Parallel Trellis-Stage-Combining BCJR for High-Throughput CUDA Decoder of CCSDS SCPPM

Amaael Antonini
University of California
420 Westwood Plaza
Los Angeles, CA 90095
amaael@g.ucla.edu

Egor Glukhov
University of California
420 Westwood Plaza
Los Angeles, CA 90095
e.glu@ucla.edu

Richard Wesel
University of California
420 Westwood Plaza
Los Angeles, CA 90095
wesel@g.ucla.edu

Dariush Divsalar
Jet Propulsion Laboratory
California Institute of Technology
4800 Oak Grove Dr.
Pasadena, CA 91109
dariush.divsalar@jpl.nasa.gov

Jon Hamkins
Jet Propulsion Laboratory
California Institute of Technology
4800 Oak Grove Dr.
Pasadena, CA 91109
jon.hamkins@jpl.nasa.gov

Abstract—**The Consultative Committee for Space Data Systems (CCSDS) standard for high photon efficiency uses a serially-concatenated (SC) code to encode pulse position modulated laser light. A convolutional encoder serves as the outer code and an accumulator serves as the inner code. These two component codes are connected through an interleaver. This coding scheme is called Serially Concatenated convolutionally coded Pulse Position Modulation (SCPPM) and it is used for NASA's Deep Space Optical Communications (DSOC) experiment. For traditional decoding that traverses the trellis forwards and backwards according to the Bahl Cocke Jelinek and Raviv (BCJR) algorithm, the latency is on the order of the length of the trellis, which has 10,080 stages for the rate 2/3 DSOC code. This paper presents a novel alternative approach that simultaneously processes all trellis stages, successively combining pairs of stages into a meta-stage. This approach has latency that is on the order of the log base-2 of the number of stages. The new decoder is implemented using the Compute Unified Device Architecture (CUDA) platform on an Nvidia Graphics Processing Unit (GPU). Compared to Field Programmable Gate Array (FPGA) implementations, the GPU implementation offers easier development, scalability, and portability across GPU models. The GPU implementation provides a dramatic increase in speed that facilitates more thorough simulation as well as enables a shift from FPGA to GPU processors for DSOC ground stations.**

Table of Contents

1. Introduction

The Consultative Committee for Space Data Systems (CCSDS) standard uses a Serially Concatenated convolutionally coded Pulse Position Modulation (SCPPM) that features a convolutional code with only four states but 7560 trellis stages for rate-1/2. Traditional decoders work sequentially through these many stages, constraining throughput. This paper presents an alternative method that simultaneously processes all trellis stages allowing a dramatic increase in throughput and lowering the latency of decoding. Parallel trellis processing can be realized on both Graphics Processing Unit (GPU) and Field Programmable Gate Array (FPGA) implementations. GPU implementation can provide multiple benefits: easier development, scalability, and portability across GPU models. With the rapid advancement of technology, the same implementation remains functional on future GPU devices and compiler versions.

Background

Convolutional codes are typically decoded via the Viterbi algorithm [1], [2] that minimizes the frame error rate (FER), or the algorithm introduced by Bahl, Cocke, Jelinek, and Raviv [3] (BCJR), that minimizes the symbol error rate. Low delay requirements have been a limiting factor for Viterbi decoders. Several works have achieved a level of parallelism to address this problem. Fettweis and Meyr [4] combined every P stages of an K-stage trellis using a smaller trellis per each of the M states, and achieved a linear speedup of P. Mohammadidoost and Hashemi [5] optimized the memory access for both the surviving path and the trace-back path on a GPU. Peng *et al.* [6] introduce a method to achieve a high throughput also using GPU. Their method is flexible enough for application on general block codes and is shown to attain a throughput of up to 1.8 Gb/s on convolutional codes. Zhihui *et al.* [7] and Hanif *et al.* [8] show that part of the computations can be done independently. In [7] a sequence is split into smaller segments and the processing of each segment is further split into an independent part and a dependent part. A matrix representation of the process is used in [8] that allows a GPU to exploit the independence of some operations. Seshadri and Sundberg [9] compare convolutional codes with Low Density Parity Check (LDPC) codes under delay constraints and show that convolutional codes outperform LDPC in the more restrictive regime. An FPGA version of a Viterbi decoder is also provided by Ben Asher *et al.* in [10], with parallelism that can be tuned via a parameter P.

Concatenated convolutional codes like the SCPPM code are often decoder iteratively, see the parallel concatenated Turbo codes by Berrou *et al.* [11] and serially concatenated codes by Benedetto *et al.* [12]. In iterative decoding, two or more decoders alternate to produce a-posteriori probabilities that are used as input to the other decoders in the next

979-8-3503-5597-0/25/$31.00 ©2025 IEEE

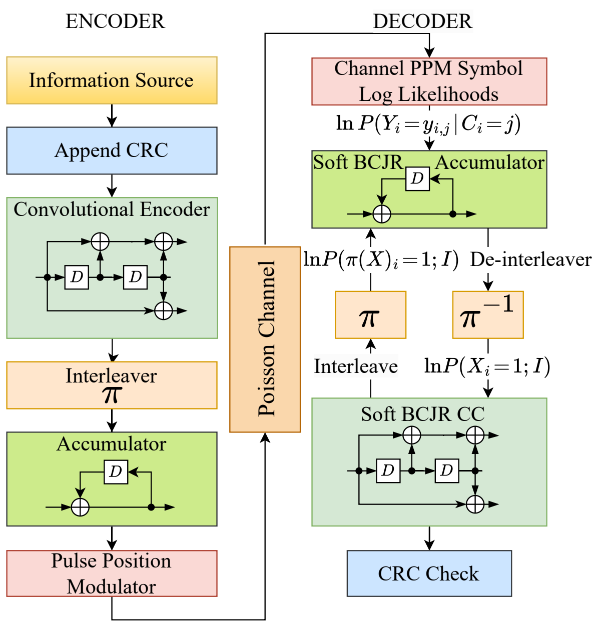


Figure 1: **SCPPM Encoder model. The information generates a bit sequence to which a 32-bit error detecting CRC sequence and two zeros are appended, to obtain a K-bit, e.g. $K = 7560$ when the rate is $1/2$. The K-bit sequence is encoded with an outer convolutional code and the output is interleaved with a quadratic interleaver. The interleaved sequence is encoded with a two-state recursive convolutional code or accumulator, and the output sequence is modulated with pulse-position modulation.**

iteration. The process repeats until a stopping condition is met. The SCPPM code stops when no error is detected by an error detecting sequence or when the maximum number of iterations is exceeded. The serially concatenated (SC) pulse-position modulation (PPM) code consists of two serially concatenated convolutional codes with an interleaver between the two codes and is depicted in Fig. 1.

A convolutional code is a linear block code that encodes an information sequence into another coded sequence of symbols. The input sequence could be a sequence of bits, and the outputs could be bits or words comprised of two or more bits each. A convolutional code (CC) consists of a set of states $\{0, \ldots, M-1\}$ and two sets of linear functions $\{f_0, f_1, \ldots, f_{M-1}\}$, $\{h_0, h_1, \ldots, h_{M-1}\}$, where f_i maps an input symbol to one or more output symbols and h_i defines a state transition that maps input symbol from source state i to a destination state j. The ratio of k input symbols to n output symbols per encoding operation defines the code rate $\frac{k}{n}$.

The model of the SCPPM encoder is depicted in the left side of Fig. 1. At the beginning, an error detecting 32-bit cyclic redundancy check (CRC) code sequence is appended to the information sequence. Then, the information sequence, with the 32-bit CRC sequence, is encoded with a four-state, non-recursive, zero-terminated convolutional code, which will be called the "outer code." The output of the outer code is interleaved with a quadratic interleaver, and the interleaved sequence is encoded with a two state recursive convolutional

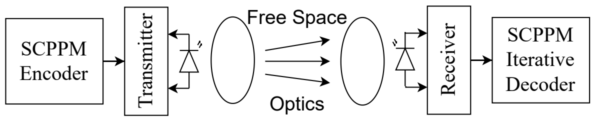


Figure 2: **Illustration of the free-space optical channel. The transmitter emits a pulse of light at each symbol's pulsed slot, and the receiver counts the number of photos received at each slot. The input to the decoder are the photon counts for every slot of every symbol.**

code, which will be called the inner code or accumulator. The output sequence of the inner code is split into symbols that are between two and eight bits long, and each symbol is modulated with a pulse position modulation.

Contribution

Serially concatenated convolutional codes are traditionally decoded by message-passing decoders, with a separate BCJR decoder implemented for each component convolutional code. Each component BCJR decoder takes as input the output of the other component BCJR decoders and the channel symbol probabilities, according to the encoding graph. Each decoder computes new a-posteriori probabilities that are passed along to the other component BCJR decoders. The BJCR decoders traditionally compute the reliabilities by working sequentially through the trellis first in the forward direction and then in the backward direction [3]. This paper presents a method to break the BCJR algorithm [3] dependency on the values at each previous stage and instead simultaneously compute many stages to increase throughput and lower the latency of decoding.

Organization

The rest of the paper proceeds as follows: Sec. 2 describes the SCPPM code and a version of the BCJR algorithm by Bahl *et al.* [3] used by the standard SCPPM decoder. Sec. 3 introduces our method to transform the BCJR algorithm to allow higher parallelism and lower the decoding time. Sec. 4 describes the implementation of the algorithms using a GPU and CUDA. Sec. 5 shows simulation results in terms of runtime performance and frame error rate vs. average photon rates, and Sec. 6 concludes the paper.

2. The SCPPM Code

Let the input of a code be a sequence with entries in a finite field $\mathcal{U}$ and outputs from a finite field $\mathcal{F}$, which could be the as $\mathcal{U}$. A K-symbol input sequence $\mathbf{U} = U_0, U_1, \ldots, U_{K-1}$ is mapped to an N output sequence $\mathbf{X}$ where $N = K\frac{n}{k}$. In this work, the value of k will be $k = 1$ and the output symbols are n-symbol words. Given an initial state S_0, a convolutional code maps a K-symbol input sequence $\mathbf{U}$ to a unique output sequence $\mathbf{X}$ and a unique sequence of state $S_1, S_2, \ldots, S_K$, via the output and state transition functions. If the code is terminated, additional symbols are appended to the input symbol sequence $\mathbf{U}$, to ensure that the last state S_{end}, generally same as S_0, is reached. In the next two sections, we will call the input to the inner code a K-bit sequence $\mathbf{U}$ and the output a N-bit sequence $\mathbf{X}$, which is mapped to $M-$ary PPM symbols $C_1, C_2, \ldots$ with $M \in \{4, 8, 32, 64, 128, 256\}$ each encoding m bits, where $m \in \{2, 3, \ldots, 8\}$. The encoder's modulated output is a vector of symbols $\mathbf{C}_1, \ldots, \mathbf{C}_{N/m}$, where symbol $\mathbf{C}_i$ is a

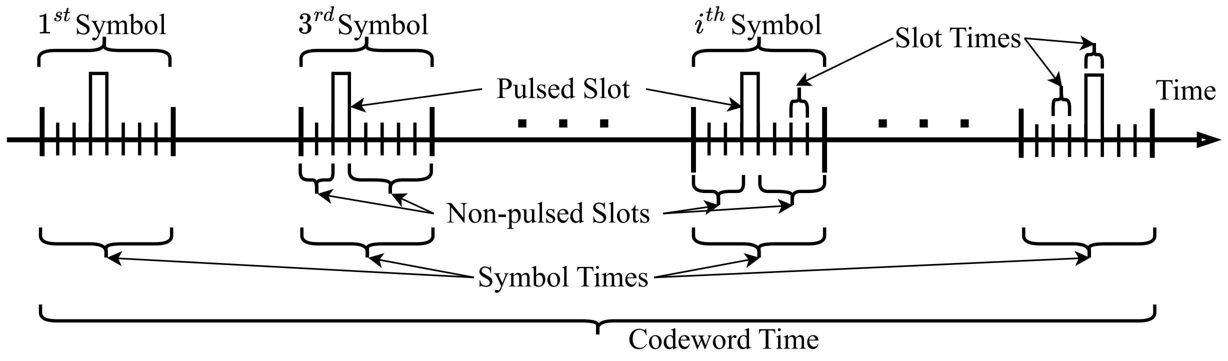


Figure 3: **Illustration of pulse position modulation (PPM) of a codeword with 8-PPM slots per symbol. Each symbol consists of 3-bits, that represent a position in $0, 1, \ldots, 7$. The 1^{st} modulated symbol is $011 = 3$, the 3^{rd} modulated symbol is $010 = 2$, the i-th modulated symbol is $100 = 4$, and the last symbol is $101 = 5$. A pulse is transmitted for each symbol, at positions $3, 2, 4, 5$ for the 1^{st}, 3^{rd}, i^{th} and last symbols respectively. No pulse is sent at any of the other slot positions on each symbol.**

vector $\mathbf{C}_i = \{C_{i,1}, C_{i,2}, \ldots, C_{i,2^m}\}$, where $C_{i,j} \in \{0, 1\}$ and $\mathbf{C}_i$ has a single 1 entry. For convenience, each m-bits symbol constructed from the output of the inner code will be called an edge E, where $E \in \{0, 1, \ldots, 2^m - 1\}$ and the input to the inner code will also be grouped into symbols $U_i \in \{0, 1, \ldots, 2^k - 1\}$. Since the inner code is rate 1, the value of k for the inner code is will be $k = m$.

Channel Model

The channel model is a Poisson channel defined by a background noise rate of K_b average photons per non-pulsed PPM slot and $K_s + K_b$ average photons per pulsed PPM slot.

The received signal for each transmitted symbol i is vector $Y_{i,j}$ and for each PPM slot $j \in \{0, \ldots, 2^m - 1\}$ $Y_{i,j}$ is modeled by the probability mass functions (p.m.f.) $\Pr(Y_{i,j} = y \mid C_{i,j} = 0)$ and $\Pr(Y_{i,j} = y \mid C_{i,j} = 1)$ given by:

$$\Pr(Y_{i,j} = y \mid C_{i,j} = 0) = \frac{K_b^y e^{-K_b}}{y!} \tag{1}$$

$$\Pr(Y_{i,j} = y \mid C_{i,j} = 1) = \frac{(K_b + K_s)^y e^{-(K_b+K_s)}}{y!}\,. \tag{2}$$

From now on, denote by $p_{i,j}(y) \triangleq \Pr(Y_{i,j} = y \mid C_{i,j} = 1)$ the channel's p.m.f. in Eq. (2).

The BCJR Algorithm

The BCJR algorithm of Bahl *et al.* [3] is a standard component of the decoder of concatenated codes because it produces a-posteriori probabilities of its input and output sequences. The inputs to the BCJR algorithm are the a-priori probabilities of the input and output sequences, where one of the two might be missing. The channel sequence symbol probabilities in (1) and (2) provide the output symbol probabilities for the inner-most decoder. In serially concatenated codes, the information sequence symbol a-posteriori probabilities are the output of only the outer-most decoder, thus the outer-most decoder does not get updated a-priori probabilities of the information sequence from other decoders. The outputs of the BCJR algorithm are the vector of a-posteriori probabilities of the input sequence $U_1, U_2, \ldots, U_K$ given by $\Pr(U_i = u \mid \mathbf{Y} = \mathbf{y})$ for each $u \in \mathcal{U}$ and the vector of a-posteriori probabilities of the output sequence $X_1, X_2, \ldots, X_{nK}$ given by $\Pr(X_i = x \mid \mathbf{Y} = \mathbf{y})$ for each $x \in \mathcal{F}$. Given code functions f_i and the channel p.m.f., we can directly compute the probabilities $\Pr(\mathbf{Y}_i = \mathbf{y}_i \mid C_{i,j} = 1)$ via:

$$\Pr(\mathbf{Y}_i = \mathbf{y} \mid C_{i,j} = 1) \tag{3}$$

$$= \Pr(Y_{i,j} = y_{i,j} \mid C_{i,j} = 1) \prod_{l \neq j} \Pr(Y_{i,j} = y_{i,l} \mid C_{i,l} = 0) \tag{4}$$

$$= \frac{\Pr(Y_{i,j} = y_j \mid C_{i,j} = 1)}{\Pr(Y_{i,j} = y_{i,j} \mid C_{i,j} = 0)} \prod_{l} \Pr(Y_{i,j} = y_{i,l} \mid C_{i,l} = 0) \tag{5}$$

For the rest of this section we explain how to compute the output probabilities $\Pr(U_i = u \mid \mathbf{Y} = \mathbf{y})$ from the channel probabilities $\Pr(\mathbf{Y}_i = \mathbf{y} \mid C_{i,j} = 1)$ and a-priori probabilities $\Pr(U_i = u; I)$ produced by the outer decoder, where I stands for intrinsic input probabilities. Note that for the inner decoder the input sequence $\mathbf{U}$ is the output sequence of the outer decoder. In the next two sections we will refer to the input of the inner decoder by $\mathbf{U} = \mathbf{U}_1, \mathbf{U}_2, \ldots, \mathbf{U}_{N/m}$, with $\mathbf{U_i} = U_{i,1}, U_{i,2}, \ldots, U_{i,m}$ and the output a sequence of edges $\mathbf{E} = E_1, E_2, \ldots, E_{N/m}$. The probabilities $\Pr(E_i = e \mid \mathbf{Y} = \mathbf{y})$ that would be produced by the inner are not used and thus are omitted. These probabilities are straightforward to obtain, see [12]. Next we express the channel probabilities $\Pr(\mathbf{Y}_i = \mathbf{y}_i \mid C_{i,j} = 1)$ as functions of the input and state S_i, that is, as $\Pr(\mathbf{Y}_i = \mathbf{y}_i \mid S_i = s, U_i = f_s^{-1}(j))$ for each symbol $i = 1, 2, \ldots, N/m$, each slot $j \in \{0, 1, \ldots, 2^m - 1\}$ and each state $s \in \{1, 2, \ldots, M\}$ as follows:

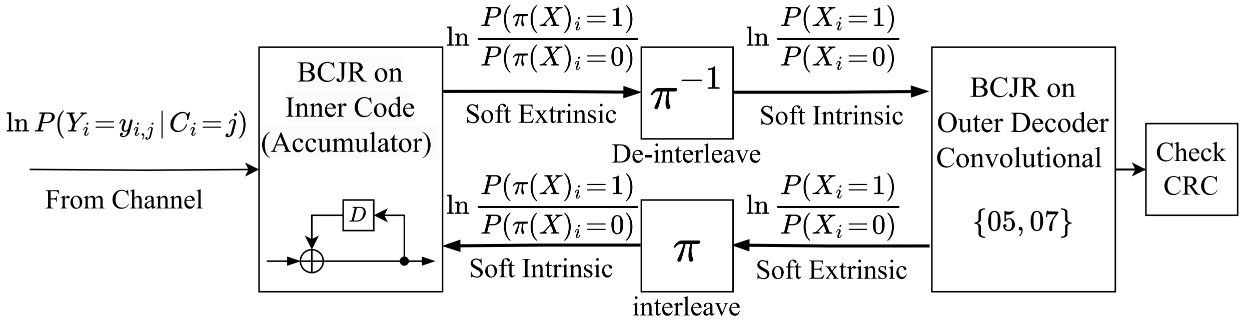
$$p_{i,j}(y) = \Pr(Y_{i,j} = y \mid C_{i,j} = 1) \tag{6}$$

$$= \Pr(Y_{i,j} = y \mid E_i = j) \tag{7}$$

$$= \Pr(Y_{i,j} = y \mid S_i = s, U_i = f_s^{-1}(j)) \tag{8}$$

We are interested in the conditional probabilities: $\Pr(U_i = u \mid \mathbf{Y} = \mathbf{y})$, which may be obtained from the joint probabilities $\Pr(U_i = u, \mathbf{Y} = \mathbf{y})$, $i = 1, 2, \ldots, N$ and the marginal $\Pr(\mathbf{Y} = \mathbf{y})$ by:

$$\Pr(U_i = u \mid \mathbf{Y} = \mathbf{y}) = \frac{\Pr(U_i = u, \mathbf{Y} = \mathbf{y})}{\Pr(\mathbf{Y} = \mathbf{y})}\,. \tag{9}$$



$\ln P(Y_i=y_{i,j}\,|\,C_i=j)$
From Channel
BCJR on Inner Code (Accumulator)
D
$\ln \frac{P(\pi(X)_i=1)}{P(\pi(X)_i=0)}$
Soft Extrinsic
π^{-1}
De-interleave
$\ln \frac{P(X_i=1)}{P(X_i=0)}$
Soft Intrinsic
BCJR on Outer Decoder Convolutional
$\{05,07\}$
Check CRC
$\ln \frac{P(\pi(X)_i=1)}{P(\pi(X)_i=0)}$
Soft Intrinsic
π
interleave
$\ln \frac{P(X_i=1)}{P(X_i=0)}$
Soft Extrinsic

Figure 4: **SCPPM $\log(\cdot)$ domain iterative decoder. The inputs to the inner decoder are the $\log(\cdot)$ of the channel symbol probabilities $\Pr(Y_{i,j} = y_{i,j} \mid C_i = j)$. The inner decoder produces "extrinsic" log-likelihood ratios of its input, which is de-interleaved and used as input by the outer decoder. The outer decoder produces an estimate of the input bits and extrinsic log-likelihood ratios of its output sequence. If the estimate sequence is not a codeword of the CRC code, the extrinsic output of the outer code is interleaved and combined as intrinsic input with the channel probability logs and used as input in next iterations by the inner decoder. This process continues until a codeword of the CRC code is found or the max iterations are exceeded.**

The bottom $\Pr(\mathbf{Y} = \mathbf{y})$ is a constant given $\mathbf{Y}$ that can be ignored in many cases, or it can be computed via $\sum_{u\in\mathcal{U}} \Pr(U_i = u, \mathbf{Y} = \mathbf{y})$. Then, is suffices to compute the probabilities $\Pr(U_i = u, \mathbf{Y} = \mathbf{y})$ as follows.

$$\Pr(U_i = u, \mathbf{Y} = \mathbf{y}) = \sum_s \Pr(\mathbf{Y} = \mathbf{y}, U_i = u, S_i = s) \tag{10}$$
$$= \sum_s \sum_{e:f_s(u)=e} \Pr(\mathbf{Y} = \mathbf{y}, E_i = e, U_i = u, S_i = s)$$

Since every next output is $E_i = f_s(u)$ and every next state is $S_{i+1} = h_{s_i}(u)$, then the event $\{U_i = u, S_i = s\}$ guarantees that $\Pr(\mathbf{Y} = \mathbf{y}, U_i = u, S_i = s, S_{i+1} \neq h_s(u)) = 0$, and:

$$\Pr(\mathbf{Y} = \mathbf{y}, U_i = u, S_{i-1} = s) \tag{11}$$
$$= \Pr(\mathbf{Y} = \mathbf{y}, U_i = u, S_{i-1} = s, S_i = h_s(u))$$
$$+ \Pr(\mathbf{Y} = \mathbf{y}, U_i = u, S_{i-1} = s, S_i \neq h_s(u))$$
$$= \Pr(\mathbf{Y} = \mathbf{y}, U_i = u, S_{i-1} = s, S_i = h_s(u)) \tag{12}$$
$$= \Pr(\mathbf{Y} = \mathbf{y}, E_i = f_s(u), S_{i-1} = s, S_i = h_s(u))\,. \tag{13}$$

The rest of this section focuses on computing the probabilities: $\Pr(\mathbf{Y} = \mathbf{y}, E_i = f_s(u), S_{i-1} = s, S_i = h_s(u))$ for each $i = 1, 2, \ldots, N/m$ and each $E_i = \{0, 1, \ldots, 2^m - 1\}$. Let the beginning state of an edge e be $b(e)$ and the end state be $t(e)$. Also denote by $\mathbf{Y}_i^j = \mathbf{y}_i, Y_{i+1}, \ldots, Y_j$, then:

$$\Pr(\mathbf{Y} = \mathbf{y}, E_i = f_s(u), S_{i-1} = s, S_i = h_s(u)) \tag{14}$$
$$= \Pr(\mathbf{Y} = \mathbf{y}, E_i = e, S_{i-1} = b(e), S_i = t(e))$$
$$= \Pr(E_i = e, \mathbf{Y}_i^N = \mathbf{y}_i^N \mid S_{i-1} = b(e), \mathbf{Y}_1^{i-1} = \mathbf{y}_1^{i-1})$$
$$\cdot \Pr(S_{i-1} = b(e), \mathbf{Y}_1^{i-1} = \mathbf{y}_1^{i-1})\,. \tag{15}$$

Need $\Pr(E_i = e, \mathbf{Y}_i^N = \mathbf{y}_i^N \mid S_{i-1} = b(e), \mathbf{Y}_1^{i-1} = \mathbf{y}_1^{i-1})$ and $\Pr(S_{i-1} = b(e), \mathbf{Y}_1^{i-1} = \mathbf{y}_1^{i-1})$ as functions of the channel signal $\mathbf{Y}$, the channel p.m.f and the a-priori probabilities $P(U_i = u)$, $i = 1, 2, \ldots, N/m$. The first probability $\Pr(E_i = e, \mathbf{Y}_i^N = \mathbf{y}_i^N \mid S_{i-1} = b(e), \mathbf{Y}_1^{i-1} = \mathbf{y}_1^{i-1})$ may be computed by:

$$\Pr(E_i = e, \mathbf{Y}_{i+1}^N = \mathbf{y}_{i+1}^N \mid S_{i-1} = b(e), \mathbf{Y}_1^{i-1} = \mathbf{y}_1^{i-1}) \tag{16}$$
$$= \Pr(E_i = e, \mathbf{Y}_i = \mathbf{y}_i \mid S_{i-1} = b(e), \mathbf{Y}_1^{i-1} = \mathbf{y}_1^{i-1})$$
$$\cdot \Pr(\mathbf{Y}_{i+1}^N = \mathbf{y}_{i+1}^N \mid S_{i-1} = b(e), E_i = e, \mathbf{Y}_1^i = \mathbf{y}_1^i)$$
$$= \Pr(E_i = e, \mathbf{Y}_i = \mathbf{y}_i \mid S_{i-1} = b(e)) \tag{17}$$
$$\cdot \Pr(\mathbf{Y}_{i+1}^N = \mathbf{y}_{i+1}^N \mid S_i = t(e)) \tag{18}$$
$$= \Pr(\mathbf{Y}_i = \mathbf{y}_i \mid E_i = e, S_{i-1} = b(e)) \tag{19}$$
$$\cdot \Pr(E_i = e \mid S_{i-1} = b(e)) \tag{20}$$
$$\cdot \Pr(\mathbf{Y}_{i+1}^N = \mathbf{y}_{i+1}^N \mid S_i = t(e)) \tag{21}$$
$$= \Pr(\mathbf{Y}_i = \mathbf{y}_i \mid E_i = e, S_{i-1} = b(e)) \Pr(U_i = U(e)) \tag{22}$$
$$\Pr(\mathbf{Y}_{i+1}^N = \mathbf{y}_{i+1}^N \mid S_i = t(e)) \tag{23}$$

Since $E_i = e \implies S_{i-1} = b(e)$, then the conditioning $S_{i-1} = b(e)$ in $\Pr(\mathbf{Y}_i = \mathbf{y}_i \,|\, E_i = e, S_{i-1} = b(e))$ is redundant, that is $\Pr(\mathbf{Y}_i = \mathbf{y}_i \,|\, E_i = e, S_{i-1} = b(e)) = \Pr(\mathbf{Y}_i = \mathbf{y}_i \,|\, E_i = e)$. The probabilities $\Pr(\mathbf{Y}_i = \mathbf{y}_i \,|\, E_i = e)$, $i = 1, 2, \ldots, N/m$ are obtained from the channel via equations (1), (2) and (7). Then:

$$\Pr(\mathbf{Y} = \mathbf{y}, E_i = e, S_{i-1} = b(e), S_i = t(e)) \tag{24}$$
$$= \Pr(S_{i-1} = b(e), \mathbf{Y}_1^{i-1} = \mathbf{y}_1^{i-1}) \tag{25}$$
$$\cdot \Pr(\mathbf{Y}_i = \mathbf{y}_i, E_i = e \mid S_{i-1} = b(e)) \tag{26}$$
$$\cdot \Pr(\mathbf{Y}_{i+1}^N = \mathbf{y}_{i+1}^N \mid S_i = t(e)) \tag{27}$$

Computing the probabilities (25), (26) *and* (27)

Let the input symbol U_i consists of m bits, that is: $U_i = b_0, b_2, \ldots, b_{m-1}$, with $b_j \in \{0, 1\}$. Denote each bit of U_i by $b_{i,j}$. We wish to compute the probabilities of each bit: $\Pr(U_{i,j} = 0 \mid \mathbf{Y} = \mathbf{y})$ and $\Pr(U_{i,j} = 1 \mid \mathbf{Y} = \mathbf{y})$. Let $\gamma_i(e), \lambda_{i,j}(s', s, b), \alpha_i(s), \beta_i(s), \sigma_i(s), \delta_{i,j}(s', s)$ be defined

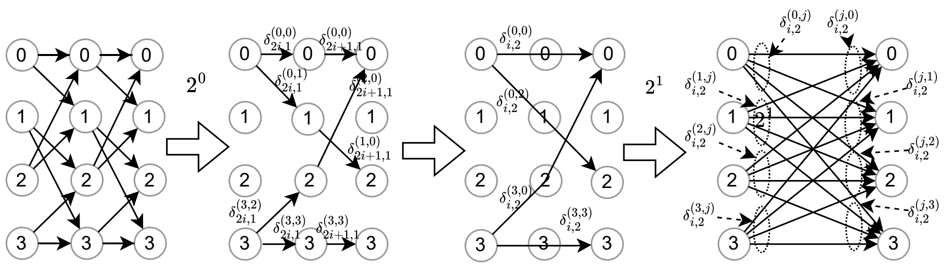


Figure 5: **Trellis stage reduction operation. The building block of the proposed algorithm is a module that takes the edge probabilities of two stages of an original trellis and computes the edges of an equivalent single step trellis.**

by:

$$\gamma_i(e) \triangleq \Pr(E_i = e, \mathbf{Y}_i = \mathbf{y}_i \mid S_{i-1} = b(e)) \quad (28)$$

$$\lambda_{i,j}(s', s, b) \triangleq \Pr(S_i = s, U_{i,j} = b \mid S_{i-1} = s') \quad (29)$$

$$\alpha_i(s) \triangleq \Pr(S_i = s, \mathbf{Y}_1^i = \mathbf{y}_1^i) \quad (30)$$

$$\beta_i(s) \triangleq \Pr(\mathbf{Y}_{i+1}^N = \mathbf{y}_{i+1}^N \mid S_i = s) \quad (31)$$

$$\sigma_i(e) \triangleq \Pr(E_i = e, \mathbf{Y} = \mathbf{y}) \quad (32)$$

$$\sigma_{i,j}(b) \triangleq \Pr(U_{i,j} = b, \mathbf{Y} = \mathbf{y}) \quad (33)$$

$$\delta_{i,j}(s', s) \triangleq \Pr(S_{i+j} = s, \mathbf{Y}_j^i = \mathbf{y}_j^i \mid S_i = s') \quad (34)$$

The probabilities $\Pr(U_{i,j} = 0 \mid \mathbf{Y} = \mathbf{y})$ and $\Pr(U_{i,j} = 1 \mid \mathbf{Y} = \mathbf{y})$ may be obtained from $\Pr(U_{i,j} = 0, \mathbf{Y} = \mathbf{y}) = \sigma_{i,j}(0)$ and $\Pr(U_{i,j} = 1, \mathbf{Y} = \mathbf{y}) = \sigma_{i,j}(1)$ via Eq. (9), with:

$$\begin{aligned}\Pr(\mathbf{Y}=\mathbf{y}) &= \Pr(U_{i,j}=0, \mathbf{Y}=\mathbf{y}) + \Pr(U_{i,j}=1, \mathbf{Y}=\mathbf{y})\\ &= \sigma_{i,j}(0) + \sigma_{i,j}(1)\,. \quad (35)\end{aligned}$$

The values of $\sigma_{i,j}(b)$, for $b = 0, 1$ and $j = 1, 2, \ldots, n$, $i = 1, 2, \ldots, N/n$ may be obtained from $\alpha_{i-1}(b(e)), \gamma_i(e), \beta_i(t(e))$ as follows:

$$\begin{aligned}\sigma_{i,j}(b) &= \Pr(\mathbf{Y} = \mathbf{y}, U_{i,j} = b) \quad (36)\\ &= \sum_{e:u_j(e)=b} \Pr(\mathbf{Y}=\mathbf{y}, E_i = e) \quad (37)\\ &= \sum_{(s,e):u_j(e)=b} \Pr(\mathbf{Y}=\mathbf{y}, E_i=e, S_{i-1}=s, S_i=h_s(e))\\ &= \sum_{(s,e):u_j(e)=b} \alpha_{i-1}(b(e))\gamma_i(e)\beta_i(t(e)) \quad (38)\\ &= \sum_{e:u_j(e)=b} \sigma_i(e) \quad (39)\end{aligned}$$

We have considered a set of edges that necessarily includes repeated edge patterns. That is, if the number of inputs is 2^k and the number of states is 2^ν, then there are $2^\nu 2^k$ edges to consider. If the length of the edges is $n < \nu + k$ then there are $2^{\nu+k-n}$ edges with the same bit sequence. A different form that unambiguously accounts for all edges.

The values of $\sigma_{i,j}(b)$ may be computed using $\lambda_{i,j}(s', s, b)$ instead of $\gamma_i(e)$ as follows:

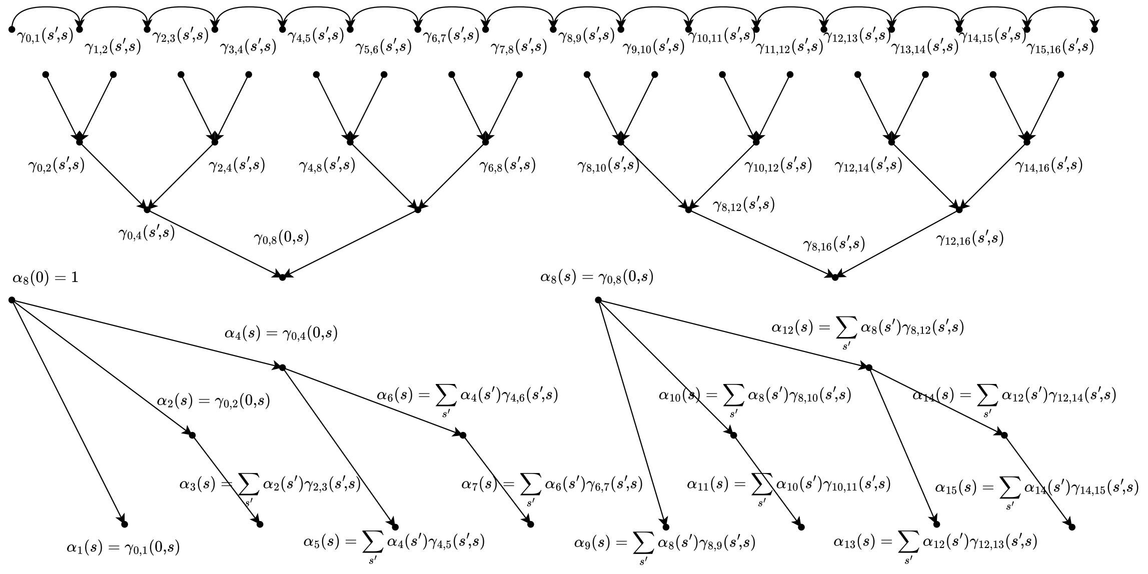
$$\begin{aligned}\sigma_{i,j}(b) &= \Pr(\mathbf{Y} = \mathbf{y}_{,i,j} = b) \quad (40)\\ &= \sum_{u:u_j=b} \Pr(\mathbf{Y} = \mathbf{y}, U_i = u) \quad (41)\\ &= \sum_{s'} \sum_{u:u_j=b} \Pr(\mathbf{Y} = \mathbf{y}, U_i = u, S_{i-1} = s') \quad (42)\\ &= \sum_{s'} \sum_{u:u_j=b} \Pr(\mathbf{Y}=\mathbf{y}, U_i=u, S_{i-1}=s', S_i=h_{s'}(u))\\ &= \sum_{s'} \sum_{s} \Pr(\mathbf{Y} = \mathbf{y}, U_{i,j} = b, S_{i-1} = s', S_i = s)\\ &= \sum_{s'} \sum_{s} \Pr(S_{i-1} = s', \mathbf{Y}_1^{i-1} = \mathbf{y}_1^{i-1}) \quad (43)\\ &\qquad \cdot \Pr(\mathbf{Y}_i=\mathbf{y}_i, S_i=s \mid S_{i-1}=s') \quad (44)\\ &\qquad \cdot \Pr(\mathbf{Y}_{i+1}^N=\mathbf{y}_{i+1}^N \mid S_i=s) \quad (45)\\ &= \sum_{s'} \sum_{s} \alpha_{i-1}(s')\lambda_{i,j}(s', s, b)\beta_i(s) \quad (46)\end{aligned}$$

Let the input that produces edge e as output be $u(e) \triangleq f_{b(e)}^{-1}(e)$. Then, for each edge e and each $i = 1, 2, \ldots, N/n$ the values of $\gamma_i(e)$ are computed from the channel probabilities $\Pr(\mathbf{Y}_i = \mathbf{y}_i \mid E_i = e)$ and the a-priori probabilities $\Pr(U_i = u(e))$ as follows:

$$\begin{aligned}\gamma_i(e) &= \Pr(\mathbf{Y}_i = \mathbf{y}_i, E_i = e \mid S_{i-1} = b(e)) \quad (47)\\ &= \Pr(\mathbf{Y}_i = \mathbf{y}_i \mid E_i = e, S_{i-1} = b(e))\\ &\qquad \cdot \Pr(E_i = e \mid S_{i-1} = b(e)) \quad (48)\\ &= \Pr(\mathbf{Y}_i=\mathbf{y}_i \mid E_i=e) \Pr(U_i=u(e) \mid S_{i-1}=b(e))\\ &= \Pr(\mathbf{Y}_i = \mathbf{y}_i \mid E_i = e) \Pr(U_i = u(e)) \quad (49)\end{aligned}$$

Note that two edges e and e' may have the same value but have different source state $b(e) \neq b(e')$ or destination state $t(e) \neq t(e')$. In such case, we will still treat them as different so that the source and destination states of each edge are unique and the event $\{S_{i-1} = b(e)\}$ is unambiguous. However, the channel probabilities will be shared by these edges.

Let $\gamma_i(s', u) \triangleq \gamma_i(f_{s'}(u))$, then the probabilities $\lambda_{i,j}(s, s', b)$ may be computed from the values of $\gamma_i(e)$ or $\gamma_i(s', u)$ as



$\gamma_{0,1}(s',s)$
$\gamma_{1,2}(s',s)$
$\gamma_{2,3}(s',s)$
$\gamma_{3,4}(s',s)$
$\gamma_{4,5}(s',s)$
$\gamma_{5,6}(s',s)$
$\gamma_{6,7}(s',s)$
$\gamma_{7,8}(s',s)$
$\gamma_{8,9}(s',s)$
$\gamma_{9,10}(s',s)$
$\gamma_{10,11}(s',s)$
$\gamma_{11,12}(s',s)$
$\gamma_{12,13}(s',s)$
$\gamma_{13,14}(s',s)$
$\gamma_{14,15}(s',s)$
$\gamma_{15,16}(s',s)$
$\gamma_{0,2}(s',s)$
$\gamma_{2,4}(s',s)$
$\gamma_{4,8}(s',s)$
$\gamma_{6,8}(s',s)$
$\gamma_{8,10}(s',s)$
$\gamma_{10,12}(s',s)$
$\gamma_{12,14}(s',s)$
$\gamma_{14,16}(s',s)$
$\gamma_{8,12}(s',s)$
$\gamma_{0,4}(s',s)$
$\gamma_{0,8}(0,s)$
$\gamma_{8,16}(s',s)$
$\gamma_{12,16}(s',s)$
$\alpha_8(0) = 1$
$\alpha_8(s) = \gamma_{0,8}(0,s)$
$\alpha_{12}(s) = \sum_{s'} \alpha_8(s')\gamma_{8,12}(s',s)$
$\alpha_4(s) = \gamma_{0,4}(0,s)$
$\alpha_6(s) = \sum_{s'} \alpha_4(s')\gamma_{4,6}(s',s)$
$\alpha_{10}(s) = \sum_{s'} \alpha_8(s')\gamma_{8,10}(s',s)$
$\alpha_{14}(s) = \sum_{s'} \alpha_{12}(s')\gamma_{12,14}(s',s)$
$\alpha_2(s) = \gamma_{0,2}(0,s)$
$\alpha_3(s) = \sum_{s'} \alpha_2(s')\gamma_{2,3}(s',s)$
$\alpha_7(s) = \sum_{s'} \alpha_6(s')\gamma_{6,7}(s',s)$
$\alpha_{11}(s) = \sum_{s'} \alpha_{10}(s')\gamma_{10,11}(s',s)$
$\alpha_{15}(s) = \sum_{s'} \alpha_{14}(s')\gamma_{14,15}(s',s)$
$\alpha_1(s) = \gamma_{0,1}(0,s)$
$\alpha_5(s) = \sum_{s'} \alpha_4(s')\gamma_{4,5}(s',s)$
$\alpha_9(s) = \sum_{s'} \alpha_8(s')\gamma_{8,9}(s',s)$
$\alpha_{13}(s) = \sum_{s'} \alpha_{12}(s')\gamma_{12,13}(s',s)$

Figure 6: **Binary tree representation of the process to compute the values of $\alpha_i(s)$ for each $i = 1, 2, \ldots, N/m$ in a logarithmic number of steps, represented by the tree depth. First the values of $\delta_{i2^{j+1}, i2^{j+1}+2^j}(s', s)$ are obtained for $j = 1, 2, \ldots, L$, and each $i = 1, 2, \ldots, 2^{L-j}$ and each $s, s' = 1, 2, \ldots, M$ and then used to compute all $\alpha_i(s)$.**

follows:

$$\lambda_{i,j}(s, s', b) = \sum_{e: b(e)=s', t(e)=s, u_j(e)=b} \gamma_i(e) \tag{50}$$

$$= \sum_{s'} \sum_{u: u_j=b, h_{s'}(u)=s} \gamma_i(f_{s'}(u)) \tag{51}$$

$$= \sum_{s'} \sum_{u: u_j=b, h_{s'}(u)=s} \gamma_i(s', u) \tag{52}$$

The values of $\alpha_i(s)$ and $\beta_i(s)$ are generally computed recursively, where $\alpha_i(s),\ s = 1, \ldots, M$ are used to obtain $\alpha_{i+1}(s)$ and $\beta_{i+1}(s),\ s = 1, \ldots, M$ are used to obtain $\beta_i(s)$, via the Forward and Backwards algorithm [3]. In this paper, we develop an algorithm to break the recursive dependencies, which is explained in the next section. The standard method to compute $\alpha_i(s)$ and $\beta_i(s)$ for all $i = 1, \ldots, N$ and all $s = 1, \ldots, M$ follows:

$$\alpha_i(s) = \Pr(S_i = s, \mathbf{Y}_1^i = \mathbf{y}_1^i) \tag{53}$$

$$= \sum_{s'} \Pr(S_{i-1} = s', \mathbf{Y}_1^{i-1} = \mathbf{y}_1^{i-1}, S_i = s, \mathbf{Y}_i = \mathbf{y}_i)$$

$$= \sum_{s'} \Pr(S_{i-1} = s', \mathbf{Y}_1^{i-1} = \mathbf{y}_1^{i-1})$$

$$\cdot \Pr(S_i = s, \mathbf{Y}_i = \mathbf{y} \mid \mathbf{Y}_1^{i-1} = \mathbf{y}_1^{i-1}, S_{i-1} = s'),$$

Note that $\Pr(S_{i-1} = s', \mathbf{Y}_1^{i-1} = \mathbf{y}_1^{i-1}) = \alpha_{i-1}(s')$ and given $S_{i-1} = s'$ the events $\{S_i = s, \mathbf{Y}_i = \mathbf{y}_i\}$ are independent of $\mathbf{Y}_1^{i-1} = \mathbf{y}_1^{i-1}$, then:

$$\alpha_i(s) = \sum_{s'} \alpha_{i-1}(s') \Pr(S_i = s, \mathbf{Y}_i = \mathbf{y} \mid S_{i-1} = s')$$

$$= \sum_{s'} \alpha_{i-1}(s') \delta_{i,i+1}(s', s). \tag{54}$$

The values of $\beta_i(s)$ are computed starting from $\beta_{N/n}(s)$ backwards as follows:

$$\beta_{i-1}(s) = \Pr(\mathbf{Y}_i^N = \mathbf{y}_i^N \mid S_{i-1} = s) \tag{55}$$

$$= \Pr(\mathbf{Y}_i = \mathbf{y}_i, \mathbf{Y}_{i+1}^N = \mathbf{y}_{i+1}^N \mid S_{i-1} = s) \tag{56}$$

$$= \sum_{s'} \Pr(\mathbf{Y}_i = \mathbf{y}_i, S_i = s', \mathbf{Y}_{i+1}^N = \mathbf{y}_{i+1}^N \mid S_{i-1} = s)$$

$$= \sum_{s'} \Pr(\mathbf{Y}_i = \mathbf{y}_i, S_i = s' \mid S_{i-1} = s) \tag{57}$$

$$\cdot \Pr(\mathbf{Y}_{i+1}^N = \mathbf{y}_{i+1}^N \mid S_i = s', \mathbf{Y}_i = \mathbf{y}_i, S_{i-1} = s)$$

Note that $\Pr(\mathbf{Y}_i = \mathbf{y}_i, S_i = s' \mid S_{i-1} = s) = \delta_{i-1,i}(s, s')$, and the event $\{\mathbf{Y}_{i+1}^N = \mathbf{y}_{i+1}^N\}$ is conditionally independent of $\{\mathbf{Y}_i = \mathbf{y}_i, S_{i-1} = s\}$ given $\{S_i = s'\}$, then:

$$\beta_{i-1}(s) = \sum_{s'} \Pr(\mathbf{Y}_i = \mathbf{y}_i, U_i = S_i = s' \mid S_{i-1} = s) \tag{58}$$

$$\cdot \Pr(\mathbf{Y}_{i+1}^N = \mathbf{y}_{i+1}^N \mid S_i = s') \tag{59}$$

$$= \sum_{e: b(e)=s} \gamma_i(e) \beta_i(t(e)) \tag{60}$$

$$= \sum_{s'} \delta_{i,i+1}(s, s') \beta_i(s') \tag{61}$$

Equation (61) concludes the review of the standard method to obtain the values of $\alpha_i(s)$, $\beta_i(s)$ and $\gamma_i(e)$ needed to compute each $\sigma_{i,j}(b)$. However, with the recursive equations (54) and (61) the values of $\alpha_i(s)$ need to be computed before the values of $\alpha_{i+1}(s)$ and the values of $\beta_i(s)$ need to be computed before the values of $\beta_{i-1}(s)$, for each $i = 1, 2, \ldots, N$ and each $s = 1, 2, \ldots, M$. In the next section, we break this dependency to enable computing the values of $\alpha_i(s)$ and $\beta_i(s)$ for many values of i at the same time.

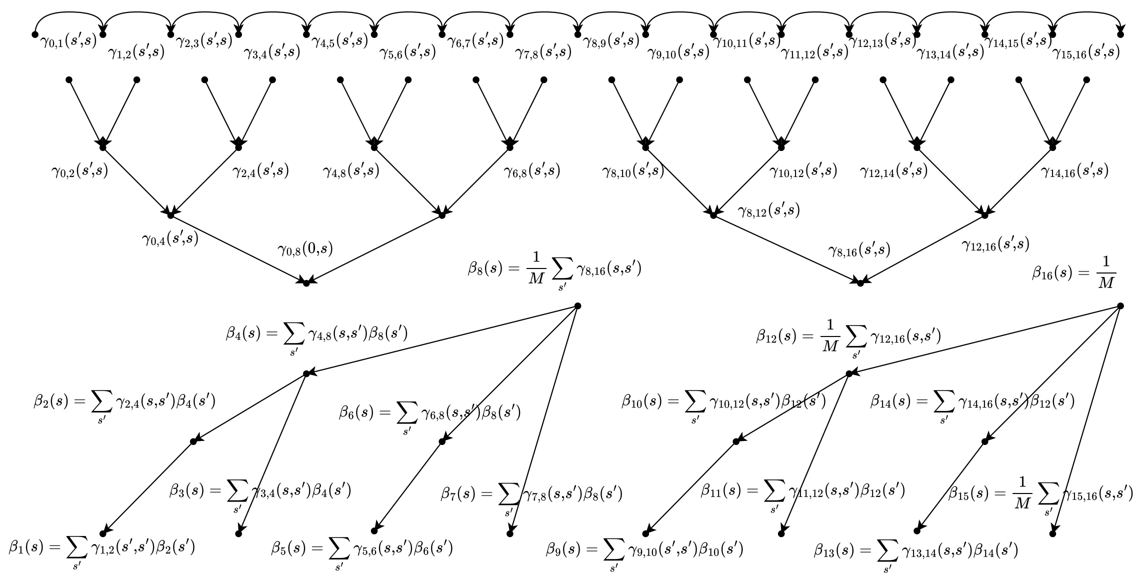


Figure 7: **Binary tree representation of the process to compute the values of $\beta_i(s)$ for each $i = 1, 2, \ldots, N/m$ in a logarithmic number of steps, represented by the tree depth. The same values of $\delta_{i2^{j+1}, i2^{j+1}+2^j}(s', s)$ used to compute each $\alpha_i(s)$ are used to compute each $\beta_i(s)$.**

3. Trellis Reduction Operations

In this section, we describe a method to compute the values of $\alpha_i(s)$, $\beta_i(s)$ for many stages i simultaneously. The goal is to break the dependency of $\alpha_{i+1}(s)$ on $\alpha_i(s)$ and of $\beta_{i-i}(s)$ on $\beta_i(s)$ and enable most of the speedup that parallel processing hardware would provide if these values could be computed independently.

The proposed method consists of a sequence of trellis reduction steps. These steps are shown in Fig. 5. At each step values of $\delta_{i,i+1}(s', s)$ of a "reduced" trellis, that skips every other stage, are computed from pairs of these values of an original trellis. First the values of $\delta_{i,i+1}(s', s)$ are computed as follows:

$$\begin{aligned}\delta_{i,i+1}(s, s') &= \sum_{e:b(e)=s', t(e)=s} \Pr(E_i = e, \mathbf{Y}_i = \mathbf{y}_i \mid S_i = b(e)) \\ &= \sum_{e:b(e)=s', t(e)=s} \gamma_i(e) \qquad (62) \\ &= \lambda_{i,j}(s, s', 0) + \lambda_{i,j}(s, s', 1)\,, \qquad (63)\end{aligned}$$

where j can be any single value from $0, 1, \ldots, k-1$. Given the values of $\delta_{i,i+1}(s', s)$ for all i and all s of an original trellis, the values of $\delta_{j,j+2}(s', s)$ of a new trellis that joins every two stages of the original trellis may be obtained by the following recursion:

$$\begin{aligned}\delta_{i,i+2}(s', s) &= \Pr(S_{i+2} = s, \mathbf{Y}_i^{i+2} = \mathbf{y}_i^{i+2} \mid S_i = s') \qquad (64) \\ &= \Pr(S_{i+2} = s, \mathbf{Y}_i^{i+1} = \mathbf{y}_i^{i+1}, \mathbf{Y}_{i+1}^{i+2} = \mathbf{y}_{i+1}^{i+2} \mid S_i = s') \\ &= \sum_r \Pr(S_{i+1} = r, \mathbf{Y}_i^{i+1} = \mathbf{y}_i^{i+1} \mid S_i = s') \qquad (65) \\ &\quad \cdot \Pr(S_{i+1} = s, \mathbf{Y}_{i+1}^{i+2} = \mathbf{y}_{i+1}^{i+2} \mid S_{i+1} = r, \mathbf{Y}_{i+1}^{i+2}, S_i) \\ &= \sum_r \Pr(S_{i+2} = s, \mathbf{Y}_{i+1}^{i+2} = \mathbf{y}_{i+1}^{i+2} \mid S_{i+1} = r) \qquad (66) \\ &\quad \cdot \Pr(S_{i+1} = r, \mathbf{Y}_i^{i+1} = \mathbf{y}_i^{i+1} \mid S_i = s') \qquad (67) \\ &= \sum_r \delta_{i+1,i+2}(r, s)\delta_{i,i+1}(s', r) \qquad (68)\end{aligned}$$

Equation (68) provides a method to compute the $\delta_{i,i+2}(s, s')$ given the values of deltas $\delta_{i,i+1}(s, s')$ and $\delta_{i+1,i+2}(s, s')$ of an original trellis. The initial values of $\delta_{i,i+1}(s, s')$ are obtained from Eq. (62) and Eq. (68). Then the values of $\delta_{i,i+2}(s, s')$ are obtained using Eq. (68), which become the values of $\delta_{i,i+1}(s, s')$ on a new trellis with half the stages. To avoid describing the trellis we are operating on, we will define by $\delta_{i,i+2}(s', s)$ the values of $\delta_{i,i+1}(s', s)$ on the new trellis obtained after the first trellis reduction step, then define $\delta_{i,i+4}(s, s')$ the values of the new trellis after the second reduction step, and so on. Note that the values of i must be a multiple of number stages of the original trellis that are combined in the most recent "new trellis," and must be a power of 2.

For ease of notation suppose that $n = 1$ and the number of trellis stages is a power of 2 given by $N = 2^{L+1}$ for some L. The method to construct all values of $\delta_{i,j}(s', s)$ of the form $\delta_{2^l i, 2^l i + 2^l}(s', s)$ is illustrated in the trees at the top of Fig. 6 and Fig. 7, and is summarized in the following equations:

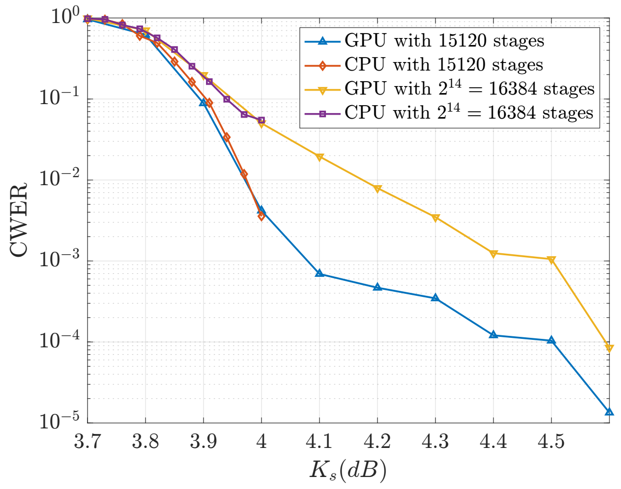


Figure 8: **CWER comparison for two implementations with** 15120 **and** 16384 **stages. CPU version features the original BCJR algorithm with no parallelization. GPU uses a trellis reduction algorithm with the implementation described in Section 4.** $K_b = 0\,dB$**,** 4 **PPM**

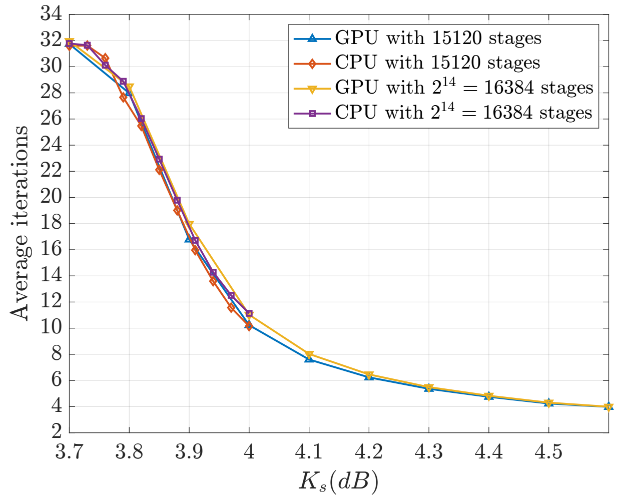


Figure 9: **Average iterations comparison for two implementations with** 15120 **and** 16384 **stages. The maximum iteration number is set to 32. CPU version features the original BCJR algorithm with no parallelization. GPU uses a trellis reduction algorithm with the implementation described in Section 4.** $K_b = 0\,dB$**,** 4 **PPM**

The steps to construct such $\delta_{i,j}(r, s')$ where $i = 2^l$ and $j = 2^{l-1}(i + 1)$ are as follows:

$$\delta_{2i,2(i+1)}(s', s) = \sum_r \delta_{2i,2i+1}(s', r)\delta_{2i+1,2i+2}(r, s) \quad (69)$$

$$\delta_{2^2 i,2^2(i+1)}(s', s) = \sum_r \delta_{2^2 i,2^2 i+2}(s', r)\delta_{2^2 i+2,2^2(i+1)}(r, s)$$

$$\delta_{2^3 i,2^3(i+1)}(s', s) = \sum_r \delta_{2^3 i,2^3 i+2^2}(s', r) \cdot \delta_{2^3 i+2^2,2^3(i+1)}(r, s) \quad (70)$$

$$\vdots \qquad \vdots$$

$$\delta_{2^L i,2^L(i+1)}(s', s) = \sum_r \delta_{2^L i,2^L i+2^{L-1}}(s', r) \cdot \delta_{2^L i+2^{L-1},2^L(i+1)}(r, s) \quad (71)$$

The $\alpha_i(s)$ and $\beta_i(s)$, $i = 1, 2, \ldots, 2^{L+1}$, $s = 1, 2, \ldots, M$ may also be computed are also computed in a logarithmic number of steps, using the values of $\delta_{2^l i,2^l i+2^l}(s', s)$, $l = 0, 1, \ldots, L$, once the values of $\delta_{0,2^L}(s', s)$ and $\delta_{2^L,2^{L+1}}(s', s)$ are obtained. This process is illustrated in the bottom trees of Fig. 6 and Fig. 7. To start set $\alpha_{2^i}(s) = \delta_{2^i,0}(s, 0)$, $\beta_{N-2^i}(s) = \delta_{N-2^i,0}(s, 0)$, then, for each $j = 0, 1, \ldots, L-1$ and each $i = 0, 1, \ldots, 2^{L-j}$ compute:

$$\alpha_{2^{j+1}i+2^j}(s) = \sum_r \alpha_{2^{j+1}i}(r)\delta_{2^{j+1}i,2^{j+1}i+2^j}(s, r) \quad (72)$$

$$\beta_{2^{j+1}i-2^j}(s) = \sum_r \delta_{i2^j-2^j,i2^{j+1}}(s, r)\beta_{i2^{j+1}}(r) \quad (73)$$

With Eq. (72) and Eq. (73) the values of $\alpha_{2^{j+1}-2^j}(s)$ and $\beta_{i2^{j+1}-2^j}(s)$ for fixed j can be computed in parallel, starting from $j = L, L-1, \ldots, 1$, where $L = \log_2(N)$. This method allows to compute the $\beta_i(s)$ and $\beta_i(s)$ from which future values depend in a logarithmic number of steps given by $L = \log_2(N)$. The same holds for the $\delta_{i2^j,(i+1)2^j}(s', s)$ values.

In the case where the code length is not a power of 2, like the case of the SCPPM code with standard length $N = 15120$, two sets of $\delta_{2^l i,2^l i+2^l}(s', s)$ may need to be computed, one to compute all $\alpha_i(s)$ and one to compute all $\beta_i(s)$. The set to compute the values of $\alpha_i(s)$ starts at the first stage, with $i = 0, 1, \ldots, 2^{L-j}$ for fixed j, and the set to compute the values of $\beta_i(s)$ starts at the last stage N. That is, computing instead $\delta_{N-2^l i,N-2^l i+2^l}(s', s)$ for $i = 0, 1, \ldots, 2^{L-j}$ for fixed j. However, if N is multiple of a power of 2, say 2^l with $l < L$, then majority of the values of $\delta_{2^j i,2^j i+2^j}(s', s)$ may be re-used, that is, all for which $j \leq l$.

4. GPU IMPLEMENTATION VIA CUDA

The Graphics Processing Unit is a massively parallel computing device with thread numbers rising to thousands. Thread is a single unit of execution that completes instructions sequentially. While threads are not completely independent, separate cores use the same instructions making development easier. We choose CUDA as it allows control of individual threads in contrast to other programming platforms supporting GPU such as PyTorch. Synchronization may be performed even on a subset of threads and memory is manually managed. Moreover, using CUDA's finer memory management interface we utilized data locality in trellis reduction operations.

GPU groups threads into Streaming Multiprocessors (SMs) that can be used to process relevant tasks using the same data. Additionally, synchronization within an SM is faster, compared to synchronization of all threads on a GPU. While "global" memory is accessible from all threads, each SM contains an additional faster "shared" memory accessible only from threads within an SM. Another memory level, "local", is only available within a thread and is the fastest. The whole workload is divided among available SMs while only the most accessed data is stored at local and shared

Algorithm 1: Inner Decoder Function Compute $\sigma_{i,j}[b]$

```
Input: $edges[1 << \nu][1 << k]$        ▷ $edges[s][u]$ is the code output when state is $s$ and input $u$
Input: $nexts[1 << \nu][1 << k]$        ▷ $nexts[s][u]$ is the state $S_{i+1}$ for state $S_i = s$ and $U_i = u$
Input: $\mathbf{y}[N][1 << k]$        ▷ $\mathbf{y}[i][e]$: channel edge probability: $\Pr(\mathbf{Y}_{ni}^{n(i+1)} = \mathbf{y}_i \mid edges[S_i][\mathbf{U}_i] = e)$
Input: $AAP[N][1 << k]$        ▷ $AAP[i][u]$: A-posteriori probability: $\Pr(\mathbf{U}_i = u)$
Output: $\sigma[N][k][2]$        ▷ $extout[i][u]$: Extrinsic probabiity: $\Pr(\mathbf{U}_i = u \mid \mathbf{Y} = \mathbf{y}, \mathbf{U} = AAP[N][1 << k])$

for $i = 0, \dots, N-1$ do
    for $s = 0, \dots, 2^\nu - 1$ do
        for $u = 0, \dots, (1 << k) - 1$ do
            for $j = 0, \dots, k-1$ do
                $s' = nexts[s][u]$        ▷ State at stage $i+1$ when state at stage $i$ is $s$ and input is $u$
                $e = edges[s][u]$        ▷ edge $e$ when state is $s$ and input is $u$
                $b = (u >> j)\&1$        ▷ binary value of $j$-th entry of $u$: $b \in \{0, 1\}$
                $\lambda[i][j][s][s'][b] \leftarrow self + y[i][e] \times app[u]$        ▷ Using (36)
        for $s' = 0, \dots, 2^\nu - 1$ do
            $\delta_{i,i+1}[s, s'] \leftarrow \lambda[i][0][s][s'][0] + \lambda[i][0][s][s'][1]$        ▷ Using (65)
        ▷ Suppose that $N = 1 << LOGN$
for $l = 0, 1, \dots, LOGN - 1$ do
    for $i = 0, 1, \dots, 2^{LOGN-l}$ do
        $d \leftarrow 2^l$        ▷ mid point between start and end stage
        $t \leftarrow i \times 2d$
        for $s = 0, \dots, 2^\nu - 1$ do
            for $s' = 0, \dots, 2^\nu - 1$ do
                ▷ $\delta_{t,t+2d}[s][s'] \leftarrow \sum_{r=0}^{2^\nu-1} \delta_{t,t+d}[s][r] \times \delta_{t+d,t+2d}[r][s']$ Eq. (82)
                for $r = 0, \dots, 2^\nu - 1$ do
                    $\delta_{t,t+2d}[s][s'] \leftarrow self + \delta_{t,t+d}[s][r] \times \delta_{t+d,t+2d}[r][s']$        ▷ Using (82)
$\alpha_0[0] \leftarrow 1, \beta_N[0] \leftarrow 1$
$\alpha_0[i] \leftarrow 0, \beta_N[i] \leftarrow 0\ \forall i = 1, 2, \dots, 2^\nu - 1$
for $l = 0, 1, \dots, LOGN - 1$ do
    for $i = 0, 1, \dots, 2^l - 1$ do
        $d \leftarrow 2^{LOGN-l-1}$        ▷ mid point between start and end stage
        $t \leftarrow i \times 2d$
        for $s = 0, \dots, 2^\nu - 1$ do
            for $s' = 0, \dots, 2^\nu - 1$ do
                $\alpha_{t+d}[s] \leftarrow self + \alpha_t[s'] \times \delta_{t,t+d}[s'][s]$        ▷ Using (70)
                $\beta_{N-t-d}[s] \leftarrow self + \delta_{N-t-d,N-t}[s][s'] \times \beta_{N-t}[s']$        ▷ Using (76)
for $i = 0, \dots, N-1$ do
    for $s = 0, \dots, 2^\nu - 1$ do
        for $s' = 0, \dots, 2^\nu - 1$ do
            for $j = 0, \dots, k-1$ do
                $\sigma[i][j][0] \leftarrow self + \alpha_i[s]\lambda[i][j][s][s'][0]\beta_{i+1}[s']$        ▷ Using (63)
                $\sigma[i][j][1] \leftarrow self + \alpha_i[s]\lambda[i][j][s][s'][1]\beta_{i+1}[s']$        ▷ Using (63)
```

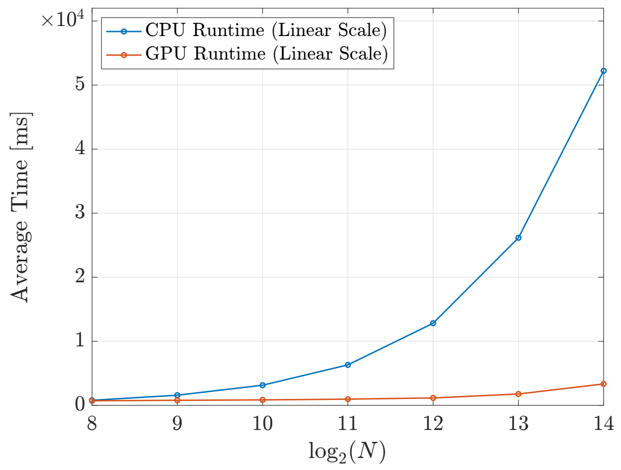


Figure 10: **Runtime of GPU and CPU decoders on 1000 codewords with 10 iterations each. Only decoder runtime is measured (Linear scale runtime).**

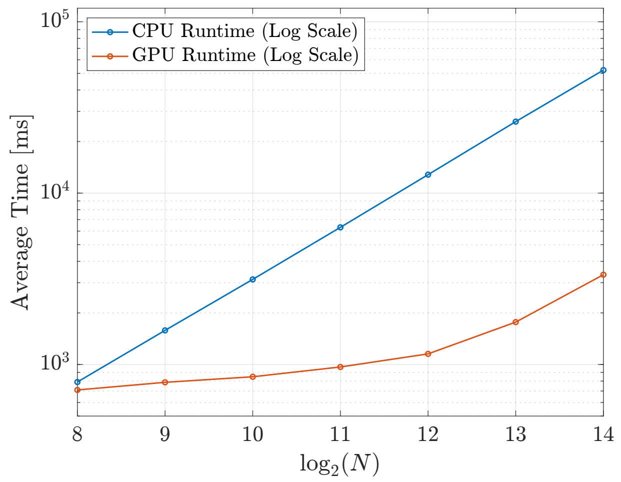


Figure 11: **Runtime of GPU and CPU decoders on 1000 codewords with 10 iterations each. Only decoder runtime is measured (Log scale runtime).**

levels.

A naive implementation of the trellis reduction algorithm, as described in Algorithm 1, stores all data in global memory and assigns each thread a stage to process. As global memory is significantly larger than the size of all required variables, this approach is easily scaled up for larger number of stages, but runtime is limited by the memory bandwidth.

The memory hierarchy suggests further optimizations. Large arrays with rare accesses including the $\alpha_i(s)$ and $\beta_i(s)$ values are located in global memory, while the $\gamma_i(e)$ values could remain local to the thread as the values are unused by other threads. On the other hand, the values of $\delta_{i,j}(s', s)$ are both compute-intensive and used by multiple threads. We place it in shared memory by exploiting the tree structure of trellis. The upper part of trellis as seen in Fig. 6 is divided into blocks equal to the desired amount of SMs. Each block uses only its own shared memory. This approach not only improves memory access time but also removes the need for synchronization between SMs during trellis processing, as separate branches of the tree can be computed independently.

Shared memory usage acts as the main constraint for the implementation on modern Nvidia GPU architectures. The data structure containing the values of $\delta_{i,j}(s', s)$ is proportional in size to the code length and is divided between SMs. Given a GPU with enough SMs, any code length can be decoded using our implementation.

5. Simulation Results

To verify the functionality of the proposed GPU implementation we compare GPU and baseline CPU implementations in terms of codeword error rate (CWER) performance. For the GPU implementation, we used 32 SMs with 256 threads each. CWER is measured by running each simulation until 100 codeword errors are observed (except for the last point in the GPU with $N = 2^{14}$ stages and $N = 15120$, where the number of errors is about 50). We run the original BCJR algorithm on the CPU and the trellis reduction algorithm on the GPU using CUDA as described in Section 4. Performance results in terms of codeword error rate vs. average photons per pulsed slots K_s (dB) are presented in Fig. 8. The data was obtained with $K_b = 0\ dB$ and with 4 PPM. The curves are essentially identical, verifying the equivalent functioning of CPU and GPU decoders. Using this simulation framework, Fig. 9 shows the average number of iterations required to decode a single codeword. The maximum number of iterations permitted is 32. As the noise is decreases, the expected number of iterations also decreases. The curves further confirm that the CPU and GPU implementations require the same number of iterations to achieve decoding. Thus, the runtime of a given number of iterations can be used to estimate runtime for a given K_s value using either Fig. 10 or Fig. 11.

To evaluate the runtime benefits of the GPU implementation over the CPU implementation, we decode 1000 codewords with 10 iterations required to decode a single codeword on average. In total 10000 iterations are performed to obtain a single data point. Results are gathered for GPU and CPU implementations and shown in Fig. 10 and Fig. 11. These graphs show the same data; one uses a linear scale for the Y-axis and the other uses a log scale. We used powers of 2 for the values of N to demonstrate how runtime increases as a function of N for these two implementations. Both figures show the significant decrease in runtime achieved by the GPU implementation as compared with the CPU implementation. More stages allow for more parallelism so that the GPU runtime remains relatively constant even as N increases as long as there are sufficient hardware resources. The exact difference can be seen on the log scale graph in Fig. 11. Even for $2^8 = 256$ stages, GPU provides 38% speed-up. The benefit rises to 15 times runtime reduction with $2^{14} = 16384$ stages.

6. Conclusions

This work shows an efficient method to execute in parallel many of the computations of the BCJR algorithm that are generally computed sequentially. Provided sufficient hardware resource, a GPU can effectively use the benefits of the trellis reduction algorithm to significantly reduce the decoding time of each codeword. The GPU implementation decode also offers greater portability than other hardware implementations like an FPGA. The runtime of the GPU implementation is improved up to an order of magnitude compared to CPU when the blocklength reaches 2^{14}-bits.

Future Work

The current version of the algorithm relies on normalization of the values to maintain numerical stability. The log domain version of the algorithm has better stability properties, but the latest implementation does not attain the run-time performance of the original version. In the future, we plan to explore optimizations of the log domain implementation and better memory management to improve the run-time performance. Further improvement in the run-time performance may be attained by an integer version of the log domain algorithm, aided by look-up tables.

ACKNOWLEDGMENTS

The UCLA authors thank the Office of Naval Research for funding this project. This work was carried out in part at the Jet Propulsion Laboratory, California Institute of Technology, under contract with the National Aeronautics and Space Administration. Both JPL and UCLA would like to thank JPL Center of Innovation Fund (CIF) for their support. UCLA also would like to acknowledge National Science Foundation (NSF) grant CCF-2008918.

REFERENCES

[1] A. Viterbi, "Error bounds for convolutional codes and an asymptotically optimum decoding algorithm," *IEEE Trans. Inf. Theory*, vol. 13, no. 2, pp. 260–269, 1967.

[2] G. Forney, "The Viterbi algorithm," *Proceedings of the IEEE*, vol. 61, no. 3, pp. 268–278, 1973.

[3] L. Bahl, J. Cocke, F. Jelinek, and J. Raviv, "Optimal decoding of linear codes for minimizing symbol error rate (corresp.)," *IEEE Trans. on Information Theory*, vol. 20, no. 2, pp. 284–287, 1974.

[4] G. Fettweis and H. Meyr, "Parallel viterbi algorithm implementation: breaking the acs-bottleneck," *IEEE Trans. on Com.*, vol. 37, no. 8, pp. 785–790, 1989.

[5] A. Mohammadidoost and M. Hashemi, "High-throughput and memory-efficient parallel viterbi decoder for convolutional codes on gpu [arxiv]," *arXiv (USA)*, pp. 8 pp. –, 2020/11/18.

[6] H. Peng, R. Liu, Y. Hou, and L. Zhao, "A gb/s parallel block-based viterbi decoder for convolutional codes on gpu," in *2016 8th Intl. Conf. on Wireless Comms. & Signal Processing (WCSP)*, 2016, pp. 1–6.

[7] Z. Du, Z. Yin, and D. A. Bader, "A tile-based parallel viterbi algorithm for biological sequence alignment on gpu with cuda," in *2010 IEEE Int. Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*, 2010, pp. 1–8.

[8] M. Hanif and K.-H. Zimmermann, "Accelerating viterbi algorithm on graphics processing units," *Computing (Germany)*, vol. 99, no. 11, pp. 1105 – 23, 2017/11/. [Online]. Available: http://dx.doi.org/10.1007/s00607-017-0557-6

[9] N. Seshadri and C.-E. Sundberg, "List Viterbi decoding algorithms with applications," *IEEE Tran. Comm. Theory*, vol. 42, pp. 313–323, 1994.

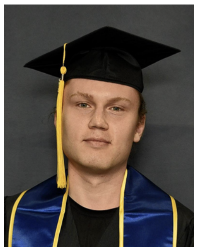
[10] Y. Ben Asher, V. Tartakovsky, K. Portman, O. Zilberman, and A. Hadar, "An fpga scalable parallel viterbi decoder," in *2018 IEEE 12th Intl. Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*, 2018, pp. 8–15.

[11] C. Berrou, A. Glavieux, and P. Thitimajshima, "Near shannon limit error-correcting coding and decoding: Turbo-codes. 1," in *Proceedings of ICC '93 - IEEE International Conference on Communications*, vol. 2, 1993, pp. 1064–1070 vol.2.

[12] S. Benedetto, D. Divsalar, G. Montorsi, and F. Pollara, "Serial concatenation of interleaved codes: performance analysis, design, and iterative decoding," *IEEE Transactions on Information Theory*, vol. 44, no. 3, pp. 909–926, 1998.

BIOGRAPHY

Amaael Antonini *(Student Member, IEEE) received the B.S. degree in electrical engineering and in applied mathematics in 2018 and the M.S. and Ph.D. degrees in electrical engineering from the University of California at Los Angeles (UCLA) and 2022, and 2024. His research interests include channel coding in the short block regime with and without feedback, trellis codes, list decoding, and low latency decoders.*



Egor Glukhov *received the B.S. degree in electrical engineering from the University of California at Los Angeles (UCLA) in 2024. His research interests include parallel computing and computer architecture. He is planning to pursue M.S. and Ph.D. degrees.*

Richard Wesel *received the B.S. and M.S. degrees in electrical engineering from the Massachusetts Institute of Technology in 1989 and the Ph.D. degree in electrical engineering from Stanford University in 1996. He is currently a Professor with the Electrical and Computer Engineering Department, Henry Samueli School of Engineering and Applied Science, UCLA, where he is also the Associate Dean of Academic and Student Affairs. His research interests include communication theory with a particular interests include short-blocklength communication with and without feedback, list decoding, low-density parity-check codes, and optimal modulation design. Dr. Wesel is a Fellow of the IEEE and has received the National Science Foundation CAREER Award, the Okawa Foundation Award for Research in Information Theory and Telecommunications, and the Excellence in Teaching Award from the Samueli School of Engineering. He has served as an Associate Editor for Coding and Coded Modulation and IEEE Transactions on Communications and as an Associate Editor for Coding and Decoding and IEEE Transactions on Information Theory.*

***Dariush Divsalar** received the Ph.D. degree in electrical engineering from UCLA, in 1978. Since then, he has been with the Jet Propulsion Laboratory (JPL), California Institute of Technology (Caltech), Pasadena, where he is a Fellow. At JPL, he has been involved with developing state-of-the-art technology for advanced deep-space communications systems and future NASA space exploration. Since 1986, he has taught graduate courses in communications and coding at UCLA and Caltech. He has published more than 290 papers, coauthored a book entitled* An Introduction to Trellis Coded Modulation with Applications*, contributed to three other books, and holds 30 U.S. patents. Dr. Divsalar was a co-recipient of the 1986 paper award of the IEEE Transactions on Vehicular Technology. He was also a co-recipient of the joint paper award of the IEEE Information Theory and IEEE Communication Theory societies in 2008. The IEEE Communication Society has selected one of his papers for inclusion in a book entitled The Best of the Best: Fifty Years of Communications and Networking Research. He served as an Editor for the IEEE Transactions on Communications from 1989 to 1996. A Fellow of IEEE since 1997. He has received over 50 NASA Tech Brief awards, a NASA Exceptional Engineering Achievement Medal in 1996, IEEE Alexander Graham Bell Medal in 2014, an Ellis Island Medal of Honor in 2023, and a NASA Distinguished Public Service Medal in 2023. He was elected to the National Academy of Engineering in 2024.*

***Jon Hamkins** received the B.S. degree in electrical engineering from the California Institute of Technology (Caltech), Pasadena, in 1990 and the M.S. and Ph.D. degrees in electrical and computer engineering from the University of Illinois at Urbana- Champaign in 1993 and 1996, respectively. Since then, he has been with the Jet Propulsion Laboratory, Caltech, where he is now the Chief Technologist of the Communications, Tracking, and Radar Division. His research interests include channel coding theory, information theory, signal processing, autonomous receivers, ranging, and optical communications.*