Integrated Computer-Aided Engineering 19 (2012) 341–350
DOI 10.3233/ICA-2012-0410
IOS Press

An efficient GPU implementation of fixed-complexity sphere decoders for MIMO wireless systems

Sandra Roger*, Carla Ramiro, Alberto Gonzalez, Vicenc Almenar and Antonio M. Vidal
Institute of Telecommunications and Multimedia Applications, Universitat Politecnica de Valencia, Valencia, Spain

Abstract. The use of many-core processors such as general purpose Graphic Processing Units (GPUs) has recently become attractive for the efficient implementation of signal processing algorithms for communication systems. This is due to the cost-effectiveness of GPUs together with their potential capability of parallel processing. This paper presents an implementation of the widely employed fixed-complexity sphere decoder on GPUs, which allows to considerably decrease the computational time required for the data detection stage in multiple-input multiple-output systems. Both, the hard- and soft-output versions of the method have been implemented. Speedup results show the proposed GPU implementation boosts the runtime of the parallel execution of the methods in a high performance multi-core CPU. In addition, the throughput of the algorithm is evaluated and is shown to outperform other recent implementations and to fulfill the real-time requirements of several LTE configurations.

Keywords: GPU, MIMO detection, sphere decoding

1. Introduction

Multiple-input multiple-output (MIMO) wireless communications have been widely studied during the last decade [24]. MIMO systems can provide high spectral efficiency by means of spatially multiplexing as many data streams as transmitting antennas are used in the system. Furthermore, MIMO techniques can be used to enhance the performance of Orthogonal Frequency Division Multiplexing (OFDM) systems by exploiting the spatial domain, since MIMO-OFDM allows to transmit different streams over different subcarriers and spatial beams [17]. This increase in the spectral efficiency makes MIMO systems promising to fulfill the high-throughput requirements of current wireless standards such as WiMAX and LTE [2].

On the other hand, the use of MIMO communication systems complicates the receiver stage, which has the task of processing the received mixture of signals affected by the channel in order to recover the transmitted data with the accuracy required by the considered application. If nearly optimal detection is desired, this stage becomes often the most computationally expensive within a MIMO system and, thus, the search for high-throughput receiver implementations is imperative. Furthermore, scalability in the number of subcarriers per MIMO-symbol and in the system size are key factors in LTE and 4G wireless standards [2].

The use of many-core processors such as general purpose Graphic Processing Units (GPUs) has recently become attractive for the efficient implementation of signal processing algorithms with high computation requirements, such as the massive convolution implementation in [8], the GPU-based 3D human interface addressed in [12], the visualization system in [26] and many other image-related applications [11,20,22]. Signal processing for wireless communication systems is also a field which requires high computation capabilities. Additionally to the traditional use of digital sig-

*Corresponding author: Sandra Roger, Institute of Telecommunications and Multimedia Applications, Universitat Politecnica de Valencia, Camino de Vera s/n, 8G Building, Access D, Valencia, 46022, Spain. Tel.: +34 963877304; Fax: +34 963877309; E-mail: sanrova@iteam.upv.es.

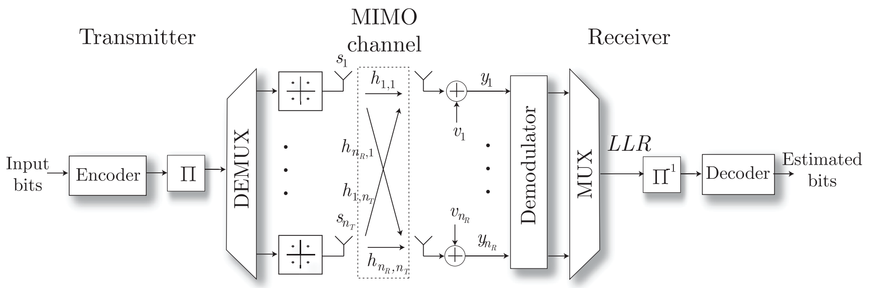

Fig. 1. Block diagram of a MIMO-BICM system.

nal processors (DSP) and field programmable gate arrays (FPGA), GPUs are becoming very useful to develop software-defined-radio platforms [3,4,19], high-throughput schemes such as the trellis-based hard- and soft-output MIMO detectors proposed in [23,28,29] or the fast decoding schemes for LDPC codes presented in [13,21]. This is due to the cost-effectiveness of GPUs together with their huge capability of parallel processing. Nevertheless, GPU devices involve a usually higher power consumption [10], which may be a shortcoming in practice.

In this paper, we present an implementation on GPUs of a widely employed fixed-complexity sphere decoder proposed in [7], which allows to ensure optimal detection performance in MIMO systems with high throughput. Both, the hard- and soft-output [5] versions of the method have been implemented and compared to the implementation of the same algorithms on a high-performance multi-core CPU. Focusing on real-time applications, the throughput of the algorithm is also evaluated and compared to that of some other recent implementations in the literature. In addition, the runtime is compared with the requirements of current wireless standards and the transmission configurations supported by the proposed GPU implementation are discussed.

2. System model

Let us consider a MIMO system with n_T transmit antennas, n_R receive antennas ($n_R \geqslant n_T$) and a certain signal-to-noise ratio (SNR). The input data stream is split equally into the n_T transmit antennas and sent simultaneously through the channel, thus overlapping in time and frequency. The baseband equivalent model for this system is given by

$$\mathbf{y} = \mathbf{H}\mathbf{s} + \mathbf{v}, \tag{1}$$

where $\mathbf{s}$ represents the transmitted signal vector composed of the elements resulting of mapping sets of information bits to symbols belonging to a certain constellation $\mathbf{\Omega}$ of size M, such as Quadrature Amplitude Modulation (QAM). Vector $\mathbf{y}$ in Eq. (1) denotes the received symbol vector, and $\mathbf{v}$ is a complex additive white Gaussian noise vector. The Rayleigh fading channel matrix $\mathbf{H}$ is assumed to be known at the receiver and is formed by $n_R \times n_T$ complex-valued elements, h_{ij}, which represent the fading gain from the j-th transmit antenna to the i-th receive antenna. It is important to note that, in this work, a block fading channel is considered, which means that it is possible to transmit a long data frame without estimating a new channel matrix. Since MIMO-OFDM transmission is considered, the system model holds for the transmission through a single subcarrier out of the N_c subcarriers used in the system.

The above system description can be easily extended to describe a MIMO-BICM [9] system with the same number of antennas. In this system (shown in Fig. 1), the sequence of information bits is encoded using an error-correcting code, passed through a bitwise interleaver Π previously to being demultiplexed and mapped into complex symbols.

3. Hard and soft-output FSD detection

Given the received signal $\mathbf{y}$, the detection problem in MIMO systems consists in determining the transmitted vector $\hat{\mathbf{s}}$ with the highest a posteriori probability [24]. In practice, this hard-output *maximum-likelihood* (ML) detection problem is carried out by solving the following least squares problem

$$\hat{\mathbf{s}} = \arg \min_{\mathbf{s} \in \Omega^{n_T}} \|\mathbf{y} - \mathbf{H}\mathbf{s}\|^2. \tag{2}$$

Equation (2) can be solved by an exhaustive search over the total n_T-dimensional lattice points $\mathbf{s}$. This implementation is cumbersome for practical systems; however, its complexity can be substantially reduced by means of tree search detection methods such as the fixed-complexity Sphere Decoder (FSD) [7] or many other sphere decoding methods [14].

A QR factorization of the channel matrix ($\mathbf{H} = \mathbf{QR}$) allows to transform Eq. (2) into an equivalent expression that can be solved using a tree structure [14]. Calling $\mathbf{y}' = \mathbf{Q}^H\mathbf{y}$, the problem Eq. (2) becomes

$$\hat{\mathbf{s}} = \arg\min_{\mathbf{s}\in\Omega^{n_T}} \left| \sum_{i=1}^{n_T} |y_i' - \sum_{j=i}^{n_T} R_{ij}s_j|^2 \right|. \qquad (3)$$

In order to solve Eq. (3) via a tree search, the following recursion is performed for $i = n_T, \ldots, 1$:

$$d_i(S^{(i)}) = d_{i+1}(S^{(i+1)}) + |e_i(S^{(i)})|^2 \qquad (4)$$

$$e_i(S^{(i)}) = y_i - \sum_{j=i}^{n_T} R_{i,j}s_j, \qquad (5)$$

where i denotes each tree level, $S^{(i)} = [s_i, s_{i+1}, \ldots, s_{n_T}]$, $d_i(S^{(i)})$ is the accumulated Partial Euclidean Distance (PED) up to level i, with $d_{n_T+1}(S^{(n_T+1)}) = 0$. Note that $|e_i(S^{(i)})|^2$ is the distance between levels i and $i+1$ in the decoding tree. Each partial solution is represented as a node in the tree.

The FSD performs a predetermined tree-search composed of two different stages: a full expansion of the tree (FE) in the first (highest) T levels and a single-path expansion (SE) in the remaining tree-levels $n_T - T$ [7]. At the FE stage, for each survivor path, all the possible values of the constellation are assigned to the symbol at the current level. The SE stage starts from each retained path and proceeds down in the tree calculating the solution of the remaining successive-interference-cancellation (SIC) problem [16] as:

$$\begin{aligned} \hat{s}_i &= \mathcal{Q}\left\{ \frac{\left(y_i' - \sum_{l=i+1}^{n_T} R_{i,l}\hat{s}_l\right)}{R_{i,i}} \right\}, \\ i &= n_T - T, \ldots, 1. \end{aligned} \qquad (6)$$

The function $\mathcal{Q}(\cdot)$ assigns the closest constellation value.

The symbols are detected following a specific ordering also proposed by the authors in [7]. As it was shown in [16], the maximum detection diversity can be achieved with the FSD if the following value of T is chosen:

$$T \geqslant \sqrt{n_T} - 1. \qquad (7)$$

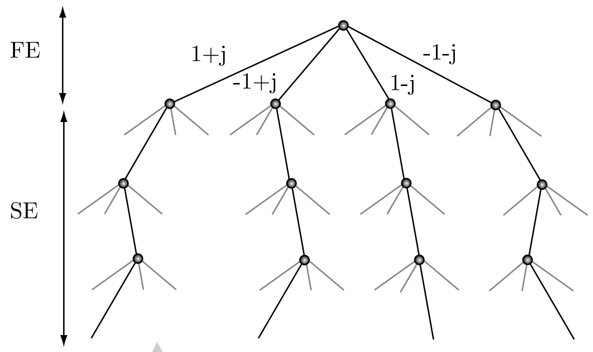

Fig. 2. Decoding tree of the hard-output FSD algorithm.

Figure 2 shows the search tree of the FSD algorithm for the case with $n_T = 4$ ($T = 1$) and QPSK symbols. It can be seen that in the first level all branches are taken into account, meanwhile in the following levels only one branch per level is explored. Regarding the achieved bit-error-ratio (BER) performance, the performance of the FSD is very close to that of an optimal ML decoder [7].

If a MIMO-BICM system [9] is considered, where channel coding is employed, the demodulation and channel decoding are not performed jointly at the receiver but in two differentiated stages (see Fig. 1). First, the demodulator provides reliability information about the transmitted coded bits, which is commonly known as soft information. The delivered soft information (usually a real-valued log-likelihood ratio (LLR) for every coded bit) is used by the channel decoder to make final decisions about the transmitted sequence bits. The LLR for each transmitted coded bit equals

$$L_{j,b} = \log \frac{f(x_{j,b} = 1|\mathbf{y}, \mathbf{H})}{f(x_{j,b} = 0|\mathbf{y}, \mathbf{H})}, \qquad (8)$$

where $x_{j,b}$ denotes the bth bit in the bit label of symbol s_j and $f(x_{j,b}|\mathbf{y}, \mathbf{H})$ is the probability mass function of $x_{j,b}$ conditioned on $\mathbf{y}$ and $\mathbf{H}$.

If the max-log approximation is assumed [15], the LLR of the bth bit in the bit label of symbol s_j equals:

$$\begin{aligned} L_{j,b} = &\frac{1}{\sigma^2}\Bigg[\min_{\mathbf{s}\in\mathcal{X}_{j,b}^{(0)}} \|\mathbf{y} - \mathbf{Rs}\|^2 \\ &- \min_{\mathbf{s}\in\mathcal{X}_{j,b}^{(1)}} \|\mathbf{y} - \mathbf{Rs}\|^2 \Bigg], \end{aligned} \qquad (9)$$

where $\mathcal{X}_{j,b}^{(c)}$ denotes the set of symbol vectors for which the bth bit in layer j equals c.

In [15], a list-based sphere decoding scheme was proposed to approximate the sets $\mathcal{X}_{j,b}^{(0)}$ and $\mathcal{X}_{j,b}^{(1)}$ by a list of candidates with a certain size, smaller than the whole set of M^{n_T}. Two parameters traded the complexity of the method versus performance: the list size and the sphere radius, both selected in a somewhat ad-hoc manner.

It is useful to realize that the Euclidean distance of the solution of the hard-output ML detection problem Eq. (2) to the received vector directly provides one of the two minima in expression Eq. (9), denoted in what follows as d^{ML}. Denote by $x_{j,b}^{\mathrm{ML}}$ the bth bit associated with s_j^{ML}. For each j and b, the second minimum in Eq. (9) can be computed as

$$\bar{d}_{j,b} = \min_{\mathbf{s}\in\mathcal{X}_{j,b}^{(\overline{x}_{j,b}^{\mathrm{ML}})}} \|\mathbf{y} - \mathbf{Hs}\|^2, \tag{10}$$

where $\overline{x}$ denotes the complement of bit x. Note that $\mathbf{s} \in \mathcal{X}_{j,b}^{(\overline{x}_{j,b}^{\mathrm{ML}})}$ represents the counter-hypothesis to the ML solution for bit b in layer j. Once Eqs (2) and (10) are calculated, the LLRs are obtained as:

$$L_{j,b} = \frac{1}{\sigma^2}(d^{ML} - d_{j,b}^{\overline{ML}})(1 - 2x_{j,b}^{ML}), \tag{11}$$

where the term at the end adjusts the sign depending on whether d^{ML} corresponds to the first or the second minimum in Eq. (9).

Note that tree-search detection methods can be employed to obtain the max-log-approximated LLRs efficiently. The straight way to do this is to carry out an initial hard-output ML detection to obtain d^{ML} and $\mathbf{x}^{ML}$ and subsequently $n_T \times \log_2 M$ new tree-searches. Each additional tree-search must force the detector to keep one of the bits in $x^{\overline{ML}}$ fixed. This strategy, which is known as Repeated Tree Search (RTS) [27], is very intuitive but performs several redundant calculations throughout the tree. A more efficient way to calculate the exact max-log-map LLRs without neither list-size constraints nor sphere radius was proposed in [25]. This scheme obtains all the necessary distances by performing a single tree-search (STS) and was shown to achieve meaningful results specially when combined with LLR clipping. However, its computational cost varies depending on the channel matrix and it is based on a purely sequential tree-search, which is clearly a drawback for parallel implementation.

Focusing on detection methods with a fixed number of visited nodes, in [5,6] the hard-output FSD in [7] was extended to provide soft information by obtaining an improved list of candidates with respect to the one in [15]. Between the two soft-output-FSD proposals, the algorithm shown in [5] achieved good demodulation results in a turbo encoded system expanding a lower number of candidates than the one in [6]. Therefore, we selected the method in [5] among the previously described algorithms for our implementation on GPU. The high parallel processing capabilities of the FSD against the high complexity of the RTS and the variable complexity of the STS motivated this selection.

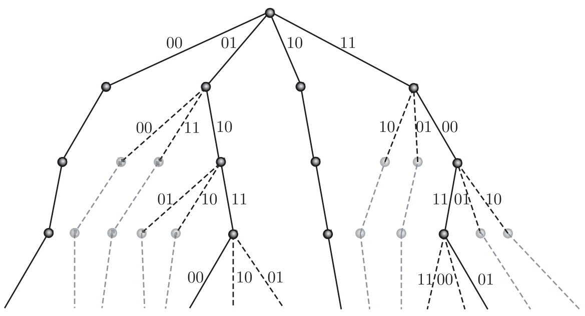

Fig. 3. Decoding tree of the SFSD algorithm.

Figure 3 shows the search-tree of the soft-output FSD (SFSD) in [5] again for the case with $n_T = 4$ and QPSK symbols. The method starts from the list of candidates that the hard-output FSD in [7] obtains (in Fig. 2) and adds new candidates to provide more information about the counter bits. Note that, since the first level of the hard-output FSD tree is already totally expanded, all the necessary values to compute the LLRs of the symbol bits in the first levels are available. Therefore, the list extension must start from the second level of such path.

To begin the list extension, the best N_{iter} paths are selected from the initial hard-output FSD list (in this example, $N_{iter} = 2$). This is based on the heuristics that the lowest-distance paths may be candidates differing from the best paths in only some bits. The symbols belonging to these N_{iter} paths are picked up from the root up to a certain level l, and, at level $l-1$, additional $\log_2 M$ branches are explored, each of them having one of the bits of the initial path symbol negated. Afterwards, these new partial paths are completed following the SIC path, as done in the hard-output FSD scheme. The same operation is repeated until the lowest level of the tree is reached. Note that N_{iter} depends on the symbol constellation. In this work, the values that achieved almost max-log-map performance for a 4×4 system in [5] were considered, being these $N_{iter} = \{2, 4, 6\}$ for QPSK, 16-QAM and 64-QAM, respectively.

4. Implementation of the hard and soft-output FSD in CUDA

4.1. GPU and CUDA

Compute Unified Device Architecture (CUDA) [1] is a software programming model that exploits the massive computation potential offered by GPUs. A GPU can have multiple stream multiprocessors (SM) with a certain number of pipelined cores each [1]. A CUDA device has a large amount of off-chip device memory (*global memory*) and a fast on-chip memory called *shared memory*. In this model, the programmer defines the kernel function which contains a set of common operations. At runtime, the kernel is called from the main central processing unit (CPU) and spawns a large number of threads blocks, which is called a grid. Each thread block contains multiple threads, usually up to 512, and all the blocks within a grid must have the same size. Each thread can select a set of data using its own unique ID and execute the kernel function on the selected set of data independently. Nevertheless, threads within a block can synchronize through a barrier and write simultaneously to shared memory to share data between them. In contrast, thread blocks are completely independent and can only share data through the global memory once the kernel ends. Also, each thread has a private local memory.

We employed for the implementations the Nvidia Tesla C2070 GPU with 14 SM (448 cores), 6 GB of global memory and 48 kB of shared memory per block. The architecture of the GPU is Fermi and hence it supports the maximum parallelism level with several kernel execution overlapping, data copy and kernel execution overlapping, simultaneous CPU to GPU and GPU to CPU data copy, etc.

4.2. Configuration parameters and performance measures

Two different performance measures were considered to evaluate the proposed implementations:

– *Speedup*, which is defined as the ratio between the computational time resulting of executing the algorithms on a high-performance multi-core CPU and the time to execute the same algorithms on the GPU. This calculation shows how advantageous the use of GPUs is to process large amounts of data compared to CPUs. The selected CPU has two Intel Xeon X5680 hexacore processors at 3.33 GHz with 96 GB of DDR3 main memory and 12 MB of cache memory.
– *Throughput*, which is defined as the number of processed information bits per second. Note that this parameter was previously used to assess other implementations on GPU such as the ones in [23, 28], thus, it allows a fair comparison among implementations. Moreover, the runtime evaluation exposes whether a given implementation guarantees the real-time requirements of a certain wireless standard.

A unidimensional grid configuration with N_B blocks is considered for the kernel. Tridimensional blocks are chosen with size depending on the number of threads per dimension, denoted by N_{th_x}, N_{th_y} and N_{th_z}. Each thread block is in charge of processing a group of subcarriers, denoted by N_{cb}. Once N_{th_x}, N_{th_y} and N_{th_z} are selected, the value of N_B is obtained as $N_B = \lceil N_c/N_{cb} \rceil$.

We considered for our implementation the highest system size contemplated by the LTE standard [2], i.e. 4×4. Besides, 4-QAM, 16-QAM and 64-QAM constellations are used. According to the LTE standard specifications, a 0.5 ms time slot is composed of 7 MIMO-OFDM symbols plus their respective cyclic prefixes [2]. Furthermore, the performances with the different N_c values reported in the LTE standard, i.e. $N_c = \{150, 300, 600, 900, 1200\}$, are investigated. In the proposed implementation all the symbols in the same time slot are detected by the same thread block.

4.3. Implementation details

The proposed GPU implementation is composed of two CUDA kernels. Assuming a previous preprocessing stage, the matrices resulting from the QR factorizations of the channel matrices of all the subcarriers are copied into the global memory before starting to execute the first kernel. Since our GPU allows asynchronous memory copy, we defined two CUDA *stream* objects [1] and dedicated one of them to the transfers from CPU to GPU and the other one to launch the kernel execution. As each stream is processed independently, this allows overlapping data transfer and kernel execution. Then, each thread block copies the $\mathbf{R}$ matrices for all the subcarriers within the block and the $\mathbf{y}'$ values for all symbols in the time slot into shared memory.

As said above, each thread block is in charge of processing a group of subcarriers. To maximize the number of simultaneous parallel calculations, the M independent branches of the hard-output FSD detection of the 7 symbols of the MIMO-OFDM time slot for the

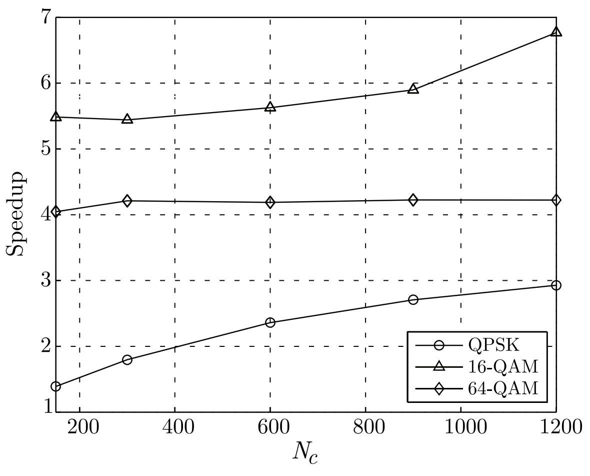

Fig. 4. Speedup for the hard-output FSD with different constellations and number of subcarriers for a 4 × 4 MIMO system.

Algorithm 1 Calculation of one of the branches of the hard-output FSD by the $\{i, j, k\}$th thread.

```
1: Get one of the values of $\Omega$ (denoted by $\Omega_k$) from constant memory,
2: Assign $s_{\{i,j,k\cdot n_T+n_T\}} = \Omega_k$ and store it in shared memory,
3: Get $\mathbf{R}_{\{i,:\}}$ and $\mathbf{y}'_{\{i,j,:\}}$ from shared memory,
4: Compute the PED $d_{\{i,j,k\cdot n_T+n_T\}}$ with Eqs (4) and (5) and store it in shared memory,
5: for $m = n_T - 1, \ldots, 1$ do
6:     Compute the $s_{\{i,j,k\cdot n_T+m\}}$ symbol using SIC Eq. (6) and store it in global memory,
7:     Update path distance $d_{\{i,j,k\cdot n_T+m\}}$ and store it in global memory,
8: end for
9: Sync barriers
```

N_{cb} subcarriers within the same group are executed by different threads simultaneously. To this end, the following numbers of threads per dimensions are set for the first kernel:

$$N_{th_x} = N_{cb}; \quad N_{th_y} = 7; \quad N_{th_z} = M. \qquad (12)$$

Here the value of N_{cb} is selected to maximize the occupancy of the device, which is assessed using the CUDA Visual Profiler. The resulting values were $N_{cb} = 4$ for QPSK and $N_{cb} = 1$ for both 16-QAM and 64-QAM.

Considering this tridimensional block structure, each thread has a unique identifier related to its position within a block, which is denoted by $\{i, j, k\}$. Algorithm 1 shows the operations carried out by thread $\{i, j, k\}$, where i ranges between 1 and N_{th_x}, j goes from 1 to N_{th_y} and k from 1 to N_{th_z}. Note that the variables involved in the process are also tridimensional, except matrix $\mathbf{R}$, which is common for the 7

Algorithm 2 Calculation of new candidates for the SFSD by the $\{i, j, k\}$ thread.

```
1: if $k == 0$ then
2:     Search $N_{iter}$ minimum distances from $d_{\{i,j,:\}}$ and store their indices in $ind_{\{i,j,:\}}$ in shared memory,
3: end if
4: for $l = 1, \ldots, n_T$ do
5:     Calculate bit position as $b = k$ rem $(\log_2 M)$
6:     Calculate selected path as $N_{it} = k$ mod $(\log_2 M)$
7:     Get index of the selected path $i' = ind_{\{i,j,N_{it}\}}$
8:     Copy symbols from level $l$ to $n_T$ from shared memory to $ss_{\{l+1:n_T\}} = s_{\{i,j,i'\cdot n_T+l+1:i'\cdot n_T+n_T\}}$ in local memory,
9:     Negate $b$th bit, keep the rest of bits as in the $N_{it}$-path symbol and assign the associated symbol to $ss_{\{l\}}$,
10:    Get partial distance $d_{\{i,j,i'\cdot n_T+l+1\}}$ from shared memory, compute the PED with Eqs (4) and (5) and store it in the local variable $dd_{\{l\}}$,
11:    if $l > 1$ then
12:        for $m = l - 1, \ldots, 1$ do
13:            Compute the $ss_{\{m\}}$ using SIC Eq. (6) and store it in global memory,
14:            Update path distance $dd_{\{m\}}$ and store it in global memory,
15:        end for
16:    end if
17: end for
```

Algorithm 3 Computation of the LLR by the $\{l, j, k\}$ thread and block i.

```
1: $d_{\min} = 1e6$,
2: for $m = 1, \ldots, P$ do
3:     if ($d_{\{i,j,m\}} < d_{\min}$) and ($k$th bit of $s_{\{i,j,m\cdot n_T+i\}}$ is equal to $x^{ML}_{l,j,k}$) then
4:         $d_{\min} = d_{\{i,j,m\}}$
5:     end if
6: end for
7: $L^{(i)}_{\{l,j,k\}} = (d^{ML} - d_{\min})(1 - 2x^{ML}_{l,j,k})/\sigma$
```

symbols within the same time slot. The notation $\{:\}$ is used to reference all the possible indices within the selected dimension.

After the hard-output part is finished, the first thread of each subcarrier calculates the N_{iter} minimum distances. The indices of the N_{iter} best paths are stored in shared memory. Then, the $N_{iter} \cdot \log_2 M \cdot (n_T - 1)$ new candidates to be obtained per subcarrier and MIMO symbol are equally distributed among all the threads of the block. Next, each thread calculates a new candidate as shown in Algorithm 2.

After this, a list with $P = M + N_{iter} \cdot \log_2 M \cdot (n_T - 1)$ paths and distances is stored in global memory. Thus, the final step finds within this list the min-

Table 1
Throughput and runtime of the FSD proposed implementation in the Nvidia Tesla C2070 GPU for a 4×4 system for different configurations compared to trellis-based detector results in [28] and the FSD results in [18].

	QPSK	16-QAM	64-QAM
FSD ($N_c = 150$)	182.61 Mbps/0.046 ms	311.11 Mbps/0.054 ms	115.07 Mbps/0.219 ms
FSD ($N_c = 300$)	311.11 Mbps/0.054 ms	436.36 Mbps/0.077 ms	123.23 Mbps/0.409 ms
FSD ($N_c = 600$)	430.77 Mbps/0.078 ms	533.33 Mbps/0.126 ms	130.57 Mbps/0.772 ms
FSD ($N_c = 900$)	494.12 Mbps/0.102 ms	579.31 Mbps/0.174 ms	131.25 Mbps/1.15 ms
FSD ($N_c = 1200$)	537.60 Mbps/0.125 ms	610.91 Mbps/0.220 ms	132.81 Mbps/1.52 ms
FSD [18] ($N_c = 300$)	284.75 Mbps/0.059 ms	112 Mbps/0.300 ms	15.13 Mbps/3.33 ms
FSD ($N_c = 300$)*	73.37 Mbps/0.229 ms	102.92 Mbps/0.326 ms	29.06 Mbps/1.73 ms
Trellis ($N_c = 300$)**	46.16 Mbps/0.350 ms	74.30 Mbps/0.427 ms	14.50 Mbps/3.31 ms

*Results weighted by α_h. **Results using Nvidia 9600 GT GPU.

imum distances of paths having the counter bits and computes the $\log_2 M \cdot n_T$ LLRs.

These operations are executed in a second kernel. The occupancy of the device is maximized if only one subcarrier per block is considered. Thus, the new values for the numbers of threads per dimensions are set as:

$$N_{th_x} = n_T; \quad N_{th_y} = 7; \quad N_{th_z} = \log_2 M. \quad (13)$$

The threads within the ith block are now accessed using the variables $\{l, j, k\}$.

4.4. Hard-output FSD: Results

The parallel implementation of the hard-output FSD on GPU was compared to the implementation in the above-mentioned multi-core CPU. Figure 4 shows the speedup for a 4×4 system and the three considered constellations as a function of the number of subcarriers. It can be observed that the higher the number of subcarriers, the higher the achieved speedup. Therefore, GPUs take advantage over multi-core CPU as the problem size does, making the use of GPUs very promising for LTE configurations dealing with a large amount of subcarriers. Regarding the constellation size, the speedup achieved for the 16-QAM case is higher than the speedup achieved for the 64-QAM case.

As said in [1], the number of registers and shared memory used by a kernel can have a significant impact on the number of resident warps. Moreover, the occupancy of the device for a certain configuration gives an idea of how well an algorithm exploits its parallel processing capabilities. However, a higher occupancy does not mean higher performance, since it depends on other factors such as global memory accesses, divergent branches, etc. Nevertheless, we evaluated the occupancy of the device for each configuration using the CUDA Visual Profiler, resulting in 67% for the QPSK and 16-QAM cases and 58% for 64-QAM. The lower occupancy for the 64-QAM may justify the lower speedup achieved.

Next, the throughput of the hard-output FSD implementation for a 4×4 was evaluated and compared to the one of the trellis-based detector proposed in [28], which considered the Nvidia 9600 GT GPU with 64 cores at 1.9 GHz. In addition, the proposed approach was also compared to the FSD GPU implementation reported in [18], by executing the CUDA code delivered by the authors in our GPU for the $N_c = 300$ example case.

Focusing on the comparison with the trellis-based detector, we defined a factor α which gathers the differences between two GPU (a and b) both in number of cores (N_{core}) and their clock frequencies (f), as follows:

$$\alpha = \frac{N_{core}^{(a)} \cdot f_a}{N_{core}^{(b)} \cdot f_b}. \quad (14)$$

Note that α compares the total processing capability of a certain GPU with another. Since our implementation uses 448 cores against the 64 cores used in [28] (7 times more) but the cores of our device (Nvidia Tesla C2070 GPU) are much slower (1.15 GHz against 1.9 GHz), this gives a value of $\alpha_h \simeq 4.24$. Thus, we included among the results in Table 1 some results weighted by α_h to allow a fairer comparison with the trellis-based implementation.

It can be seen that our proposed approach outperforms the scheme in [28] for all the constellation values. Moreover, recall that while the hard-output FSD algorithm achieves the same performance as the optimal ML detector, the trellis-based architecture is not shown to have this behavior and may not reach the ML solution in all cases. Thus, our proposed GPU implementation is more efficient than the trellis-based ap-

Table 2
Throughput and runtime of the SFSD proposed implementation in the Nvidia Tesla C2070 GPU for a 4×4 system for different configurations compared to trellis-based detector results in [29] and the STS results in [18].

	QPSK	16-QAM	64-QAM
SFSD ($N_c = 150$)	73.04 Mbps/0.115 ms	45.04 Mbps/0.373 ms	9.82 Mbps/2.57 ms
SFSD ($N_c = 300$)	82.35 Mbps/0.204 ms	49.41 Mbps/0.680 ms	9.74 Mbps/5.17 ms
SFSD ($N_c = 600$)	105.99 Mbps/0.317 ms	52.09 Mbps/1.29 ms	9.68 Mbps/10.40 ms
SFSD ($N_c = 900$)	118.31 Mbps/0.426 ms	51.88 Mbps/1.94 ms	9.59 Mbps/15.77 ms
SFSD ($N_c = 1200$)	125.61 Mbps/0.535 ms	52.09 Mbps/2.58 ms	9.60 Mbps/21 ms
STS [18] ($N_c = 300$)	10.10 Mbps/1.67 ms	0.86 Mbps/39.04 ms	0.46 Mbps/1.10 s
SFSD (8×8192 sym)*	87.36 Mbps/1.56 ms	32.86 Mbps/8.28 ms	6.01 Mbps/67.88 ms
Trellis (8×8192 sym)**	74.47 Mbps/1.76 ms	27.94 Mbps/8.31 ms	3.15 Mbps/124.62 ms

*Results weighted by α_s. **Results using Nvidia Tesla C1060 GPU.

proach both in terms of throughput and BER performance.

The comparison between our proposed approach and the implementation in [18] is, in principle, fair, as in both cases the same FSD MIMO detector is considered. Note that our proposed GPU implementation considers a different channel matrix per slot, while the code in [18] maintains the same channel matrix for the whole transmission. This difference puts our proposal in a disadvantaged position, since it affects both memory storage and number of accesses. Despite this, it can be observed in Table 1 that our proposed implementation is slightly faster for the QPSK case and much faster for the 16-QAM and 64-QAM cases. These results confirm that the parallel execution of the FSD branches considered in our implementation is more efficient that the sequential computation of the FSD branches performed by the approach in [18].

Regarding the LTE real-time requirements, for the QPSK and 16-QAM cases a whole slot can be processed before having received the following one (i.e. in $\leqslant 0.5$ ms) for all the considered subcarrier configurations. For the 64-QAM case, the runtime is $\leqslant 0.5$ ms for the $N_c = 150$ and $N_c = 300$ cases. Therefore, the proposed GPU implementation can manage all the LTE configurations except the three having the highest values of N_c when using 64-QAM symbols. Thus, further work is needed to decrease the runtime of the latter configurations, either by further optimizing the code or by making use of more powerful GPUs.

We used the CUDA Compute Visual Profiler to assess our implementation and observed that the bottleneck of our implementation is the size of the shared memory.

4.5. *Soft-output FSD: Results*

The throughput achieved by the SFSD GPU implementation for a 4×4 was evaluated and compared to the one of the trellis-based soft MIMO detector proposed in [29], which considered the detection of 8 streams of 8192 symbols simultaneously in a Nvidia Tesla C1060 GPU with 240 cores at 1.3 GHz. Moreover, the proposed approach was also compared to the soft-output single tree-search sphere decoder (STS) GPU implementation reported in [18], by executing the CUDA code provided by the authors using our GPU for the $N_c = 300$ example case.

We first compare the proposed SFSD implementation with the trellis-based soft detector, taking into account that the comparison between our device and the one used in [29] leads to a value of $\alpha_s \simeq 1.65$. Again we included among the results in Table 2 some results weighted by α_s to allow a fairer comparison with the trellis-based implementation. The comparison in Table 2 reveals that, considering the weighted results, the proposed SFSD implementation would achieve higher throughput than the trellis-based approach for all cases. Nevertheless, the throughput differences are not very high and might be compensated in an actual setup over the same GPU.

The comparison between our proposed SFSD implementation and the STS GPU implementation in [18] is fair because both algorithms achieve max-log-map performance. The throughput results in Table 2 reveal that our proposed implementation is much faster than the one in [18] for all constellation sizes, which shows a clear advantage of the SFSD parallel structure over the STS sequential nature.

Regarding the LTE real-time requirements, for the QPSK case, the proposed SFSD GPU implementation can process a whole slot before having received the following one for all N_c values except for $N_c = 1200$ case. Nevertheless, real-time is nearly achieved for the latter case. For the 16-QAM case, however, the runtime is $\leqslant 0.5$ ms only for the $N_c = 150$ case. Therefore, the SFSD GPU implementation requires further optimizations than the hard-output FSD to meet real-time. In fact, the occupancy for the SFSD scored 50% for QPSK and 16-QAM and only 33% for 64-QAM, being these values lower than their respective ones in the FSD implementation.

5. Conclusion

In this paper, a GPU-based implementation of the fixed-complexity sphere decoder providing either hard or soft outputs has been presented. The detection stage is highly accelerated through exploiting two parallelism levels: first, the FSD branches associated to independent SIC detection problems are processed in parallel and, second, the detection step is processed simultaneously for all the subcarriers through forwarding each parallel stream to a different thread.

The efficiency of the proposed approach was first assessed by comparing its computational time to the one taken by an equivalent implementation on a high-performance multi-core CPU. Results showed that the GPU-based hard-FSD implementation performs up to 7 times faster than its CPU-equivalent for some cases. Moreover, the speedup increased slightly with the number of subcarriers, showing the interest of GPU implementation for configurations managing many data streams simultaneously.

Finally, the throughput of the method was calculated for some LTE configurations and compared to the one of previously proposed GPU MIMO detectors. For a 4×4 MIMO system, the proposed GPU implementation of the FSD achieved higher throughput than the other proposals for both the hard-output FSD and SFSD. The throughput differences among the implementations are higher in the hard-output case.

Future work is needed to decrease the runtime for those configurations not attaining real-time. The use of either more powerful GPU and/or more than one GPU might be promising solutions for this purpose. Another interesting topic for future research is to analyze the amount of energy consumed by the proposed GPU implementation.

Acknowledgment

This work was partially funded by the TEC2009-13741 project of the Spanish Ministry of Science and by the PROMETEO/2009/013 project of the Generalitat Valenciana.

References

[1] *NVIDIA CUDA C programming guide*, Online at: http://developer.download.nvidia.com/compute/cuda/4_0_rc2/toolkit/docs/CUDA_C_Programming_Guide.pdf.

[2] 3GPP TS 36.300, V8.9.0, *Evolved Universal Terrestrial Radio Access (E-UTRA); Physical Layer – General Description,* (2009).

[3] C. Ahn, J. Kim, J. Ju, J. Choi, B. Choi and S. Choi, Implementation of an SDR platform using GPU and its application to a 2×2 MIMO WiMAX system, *Journal of Analog Integrated Circuits and Signal Processing* **69** (2011), 107–117.

[4] P. Andelfinger, J. Mittag and H. Hartenstein, GPU-based architectures and their benefit for accurate and efficient wireless network simulations, in *International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems*, Washington, DC, USA, July 2011.

[5] L.G. Barbero, T. Ratnarajah and C. Cowan, A low-complexity soft-MIMO detector based on the fixed-complexity sphere decoder, in *IEEE International Conference on Acoustics, Speech and Signal Processing*, Las Vegas, Nevada (USA), 2008.

[6] L.G. Barbero and J.S. Thompson, Extending a fixed-complexity sphere decoder to obtain likelihood information for turbo-MIMO systems, *IEEE Transactions on Vehicular Technology* **57** (2008), 2804–2814.

[7] L.G. Barbero and J.S. Thompson, Fixing the complexity of the sphere decoder for MIMO detection, *IEEE Transactions on Wireless Communications* **7** (2008), 2131–2142.

[8] J.A. Belloch, A. Gonzalez, F.J. Martínez-Zaldívar and A.M. Vidal, Real-time massive convolution for audio applications on GPU, *Journal of Supercomputing* **58** (2011), 449–457.

[9] J.J. Boutros, F. Boixadera and C. Lamy, Bit-interleaved coded modulations for multiple-input multiple-output channels, in *IEEE Sixth International Symposium on Spread Spectrum Techniques and Applications*, New Jersey, USA, September 2000.

[10] S. Collange, D. Defour and A. Tisserand, Power consumption of GPUs from a software perspective, *Lecture Notes in Computer Science: ICCS* **5544** (2009), 914–923.

[11] L. D'Amore, D. Casaburi, A. Galletti, L. Marcellino and A. Murli, Integration of emerging computer technologies for an efficient image sequences analysis, *Integrated Computer-Aided Engineering* **18** (2011), 365–378.

[12] R. del Riego, J. Otero and J. Ranilla, A low-cost 3D human interface device using GPU-based optical flow algorithms, *Integrated Computer-Aided Engineering* **18** (2011), 391–400.

[13] G. Falcao, V. Silva and L. Sousa, How GPUs can outperform ASICs for fast LDPC decoding, *International Conference on Supercomputing,* Yorktown Heights, New York (USA), 2009.

[14] B. Hassibi and H. Vikalo, On Sphere Decoding algorithm, Part I, the expected complexity, *IEEE Transactions on Signal Processing* **54** (2005), 2806–2818.

[15] B.M. Hochwald and S.T. Brink, Achieving near-capacity on a multiple-antenna channel, *IEEE Transactions on Communications* **51** (2003), 389–399.

[16] J. Jalden, L.G. Barbero, B. Ottersten and J.S. Thompson, The error probability of the fixed-complexity sphere decoder, *IEEE Transactions on Signal Processing* **57** (2009), 2711–2720.

[17] M. Jiang and L. Hanzo, Multiuser MIMO-OFDM for next-generation wireless systems, *Proceedings of the IEEE* **95** (2007), 1430–1469.

[18] M. Khairy, C. Mehlführer and M. Rupp, Boosting sphere de-

coding speed through graphic processing units, in *European Wireless Conference*, Lucca, Italy, April 2010.

[19] J. Kim, S. Hyeon and S. Choi, Implementation of an SDR system using Graphics Processing Unit, *IEEE Communications Magazine* **48** (2010), 156–162.

[20] L. Lattari, A. Montenegro, A. Conci, E. Clua, V. Mota, M. Bernardes-Vieira and G. Lizarraga, Using graph cuts in GPUs for color based human skin segmentation, *Integrated Computer-Aided Engineering* **18** (2011), 41–59.

[21] F.J. Martínez-Zaldívar, A.M. Vidal, A. Gonzalez and V. Almenar, Tridimensional block multiword LDPC decoding on GPUs, *Journal of Supercomputing* **58** (2011), 314–322.

[22] J.C. Noyer, P. Lanvin and M. Benjelloun, Correlation-based particle filter for 3D object tracking, *Integrated Computer-Aided Engineering* **16** (2011), 165–177.

[23] T. Nylanden, J. Janhunen, O. Silven and M. Juntti, A GPU implementation for two MIMO-OFDM detectors, in *International Conference on Embedded Computer Systems*, Samos, Greece, July 2010.

[24] A.J. Paulraj, D.A. Gore, R.U. Nabar and H. Bölcskei, An overview of MIMO communications – a key to Gigabit wireless, *Proceedings of the IEEE* **92** (2004), 198–218.

[25] C. Studer, A. Burg and H. Bolcskei, Soft-output sphere decoding: Algorithms and VLSI implementation, *IEEE Journal on Selected Areas in Communications* **26** (2008), 290–300.

[26] G. Vigueras, J.M. Orduña, M. Lozano and Y. Chrysanthou, A distributed visualization system for crowd simulations, *Integrated Computer-Aided Engineering* **18** (2011), 349–363.

[27] R. Wang and G.B. Giannakis, Approaching MIMO channel capacity with reduced-complexity soft sphere decoding, *IEEE Transactions on Communications* **51** (2003), 389–399.

[28] M. Wu, Y. Sun, S. Gupta and J. Cavallaro, A GPU implementation of a real-time MIMO detector, in *IEEE Workshop on Signal Processing Systems*, Tampere, Finland, October 2009.

[29] M. Wu, Y. Sun, S. Gupta and J. Cavallaro, Implementation of a high throughput soft MIMO detector on GPU, *Journal of Signal Processing Systems* **64** (2011), 123–136.