UNIVERSITY OF Southampton

University of Southampton Research Repository
ePrints Soton

Copyright © and Moral Rights for this thesis are retained by the author and/or other copyright owners. A copy can be downloaded for personal non-commercial research or study, without prior permission or charge. This thesis cannot be reproduced or quoted extensively from without first obtaining permission in writing from the copyright holder/s. The content must not be changed in any way or sold commercially in any format or medium without the formal permission of the copyright holders.

When referring to this work, full bibliographic details including the author, title, awarding institution and date of the thesis must be given e.g.

AUTHOR (year of submission) "Full thesis title", University of Southampton, name of the University School or Department, PhD Thesis, pagination

http://eprints.soton.ac.uk

Implementation of a Fully-Parallel Turbo Decoder on a General-Purpose Graphics Processing Unit

An Li[1], Robert G. Maunder[1], Bashir M. Al-Hashimi[2], and Lajos Hanzo[1]

[1]*Southampton Wireless Group, Department of Electronics and Computer Science, University of Southampton, Southampton SO17 1BJ, U.K.*
[2]*Faculty of Physical Sciences and Engineering, University of Southampton, Southampton SO17 1BJ, U.K.*

***Abstract*—Turbo codes comprising a parallel concatenation of upper and lower convolutional codes are widely employed in state-of-the-art wireless communication standards, since they facilitate *transmission* throughputs that closely approach the channel capacity. However, this necessitates high *processing* throughputs in order for the turbo code to support real-time communications. In state-of-the-art turbo code implementations, the processing throughput is typically limited by the data dependencies that occur within the forward and backward recursions of the Log-BCJR algorithm, which is employed during turbo decoding. In contrast to the highly-serial Log-BCJR turbo decoder, we have recently proposed a novel Fully Parallel Turbo Decoder (FPTD) algorithm, which can eliminate the data dependencies and perform fully parallel processing. In this paper, we propose an optimized FPTD algorithm, which reformulates the operation of the FPTD algorithm so that the upper and lower decoders have identical operation, in order to support Single Instruction Multiple Data (SIMD) operation. This allows us to develop a novel General Purpose Graphics Processing Unit (GPGPU) implementation of the FPTD, which has application in Software-Defined Radios (SDRs) and virtualized Cloud-Radio Access Networks (C-RANs). As a benefit of its higher degree of parallelism, we show that our FPTD improves the higher processing throughput of the Log-BCJR turbo decoder by between 2.3 and 9.2 times, when employing a high-specification GPGPU. However, this is achieved at the cost of a moderate increase of the overall complexity by between 1.7 and 3.3 times.**

***Index Terms*—fully-parallel turbo decoder, parallel processing, GPGPU computing, software defined radio, cloud radio access network**

I. INTRODUCTION

Channel coding has become an essential component in wireless communications, since it is capable of correcting the transmission errors that occur when communicating over noisy channels. In particular, turbo coding [1]–[3] is a channel coding technique that facilitates near-theoretical-limit transmission throughputs, which approach the capacity of a wireless channel. Owing to this, turbo codes comprising a concatenation of upper and lower convolutional codes are widely employed in state-of-the-art mobile telephony standards, such as WiMAX [4] and LTE [5]. However, the *processing* throughput of the turbo decoding can impose a bottleneck on the *transmission* throughput in real-time or very throughput-demanding applications, such as flawless, high-quality video conferencing. In dedicated receiver hardware, a state-of-the-art turbo decoder Application-Specific Integrated Circuits (ASICs) may be used for eliminating the bottleneck of the turbo decoding. However, this bottleneck is a particular problem in the flexible receiver architectures of Software-Defined Radio (SDR) [6] and virtualized Cloud-Radio Access Network (C-RAN) [7], [8] systems that employ only programmable devices, such as Central Processing Unit (CPU) or Field-Programmable Gate Array (FPGA), which typically exhibit a limited processing performance capability or a high-cost. Although CPUs are capable of carrying out most of the LTE and WiMAX baseband operations, they are not well-suited to the most processor-intensive aspect, namely turbo decoding [9], [10]. Likewise, while high-performance and large-size FPGAs are well-suited to the parallel processing demands of state-of-the-art turbo decoding algorithms, they are relatively expensive. By contrast, General-Purpose Graphics Processing Units (GPGPUs) offer the advantages of high performance parallel processing at a low cost. Owing to this, GPGPUs have been favoured over CPUs and FPGAs as the basis of SDRs, where a high processing throughput at a low cost is required [11], [12]. This motivates the implementation of the turbo decoding algorithm on GPGPU, as was first demonstrated in [13], [14].

However, turbo decoder implementations typically op-

The financial support of the EPSRC, Swindon UK under the grants EP/J015520/1, EP/L010550/1 and the TSB, Swindon UK under the grant TS/L009390/1 is gratefully acknowledged. The research data for this paper is available at http://dx.doi.org/10.5258/SOTON/385323.

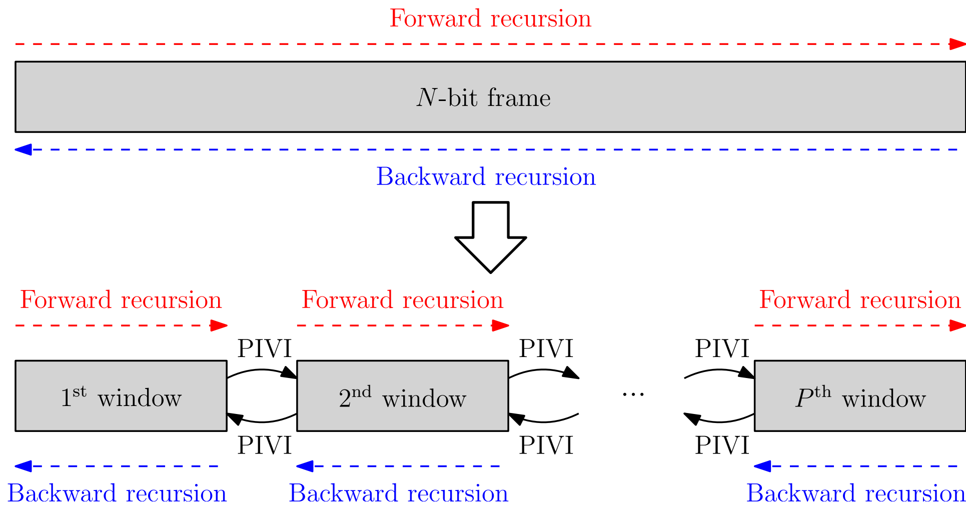


Fig. 1: Schematic of the Log-BCJR turbo decoder using windowing and PIVI techniques.

erate on the basis of the serially-oriented Logarithmic Bahl-Cocke-Jelinek-Raviv (Log-BCJR) algorithm [15]. More specifically, this algorithm processes the bits of the message frame using both forward and backward recursions [16], which impose strict data dependencies and hence require processing, which is spread over numerous consecutive clock cycles. In order to mitigate the inherent bottleneck that the serial nature of the Log-BCJR algorithm imposes on the achievable processing throughput, the above-mentioned GPGPU implementation of [13] invoke a variety of methods for increasing the parallelism of the algorithm. For example, the windowing technique of [17], [18] decomposes each frame of N bits into P equal-length windows, giving a window length of $W = \frac{N}{P}$, as shown in Figure 1. The processing throughput may be increased by a factor of P upon processing the windows concurrently, each using separate forwards and backwards recursions. Here, the Previous Iteration Value Initialization (PIVI) technique of [17], [18] may be employed for allowing the adjacent windows to assist each others' operation. However, the error correction capability of the PIVI Log-BCJR turbo decoder is degraded as the number P of windows is increased. For this reason, the maximum number of windows employed in previous GPGPU implementations of the LTE turbo decoder associated with $N = 6144$ was $P = 192$ [13], [17], [18], which avoids any significant error correction performance degradation and facilitates a 192-fold increase in the grade of parallelism [19]. Furthermore, the concept of trellis state-level parallelism may be employed [12]. More specifically, the forward and backward recursions of the Log-BCJR algorithm operates on the basis of trellises comprising M states per bit [15]. Since there are no data dependencies amongst the calculations performed for each of the M states, these can be performed concurrently. Since the LTE turbo decoder relies on $M = 8$ states, the combination of the trellis state-level parallelism and windowing facilitates a degree of parallelism up to $P \times M = 1536$, occupying 1536 concurrent threads on a GPGPU. However GPGPUs are typically capable of exploiting much higher degrees of parallelism than this [20], implying that the existing GPGPU based turbo decoder implementations do not exploit the full potential for achieving a high processing throughput. Although a higher degree of parallelism may be achieved by processing several frames in parallel [21], this would only be useful when several frames were available for simultaneous decoding. Furthermore, the act of processing frames in parallel does not improve the processing latency of the turbo decoder, which hence exceeds the tolerable transmission latency of many applications.

Motivated by these issues, we previously proposed a Fully-Parallel Turbo Decoder (FPTD) algorithm [22], which dispenses with the serial data dependencies of the conventional Log-BCJR turbo decoder algorithm. This enables every bit in a frame to be processed concurrently, hence achieving a much higher degree of parallelism than the previously demonstrated in the literature. Thus, the FPTD is well suited for multi-core processors [23], potentially facilitating a significant processing throughput gain, relative to the state-of-the-art. However, our previous work of [22] considered the FPTD at a purely algorithmic level, without addressing its hardware implementation. Against this background, the contribution of this paper is follows.

1) We propose a beneficial enhancement of the FPTD algorithm of [22] so that it supports Single Instruc-

tion Multiple Data (SIMD) operation and therefore it becoming better suited for implementation on a GPGPU. More specifically, we reformulate the FPTD algorithm so that the operations performed for the upper decoder are identical to those carried out by the lower decoder, despite the differences between the treatment of the systematic bits in the upper and lower encoders. The proposed SIMD FPTD algorithm also requires less high-speed memory and has a lower computational complexity compared to the FPTD algorithm of [22], which are desirable characteristics for GPGPU implementations.

2) We propose a beneficial GPGPU implementation of our SIMD FPTD for the LTE turbo code, achieving a throughput of up to 18.7 Mbps. Furthermore, our design overcomes a range of significant challenges related to topological mapping, data rearrangement and memory allocation.
3) We implement a PIVI Log-BCJR LTE turbo decoder on a GPGPU as a benchmarker, achieving a throughput of up to 8.2 Mbps, while facilitating the same BER as our SIMD FPTD having a window size of $N/P = 32$, which is comparable to the throughputs of 6.8 Mbps and 4 Mbps, demonstrated in the pair of state-of-the-art benchmarkers of [13] and [17], respectively.
4) We show that when used for implementing the LTE turbo decoder, the proposed SIMD FPTD achieves a degree of parallelism that is between 4 and 24 times higher, representing a processing throughput improvement between 2.3 to 9.2 times as well as a latency reduction between 2 to 8.2 times. However, this is achieved at the cost of increasing the overall complexity by a factor between 1.7 and 3.3.

The rest of the paper is organized as follows. Section II provides an overview of GPGPU computing and its employment for the Log-BCJR turbo decoder. Section III introduces our SIMD FPTD algorithm proposed for the implementation of the LTE turbo decoder. Section IV discusses the implementation of the proposed SIMD FPTD using a GPGPU, considering topological mapping, data rearrangement and memory allocation. Section V presents our simulation results, including error correction performance, degree of parallelism, processing latency, processing throughput and complexity. Finally, Section VI offers our conclusions.

II. GPU COMPUTING AND IMPLEMENTATIONS

GPUs offer a flexible throughput-oriented processing architecture, which was originally designed for facilitating massively parallel numerical computations, such as 3D image graphics [9] and physics simulations [24]. Additionally, the GPGPU technology provides an opportunity to utilize the GPU's capability to perform several trillion Floating-point Operations Per Second (FLOPS) for general-purpose applications, such as used for the computations performed in an SDR platform. In particular, the Compute Unified Device Architecture (CUDA) [20] platform offers a software programming model, which enables programmers to efficiently exploit a GPGPU's computational units to be exploited for general-purpose computations. As shown in Figure 2, a programmer may specify GPGPU instructions using CUDA *kernels*, which are software subroutines that may be called by the host CPU and then executed on the GPU's computational units. CUDA manages these computational units at three levels, corresponding to the *grids*, *thread blocks* and *threads*. Each call of a kernel invokes a *grid*, which typically comprises of many *thread blocks*, each of which typically comprises many *threads*, as shown in Figure 2. However, during the kernel's execution, all threads are grouped into warps, each of which comprises 32 threads. Each warp is operated in a Single Instruction Multiple-Data (SIMD) [25] fashion, with all of the 32 constituent threads executing identical instructions at the same time, but on different data elements.

There are several different types of memory in a GPU, including global memory, shared memory and registers, as shown in Figure 2. Each different type of memory has different properties, which may be best exploited in different circumstances in order to optimize the performance of the application. More specifically, global memory is an off-chip memory that typically has a large capacity accessible from the host CPU, as well as from any thread on the GPU. Global memory is typically used for exchanging data between the CPU and the GPU, although it has the highest access latency and most limited bandwidth, compared to the other types of GPU memory. By contrast, shared memory is user-controlled on-chip cache, which has very high bandwidth (bytes/second) and extremely low access latency. However, shared memory has a limited capacity and an access scope that is limited to a single thread block. Owing to this, a thread in a particular thread block cannot access any shared memory allocated to any other thread block. Furthermore, all data stored in shared memory will be released automatically once the execution of the corresponding thread block is completed. In comparison to global and shared memory, registers have the largest bandwidth and the smallest access latency. However, registers have very limited capacity and their access scope is limited to a single thread.

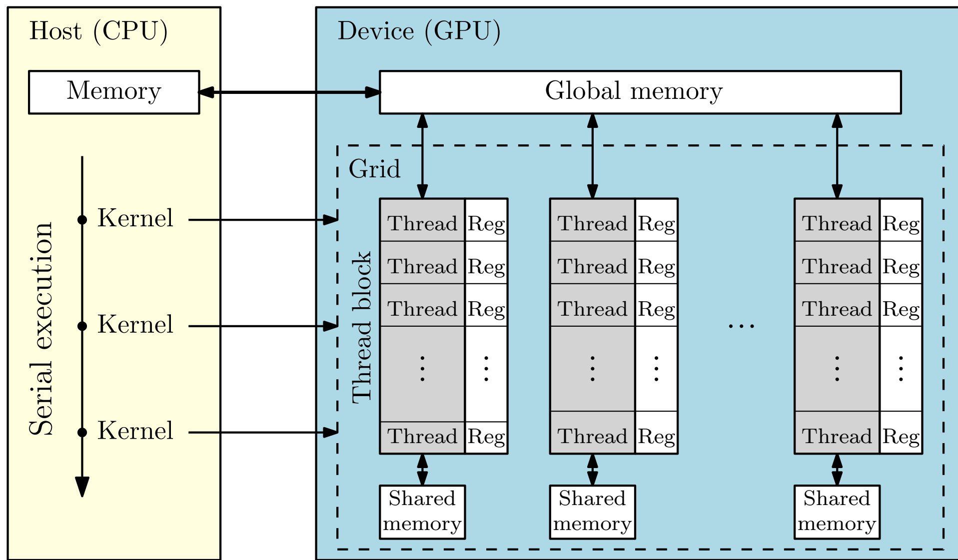


Fig. 2: Schematic of GPU computing

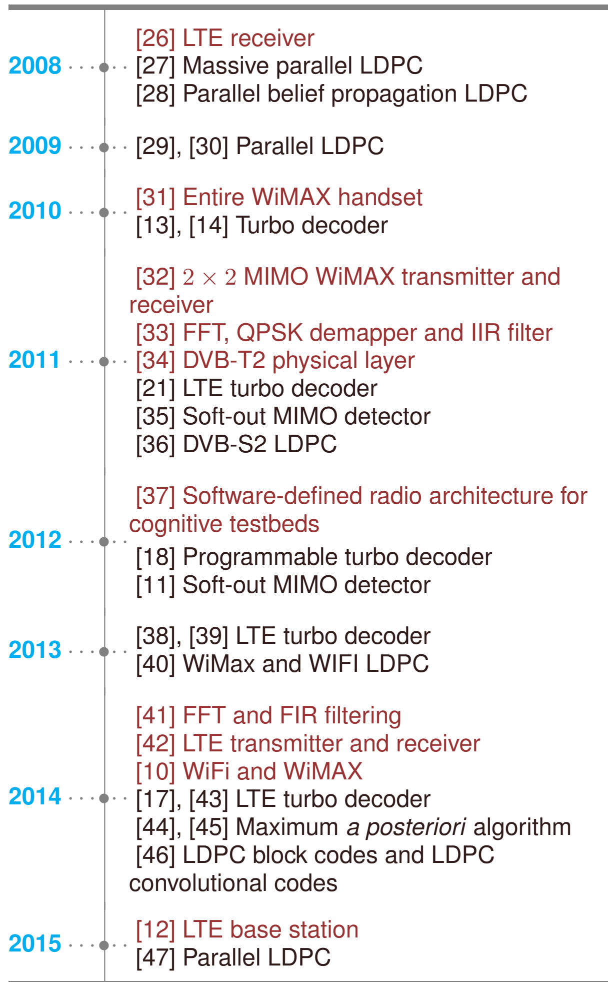


Fig. 3: Selected implementations of GPU based SDRs, where the implementations of an entire transmitting system is colored in red, whilst the implementations focusing on a particular application is colored in black.

Considering these features, many previous research projects have explored the employment of GPGPUs in SDR applications, as shown in Figure 3. Note that the GPGPU-based virtualized C-RAN implementation has not been exploited, although a C-RAN system has been implemented on the Amazon Elastic Compute Cloud (Amazon EC2) [48] using only CPUs. More specifically, [37] compared several different SDR implementation approaches in terms of programmability, flexibility, energy consumption and computing power. In particular, [37] recommended the employment of GPGPU as a co-processor to complement an ASIC, FPGA or Digital Signal Processor (DSP). Additionally, [33] characterized the performance of GPGPUs, when employed for three different operations, namely Fast Fourier Transform (FFT), Quadrature Phase Shift Keying (QPSK) demapper and Infinite Impulse Response (IIR) filter. Similarly, [41] compared the processing throughput and energy efficiency of a particular FPGA and a particular GPGPU, when implementing both the FFT and a Finite Impulse Response (FIR) filter.

As shown in Figure 3, [12], [26], [31], [32], [42] implemented an entire transmitter, receiver or transceiver for the LTE or WiMAX standard on a SDR platform that employs GPGPUs. Additionally, [11], [35] implemented a soft-output Multiple-Input Multiple-Output (MIMO) detector, while [34] implemented the Digital Video Broadcasting (DVB) physical layer on a GPGPU. All of these previous research efforts demonstrated that GPGPUs offer an improved processing throughput, compared to the family of implementations using only a CPU. Furthermore, [12] showed that an LTE base station supporting a peak data rate of 150 Mbps can be implemented

using four NVIDIA GTX 680 GPUs, achieving a similar energy efficiency to a particular dedicated LTE baseband hardware. However, [12], [42] demonstrated that turbo decoding is the most processor-intensive operation of basestation processing, requiring at least 64% of the processing resources used for receiving a message frame, where the remaining 36% includes the FFT, demapping, demodulation and other operations. Motivated by this, a number of previous research efforts [13], [14], [17], [18], [21], [38], [39], [43]–[45] have proposed GPGPU implementations dedicated to turbo decoding, as shown in Figure 3. Additionally, the authors of [27]–[30], [36], [40], [46], [47] have proposed GPGPU implementations of LDPC decoders.

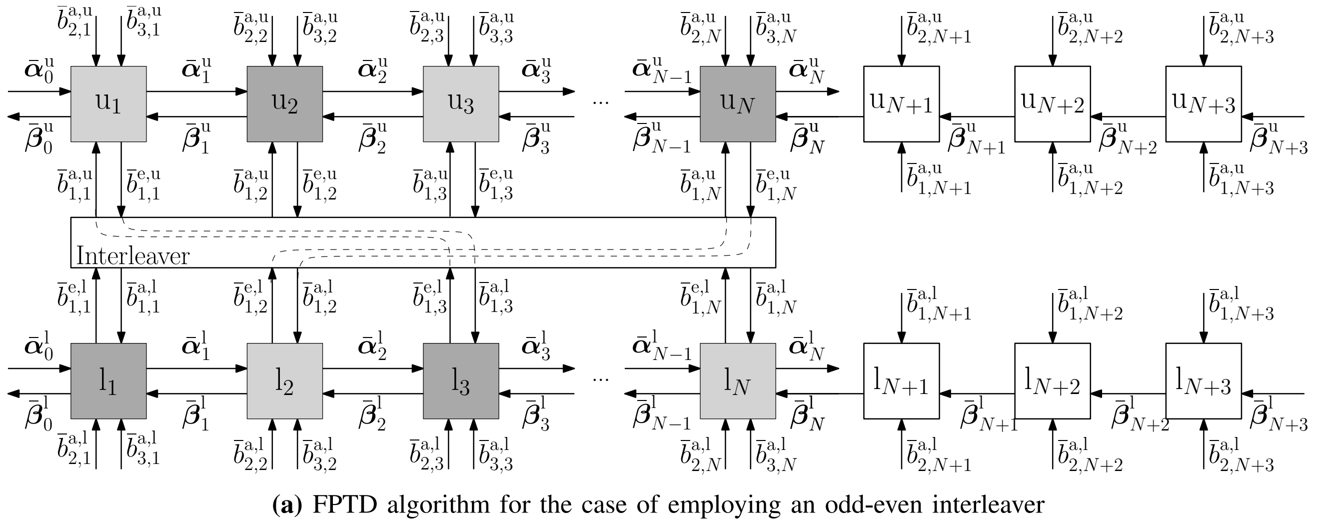
III. Single-instruction-multiple-data fully-parallel turbo decoder algorithm

In this section, the operation of the proposed SIMD FPTD algorithm is detailed in Section III-A and it is compared with the FPTD algorithm of [22] in Section III-B.

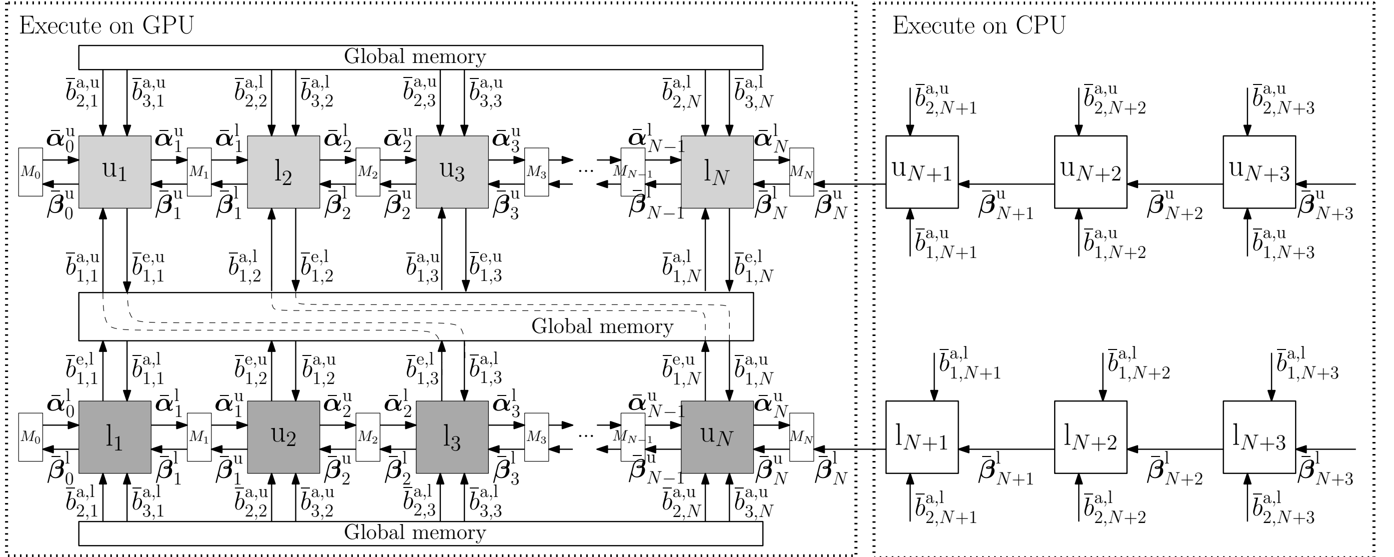
A. Operation of the proposed SIMD FPTD algorithm

In this section, we detail our proposed SIMD FPTD algorithm for the LTE turbo decoder, using the schematic of Figure 4(a). The corresponding turbo encoder is not illustrated in this paper, since it is identical to the conventional LTE turbo encoder [5]. As in the PIVI Log-BCJR turbo decoder, the proposed SIMD FPTD employs an upper decoder and a lower decoder, which are separated by an interleaver. Accordingly, Figure 4(a) shows two rows of so-called algorithmic blocks, where the upper row constitutes the upper decoder, while the lower decoder is comprised of the lower row of algorithmic blocks. Like the PIVI Log-BCJR turbo decoder, the input to the proposed SIMD FPTD comprises Logarithmic Likelihood Ratios (LLRs) [49], where each LLR $\bar{b} = \ln[\Pr(b = 1)/\Pr(b = 0)]$ conveys soft information pertaining to the corresponding bit b within the turbo encoder. More specifically, when decoding frames comprising N bits, this input comprises the six LLR vectors shown in Figure 4(a): (a) a vector $[\bar{b}^{\text{a,u}}_{2,k}]^{N+3}_{k=1}$ comprising $N+3$ *a priori* parity LLRs for the upper decoder; (b) a vector $[\bar{b}^{\text{a,u}}_{3,k}]^{N}_{k=1}$ comprising N *a priori* systematic LLRs for the upper decoder; (c) a vector $[\bar{b}^{\text{a,u}}_{1,k}]^{N+3}_{k=N+1}$ comprising three *a priori* message termination LLRs for the upper decoder; (d) a vector $[\bar{b}^{\text{a,l}}_{2,k}]^{N+3}_{k=1}$ comprising $N+3$ *a priori* parity LLRs for the lower decoder; (e) a vector $[\bar{b}^{\text{a,l}}_{3,k}]^{N}_{k=1}$ comprising N *a priori* systematic LLRs for the lower decoder; (f) a vector $[\bar{b}^{\text{a,l}}_{1,k}]^{N+3}_{k=N+1}$ comprising three *a priori* message termination LLRs for the lower decoder. Note that vectors $[\bar{b}^{\text{a,u}}_{2,k}]^{N+3}_{k=1}$ and $[\bar{b}^{\text{a,l}}_{2,k}]^{N+3}_{k=1}$ include LLRs pertaining to the three parity termination bits of the two component codes [5]. Furthermore, the vector $[\bar{b}^{\text{a,l}}_{3,k}]^{N}_{k=1}$ is not provided by the channel, but may instead be obtained by rearranging the order of the LLRs in the vector $[\bar{b}^{\text{a,u}}_{3,k}]^{N}_{k=1}$ using the interleaver π, where $\bar{b}^{\text{a,l}}_{3,k} = \bar{b}^{\text{a,u}}_{3,\pi(k)}$. Moreover, as in the PIVI Log-BCJR turbo decoder, the SIMD FPTD algorithm also employs the iterative operation of the upper and lower decoders. As shown in Figure 4(a), these iteratively exchange vectors $[\bar{b}^{\text{e,u}}_{1,k}]^{N}_{k=1}$ and $[\bar{b}^{\text{e,l}}_{1,k}]^{N}_{k=1}$ of extrinsic LLRs through the interleaver π, in order to obtain *a priori* message vectors $[\bar{b}^{\text{a,u}}_{1,k}]^{N}_{k=1}$ and $[\bar{b}^{\text{a,l}}_{1,k}]^{N}_{k=1}$ for the upper and lower decoders respectively [19], where $\bar{b}^{\text{a,l}}_{1,k} = \bar{b}^{\text{e,u}}_{1,\pi(k)}$ and $\bar{b}^{\text{a,u}}_{1,\pi(k)} = \bar{b}^{\text{e,l}}_{1,k}$. Following the completion of the iterative decoding process, a vector $[\bar{b}^{\text{p}}_{1,k}]^{N}_{k=1}$ of *a posteriori* LLRs can be obtained, where $\bar{b}^{\text{p}}_{1,k} = \bar{b}^{\text{a,u}}_{1,k} + \bar{b}^{\text{a,u}}_{3,k} + \bar{b}^{\text{e,u}}_{1,k}$. Throughout the remainder of this paper, the superscripts 'u' and 'l' are used only when necessary to explicitly distinguish between the upper and lower components of the turbo code and are omitted when the discussion applies equally to both.

As in the PIVI Log-BCJR turbo decoder, the proposed SIMD FPTD algorithm employs two half-iterations per decoder iteration. However, the two half-iterations do not correspond to the separate operation of the upper and lower decoders, like in the PIVI Log-BCJR turbo decoder. Furthermore, during each half-iteration, the proposed SIMD FPTD algorithm does not operate the algorithmic blocks of Figure 4(a) in a serial manner, using forward and backward recursions. Instead, the first half-iteration performs the fully-parallel operation of the lightly-shaded algorithmic blocks of Figure 4(a) concurrently, namely the odd-indexed blocks of the upper decoder and the even-indexed blocks of the lower decoder. Furthermore, the second half-iteration performs the concurrent operation of the remaining darkly-shaded algorithmic blocks of Figure 4(a), in a fully-parallel manner. This decomposition of the algorithmic blocks into odd-even algorithmic blocks is motivated by the odd-even nature of the Quadratic Permutation Polynomial (QPP) interleaver [19] used by the LTE turbo code and the Almost Regular Permutation (ARP) interleaver used by the WiMAX turbo code [4]. More explicitly, QPP and ARP interleavers only connect algorithmic blocks in the upper decoder that have an odd index k to specific blocks that also have an odd index in the lower decoder. Similarly, even-indexed blocks in the upper decoder are only connected to even-indexed blocks in the lower decoder. It is this fully-parallel operation of algorithmic blocks that yields a significantly higher degree of parallelism than the PIVI Log-BCJR turbo decoder algorithm, as



(a) FPTD algorithm for the case of employing an odd-even interleaver



(b) Mapping of the proposed SIMD FPTD algorithm onto the GPGPU

Fig. 4: Schematics of the proposed SIMD FPTD algorithm and its mapping for the GPGPU, where M_k represents global memory on the GPGPU device.

well as a significantly higher decoding throughput. More specifically, rather than requiring 10s or 100s of consecutive time periods to complete the forward and backward recursions in each window of the PIVI Log-BCJR turbo decoder, the proposed SIMD FPTD algorithm completes each half-iteration using only a single time period, during which all algorithmic blocks in the corresponding set are operated concurrently. Note also that this odd-even concurrent operation of algorithmic blocks in the upper and lower decoder represents a significant difference between the FPTD algorithm and a PIVI Log-BCJR decoder employing a window length of $W = 1$, as considered in [38]. More specifically, a PIVI Log-BCJR decoder having a window length of $W = 1$ may require as many as $I = 65$ iterations to maintain a similar BER performance as a PIVI Log-BCJR decoder having a window length of $W = 32$ and $I = 7$ iterations [38]. By contrast, taking advantage of the odd-even feature our FPTD algorithm requires only $I = 36$ iterations to achieve this, as it will be detailed in Section V-A.

In the tth time period of proposed SIMD FPTD, each algorithmic block of the corresponding odd or even shading having an index $k \in \{1, 2, 3, \ldots, N\}$ accepts five inputs and generates three outputs, as shown in Figure 4(a). In addition to the LLRs $\bar{b}^{\text{a},t-1}_{1,k}$, $\bar{b}^{\text{a}}_{2,k}$ and $\bar{b}^{\text{a}}_{3,k}$, the kth algorithmic block requires the vectors

$$\bar{\gamma}_k^t(S_{k-1}, S_k) = b_1(S_{k-1}, S_k) \cdot \bar{b}_{1,k}^{\mathrm{a},t-1} + b_2(S_{k-1}, S_k) \cdot \bar{b}_{2,k}^{\mathrm{a}} + b_3(S_{k-1}, S_k) \cdot \bar{b}_{3,k}^{\mathrm{a}} \tag{1}$$

$$\bar{\alpha}_k^t(S_k) = \max^*_{\{S_{k-1}|c(S_{k-1},S_k)=1\}} \left[\bar{\gamma}_k^t(S_{k-1}, S_k) + \bar{\alpha}_{k-1}^{t-1}(S_{k-1})\right] \tag{2}$$

$$\bar{\beta}_{k-1}^t(S_{k-1}) = \max^*_{\{S_k|c(S_{k-1},S_k)=1\}} \left[\bar{\gamma}_k^t(S_{k-1}, S_k) + \bar{\beta}_k^{t-1}(S_k)\right] \tag{3}$$

$$\begin{aligned} \bar{b}_{1,k}^{\mathrm{e},t} = & \left[\max^*_{\{(S_{k-1},S_k)|b_1(S_{k-1},S_k)=1\}} \left[b_2(S_{k-1}, S_k) \cdot \bar{b}_{2,k}^{\mathrm{a}} + \bar{\alpha}_{k-1}^{t-1}(S_{k-1}) + \bar{\beta}_k^{t-1}(S_k)\right]\right] \\ & - \left[\max^*_{\{(S_{k-1},S_k)|b_1(S_{k-1},S_k)=0\}} \left[b_2(S_{k-1}, S_k) \cdot \bar{b}_{2,k}^{\mathrm{a}} + \bar{\alpha}_{k-1}^{t-1}(S_{k-1}) + \bar{\beta}_k^{t-1}(S_k)\right]\right] \end{aligned} \tag{4}$$

$\bar{\boldsymbol{\alpha}}_{k-1}^{\text{t-1}} = [\bar{\alpha}_{k-1}^{\text{t-1}}(S_{k-1})]_{S_{k-1}=0}^{M-1}$ and $\bar{\boldsymbol{\beta}}_k^{\text{t-1}} = [\bar{\beta}_k^{\text{t-1}}(S_k)]_{S_k=0}^{M-1}$. Here, $\bar{\alpha}_{k-1}^{\text{t-1}}(S_{k-1})$ is the forward metric provided for the state $S_{k-1} \in [0, M-1]$ in the previous time period t-1 by the preceding algorithmic block, where the LTE turbo code employs $M = 8$ states. Similarly, $\bar{\beta}_k^{\text{t-1}}(S_k)$ is the backward metric provided for the state $S_k \in [0, M-1]$ in the previous time period by the following algorithmic block.

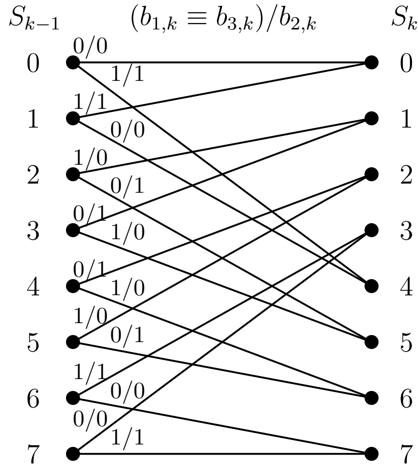


Fig. 5: State transition diagram of the LTE turbo code.

The kth algorithmic block combines these inputs using four steps, which correspond to Equations (1), (2), (3) and (4), respectively. As in the conventional Log-BCJR turbo decoder, (1) obtains an *a priori* metric $\bar{\gamma}_k^t(S_{k-1}, S_k)$ for the transition between a particular pair of states S_{k-1} and S_k. As shown in Figure 5, for the case of the LTE turbo code, this transition implies a particular binary value for the corresponding message bit $b_{1,k}$, parity bit $b_{2,k}$ or systematic bit $b_{3,k}$. Note that since $b_3(S_{k-1}, S_k) \equiv b_1(S_{k-1}, S_k)$ for the LTE turbo code, there are only four possible values for $\bar{\gamma}_k^t(S_{k-1}, S_k)$, namely $\bar{b}_{2,k}^{\mathrm{a}}$, $(\bar{b}_{1,k}^{\mathrm{a},t-1} + \bar{b}_{3,k}^{\mathrm{a}})$, $(\bar{b}_{1,k}^{\mathrm{a},t-1} + \bar{b}_{2,k}^{\mathrm{a}} + \bar{b}_{3,k}^{\mathrm{a}})$ and zero. All four of these possible values can be calculated using as few as two additions, as shown in Figure 6, which provides an optimized datapath for the kth algorithmic block of the proposed SIMD FPTD. Following this, (2) and (3) may be employed to obtain the state metrics $\bar{\boldsymbol{\alpha}}_k^t$ and $\bar{\boldsymbol{\beta}}_{k-1}^t$, respectively. Here, $c(S_{k-1}, S_k)$ adopts a binary value of 1, if there is a transition between the states S_{k-1} and S_k in the state transition diagram of Figure 5, while

$$\max^*(\bar{\delta}_1, \bar{\delta}_2) = \max(\bar{\delta}_1, \bar{\delta}_2) + \ln(1 + e^{-|\bar{\delta}_1 - \bar{\delta}_2|}) \tag{5}$$

is the Jacobian logarithm [16], as is employed by the Log-BCJR decoder. Note that the Jacobian logarithm may be approximated as

$$\max^*(\bar{\delta}_1, \bar{\delta}_2) \approx \max(\bar{\delta}_1, \bar{\delta}_2) \tag{6}$$

in order to reduce the computational complexity, as in the Max-Log-BCJR. Note that for those transitions having a metric $\bar{\gamma}_k^t(S_{k-1}, S_k)$ of zero, the corresponding terms in (2) and (3) can be ignored, hence reducing the number of additions required. This is shown in the optimized datapath of Figure 6. Finally, (4) may be employed for obtaining the extrinsic LLR $\bar{b}_{1,k}^{\mathrm{e},t}$, as shown in Figure 6. This LLR may then be output by the algorithmic block, as shown in Figure 4(a).

When operating the kth algorithmic block in the first half-iteration of the iterative decoding process, the *a priori* message LLR provided by the other row is unavailable, hence it is initialized as $\bar{b}_{1,k}^{\mathrm{a},t-1} = 0$, accordingly. Likewise, the forward state metrics from the neighboring algorithmic blocks are unavailable, hence these are initialized as $\bar{\boldsymbol{\alpha}}_{k-1}^{t-1} = [0, 0, 0, \ldots, 0]$ for $k \in [2, N]$. However, in the case of the $k = 1^{\text{st}}$ algorithmic block, we employ $\bar{\boldsymbol{\alpha}}_0^{t-1} = [0, -\infty, -\infty, ..., -\infty]$ in all decoding iterations, since the LTE trellis is guaranteed to start from an initial state of $S_0 = 0$. Similarly, before operating the kth algorithmic block in the first half-iteration, we employ $\bar{\boldsymbol{\beta}}_k^{t-1} = [0, 0, 0, \ldots, 0]$ for $k \in [1, N-1]$. Further-

more, we employ $\bar{\boldsymbol{\beta}}^{t-1}_{N+3} = [0, -\infty, -\infty, ..., -\infty]$, since the LTE turbo coding employs three termination bits to guarantee $S_{N+3} = 0$. Note that (1), (2), (3) and (4) reveal that $\bar{\boldsymbol{\beta}}_N$ is independent of $\bar{\boldsymbol{\alpha}}_N$. Therefore, the algorithmic blocks with indices $k \in [N+1, N+3]$, shown as unshaded blocks in Figure 4(a), can be processed before and independently of the iterative decoding process. This may be achieved by employing only equations (1) and (3), where the term $b_3(S_{k-1}, S_k) \cdot \bar{b}^{\mathrm{a}}_{3,k}$ is omitted from (1). More specifically, these equations are employed in a backward recursion, in order to successively calculate $\bar{\boldsymbol{\beta}}_{N+2}$, $\bar{\boldsymbol{\beta}}_{N+1}$ and $\bar{\boldsymbol{\beta}}_N$, the latter of which is employed throughout the iterative decoding process by the Nth algorithmic block.

B. Comparison with the FPTD algorithm of [22]

In this section, we compare the proposed SIMD FPTD algorithm with the original FPTD algorithm of [22]. In particular, we compare the operation, temporary storage requirements and computational complexity of these decoders. Note that in analogy to (1), the FPTD algorithm of [22] employs a summation of three *a priori* LLRs, when operating the algorithmic blocks of the upper row having an index $k \in \{1, 2, 3, \ldots, N\}$. However, a summation of just two *a priori* LLRs is employed for the corresponding blocks in the lower row of the FPTD algorithm of [22], since in this case the term $b_3(S_{k-1}, S_k) \cdot \bar{b}^{\mathrm{a}}_{3,k}$ is omitted from the equivalent of (1). By contrast, the proposed SIMD FPTD algorithm employs (1) in all algorithmic blocks, ensuring that all of them operate in an identical manner, hence facilitating SIMD operation, which is desirable for GPGPU implementations. This is achieved by including the *a priori* systematic LLR $\bar{b}^{\mathrm{a}}_{3,k}$ in the calculation of (1), regardless of whether the algorithmic block appears in the upper or the lower row. Furthermore, in contrast to the FPTD algorithm of [22], $\bar{b}^{\mathrm{a}}_{3,k}$ is omitted from the calculation of (4), regardless of which row the algorithmic blocks appears in.

A further difference between the proposed SIMD FPTD algorithm and the original FPTD algorithm of [22], is motivated by reductions in memory usage and computational complexity. More specifically, the algorithmic blocks of the proposed SIMD FPTD algorithm are redesigned to use fewer intermediate variables and computations. In particular, the transition metric $\bar{\gamma}_k(S_{k-1}, S_k)$ of (1) can only adopt three non-zero values, as described above. By contrast, the original FPTD algorithm of [22] needs to calculate and store a different transition metric $\bar{\delta}_k(S_{k-1}, S_k)$ for each of the sixteen transitions. The proposed approach allows a greater

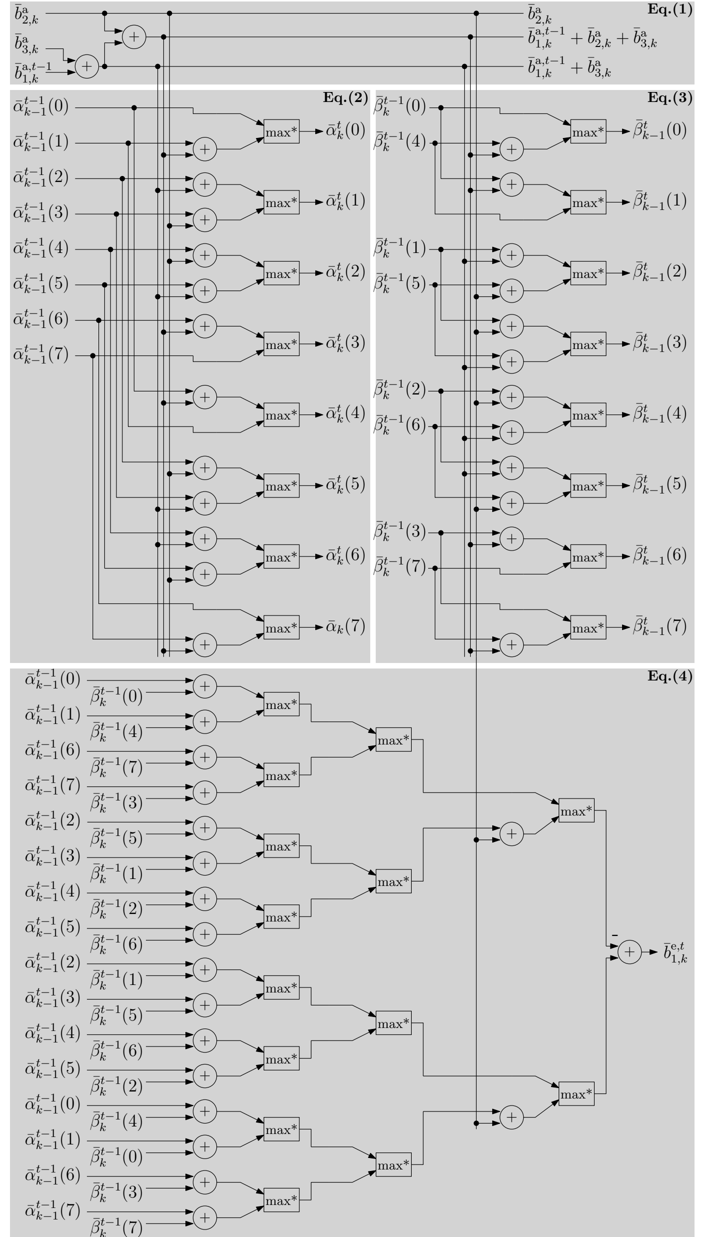


Fig. 6: The optimized datapath inside the kth algorithmic block of the proposed SIMD FPTD algorithm for the LTE turbo decoder.

proportion of the intermediate variables to be stored in the GPGPU's limited number of low-latency registers, with less reliance on its high-latency memory. Since the GPGPU's low-latency registers are shared among all N algorithmic blocks, the benefit of reducing the reliance of each block on intermediate variables is magnified by N times. Owing to this, a slight reduction in the memory usage of each algorithmic block results in a huge reduction in the total memory usage, especially when N is large.

Furthermore, while the proposed SIMD FPTD algorithm, the original FPTD algorithm of [22] and the PIVI Log-BCJR decoder all require the same number

of max* operations per algorithmic blocks, the proposed SIMD FPTD algorithm requires the fewest additions and subtractions. More specifically, as shown in the optimized datapath for the LTE turbo code of Figure 6, the proposed SIMD FPTD algorithm requires only 45 additions and subtractions per algorithmic block. This is approximately 5% lower than the 47.5 additions and subtractions required by the original FPTD algorithm of [22], as well as approximately 19% lower than the 55.5 required by the PIVI Log-BCJR decoder. Note that this computational complexity reduction is achieved by exploiting the relationship max*(A + C, B + C) = max*(A, B) + C [50]. This relationship holds for both the exact max* of (5) and approximate max* of (6). More specifically, (4) requires sixteen additions for obtaining $\bar{\alpha}_{k-1} + \bar{\beta}_k$ for the sixteen transitions in the LTE trellis, eight of which also require an extra addition for obtaining $\bar{\alpha}_{k-1} + \bar{\beta}_k + \bar{b}^{\mathrm{a}}_{2,k}$, before the max* operation. By grouping the transitions carefully, the additions of $\bar{b}^{\mathrm{a}}_{2,k}$ can be moved to after the max* operation. Owing to this, only two additions are required, rather than eight, as shown in Figure 6. Note that the datapath of Figure 6 has been specifically optimized for the LTE turbo code. By contrast, the FPTD algorithm of [22] is optimized for general turbo code applicability, yielding a more desirable design in the case of the duo-binary WiMAX turbo code [4], for example.

IV. Implementation of the SIMD FPTD algorithm on a GPGPU

This section describes the implementation of the proposed SIMD FPTD algorithm using an NVIDIA GPGPU platform, adopting the Compute Unified Device Architecture (CUDA) [20]. The mapping of the SIMD FPTD algorithm onto the GPGPU and its memory allocation are discussed in Sections IV-A and IV-B, respectively. The pseudo code of the proposed GPGPU kernel designed for implementing the SIMD FPTD algorithm is described in Section IV-C.

A. Mapping the SIMD FPTD algorithm onto a GPGPU

The proposed SIMD FPTD algorithm of Figure 4(a) may be mapped onto a CUDA GPGPU using a single kernel. Here, two approaches are compared. In the first approach, each execution of the kernel performs one half iteration of the proposed algorithm, requiring $2I$ kernel repetitions in order to complete I number of decoding iterations. For this approach, the GPU kernel repetitions are scheduled serially by the CPU, achieving synchronization between each pair of consecutive half iterations by the CPU. This synchronization ensures that all parts of a particular half iteration are completed, before any parts of the next half iteration begin. However, this synchronization occupies an average of 31.3% of the total processing time, which is due to the communication overhead between the CPU and the GPU, according to our experimental results. Owing to this, our second approach performs all $2I$ half iterations within a single GPU kernel run, eliminating the requirement for any communication between the CPU and the GPU during the iterative decoding process. However, the inter-block synchronization has to be carried out by the GPU in order to maintain the odd-even nature of the operation. Since CUDA GPGPUs do not have any native support for inter-block synchronization, here we include the lock-free inter-block synchronization technique of [51]. We perform this synchronization at end of every half iteration, which reduces the time dedicated to the synchronization from 31.3% to 15.5%, according to our experimental results. Owing to this superior performance compared to CPU synchronization, inter-block synchronization on the GPU is used for our proposed FPTD implementation and its performance is characterized in Section V.

Our kernel employs N number of threads, with one for each of the N algorithmic blocks that are operated within each half iteration of Figure 4(a). Here, the kth thread processes the kth algorithmic block in the upper or lower row according to the odd-even arrangement of Figure 4(a), where $k \in [1, N]$. Note that it would be possible to achieve further parallelism by employing eight threads per algorithmic block, rather than just one. This would facilitate state-level parallelism as described in Section I for the conventional GPGPU implementation of the PIVI Log-BCJR turbo decoder. However, our experiments reveal that state-level parallelism offers no advantage for the proposed SIMD FPTD algorithm. More specifically, according to the Nsight profiler of [52], the processing throughput of the proposed FPTD implementation is bounded by the memory bandwidth rather than memory access latency, which implies that the parallelism of N is already large enough to make the most of the GPGPUs computing resource. Furthermore, employing state-level parallelism would result in a requirement for more accesses of the global memory, in order to load the *a priori* LLRs $\bar{b}^{\mathrm{a}}_{1,k}$, $\bar{b}^{\mathrm{a}}_{2,k}$ and $\bar{b}^{\mathrm{a}}_{3,k}$, which would actually degrade the throughput.

The algorithmic blocks of the proposed SIMD FPTD algorithm are arranged in groups of 32, in order for the corresponding threads to form warps, which are particularly suited to SIMD operation. In order to maximize the computation throughput, special care must be taken to avoid *thread divergence*. This arises when 'if' and 'else' statements cause the different threads of a warp to oper-

ate differently, resulting in the serial processing of each possible outcome. However, the schematic of Figure 4(a) is prone to thread divergence, since each half iteration comprises the operation of algorithmic blocks in both the upper and the lower row, as indicated using light and dark shading. More specifically, 'if' and 'else' statements are required to determine whether each algorithmic block resides in the top or bottom row of Figure 4(a), when deciding which inputs and outputs to consider. This motivates the alternative design of Figure 4(b), in which all algorithmic blocks within the same half iteration have been relocated to the same row in order to avoid these 'if' and 'else' statements. More specifically, the algorithmic blocks that have an even index in the upper row have been swapped with those from the lower row. As a result, the upper row comprises the lightly-shaded blocks labeled $\mathrm{u}_{k|k \text{ is odd}}$ and $\mathrm{l}_{k|k \text{ is even}}$, whilst the lower row comprises the darkly-shaded blocks labeled $\mathrm{u}_{k|k \text{ is even}}$ and $\mathrm{l}_{k|k \text{ is odd}}$. Consequently, the operation of alternate half iterations corresponds to the alternate operation of the upper and lower rows of Figure 4(b). Note that this rearrangement of algorithmic blocks requires a corresponding rearrangement of inputs, outputs and memory, as will be discussed in Section IV-B.

As described in Section III, the consideration of the termination bits by the three algorithmic blocks at the end of the upper and lower rows can be isolated from the operation of the iterative processes. Therefore, we recommend the processing of all termination bits using the CPU, before beginning the iterative decoding process on the GPGPU. This aids the mapping of algorithmic blocks to warps and also avoids thread divergence, since the processing of the termination bits is not identical to that of the other bits, as shown in Figure 4(b).

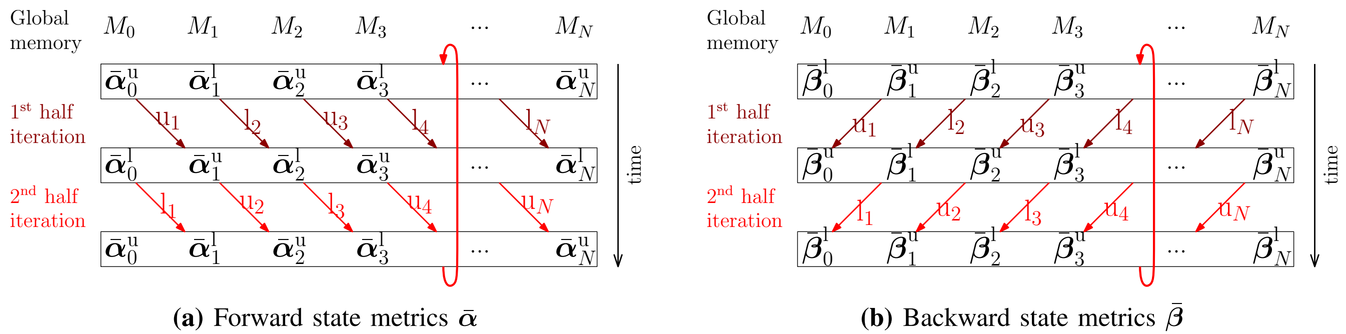
B. Data arrangement and memory allocation

Note that because the proposed SIMD FPTD employs the rearranged schematic of Figure 4(b) rather than that of Figure 4(a), the corresponding datasets must also be rearranged, using swaps and mergers. More specifically, for the *a priori* parity LLRs $\bar{b}_2^{\mathrm{a}}$ and the systematic LLRs $\bar{b}_3^{\mathrm{a}}$ the rearrangement can be achieved by swapping the corresponding elements in the upper and lower datasets, following the same rule that was applied to the algorithmic blocks of Figure 4(b). For the forward and backwards metrics $\bar{\boldsymbol{\alpha}}$ and $\bar{\boldsymbol{\beta}}$ as well as for the *a priori* message LLRs $\bar{b}_1^{\mathrm{a}}$ the rearrangement can be achieved by merging the two separate datasets for the upper and lower rows together. Furthermore, there is no need to store both the *a priori* and the extrinsic LLRs, since interleaving can be achieved by writing the latter into the memory used for storing the former, but in an interleaved order. Note that this arrangement also offers the benefit of minimizing memory usage, which is achieved without causing any overwriting, as shown in Figure 7. More explicitly, the kth memory slot M_k of Figure 4(b) may be used for passing the kth forward state metrics $\bar{\boldsymbol{\alpha}}_k^{\mathrm{u/l}}$ between the algorithmic blocks $\mathrm{u}_k/\mathrm{l}_k$ and $\mathrm{u}_{k+1}/\mathrm{l}_{k+1}$, for example. During the first half iteration, the upper algorithmic block u_k is operated to obtain $\bar{\boldsymbol{\alpha}}_k^{\mathrm{u}}$, which is stored in M_k. Then during the second half iteration, this data stored in M_k will be provided to the algorithmic block u_{k+1}, before it is overwritten by the new data $\bar{\boldsymbol{\alpha}}_k^{\mathrm{l}}$, which is provided by the algorithmic block l_k.

As illustrated in Figure 4(b), there are a total of seven datasets that must be stored throughout the decoding process, namely $[\bar{b}_{2,k}^{\mathrm{a,u}}]_{k=1}^N$, $[\bar{b}_{2,k}^{\mathrm{a,l}}]_{k=1}^N$, $[\bar{b}_{3,k}^{\mathrm{a,u}}]_{k=1}^N$, $[\bar{b}_{3,k}^{\mathrm{a,l}}]_{k=1}^N$, $[\bar{b}_{1,k}^{\mathrm{a}}]_{k=1}^N$, $[\bar{\boldsymbol{\alpha}}_k]_{k=1}^N$ and $[\bar{\boldsymbol{\beta}}_k]_{k=1}^N$, requiring an overall memory resource of $21N$ floating-point numbers. As shown in Figure 4(b), these datasets are stored in the global memory, since it has a large capacity and is accessible from the host CPU, as well as from any thread in the GPGPU device. However the global memory has a relatively high access latency and a limited bandwidth. In order to minimize the impact of this, each algorithmic block employs local low-latency registers to store all intermediate variables that are required multiple times within a half iteration. More specifically, the kth algorithmic block uses registers to store $\bar{b}_{2,k}^{\mathrm{a}}$, $(\bar{b}_{1,k}^{\mathrm{a}} + \bar{b}_{3,k}^{\mathrm{a}})$, $(\bar{b}_{1,k}^{\mathrm{a}} + \bar{b}_{2,k}^{\mathrm{a}} + \bar{b}_{3,k}^{\mathrm{a}})$, $\bar{\boldsymbol{\alpha}}_{k-1}$ and $\bar{\boldsymbol{\beta}}_k$, comprising a total of 19 floating-point numbers, as shown in Figure 6.

C. Pseudo code

Algorithm 1 describes the operation of the kth thread dedicated to the computation of the kth algorithmic block, in analogy to the datapath of Figure 6. Note that the labels of Register (R) and Global memory (G) shown in Algorithm 1 indicate the type of the memory used for storing the corresponding data. Each thread is grouped into four steps as follows. The first step caches the *a priori* LLR $\bar{b}_{2,k}^{\mathrm{a}}$ and the *a priori* state metrics $\bar{\boldsymbol{\alpha}}_{k-1}$ and $\bar{\boldsymbol{\beta}}_k$ from the global memory to the local registers. Furthermore, the first step computes $\bar{b}_{13}^{\mathrm{a}} = \bar{b}_{1,k}^{\mathrm{a},t-1} + \bar{b}_{3,k}^{\mathrm{a}}$ and $\bar{b}_{123}^{\mathrm{a}} = \bar{b}_{1,k}^{\mathrm{a},t-1} + \bar{b}_{2,k}^{\mathrm{a}} + \bar{b}_{3,k}^{\mathrm{a}}$, before storing the results in the local registers. Following this, the second and third steps compute the extrinsic forward state metrics $\bar{\boldsymbol{\alpha}}_k^t$ and the extrinsic backward state metrics $\bar{\boldsymbol{\beta}}_{k-1}^t$, in analogy to the datapath of Figure 6. The results of these computations are written directly into the corresponding memory slot M_k in the global memory, as shown in Figure 4(b). In the fourth step, the extrinsic LLR $\bar{b}_{1,k}^{\mathrm{e},t}$



(a) Forward state metrics $\bar{\boldsymbol{\alpha}}$ **(b)** Backward state metrics $\bar{\boldsymbol{\beta}}$

Fig. 7: Schematic of using the global memory to store the intermediate data of $\bar{\boldsymbol{\alpha}}$ and $\bar{\boldsymbol{\beta}}$.

is computed and stored in the global memory. Here, interleaving or deinterleaving is achieved by storing the extrinsic LLRs into particular global memory slots selected according to the design of the LTE interleaver. Note that the intermediate values of $\bar{\delta}_0$ and $\bar{\delta}_1$ require the storage of two floating-point numbers in registers, as shown in Algorithm 1. However, instead of using two new registers, they can be stored respectively in the registers that were previously used for storing the values of $\bar{b}^{\mathrm{a}}_{13}$ and $\bar{b}^{\mathrm{a}}_{123}$, since these are not required in the calculations of the fourth step. As a result, a total of 19 registers are required per thread, as discussed above.

V. RESULTS

In the following sub-sections, we compare the performance of the proposed GPGPU implementation of our SIMD FPTD algorithm with that of the state-of-the-art GPGPU turbo decoder implementation in terms of error correction performance, degree of parallelism, processing throughput and complexity. Both turbo decoders were implemented using single-precision floating-point arithmetic and both were characterized using the Windows 8 64-bit operating system, an Intel I7-2600@3.4GHz CPU, 16GB RAM and an NVIDIA GTX680 GPGPU. This GPGPU has eight Multiprocessors (MPs) and 192 CUDA cores per MP, with a GPU clock rate of 1.06 GHz and a memory clock rate of 3 GHz.

The state-of-the-art benchmarker employs the Log-BCJR algorithm, with PIVI windowing, state-level parallelism and Radix-2 operation [17], [18]. This specific combination was selected, since it offers a high throughput and a low complexity, at a negligible cost in terms of BER degradation. This algorithm was mapped onto the GPGPU according to the approach described in [13]. Furthermore, as recommended in [13] and [18], the longest LTE frames comprising $N = 6144$ bits were decomposed into $P \in \{192, 128, 96, 64, 32\}$ number of partitions. This is equivalent to having PIVI window lengths of $W = N/P \in \{32, 48, 64, 96, 192\}$, respectively.

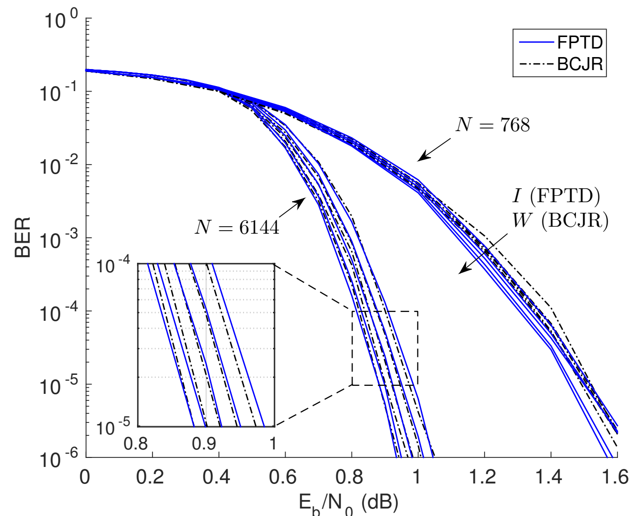


Fig. 8: BER performance for the PIVI Log-BCJR turbo decoder having window lengths of $W \in \{32, 48, 64, 96, 192\}$ and performing $I = 7$ iterations, as compared with that of the proposed SIMD FPTD when performing $I \in \{36, 39, 42, 46, 49\}$ iterations. Here, both decoders use the approximate max* operation.

A. BER performance

Figure 8 compares the BER performance of both the PIVI Log-BCJR turbo decoder and the proposed SIMD FPTD algorithm, when employing the approximate max* operation of (6). Here, BPSK modulation was employed for transmission over an AWGN channel. For both decoders, the BER performance is provided for a relatively short frame length of $N = 768$ bits, as well as for the longest frame length that is supported by the LTE standard, namely $N = 6144$ bits. We have not included the BER performance of the two decoders when employing the exact max* operation of (5), but we found that they obey the same trends as Figure 8.

Algorithm 1 A kernel for computing a half-iteration of the proposed SIMD FPTD algorithm

Step 1: Loading data
for $i = 0$ to 7 **do**
(R) $\bar{\alpha}(i) \leftarrow$ (G) $\bar{\alpha}_{k-1}^{t-1}(i)$
(R) $\bar{\beta}(i) \leftarrow$ (G) $\bar{\beta}_{k}^{t-1}(i)$
end for
(R) $\bar{b}_{13}^{\mathrm{a}} \leftarrow$ (G) $\bar{b}_{1,k}^{\mathrm{a},t-1}$ + (G) $\bar{b}_{3,k}^{\mathrm{a}}$
(R) $\bar{b}_{2}^{\mathrm{a}} \leftarrow$ (G) $\bar{b}_{2,k}^{\mathrm{a}}$
(R) $\bar{b}_{123}^{\mathrm{a}} \leftarrow \bar{b}_{2}^{\mathrm{a}} + \bar{b}_{13}^{\mathrm{a}}$

Step 2:Computing forward state metrics
(G) $\bar{\alpha}_{k}^{t}(0) \leftarrow \max^*(\bar{\alpha}(0)\ ,\ \bar{\alpha}(1) + \bar{b}_{123}^{\mathrm{a}})$
(G) $\bar{\alpha}_{k}^{t}(1) \leftarrow \max^*(\bar{\alpha}(2) + \bar{b}_{13}^{\mathrm{a}}\ ,\ \bar{\alpha}(3) + \bar{b}_{2}^{\mathrm{a}})$
(G) $\bar{\alpha}_{k}^{t}(2) \leftarrow \max^*(\bar{\alpha}(4) + \bar{b}_{2}^{\mathrm{a}}\ ,\ \bar{\alpha}(5) + \bar{b}_{13}^{\mathrm{a}})$
(G) $\bar{\alpha}_{k}^{t}(3) \leftarrow \max^*(\bar{\alpha}(6) + \bar{b}_{123}^{\mathrm{a}}\ ,\ \bar{\alpha}(7))$
(G) $\bar{\alpha}_{k}^{t}(4) \leftarrow \max^*(\bar{\alpha}(0) + \bar{b}_{13}^{\mathrm{a}}\ ,\ \bar{\alpha}(1))$
(G) $\bar{\alpha}_{k}^{t}(5) \leftarrow \max^*(\bar{\alpha}(2) + \bar{b}_{2}^{\mathrm{a}}\ ,\ \bar{\alpha}(3) + \bar{b}_{13}^{\mathrm{a}})$
(G) $\bar{\alpha}_{k}^{t}(6) \leftarrow \max^*(\bar{\alpha}(4) + \bar{b}_{13}^{\mathrm{a}}\ ,\ \bar{\alpha}(5) + \bar{b}_{2}^{\mathrm{a}})$
(G) $\bar{\alpha}_{k}^{t}(7) \leftarrow \max^*(\bar{\alpha}(6)\ ,\ \bar{\alpha}(7)) + \bar{b}_{123}^{\mathrm{a}})$

Step 3:Computing backward state metrics
(G) $\bar{\beta}_{k-1}^{t}(0) \leftarrow \max^*(\bar{\beta}(0)\ ,\ \bar{\beta}(4) + \bar{b}_{123}^{\mathrm{a}})$
(G) $\bar{\beta}_{k-1}^{t}(1) \leftarrow \max^*(\bar{\beta}(0) + \bar{b}_{123}^{\mathrm{a}}\ ,\ \bar{\beta}(4))$
(G) $\bar{\beta}_{k-1}^{t}(2) \leftarrow \max^*(\bar{\beta}(1) + \bar{b}_{13}^{\mathrm{a}}\ ,\ \bar{\beta}(5) + \bar{b}_{2}^{\mathrm{a}})$
(G) $\bar{\beta}_{k-1}^{t}(3) \leftarrow \max^*(\bar{\beta}(1) + \bar{b}_{2}^{\mathrm{a}}\ ,\ \bar{\beta}(5) + \bar{b}_{13}^{\mathrm{a}})$
(G) $\bar{\beta}_{k-1}^{t}(4) \leftarrow \max^*(\bar{\beta}(2) + \bar{b}_{2}^{\mathrm{a}}\ ,\ \bar{\beta}(6) + \bar{b}_{13}^{\mathrm{a}})$
(G) $\bar{\beta}_{k-1}^{t}(5) \leftarrow \max^*(\bar{\beta}(2) + \bar{b}_{13}^{\mathrm{a}}\ ,\ \bar{\beta}(6) + \bar{b}_{2}^{\mathrm{a}})$
(G) $\bar{\beta}_{k-1}^{t}(6) \leftarrow \max^*(\bar{\beta}(3) + \bar{b}_{123}^{\mathrm{a}}\ ,\ \bar{\beta}(7))$
(G) $\bar{\beta}_{k-1}^{t}(7) \leftarrow \max^*(\bar{\beta}(3)\ ,\ \bar{\beta}(7)) + \bar{b}_{123}^{\mathrm{a}})$

Step 4:Computing extrinsic LLR
(R) $\bar{\delta}_0 \leftarrow \max^*(\bar{\alpha}(2) + \bar{\beta}(5)\ ,\ \bar{\alpha}(3) + \bar{\beta}(1))$
$\bar{\delta}_0 \leftarrow \max^*(\bar{\delta}_0\ ,\ \bar{\alpha}(4) + \bar{\beta}(2))$
$\bar{\delta}_0 \leftarrow \max^*(\bar{\delta}_0\ ,\ \bar{\alpha}(5) + \bar{\beta}(6))$
$\bar{\delta}_0 \leftarrow \bar{\delta}_0 + \bar{b}_{2}^{\mathrm{a}}$
$\bar{\delta}_0 \leftarrow \max^*(\bar{\delta}_0\ ,\ \bar{\alpha}(0) + \bar{\beta}(0))$
$\bar{\delta}_0 \leftarrow \max^*(\bar{\delta}_0\ ,\ \bar{\alpha}(1) + \bar{\beta}(4))$
$\bar{\delta}_0 \leftarrow \max^*(\bar{\delta}_0\ ,\ \bar{\alpha}(6) + \bar{\beta}(7))$
$\bar{\delta}_0 \leftarrow \max^*(\bar{\delta}_0\ ,\ \bar{\alpha}(7) + \bar{\beta}(3))$

(R) $\bar{\delta}_1 \leftarrow \max^*(\bar{\alpha}(0) + \bar{\beta}(4)\ ,\ \bar{\alpha}(1) + \bar{\beta}(0))$
$\bar{\delta}_1 \leftarrow \max^*(\bar{\delta}_1\ ,\ \bar{\alpha}(6) + \bar{\beta}(3))$
$\bar{\delta}_1 \leftarrow \max^*(\bar{\delta}_1\ ,\ \bar{\alpha}(7) + \bar{\beta}(7))$
$\bar{\delta}_1 \leftarrow \bar{\delta}_1 + \bar{b}_{2}^{\mathrm{a}}$
$\bar{\delta}_1 \leftarrow \max^*(\bar{\delta}_1\ ,\ \bar{\alpha}(2) + \bar{\beta}(1))$
$\bar{\delta}_1 \leftarrow \max^*(\bar{\delta}_1\ ,\ \bar{\alpha}(3) + \bar{\beta}(5))$
$\bar{\delta}_1 \leftarrow \max^*(\bar{\delta}_1\ ,\ \bar{\alpha}(4) + \bar{\beta}(6))$
$\bar{\delta}_1 \leftarrow \max^*(\bar{\delta}_1\ ,\ \bar{\alpha}(5) + \bar{\beta}(2))$

(G) $\bar{b}_{1,\pi(k)}^{\mathrm{e},t} \leftarrow \bar{\delta}_1 - \bar{\delta}_0$

Figure 8 characterizes the BER performance of the PIVI Log-BCJR turbo decoder when employing $I = 7$ iterations and the window lengths of $W \in \{32, 48, 64, 96, 192\}$. In addition to this, Figure 8 provides the BER performance of the SIMD FPTD algorithm when performing $I \in \{36, 39, 42, 46, 49\}$ iterations. As may be expected, the BER performance of the PIVI Log-BCJR turbo decoder improves when employing longer window lengths W. Therefore, more iterations I of the SIMD FPTD algorithm are required in order to achieve the same BER performance as the PIVI Log-BCJR turbo decoder, when W is increased. More specifically, Figure 8 shows that when employing $N = 6144$-bit frames, the SIMD FPTD algorithm requires $I \in \{36, 39, 42, 46, 49\}$ decoding iterations in order to achieve the same BER performance as the PIVI Log-BCJR turbo decoder performing $I = 7$ iterations with the window lengths of $W \in \{32, 48, 64, 96, 192\}$, respectively. Note that in all cases, the proposed SIMD FPTD algorithm is capable of achieving the same BER performance as the PIVI Log-BCJR turbo decoder, albeit at the cost of requiring a greater number of decoding iterations I. Note that the necessity for the FPTD to perform several times more iterations than the Log-BCJR turbo decoder was discussed extensively in [22].

B. Degree of parallelism

The degree of parallelism for the PIVI Log-BCJR turbo decoder may be considered to be given by $D_p^{\text{Log-BCJR}} = \frac{M \cdot N}{W}$, where N is the frame length, W is the window length and $M = 8$ is the number of states in the LTE turbo code trellis. Here, $M = 8$ threads can be employed for achieving state parallelism, while decoding each of the N/W windows simultaneously. By contrast, the degree of parallelism for the FPTD can be simply defined as $D_p^{\text{FPTD}} = N$, since all algorithmic blocks can be processed in parallel threads and because we do not exploit state parallelism in this case. Table I compares the parallelism D_p of the proposed SIMD FPTD with that of the PIVI Log-BCJR turbo decoder, when decomposing $N = 6144$-bit frames into windows comprising various numbers of bits W. Depending on the window length W chosen for the PIVI Log-BCJR turbo decoder, the degree of parallelism achieved by the proposed SIMD FPTD can be seen to be between 4 and 24 times higher.

C. Processing latency

Figure 9 compares the processing latency of both the proposed SIMD FPTD and of the PIVI Log-BCJR decoder, when decoding frames comprising $N = 6144$ bits using both the approximate max* operation of (6)

TABLE I: Comparison between the PIVI Log-BCJR turbo decoder and the proposed SIMD FPTD in terms of degree of parallelism, overall latency, pipelined throughput and complexity (IPBPHI and IPB), where $N = 6144$ for both decoders, $I = 7$ and $W \in \{32, 48, 64, 96, 192\}$ for the PIVI Log-BCJR turbo decoder, whilst $I \in \{36, 39, 42, 46, 49\}$ for the FPTD. Results are presented using the format x/y, where x corresponds to the case where the approximate max* operation of (6) is employed, while y corresponds to the exact max* operation of (5).

		Degree of Parallelism D_p	Overall Latency (μs)	Pipelined throughput (Mbps)	Complexity		
					IPW	IPBPHI	IPB
Log-BCJR	$W = 32$	1536	816.9 / 1041.5	8.2 / 6.3	2511 / 3842	19.6 / 30.0	274 / 420
	$W = 48$	1024	1111.5 / 1415.5	5.9 / 4.6	3720 / 5697	19.4 / 29.7	272 / 416
	$W = 64$	768	1338.9 / 1786.6	4.8 / 3.6	5044 / 7914	19.7 / 30.9	276 / 433
	$W = 96$	512	1945.7 / 2549.3	3.3 / 2.5	7368 / 11.3k	19.2 / 29.4	269 / 412
	$W = 192$	256	3694.6 / 4842.6	1.7 / 1.3	14.7k / 22.6k	19.1 / 29.4	267 / 412
FPTD	$I = 36$	6144	402.5 / 451.8	18.7 / 16.1	200 / 439	6.3 / 13.8	454 / 994
	$I = 39$		427.4 / 482.1	17.4 / 14.9			491 / 1076
	$I = 42$		454.6 / 516.4	16.2 / 13.9			529 / 1159
	$I = 46$		486.6 / 556.2	14.8 / 12.7			580 / 1270
	$I = 49$		513.4 / 589.6	13.9 / 12			617 / 1352

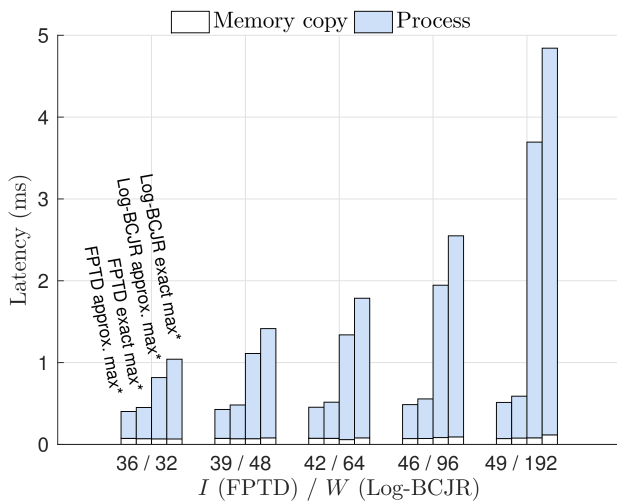


Fig. 9: Latency for the proposed SIMD FPTD with $I \in$ {36, 39, 42, 46, 49}, as compared with that of the PIVI Log-BCJR turbo decoder with I=7 and $W \in$ {32, 48, 64, 96, 192}.

and the exact max* operation of (5). Note that different numbers of iterations $I \in \{36, 39, 42, 46, 49\}$ are used for the SIMD FPTD, while $I = 7$ iterations and different window lengths $W \in \{32, 48, 64, 96, 192\}$ are employed for the PIVI Log-BCJR turbo decoder, as discussed in Section V-A. Here, the overall latency includes two parts, namely the time used for memory transfer between the CPU and the GPU, as well as the time used for the iterative decoding process. More specifically, the memory transfer includes transferring the channel LLRs from the CPU to the GPU at the beginning of the iterative decoding process and transferring the decoded results from the GPU to the CPU at the end of that process. Therefore, the time used for memory transfer depends only on the frame length N and it is almost independent of the type of decoder and the values of I and W, as shown in Figure 9. Note that the latency was quantified by averaging over the decoding of 5000 frames for each configuration. By contrast, the time used for the iterative decoding process differs significantly between the proposed SIMD FPTD and the Log-BCJR turbo decoder. More specifically, Table I shows that the overall latency of the SIMD FPTD is in the range from 402.5 μs to 513.4 μs, when the number of iterations is increased from $I = 36$ to $I = 49$, provided that the approximate max* operation of (6) is employed, hence meeting the sub 1ms requirement of the LTE physical layer [53]. By contrast, the overall latency of the PIVI Log-BCJR decoder ranges from 816.9 μs to 3694.6 μs, when the window length increases from $W = 32$ to $W = 192$, and when $I = 7$ iterations are performed, assuming that the approximate max* operation of (6) is employed. These extremities of the range are 2 times and 7.2 times worse than those of the proposed SIMD FPTD, respectively. Additionally, when the exact max* operation of (5) is employed, the overall latency of the SIMD FPTD increases by 12.3% and 14.8% for the case of $I = 36$ and $I = 49$, compared to those obtained when employing the approximate max* operation of (6). By contrast, the overall latency increases in this case by 27.5% and 31.1% for the PIVI Log-BCJR decoder associated with $W = 32$ and $W = 192$, hence further widening the gap to the latency of the proposed SIMD FPTD.

D. Processing throughput

Table I presents the processing throughputs that were measured on the GPGPU, when employing the proposed SIMD FPTD and the PIVI Log-BCJR turbo decoder

to decode frames comprising $N = 6144$ bits. Here, throughputs are presented for the case where the approximate max* operation of (6) is employed, as well as for the case of employing the exact max* operation of (5). Note that when the iterative decoding of a particular frame has been completed, its memory transfer from the GPU to CPU can be pipelined with the iterative decoding of the next frame and with the memory transfer from the CPU to GPU of the frame after that. Since Figure 9 shows that the iterative decoding is the slowest of these three processes, it imposes a bottleneck on the overall processing throughput. Owing to this, the throughput presented in Table I was obtained by considering only the iterative decoding process, based on the assumption that throughput $= N/$latency of iterative decoding. As shown in Table I, the proposed GPGPU implementation of the SIMD FPTD achieves throughputs of up to 18.7 Mbps. This exceeds the average throughput of 7.6 Mbps, which is typical in 100 MHz LTE uplink channels [54], demonstrating the suitability of the proposed implementation for C-RAN applications. Furthermore, higher throughputs can be achieved either by using a more advanced GPU or by using multiple GPUs in parallel.

Recall from Figure 8 that the proposed SIMD FPTD performing $I = 36$ iterations achieves the same BER performance as the PIVI Log-BCJR turbo decoder performing $I = 7$ iterations and having the window length of $W = 32$. Here, $W = 32$ corresponds to the maximum degree of parallelism of $P = 192$ that can be achieved for the PIVI Log-BCJR turbo decoder, without imposing a significant BER performance degradation [19]. In the case of this comparison, Table I reveals that the processing throughput of the proposed SIMD FPTD is 2.3 times and 2.5 times higher than that of the PIVI Log-BCJR turbo decoder, when the approximate max* operation and the exact max* operation are employed, respectively. An even higher processing throughput improvement is offered by the proposed SIMD FPTD, when the parallelism of the PIVI Log-BCJR turbo decoder is reduced, for the sake of improving its BER performance. For example, the proposed SIMD FPTD performing $I = 49$ iterations has a processing throughput that is 8.2 times (approximate max*) and 9.2 times (exact max*) higher than the PIVI Log-BCJR turbo decoder having a window length of $W = 192$, while offering the same BER performance. Note that owing to its lower computational complexity, the approximate max* operation of (6) facilities a higher processing throughput than the exact max* operation of (5), in the case of both decoders.

Furthermore, Table II compares the processing throughput of the proposed SIMD FPTD GPGPU implementation to that of other GPGPU implementations of the LTE turbo decoder found in the literature [13], [17], [18], [38], [43]. Here, the throughputs of all implementations are quantified for the case of decoding $N = 6144$-bit frames, when using the approximate max* operation of (6). Note that the throughputs shown in Table II for the benchmarkers employing the PIVI Log-BCJR algorithm have been linearly scaled to the case of performing $I = 7$ iterations, in order to perform a fair comparison. However, different GPUs are used for the different implementations, which complicates their precise performance comparison. In order to make fairer comparisons, we consider two different methods for normalizing the throughputs of the different implementations, namely $\frac{\text{throughput}\times 10^6}{\text{clock freq.}\times\text{mem BW}}$ and $\frac{\text{throughput}\times 10^6}{\text{clock freq.}\times\text{mem freq.}}$.

More specifically, the authors of [38] proposed a loosely synchronized parallel turbo decoding algorithm, in which the iterative operation of the partitions is not guaranteed to operate synchronously. In their contribution, the normalized throughput was obtained as $\frac{\text{throughput}\times 10^6}{\text{clock freq.}\times\text{mem BW}}$, since the GPGPU's global memory bandwidth impose the main bottleneck upon the corresponding implementation. Similarly, we suggest using the same normalization method for our proposed FPTD, since its performance is also bounded by the global memory bandwidth, according to the experimental results from the Nsight profiler, as discussed in Section IV-A. As shown in Table II, the benchmarker of [38] achieves a normalized throughput of 100.6, when performing $I = 12$ iterations for decoding $N = 6144$-bit frames, divided into $P = 768$ partitions. However, this approach results in an E_b/N_0 degradation of 0.2 dB, compared to that of the conventional Log-BCJR turbo decoding algorithm employing $P = 64$ partitions and performing $I = 6$ iterations. When tolerating this 0.2 dB degradation, our proposed SIMD FPTD algorithm requires only $I = 27$ iterations, rather than $I = 36$, as shown in Table II. In this case, the normalized throughput of our proposed SIMD FPTD algorithm is 128.8, which is 28% higher than that of the loosely synchronized parallel turbo decoding algorithm of [38]. Furthermore, our approach has the advantage of being able to maintain a constant BER performance, while the loosely synchronized parallel turbo decoding algorithm of [38] suffers from a BER performance that varies from frame to frame, owing to its asynchronous decoding process.

By contrast, using the normalization of $\frac{\text{throughput}\times 10^6}{\text{clock freq.}\times\text{mem freq.}}$ is more appropriate for the other implementations listed in Table II, since according to our experimental results, the computational latency and the global memory access latency constitute the main

TABLE II: Comparison between the proposed GPGPU implementation of our SIMD FPTD algorithm with other GPGPU implementations of the turbo decoders found in the literature.

Implementations	This work		[18]	[17]	[43]	[13]	[38]
GPU	GTX 680		GeForce 9800 GX2	Tesla K20c	GTX Titan	Tesla C1060	GTX 480
Clock freq. [MHz]	1006		1500	706	836	1296	1401
Memory freq. [MHz]	1502		1000	1300	1502	800	924
Memory bandwidth [GB/s]	192.2		128	208	288	102	177.4
Algorithm	SIMD FPTD	Log-BCJR	Log-BCJR	Log-BCJR	Log-BCJR	Log-BCJR	Parallel Log-BCJR
Windowing mechanism	odd-even	PIVI	PIVI	PIVI	PIVI	PIVI	loose synchronization
Window size W	1	32	32	32	32	32	8
Iteration I	36 (27[1])	7	7	7	7	7	12[1]
Throughput [Mbps]	18.7 (24.9[1])	8.2	1.9	4	3.2	6.8	25[1]
Normalized throughput[2]	**96.7** (**128.8**[1])	42.4	9.9	27.2	13.3	51.4	**100.6**[1]
Normalized throughput[3]	12.4 (16.5[1])	**5.4**	**1.3**	**3.0**	**4.4**	**6.6**	19.3[1]

[1] Subject to a cost of 0.2 dB degradation in frame error rate performance.
[2] The throughput is normalized by $\frac{\text{throughput}\times 10^6}{\text{clock freq.}\times\text{mem BW}}$, which is appropriate for applications where the performance is bounded by the global memory bandwidth, as used in [38].
[3] The throughput is normalized by $\frac{\text{throughput}\times 10^6}{\text{clock freq.}\times\text{mem freq.}}$, which is appropriate for applications where the performance is bounded by the compute latency and the memory latency.

bottlenecks of these implementations of the Log-BCJR algorithm. This may be attributed to the low degree of parallelism of the decoder compared to the capability of the GPGPU, particularly when only a single frame is decoded at a time. Note that the normalized throughputs obtained using the different normalization methods are not comparable to each other. As shown in Table II, the normalized throughput of 5.4 achieved by our PIVI Log-BCJR benchmarker is significantly better than those of [18], [17] and [43]. Although the benchmarker of [13] achieves a better normalized throughput of 6.6, this is achieved by decoding a batch of 100 frames at a time, which can readily achieve a higher degree of parallelism than decoding only a single frame at a time, like all of the other schemes, as discussed in [18]. Owing to this, the computing latency and memory latency maybe no longer a limiter for the throughput performance, implying that the normalized throughput for [13] may be inappropriate. Additionally, this throughput can only be achieved, when there are 100 frames available for simultaneous decoding, which may not occur frequently in practice, hence resulting in an unfair comparison with the other benchmarkers. Furthermore, while decoding several frames in parallel improves the overall processing throughput, it does not improve the processing latency of each frame.

E. Complexity

The complexity of the proposed GPGPU implementation of our SIMD FPTD algorithm may be compared with that of the PIVI Log-BCJR turbo decoder by considering the number of GPGPU instructions that are issued per bit of the message frame. This is motivated, since all GPGPU thread operations are commanded by instructions. More specifically, while performing one half iteration and one interleaving operation for each turbo decoder, the average number Instructions Per Warp (IPW) was measured using the NVIDIA analysis tool, Nsight [52]. Using this, the average number of Instructions Per Bit (IPB) may be obtained as $\text{IPB} = 2I \cdot \text{IPBPHI} = \frac{2I \cdot \text{IPW} \cdot D_p}{32N}$, where IPBPHI is the average number of Instructions Per Bit Per Half Iteration,

N is the frame length and D_p is the corresponding degree of parallelism. Here, $\frac{D_p}{32}$ represents the total number of warps, since each warp includes 32 of the D_p threads employed by the decoder.

Table I quantifies IPBPHI and IPB for both the proposed SIMD FPTD and the PIVI Log-BCJR turbo decoder, when employing both the approximate and exact max* operations of (6) and (5), respectively. As shown in Table I, the IPBPHI of the proposed SIMD FPTD is around one third that of the PIVI Log-BCJR turbo decoder, when employing the approximate max* operation, although this ratio grows to one half, when employing the exact max* operation. Note however that the proposed SIMD FPTD algorithm requires more decoding iterations than the PIVI Log-BCJR turbo decoder for achieving a particular BER performance, as quantified in Section V-A. Therefore, the overall IPB complexity of the proposed SIMD FPTD is 1.7 to 3.3 times higher than that of the PIVI Log-BCJR turbo decoder, depending on the number of iterations I, window length W and type of max* operation performed, as shown in Table I. Note that this trend broadly agrees with that of our previous work [22], which showed that the FPTD algorithm has a complexity that is 2.9 times higher than that of the state-of-the-art LTE turbo decoder employing the Log-BCJR algorithm, which was obtained by comparing the number of additions/subtractions and max* operations employed by the different algorithms. Note that the increased complexity of the FPTD represents the price that must be paid for increasing the decoding throughput by a factor up to 9.1.

VI. Conclusions

In this paper, we have proposed a SIMD FPTD algorithm and demonstrated its implementation on a GPGPU. We have also characterized its performance in terms of BER performance, degree of parallelism, GPGPU processing throughput and complexity. Furthermore, these characteristics have been compared with those of the state-of-the-art PIVI Log-BCJR turbo decoder. This comparison shows that owing to its increased degree of parallelism, the proposed SIMD FPTD offers a processing throughput that is between 2.3 and 9.2 times higher and a processing latency that is between 2 and 8.2 times better than that of the benchmarker. However, this is achieved at the cost of requiring a greater number of iterations than the benchmarker in order to achieve a particular BER performance, which may result in a 1.7 to 3.3 times increase in overall complexity. In our future work we will conceive techniques for disabling particular algorithmic blocks in the FPTD, once they have confidently decoded their corresponding bits. With this approach, we expect to significantly reduce the complexity of the FPTD, such that it approaches that of the Log-BCJR turbo decoder, without compromising the BER performance.

References

[1] J. Woodard and L. Hanzo, "Comparative Study Of Turbo Decoding Techniques: An Overview," *IEEE Transactions on Vehicular Technology*, vol. 49, pp. 2208–2233, Nov 2000.

[2] S. Chaabouni, N. Sellami, and M. Siala, "Mapping Optimization for a MAP Turbo Detector Over a Frequency-Selective Channel," *IEEE Transactions on Vehicular Technology*, vol. 63, pp. 617–627, Feb 2014.

[3] M. Brejza, L. Li, R. Maunder, B. Al-Hashimi, C. Berrou, and L. Hanzo, "20 Years of Turbo Coding and Energy-Aware Design Guidelines for Energy-Constrained Wireless Applications," *IEEE Communications Surveys & Tutorials*, vol. PP, pp. 1–1, Jun 2015.

[4] IEEE, "IEEE Standard for Local and Metropolitan Area Networks Part 16: Air Interface for Broadband Wireless Access Systems," 2012.

[5] ETSI, "LTE; Evolved Universal Terrestrial Radio Access (E-UTRA); Multiplexing and channel coding," Feb 2013.

[6] A. A. Abidi, "The Path to the Software-Defined Radio Receiver," *IEEE Journal of Solid-State Circuits*, vol. 42, pp. 954–966, May 2007.

[7] P. Demestichas, A. Georgakopoulos, D. Karvounas, K. Tsagkaris, V. Stavroulaki, J. Lu, C. Xiong, and J. Yao, "5G on the Horizon: Key Challenges for the Radio-Access Network," *IEEE Vehicular Technology Magazine*, vol. 8, pp. 47–53, Sep 2013.

[8] D. Wubben, P. Rost, J. S. Bartelt, M. Lalam, V. Savin, M. Gorgoglione, A. Dekorsy, and G. Fettweis, "Benefits and Impact of Cloud Computing on 5G Signal Processing: Flexible centralization through cloud-RAN," *IEEE Signal Processing Magazine*, vol. 31, pp. 35–44, Nov 2014.

[9] W. Chen, Y. Chang, S.g Lin, L. Ding, and L. Chen, "Efficient Depth Image Based Rendering with Edge Dependent Depth Filter and Interpolation," in *2005 IEEE International Conference on Multimedia and Expo*, (Amsterdam), pp. 1314–1317, IEEE, Jul 2005.

[10] R. Li, Y. Dou, J. Zhou, L. Deng, and S. Wang, "CuSora: Real-time Software Radio using Multi-core Graphics Processing Unit," *Journal of Systems Architecture*, vol. 60, pp. 280–292, Mar 2014.

[11] S. Roger, C. Ramiro, A. Gonzalez, V. Almenar, and A. M. Vidal, "Fully Parallel GPU Implementation of a Fixed-Complexity Soft-Output MIMO Detector," *IEEE Transactions on Vehicular Technology*, vol. 61, pp. 3796–3800, Oct 2012.

[12] Q. Zheng, Y. Chen, H. Lee, R. Dreslinski, C. Chakrabarti, A. Anastasopoulos, S. Mahlke, and T. Mudge, "Using Graphics Processing Units in an LTE Base Station," *Journal of Signal Processing Systems*, vol. 78, pp. 35–47, Jan 2015.

[13] M. Wu, Y. Sun, and J. R. Cavallaro, "Implementation of a 3GPP LTE turbo decoder accelerator on GPU," in *IEEE Workshop on Signal Processing Systems, SiPS: Design and Implementation*, (San Francisco, CA), pp. 192–197, IEEE, Oct 2010.

[14] D. Lee, M. Wolf, and H. Kim, "Design Space Exploration of the Turbo Decoding Algorithm on GPUs," in *Proceedings of the international conference on Compilers, architectures and synthesis for embedded systems*, CASES '10, (Scottsdale, AZ, USA), pp. 217–226, ACM Press, Oct 2010.

[15] L. Bahl, J. Cocke, F. Jelinek, and J. Raviv, "Optimal Decoding of Linear Codes for Minimizing Symbol Error Rate (Corresp.)," *IEEE Transactions on Information Theory*, vol. 20, pp. 284–287, Mar 1974.

[16] P. Robertson, E. Villebrun, and P. Hoeher, "A Comparison of Optimal and Sub-optimal MAP Decoding Algorithms Operating in the Log Domain," in *Proceedings IEEE International Conference on Communications ICC '95*, vol. 2, (Seattle, WA, USA), pp. 1009–1013, IEEE, Jun 1995.

[17] Y. Zhang, Z. Xing, L. Yuan, C. Liu, and Q. Wang, "The Acceleration of Turbo Decoder on the Newest GPGPU of Kepler Architecture," in *14th International Symposium on Communications and Information Technologies (ISCIT)*, (Incheon), pp. 199–203, IEEE, Sep 2014.

[18] D. R. N. Yoge and N. Chandrachoodan, "GPU Implementation of a Programmable Turbo Decoder for Software Defined Radio Applications," in *2012 25th International Conference on VLSI Design*, (Hyderabad), pp. 149–154, IEEE, Jan 2012.

[19] A. Nimbalker, T. Blankenship, B. Classon, T. Fuja, and D. Costello, "Contention-Free Interleavers for High-Throughput Turbo Decoding," *IEEE Transactions on Communications*, vol. 56, pp. 1258–1267, Aug 2008.

[20] NVIDIA, *CUDA C PROGRAMMING GUIDE*, v6.5 ed., Mar 2015.

[21] M. Wu, Y. Sun, G. Wang, and J. R. Cavallaro, "Implementation of a High Throughput 3GPP Turbo Decoder on GPU," *Journal of Signal Processing Systems*, vol. 65, pp. 171–183, Nov 2011.

[22] R. G. Maunder, "A Fully-Parallel Turbo Decoding Algorithm," *IEEE Transactions on Communications*, vol. 63, pp. 2762–2775, Aug 2015.

[23] L. Sousa, S. Momcilovic, V. Silva, and G. Falcao, "Multi-core Platforms for Signal Processing: Source and Channel Coding," in *2009 IEEE International Conference on Multimedia and Expo*, (New York, NY), pp. 1809–1812, IEEE, Jun 2009.

[24] E. Hermann, B. Raffin, F. Faure, T. Gautier, and J. Allard, *Multi-GPU and Multi-CPU Parallelization for Interactive Physics Simulations*, vol. 6272 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2010.

[25] S. Cook, *CUDA Programming: A Developer's Guide to Parallel Computing with GPUs*. Applications of GPU Computing Series, Elsevier Science, 2012.

[26] H. Berg, C. Brunelli, and U. Lucking, "Analyzing Models of Computation for Software Defined Radio Applications," in *International Symposium on System-on-Chip*, (Tampere), pp. 1–4, IEEE, Nov 2008.

[27] 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*, (New York, New York, USA), p. 83, ACM Press, 2008.

[28] S. Wang, S. Cheng, and Q. Wu, "A Parallel Decoding Algorithm of LDPC codes using CUDA," in *2008 42nd Asilomar Conference on Signals, Systems and Computers*, pp. 171–175, IEEE, Oct 2008.

[29] G. Falcão, S. Yamagiwa, V. Silva, and L. Sousa, "Parallel LDPC Decoding on GPUs Using a Stream-Based Computing Approach," *Journal of Computer Science and Technology*, vol. 24, pp. 913–924, Sep 2009.

[30] G. Falcão, V. Silva, and L. Sousa, "How GPUs Can Outperform ASICs for Fast LDPC Decoding," in *Proceedings of the 23rd international conference on Conference on Supercomputing - ICS '09*, (New York, New York, USA), p. 390, ACM Press, 2009.

[31] J. Kim, S. Hyeon, and S. Choi, "Implementation of an SDR System using Graphics Processing Unit," *IEEE Communications Magazine*, vol. 48, pp. 156–162, Mar 2010.

[32] 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 x 2 MIMO WiMAX System," *Analog Integrated Circuits and Signal Processing*, vol. 69, pp. 107–117, Dec 2011.

[33] P.-H. Horrein, C. Hennebert, and F. Pétrot, "Integration of GPU Computing in a Software Radio Environment," *Journal of Signal Processing Systems*, vol. 69, pp. 55–65, Oct 2012.

[34] S. Grönroos, K. Nybom, and J. Björkqvist, "Complexity Analysis of Software Defined DVB-T2 Physical Layer," *Analog Integrated Circuits and Signal Processing*, vol. 69, pp. 131–142, Dec 2011.

[35] M. Wu, Y. Sun, S. Gupta, and J. R. Cavallaro, "Implementation of a High Throughput Soft MIMO Detector on GPU," *Journal of Signal Processing Systems*, vol. 64, pp. 123–136, Jul 2011.

[36] G. Falcao, J. Andrade, V. Silva, and L. Sousa, "Real-time DVB-S2 LDPC Decoding on Many-core GPU Accelerators," in *2011 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 1685–1688, IEEE, May 2011.

[37] M. Dardaillon, K. Marquet, T. Risset, and A. Scherrer, "Software Defined Radio Architecture Survey for Cognitive Testbeds," in *2012 8th International Wireless Communications and Mobile Computing Conference (IWCMC)*, (Limassol), pp. 189–194, IEEE, Aug 2012.

[38] X. Jiao, C. Chen, P. Jaaskelainen, V. Guzma, and H. Berg, "A 122Mb/s Turbo Decoder using a Mid-range GPU," in *2013 9th International Wireless Communications and Mobile Computing Conference (IWCMC)*, (Sardinia), pp. 1090–1094, IEEE, Jul 2013.

[39] M. Wu, G. Wang, B. Yin, C. Studer, and J. R. Cavallaro, "HSPA;/LTE-A Turbo Decoder on GPU and Multicore CPU," in *2013 Asilomar Conference on Signals, Systems and Computers*, no. i, (Pacific Grove, CA), pp. 824–828, IEEE, Nov 2013.

[40] G. Wang, M. Wu, B. Yin, and J. R. Cavallaro, "High Throughput Low Latency LDPC Decoding on GPU for SDR Systems," in *2013 IEEE Global Conference on Signal and Information Processing*, no. 3, pp. 1258–1261, IEEE, Dec 2013.

[41] V. Adhinarayanan, T. Koehn, K. Kepa, W.-c. Feng, and P. Athanas, "On the Performance and Energy Efficiency of FPGAs and GPUs for Polyphase Channelization," in *International Conference on ReConFigurable Computing and FPGAs (ReConFig14)*, (Cancun), pp. 1–7, IEEE, Dec 2014.

[42] S. Bang, C. Ahn, Y. Jin, S. Choi, J. Glossner, and S. Ahn, "Implementation of LTE System on an SDR Platform using CUDA and UHD," *Analog Integrated Circuits and Signal Processing*, vol. 78, pp. 599–610, Mar 2014.

[43] H. Ahn, Y. Jin, S. Han, S. Choi, and S. Ahn, "Design and Implementation of GPU-based Turbo Decoder with a Minimal Latency," in *The 18th IEEE International Symposium on Consumer Electronics (ISCE 2014)*, (JeJu Island), pp. 1–2, IEEE, Jun 2014.

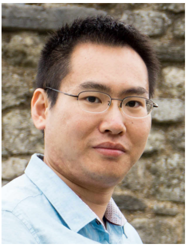
[44] J. a. Briffa, "A GPU Implementation of a MAP Decoder for Synchronization Error Correcting Codes," *IEEE Communications Letters*, vol. 17, pp. 996–999, May 2013.

[45] J. a. Briffa, "Graphics Processing Unit Implementation and Optimisation of a Flexible Maximum A-posteriori Decoder for Synchronisation Correction," *The Journal of Engineering*, pp. 1–13, Jan 2014.

[46] Z. Yue and F. C. M. Lau, "Implementation of Decoders for LDPC Block Codes and LDPC Convolutional Codes Based on GPUs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, pp. 663–672, Mar 2014.

[47] J.-H. Hong and K.-s. Chung, "Parallel LDPC Decoding on a GPU using OpenCL and Global memory for Accelerators," in *2015 IEEE International Conference on Networking, Architecture and Storage (NAS)*, pp. 353–354, IEEE, Aug 2015.

[48] F. Ge, H. Lin, A. Khajeh, C. J. Chiang, M. E. Ahmed, W. B. Charles, W.-c. Feng, and R. Chadha, "Cognitive Radio Rides on the Cloud," in *Military Communications Conference*, (San Jose, CA), pp. 1448–1453, IEEE, Oct 2010.
[49] C. Berrou and A. Glavieux, "Near Optimum Error Correcting Coding and Decoding: Turbo-codes," *IEEE Transactions on Communications*, vol. 44, pp. 1261–1271, Oct 1996.
[50] J. Erfanian, S. Pasupathy, and G. Gulak, "Reduced Complexity Symbol Detectors with Parallel Structure for ISI Channels," *IEEE Transactions on Communications*, vol. 42, pp. 1661–1671, Feb 1994.
[51] S. Xiao and W.-c. Feng, "Inter-block GPU Communication via Fast Barrier Synchronization," in *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, pp. 1–12, IEEE, Apr 2010.
[52] NVIDIA, *NVIDIA Nsight Visual Studio Edition 4.6 User Guide*, 4.6 ed., 2015.
[53] ETSI, "ETSI TS 1 136 213," *Etsi*, vol. V12.7.0, 2015.
[54] MOTOROLA, "Real-World Lte Performance for Public Safety," White paper, Sep, 2010.



An Li received his first class honors BEng degree in Electronic Engineering from the University of Southampton in 2011 and his MPhil degree from the University of Cambridge in 2012. He is currently a PhD student in Wireless Communication research group in the University of Southampton. His research interests include parallel turbo decoding algorithms and their implementations upon VLSI, FPGA and GPGPU.

Robert G. Maunder has studied with Electronics and Computer Science, University of Southampton, UK, since October 2000. He was awarded a first class honors BEng in Electronic Engineering in July 2003, as well as a PhD in Wireless Communications in December 2007. He became a lecturer in 2007 and an Associated Professor in 2013. Rob's research interests include joint source/channel coding, iterative decoding, irregular coding and modulation techniques. For further information on this research, please refer to http://users.ecs.soton.ac.uk/rm.

Bashir M. Al-Hashimi (M99-SM01-F09) is a Professor of Computer Engineering and Dean of the Faculty of Physical Sciences and Engineering at University of Southampton, UK. He is ARM Professor of Computer Engineering and Co-Director of the ARM-ECS research centre. His research interests include methods, algorithms and design automation tools for energy efficient of embedded computing systems. He has published over 300 technical papers, authored or co-authored 5 books and has graduated 31 PhD students.

Lajos Hanzo (FREng, FIEEE, FIET, Eurasip Fellow, RS Wolfson Fellow, www-mobile.ecs.soton.ac.uk) holds the Chair of Telecommunications at Southampton University, UK. He co-authored 1500+ IEEE Xplore entries, 20 IEEE Press & Wiley books, graduated 100+ PhD students and has an H-index of 59.