{H_{BN}}$, using a texture of dimension $D \times D$, $with\ D = \left\lceil \sqrt{\sum_{m=1}^{M}\sum_{n=1}^{N} \mathbf{H}_{mn}} \right\rceil$}
8: **end if**
9: **end for**
10: **end for**

A similar principle can be applied to code the information necessary for kernel 2 (vertical processing). $\mathbf{H_{CN}}$ 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). Once again, the access between adjacent elements is circular, as described in Algorithm 3 and illustrated in figure 4 for the matrix $\mathbf{H}$ given in figure 1. In this case, a careful construction of the 2D addresses in $\mathbf{H_{CN}}$ is required because every pixel texture representing an (n, m) edge must be exactly in the same position as it is in $\mathbf{H_{BN}}$. This meticulous positioning of the pixel elements in $\mathbf{H_{CN}}$ allows the processing of 2D input textures to be alternately performed for $r_{mn}^{(i)}$ and $q_{nm}^{(i)}$, using the same input textures, while it also guarantees that data being processed for every (n, m) edge is stored in the correct position for both horizontal and vertical processing. Steps 4 and 5 in Algorithm 3 depict this property by detailing the circular addressing mechanism necessary to obtain all non-null elements in a column associated to the same BN. After that, steps 6 and 7 show the element (p_{next}) being placed

in the equivalent pixel texture (or (n, m) edge) that it occupies in $\mathbf{H_{BN}}$.

In a multiprocessor and multithreaded platform, on the limit, a different thread can be allocated to any single edge. Although in figures 3 and 4 the elements are identified by their row and column addresses, the structures can be easily vectorized by convenient 1D, 2D, or even 3D reshaping according to the target architecture they apply to. The 3D representation of the new data structures shows that the same information can be used to perform the simultaneous decoding of several codewords. Therefore, data-parallelism can be exploited by using the multiple multiprocessors available on a GPU, or in any other device with Single Instruction Multiple Data (SIMD) processing capabilities, as for example the multimedia extensions of general purpose processors.

4.2 SPA Kernels for LDPC Decoding

A stream-based SPA for implementing a parallel LDPC decoder is represented by the Synchronous Data Flow (SDF) graph in figure 5. The complete updating of BNs is performed by kernel 1 and alternates with the updating of CNs performed by kernel 2, as indicated in Algorithm 4. One complete iteration involves processing in parallel all the elements of the grid (see figures 2, 6 and 7) for the two types of processing.

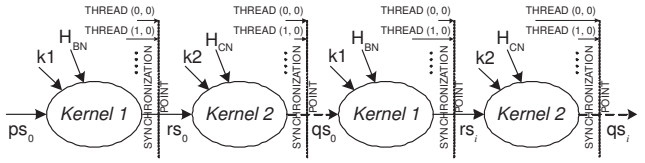

Figure 5. Synchronous Data Flow graph for a stream-based LDPC decoder: the pair kernel 1 and kernel 2 is repeated n times for an LDPC decoder performing n iterations.

Kernel 1 receives at the input the data stream ps_0 representing the original probabilities p_n, a constant k_1 indicating a dimension of the matrix and the stream $\mathbf{H_{BN}}$ that holds the necessary information to perform the SPA horizontal processing indicated in equations (1) and (2) (see Algorithm 1). It produces the output stream rs_0 which is one of the input data streams feeding kernel 2. The other inputs of this kernel are $\mathbf{H_{CN}}$, which holds information to produce the vertical processing, and constant k_2 representing the other dimension of $\mathbf{H}$. Kernel 2 produces the output stream qs_0. The compact data structures applied to kernel 1 and kernel 2 make the SPA suitable to run on a multithread platform, performing the massive decoding of a large number of elements in parallel. The threads are synchronized in the execution of each kernel in order to guarantee data reliability and the correct execution of *flooding schedule*. The pair kernel 1 and kernel 2 completes one iteration and is repeated n times for an LDPC decoder performing n iterations. After the final iteration, the last kernel conveys the codeword.

4.3 Programming the SPA on the CUDA Grid

The $\mathbf{H_{BN}}$, $\mathbf{H_{CN}}$, $r_{mn}^{(i)}$ and $q_{nm}^{(i)}$ data structures are partitioned over a grid having $X \times Y$ blocks (see figure 2). Each block contains $tx \times ty$ elements that represent threads. Threads are grouped inside warps and dispatched by one of the 128 stream processors according to the structure of the algorithm and the thread scheduler.

Figure 6 depicts the internal processing inside block (0, 0) for the example in figure 1. The update of every message represented by $r_{mn}^{(i)}$ inside the block is calculated according to the structure previously identified in $\mathbf{H_{BN}}$. In fact, both structures $\mathbf{H_{BN}}$ and $r_{mn}^{(i)}$ can be seen as perfectly overlapping one another. Every element in $r_{mn}^{(i)}$ can be updated by a different thread, which means several

Algorithm 4 SPA kernel executing on the GPU grid

```
1: {Initialize the grid block size (or number of threads per block).}
2: for all threads on a tx × ty block: do
3:     if UpdatingBNs = true then
4:         if r_mn ≠ r_NULL then
5:             Processing kernel 1
               {Compute the message sent from CN_m to BN_n.}
6:         end if
7:     else {UpdatingCNs = true}
8:         if q_nm ≠ q_NULL then
9:             Processing kernel 2
               {Compute message sent from BN_n to CN_m.}
10:        end if
11:    end if
12:    Synchronize all threads.
       {Conclude the kernel execution.}
13: end for
```

messages may in fact be updated simultaneously. Each thread can concurrently access the necessary information inside $\mathbf{H_{BN}}$ to perform the update. The circular addressing property introduced by the compact stream-based data structures $\mathbf{H_{BN}}$ and $\mathbf{H_{CN}}$ allows several threads to simultaneously access the information representing the $\mathbf{H}$ matrix. The example shown in figure 6 addresses that property. Again, the same principle applies to the update of $q_{nm}^{(i)}$ messages. As a consequence, true parallel execution takes place and the overall processing time required to decode a codeword can be significantly reduced, as it will be seen in the next section. More data parallelism can be exploited by decoding several codewords simultaneously (z axis in figures 3 and 4), but it was not considered in this work.

Furthermore, the *flooding schedule* algorithm guarantees that there are no conflicts in the order in which messages are updated. According to figure 6, the update of $r_{00}^{(i)}$ is performed by reading the corresponding values of $q_{nm}^{(i-1)}$ from addresses $(0, 1)$ and $(0, 2)$, while $r_{01}^{(i)}$ is updated by reading the $q_{nm}^{(i-1)}$ values from addresses $(0, 2)$ and $(0, 0)$, etc.

The r_{NULL} null elements were added to fill the compact data structures, ensuring that each block's row contains only information regarding one single row of $\mathbf{H}$. This increases the memory usage but prevents memory access conflicts on the other side. Experimental results report an average increase of 11% in the processing speed for the matrices under test, due to the introduction of these null elements.

The $\mathbf{H_{CN}}$ follows the same layout, since each (n, m) edge must be in the same position for both $\mathbf{H_{BN}}$ and $\mathbf{H_{CN}}$ structures. However, in this case the memory accesses in kernel 2 (representing the vertical processing) can be quite irregular. Due to the nature of LDPC codes, it is not possible to simultaneously guarantee regular memory access patterns for both $\mathbf{H_{BN}}$ and $\mathbf{H_{CN}}$ data structures. Figure 7 depicts the irregularity when updating new $q_{nm}^{(i)}$ values. In fact, it can be seen that the access to one complete column in $\mathbf{H}$ can be made, by reading several $r_{mn}^{(i-1)}$ values across different blocks of the grid. For all the cases where this happens, threads will be using less efficient global memory, which presents a higher latency. Figure 7 depicts the processing of two different columns. The oblique shaded and cross shaded pixels represent the updating of different BNs. It can be seen that both of them are accessing data spread over different blocks. Similarly to figure 6, so figure 7 shows a part of each block containing q_{NULL} null elements.

For every different application we need to estimate the best fitting number of threads per block. The minimum consists of 64, but 192 or 256 threads per block can be used. Increasing the number of threads per block often allows to achieve better

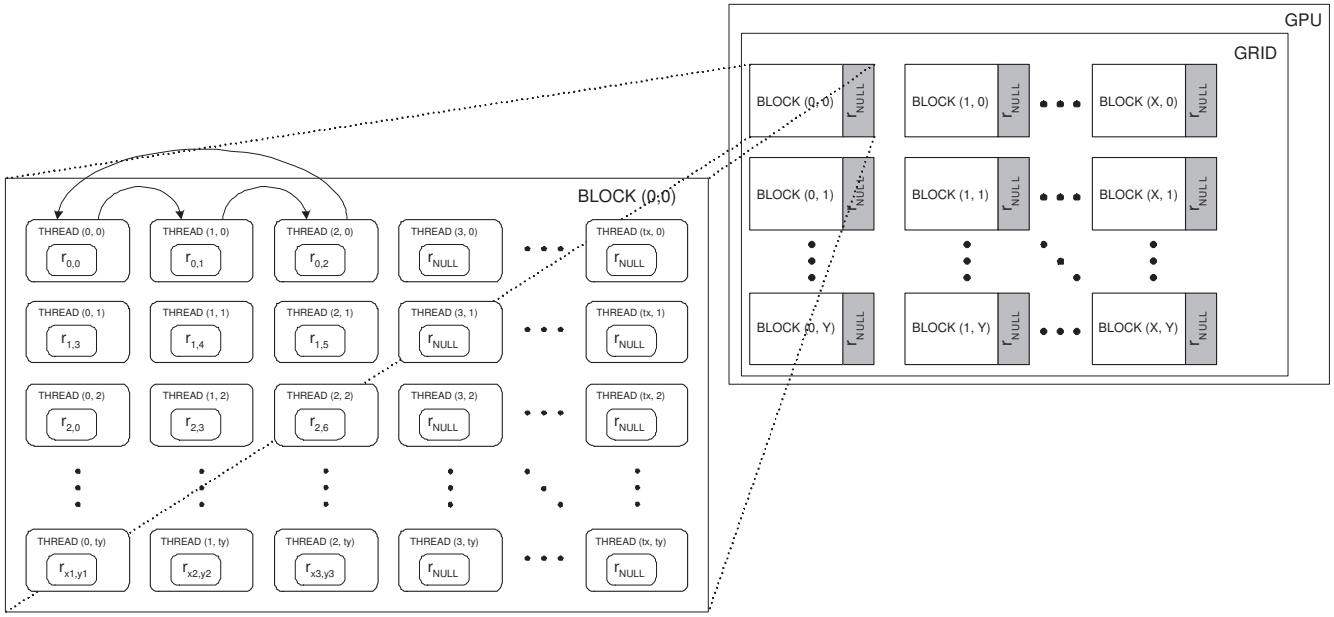

Figure 6. Detail of the $\mathbf{H_{BN}}$ structure inside a threaded block. When processing kernel 1 each pixel becomes a thread.

	CPU	GPU
Platform	Intel Core2Duo	NVidia 8800 GTX
Clock speed	2.4GHz	1.35GHz (p/ SP)
Memory	1GB	768MB
IDE	Visual Studio .NET	CUDA
Language	C	C (interoperating with OpenGL or DirectX)
OS	Windows XP SP2	

Table 1. Experimental setup.

performance as long as we guarantee that there are still enough registers per thread to compile.

The experiments carried out in this work use two distinct grid configurations: one having 64 threads per block; and another one having 256 threads per block. The results are presented in the next section.

5. Implementation and Experimental Results

To evaluate the performance of the proposed stream-based LDPC decoder, the GPU was programmed using the CUDA. A similar LDPC decoder was also implemented on a x86 CPU, using an efficient linked lists mechanism and the compact structures introduced in subsection 4.1 to represent the *Tanner* graph. The program is written in C and uses single precision floating-point arithmetic operations without Stream SIMD Extensions (SSE) instructions. The use of this kind of SIMD instructions in LDPC decoding has been presented in recent work, where the authors have reported gains between 3 and 4, as it would be expectable. Thus, such approach was considered out of scope to this work and can be addressed in [23].

The experimental setup is depicted in Table 1. Two powerful processing platforms where selected to perform LDPC decoding. The x86 CPU runs at 2.4GHz, while each one of the 128 stream processors (SP) of the GPU runs at 1.35GHz. All matrices under test were run on blocks with dimensions 8×8 each (64 threads per block) and 16×16 (256 threads per block). Matrix A has 512×1024 elements with 6 BNs per row and 3072 edges, matrix B has 2448×4896 elements with 6 BNs per row and 14688 edges, and matrix C has 2000×4000 elements with 8 BNs per row and 16000 edges. Matrix A was programmed to use 32 blocks of the grid, while matrix B uses 153 blocks and matrix C 125. The higher level of parallelism expectedly supported by the next GPU generations, advises the use of at least 100 blocks per grid in order to exploit full potential and make the solution scalable to future devices with more stream processors.

Tables 2 and 3 present the overall measured decoding times for CPU and GPU (NVidia 8800 GTX). Experimental results in these tables show that the GPU-based implementation is much faster than the CPU-based one. The difference between the two experiments in tables 2 and 3 lies in the number of threads per block used. Table 2 indicates results for 64 threads per block, while table 3 shows decoding times measured for 256 threads per block.

The obtained speedups are depicted in figures 8 and 9, for the same setup and matrices. Speedup increases as the number of edges being processed also increases, although the speedup does not only depend on the number of edges. The fastest execution times measured on the GPU suggest the use of blocks with dimension 16×16, equivalent to 256 threads per block, which is a multiple of the warp size and remains below the maximum number of allowed threads per block (512).

The GPU-based implementation shows significantly higher speedups for the LDPC decoding algorithm with intensive computation on large quantities of data, due to the fact that multicore and multithread features support a high level of parallelism. It is, however, likely that beyond a certain number of edges, the bottleneck will lie in the memory bandwidth data transfer between RAM in the host and VRAM on the GPU. A speedup above 19 is achieved for a small code with 3072 edges and executing 10 iterations, while it rises to over 45 for significantly larger codes (14688 edges and above). For small codes and a small number of iterations, better results are achieved with 64 threads per block, although the

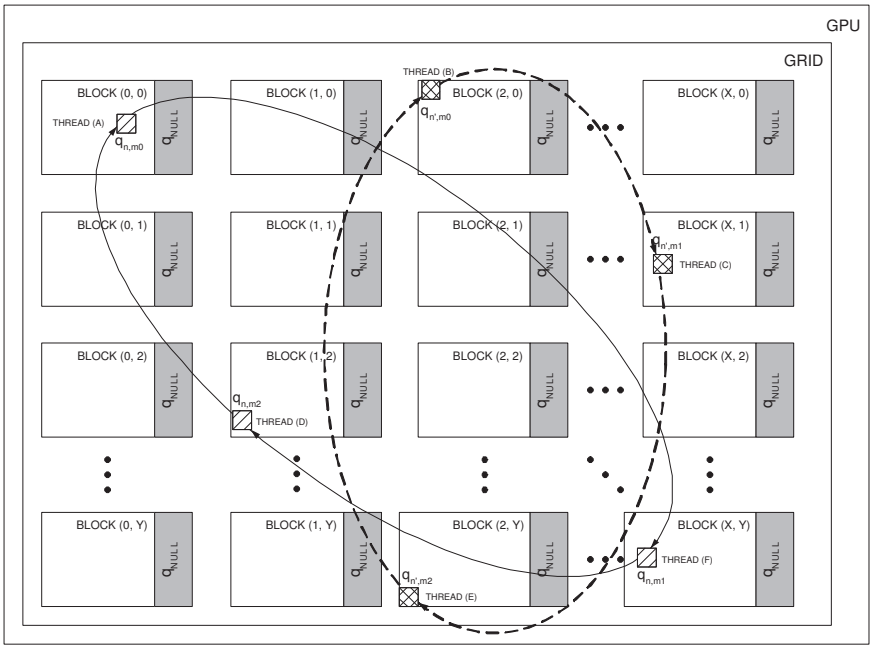

Figure 7. Processing kernel 2 using the $\mathbf{H_{CN}}$ structure. Illustration of threads accessing data memory outside their scope in different blocks.

	Matrix A - 3072 edges		Matrix B - 14688 edges		Matrix C - 16000 edges	
	CPU	**GPU**	**CPU**	**GPU**	**CPU**	**GPU**
10 iterations	27	2	131	3	161	3
25 iterations	179	3	832	6	1031	7
50 iterations	710	5	3351	11	4155	14
100 iterations	2837	12	13449	21	16680	26

Table 2. LDPC Decoding time for a CUDA programming environment using 64 threads per block (rounded to milliseconds)

	Matrix A - 3072 edges		Matrix B - 14688 edges		Matrix C - 16000 edges	
	CPU	**GPU**	**CPU**	**GPU**	**CPU**	**GPU**
10 iterations	27	2	130	3	162	3
25 iterations	180	3	831	6	1031	7
50 iterations	711	5	3348	10	4151	12
100 iterations	2828	10	13435	18	16663	24

Table 3. LDPC Decoding time for a CUDA programming environment using 256 threads per block (rounded to milliseconds)

decoding time difference is minimal. However, by comparing tables 2 and 3, it can be seen that when decoding larger codes with a higher number of iterations, the reported speedups are higher when 256 threads per block are used. Even in this situation, where more intensive processing is required, there are enough registers per thread to compile an efficient solution. Increasing the number of iterations from 25 up to 100, the GPU achieves even better performance. Speedups range from 292 up to 730 for 100 iterations and for all tested matrices.

The highest speedup is achieved for matrix B with 14688 edges, and processed on the GPU with a grid having 153 blocks and 256 threads per block. Processing under this grid provides a gain superior to 700.

It is nonetheless important to interpret such significant speedups accurately. In fact, for all matrices and for a wide range of iterations under test, the CPU always performs worse than the GPU. But for large matrices, the CPU performance degrades even further also due to data cache miss rate. The GPU achieves better performances for larger quantities of data and is, therefore, ideal for intensive LDPC decoding.

6. Conclusions

This paper proposes a stream-based LDPC decoder using a GPU instead of the conventional VLSI fixed-point dedicated implementations of the decoder. Compact data structures were designed to represent all the message exchanges between Bit ($\mathbf{BN}$) and Check

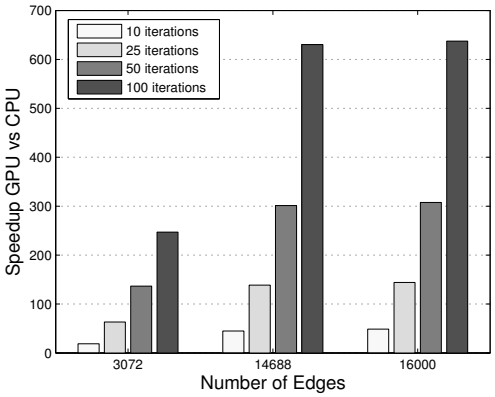

Figure 8. Speedup comparing GPU against CPU-based implementations for 3 different matrices using 64 threads per block.

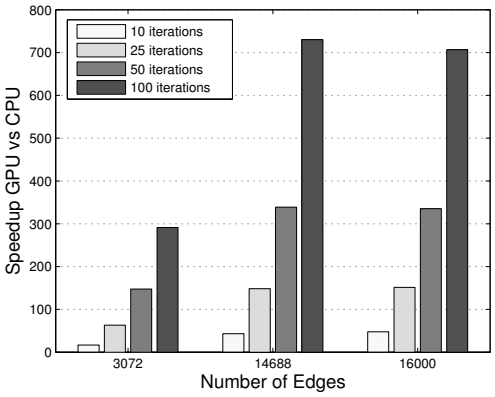

Figure 9. Speedup comparing GPU against CPU-based implementations for 3 different matrices using 256 threads per block.

(CN) connected nodes, as indicated in the *Tanner* graph. These data structures allow a significant reduction in the memory space and the processing time necessary for LDPC decoding, while at the same time they supply a circular access mechanism that allows exploitation of extra parallelism. The Sum Product Algorithm was adapted to the stream-based computing model in order to perform floating-point parallel decoding on a powerful graphics processing unit (GPU). The GPU-based LDPC decoder proved to be much faster regarding to the time required to perform LDPC decoding on a recent CPU. Speedups from 19 to 730 are achieved. The compact data structures and algorithm proposed in this paper open the perspective for future work in the area of stream-based applications dedicated to intensive error correcting codes, by exploiting the use of low-cost and flexible GPUs.

References

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

[2] 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.

[3] 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.

[4] F. Kienle, T. Brack, and N. Wehn. A Synthesizable IP Core for DVB-S2 LDPC Code Decoding. In *Design, Automation and Test in Europe (DATE'05)*, pages 100–105, Munich, Germany, 2005.

[5] 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.

[6] G. Falcão, M. Gomes, J. Gonçalves, P. Faia, and V. Silva. HDL library of processing units for an automatic LDPC decoder design. In *IEEE PhD. Research in Microelectronics and Electronics (PRIME)*, pages 349–352, Italy, June 2006.

[7] M. Gomes, G. Falcão, V. Silva, V. Ferreira, A. Sengo, and M. Falcão. Flexible Parallel Architecture for DVB-S2 LDPC Decoders. In *IEEE Globecom 2007*, November 2007.

[8] 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.

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

[10] URL - http://www.nvidia.com/page/geforce_8800.html.

[11] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Kruger, A. E. Lefohn, and T. J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, March 2007.

[12] 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.

[13] 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.

[14] Guobin Shen, Guang-Ping Gao, Shipeng Li, Heung-Yeung Shum, and Ya-Qin Zhang. Accelerate Video Decoding With Generic GPU. *IEEE Transactions on Circuits and Systems for Video Technology*, 15(5):685–693, May 2005.

[15] Jeff Bolz, Ian Farmer, Eitan Grinspun, and Peter Schroder. Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid. *ACM Transactions on Graphics*, 22(3):917–924, July 2003.

[16] Manuel Ujaldon and Joel Saltz. The GPU on irregular computing: Performance issues and contributions. In *Ninth International Conference on Computer Aided Design and Computer Graphics (CAD/CG 2005)*, December 2005.

[17] Alan Brunton, Chang Shu, and Gerhard Roth. Belief Propagation on the GPU for Stereo Vision. In *The 3rd Canadian Conference on Computer Robot Vision*, pages 76–82, June 2006.

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

[19] URL - http://developer.nvidia.com/object/cuda.html.

[20] 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.

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

[22] S. Yamagiwa and L. Sousa. Caravela: A Novel Stream-Based Distributed Computing Environment. *IEEE Computer*, 40(5):76–83, May 2007.

[23] G. Falcão, V. Silva, M. Gomes, and L. Sousa. Edge Stream Oriented LDPC Decoding. In *16th Euromicro International Conference on Parallel, Distributed and network-based Processing (PDP)*, France, February 2008.