Performance Analysis and Optimal Design of BATS Code: A Hardware Perspective

Jiaxin Qing, Philip H. W. Leong, *Senior Member, IEEE,* and Raymond W. Yeung, *Fellow, IEEE*

***Abstract*—Batched Sparse codes (BATS codes) are a class of linear network coding schemes that increase network throughput by converting a multi-hop problem into an end-to-end problem through network coding. It plays a crucial role in future wireless communication, where packet loss is inevitable. This is an enduring problem in many applications, including vehicular networks. While the theory of BATS codes has been developed over the past decade, little research has been done on its hardware implementation, which is important for practical adoption. This paper provides a systematic way to analyze the performance of a BATS hardware accelerator when the code varies. A roofline model which provides an upper bound of the performance considering both computational and input/output constraints is developed. Next, we build a model connecting the BATS code design space with the hardware execution time, which can be used to determine an optimal code. Finally, we propose and test a flexible and scalable BATS code design paradigm for hardware accelerators with numerical results. The same framework could easily be extended to other linear codes and different computing systems.**

I. Introduction

MOBILE and distributed scenarios where edge devices such as mobile phones and sensors send data to the cloud to produce an output at a remote location are becoming increasingly prevalent. Such models of computation often involve multiple machines of different types connected via one or more wireless communication systems, and applications that fall in this category include Edge Computing, Artificial Intelligence, Internet of Things, Autopilot, etc. Efficient wireless communication is central to providing the infrastructure necessary for such applications.

The designs of current network protocols like TCP/IP were based on the assumption that a reliable and low-latency link layer is available. This is normally true for wired media such as optical fiber or twisted-pair cable. However, in wireless communication, the channel is constantly changing. Packet loss is inevitable due to congestion, fading, and interference, which is even more so for mobile and vehicular communications [1], [2]. Current techniques to constrain packet loss involve either reducing the modulation and coding rate or increasing the signal transmission power, resulting in lower network throughput and higher power consumption [3]. Batched Sparse Code (BATS Code) [4] is a new class of network codes [5], [6] that increase the end-to-end throughput in a wireless multi-hop network without increasing the transmission power for each link. The BATS code differs from the traditional end-to-end channel codes in the fact that it enables coding at the intermediate nodes rather than simple packet forwarding, which is critical for a network code to achieve the capacity in a lossy network in a wide range of scenarios [7], [8].

Retransmission is the only approach to address end-to-end packet loss for the conventional store-and-forward strategy. However, this introduces extra acknowledgment and management packets that can lead to potential channel congestion and also causes an exponential increase in overhead and power consumption as the number of hops increases. For example, with IEEE 802.11a, a 20 Mbps single hop network drops to 1 Mbps when the number of hops is increased to eight [9]. Network codes, like the BATS code, trade the computational requirement at intermediate nodes for higher end-to-end throughput so that retransmission is not required and transmitters can send data at **higher code rates**. As a result, power consumption caused by retransmissions, congestion control, and channel loss control in the store-and-forward strategy can be reduced.

Application-specific integrated circuits (ASICs) and Field Programmable Gate Arrays (FPGAs) are hardware platforms that use digital circuits to accelerate algorithms with a low power consumption [10], [11]. As algorithms are implemented as digital circuits, implementations can be massively paralleled by pipelining different parts of the algorithms, which leads to a high throughput and low latency performance. On the other hand, applications running on a CPU with a complexly designed kernel are usually slow for networking applications where a large volume of data must be handled efficiently and reliably [12], [13]. Therefore, in high-throughput and delay-sensitive applications like the wireless base station [14], practical LDPC [15], high-performance computing [16], and data center [17], FPGA and ASIC are widely used to provide a reliable and efficient data processing capability.

To match the push for **higher wireless data rates and lower power consumption**, efficient hardware implementations of BATS codes are required. However, there is no efficient algorithm for finding the most suitable BATS code design for FPGA except for searching a large design space, which is not feasible. This paper considers FPGA implementation, although the same ideas directly apply to ASICs.

Copyright (c) 2015 IEEE. Personal use of this material is permitted. However, permission to use this material for any other purposes must be obtained from the IEEE by sending a request to pubs-permissions@ieee.org.

J. Qing is with the Department of Information Engineering, The Chinese University of Hong Kong, Hong Kong SAR (email: jqing@ie.cuhk.edu.hk).

P. H. W. Leong is with the School of Electrical and Information Engineering, The University of Sydney, Sydney, Australia (email: philip.leong@sydney.edu.au).

R. W. Yeung is with the Department of Information Engineering, The Chinese University of Hong Kong, Hong Kong SAR. R. W. Yeung is also with the Institute of Network Coding, The Chinese University of Hong Kong, and he is also a Principal Investigator of the Centre for Perceptual and Interactive Intelligence (CPII) Limited (email: whyeung@ie.cuhk.edu.hk).

In this work, we provide a general performance analysis of BATS codes, with few assumptions regarding its implementation. We provide a practical and achievable performance upper bound that can be used to compare different BATS code designs and choices of the finite field. We then introduce a flexible and scalable BATS code design methodology using this model, which can be used to design an optimal BATS code with minimal searches in the design space. Finally, we present implementation results that verify our presented theory. To the best of our knowledge, this is the first paper to study hardware implementations of BATS codes.

II. Background

A. BATS Code

A BATS code is a network code for wireless multi-hop networks with an erasure channel in which packets are either lost or well-received. It was first proposed in [18] and analyzed asymptotically in [4], [19]. A finite-length analysis was proposed in [20] with the asymptotic assumption relaxed, which provides a more practical analysis of the performance when the number of received batches is finite. Breaking the "multi-hop curse" in wireless communication, various variants of the BATS code are designed for different applications. For example, [21] designed an expanding-window-based BATS code for delay-sensitive applications like video streaming and vehicular communications. [22] further improved this method with an adaptive encoder. [23] proposed a BATS code with unequal error protection capability for applications where different parts of the data are of different importance. Many researchers have focused on improving BATS codes' practicality to further push for more applications. For example, [24] and [25] designed sophisticated and practical inner codes for the BATS code, while [26] designed a quasi-universal outer code for different channel conditions. However, the existing work analyzes and optimizes the BATS code without considering a specific computation platform.

A BATS code consists of an inner code and an outer code, as depicted in Fig. 1. The inner code is a random linear network code (RLNC) [27] applied at the intermediate nodes (referred to as *recoding*); and the outer code is a matrix generalization of the fountain code [28], applied at the source and destination nodes. The BATS code has lower computational and storage requirements than RLNC, and it preserves the rateless property of the fountain code [29]. Similar to the RLNC, a BATS code can be applied in a network with an unknown topology. However, we consider a simple network here for simplicity.

Consider a line network with one source node, one destination node, and multiple intermediate nodes. The source node has a file of size F to transmit, which can be divided into k packets denoted by $\boldsymbol{B} = [b_1, b_2, ..., b_k]$, where each packet consists of a fixed number of symbols taken from a finite field called the base field. It is further assumed that all the symbols in this paper are taken from the base field.

1) Encoding: A $batch$ is a set of M coded vectors generated using $\boldsymbol{B_j} \subseteq \boldsymbol{B}$ and a random generator matrix $\boldsymbol{G_j}$ where j refers to the j-th batch. The source node encodes the source packets according to Algorithm 1, where dg_j is the degree of the j-th batch, which is sampled from Ψ, the degree distribution. Encoding can be described by the following linear system:

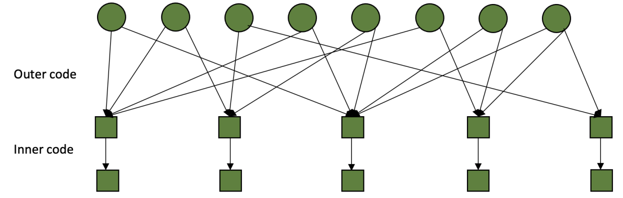

Fig. 1. Tanner graph representation for BATS. The first row represents input packets which are called variable nodes. The second row represents coded batches which are called check nodes. The third row stands for batches after being "recoded" by the intermediate nodes. Batches are generated in the outer code by taking linear combinations of the input symbols according to an optimized degree distribution.

$$\boldsymbol{X_j} = \boldsymbol{B_j} \times \boldsymbol{G_j},$$

where $\boldsymbol{X_j}$ is the j-th batch with M coded symbols.

Algorithm 1: Encoding for the j-th $batch$

Input: $\boldsymbol{B} = [b_1, b_2, ..., b_k], seed_j$
Output: $\boldsymbol{h_{j1}} + \boldsymbol{X_j}$
$dg_j \leftarrow \psi(rng(seed_j))$;
$\boldsymbol{B_j} \leftarrow$ uniformly sample dg_j pkts from $\boldsymbol{B}$;
$\boldsymbol{G_j} \leftarrow rng(seed_j)$;
$\boldsymbol{X_j} = \boldsymbol{B_j} \times \boldsymbol{G_j}$;
$\boldsymbol{h_{j1}} = \boldsymbol{I}$;

Other than the coded symbols, an identity matrix $\boldsymbol{I}$ of size $M \times M$ is appended to each batch as a "pilot" used for learning the end-to-end transfer matrix.

2) Recoding: Recoding is only done within each batch. If n_i packets are received, where $n_i \leq M$, these received packets are multiplied by a random matrix $\boldsymbol{h_j}$ of size $n_i \times M$. In the case of packet loss, recoding restores the batch to M coded symbols, which effectively introduces extra redundancy at the intermediate node. In detail, the recoding operation is performed according to Algorithm 2.

Algorithm 2: Recoding the j-th $batch$

Input: $\boldsymbol{H_j} + \boldsymbol{X_j}$
Output: $\boldsymbol{H_j} + \boldsymbol{Y_j}$
$\boldsymbol{h_j} \leftarrow rng()$;
$\boldsymbol{Y_j} = \boldsymbol{X_j} \times \boldsymbol{h_j}$;
$\boldsymbol{H_j} = \boldsymbol{H_j} \times \boldsymbol{h_j}$;

We introduce the coefficient vector $\boldsymbol{H_j} = \boldsymbol{h_{j1}} \boldsymbol{h_{j2}} ... \boldsymbol{h_{jn}}$, where $\boldsymbol{h_{jn}}$ is the linear transformation applied to the $batch_j$ by the n-th node. Effectively, $\boldsymbol{H_j}$ is the product of all random matrices involved in recoding of $batch_j$ after n hops. The encoding and recoding are described by the following linear system:

$$\boldsymbol{Y_j} = \boldsymbol{B_j} \boldsymbol{G_j} \boldsymbol{H_j},$$

where $\boldsymbol{Y_j}$ is the j-th output of this linear system.

3) Decoding: Since the code is constructed based on a pseudo-random number generator during encoding, the incidence relations between input packets and coded batches (depicted by the Tanner graph [30]), and the corresponding generator matrices can be recovered using the shared seeds at the destination node. To solve this linear system for B, belief propagation (BP) decoding is used [29]. The basic idea of BP decoding in BATS is to recursively solve a decodable batch (check node) and then substitute the decoded symbols (variable node) into other undecoded batches until all the decodable batches are decoded. The detailed decoding procedure is given in Algorithms 3 and 4.

Algorithm 3: Decoding for $\boldsymbol{B}$

```
Input: Tanner graph: T
Input: H + Y
Output: B
while exist undecoded pkts in B do
    for batch_i <- i = 1 to n do
        if batch_i is not decoded then
            if rank(G_i H_i) == rank(B_i) then
                B_i = Y_i (G_i H_i)^{-1};
                BackSubstitute(B_i, T);
            end
        end
    end
end
```

Algorithm 4: BackSubstitution

```
Input: Decoded pkt: b
Input: Tanner graph: T
for all batch_i in T that involves b do
    Y_i, H_i, g <- (batch_i, T);
    Y_i = Y_i - b · g · H_i;
end
```

Generally, in the design space of a BATS code, the following parameters are important: F, Ψ, M, pk and K as summarized in Table I. The optimal choice of Ψ based on the network topology and channel status has been extensively studied, and we refer readers to [29], [31], [32], [33], [26], [34] for detailed discussions[1]. Here, F, M and pk are usually related. In [29], it is shown that in a line network with multiple hops, the achievable rate increases with M. However, when F is small, a large M will lead to a lower efficiency assuming pk is fixed, and K is equal to $(F * (1 + overhead))/(pk * M) + 1$, where the overhead is a non-negative real number.

TABLE I
SUMMARY OF BATS CODE PARAMETERS

Symbol	Meaning	Unit
F	total size of the file	bits
φ	size of the file in finite field symbols	finite field symbols
Ψ	degree distribution	N/A
M	batch size	N/A
K	total number of batches generated	N/A
pk	packet size	finite field symbols
K	total number of batches generated	N/A

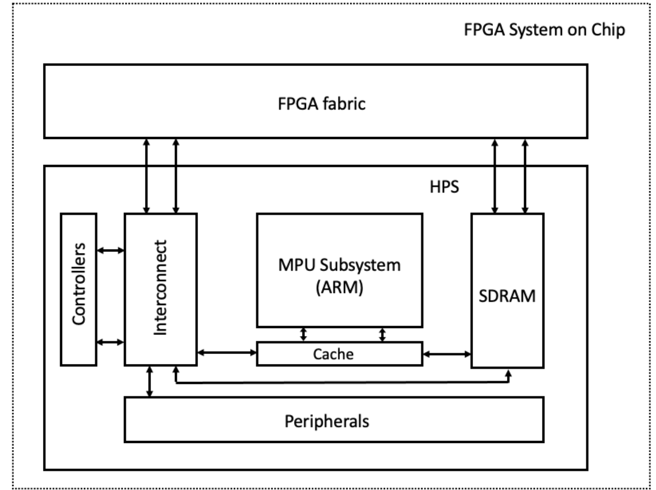

Fig. 2. Simplified architecture for a general FPGA SoC [35]. HPS: Hard Processor System. MPU: Microprocessor Unit

B. Problem Formulation

In this paper, we consider implementing a BATS code on a typical FPGA SoC architecture [35] as shown in Fig. 2. The BATS code consists of three major components: the encoder, the recoder and the decoder. We assume the following parameters to be static during encoding: the degree distribution (Ψ), the packet size (pk) and the batch size (M). We use random linear network coding for the recoder and a Belief Propagation decoding without inactivation for the decoder [29].

We also consider accelerating BATS by offloading the finite field matrix computations to an FPGA while keeping other necessary operations on the CPU. This design philosophy will be justified in Section IV. Therefore, Algorithms 1, 2 and 3 are modified to FPGA accelerated versions as in Algorithms 5, 6 and 7, where **FPGA**$(\cdot)$ indicates the algorithms to be implemented on the FPGA.

Given the above setup, we aim to find BATS parameters which leads to a design with the highest throughput and lowest latency under the same network and application constraints.

Algorithm 5: FPGA Accelerated Encoding

```
Input: B = [b_1, b_2, ..., b_k], seed_j
Output: I + X_j
dg_j <- ψ(rng(seed_j));
B_j <- uniformly sample dg_j pkts from B;
G_j <- rng(seed_j);
X_j = FPGA{B_j × G_j};
```

[1]As the first implementation of the BATS code on FPGA, we accept the original design of the BATS code as described in [29].

Algorithm 6: FPGA Accelerated Recoding

Input: $\boldsymbol{I} + \boldsymbol{X_j}$
Output: $\boldsymbol{H_j} + \boldsymbol{Y_j}$
$\boldsymbol{h_j} \leftarrow rng()$;
$\boldsymbol{Y_j} = \textbf{FPGA}\{\boldsymbol{X_j} \times \boldsymbol{h_j}\}$;
$\boldsymbol{H_j} = \textbf{FPGA}\{\boldsymbol{I} \times \boldsymbol{h_j}\}$;

Algorithm 7: FPGA Accelerated Decoding

Input: $\boldsymbol{H} + \boldsymbol{Y}$
Output: $\boldsymbol{B}$
while *exist undecoded pkts in* $\boldsymbol{B}$ **do**
  for $batch_i \leftarrow i = 1$ **to** n **do**
    if $batch_i$ *is not decoded* **then**
      if $rank(\boldsymbol{G_i H_i}) == rank(\boldsymbol{B_i})$ **then**
        $\boldsymbol{B_i} = \textbf{FPGA}\{\boldsymbol{Y_i}(\boldsymbol{G_i H_i})^{-1}\}$;
        $\textbf{FPGA}\{\boldsymbol{BackSubsitute(B_i)}\}$;
      end
    end
  end
end

III. Roofline Model

The performance of a hardware accelerator is constrained by two factors, memory bandwidth and computational power. It can be improved by: increasing the available computing resources through the use of larger capacity devices; increasing the memory access efficiency via schemes such as bursting and coalescing [36]; and improving the algorithm to better utilise the computing resources (normally through parallelism) and reduce the memory bandwidth requirements. The Roofline model [37] is a simple bound-and-bottleneck analysis methodology for analysing computer systems, providing insightful performance and optimization bounds to which a specific implementation can be compared to determine its optimality. The general construction of a roofline model is now briefly reviewed.

Consider the analysis of a computer system required to execute a given algorithm using floating-point arithmetic. Let π be the arithmetic performance in floating-point operations per second (FLOPS), and let β be the memory bandwidth in byte/s. We define the *Operational Intensity (OI)* of a particular program to be the average number of floating point operations that are executed per memory access:

Definition 1: $OI = \pi/\beta$ with the unit of FLOP/byte.

Here, *OI* is a metric connecting the computational power with the memory bandwidth. For a given computer system, the maximum computational power (π_{max}) and memory bandwidth (β_{max}) can be determined from its specifications or via benchmarking.

Fig. 3 shows an example of a roofline model[2]. For a fixed β, if the computational power is unbounded (i.e. $\pi_{max} = \infty$), we have $\pi = \beta * OI$, independent of OI. Then the diagonal line formed for $\beta = \beta_{max}$ describes the maximum performance of this platform. In a real system, however the computational power in FLOPS is constrained by some finite π_{max}, so any implementation necessarily lies under the roof $\pi = \pi_{max}$ thus formed. Reference [37] suggests the following:

[2]The example in Fig. 3 is a finite field modified version. Different from the one in [37], π is measured by finite field operations per second (OPS). See Sec. III-B.

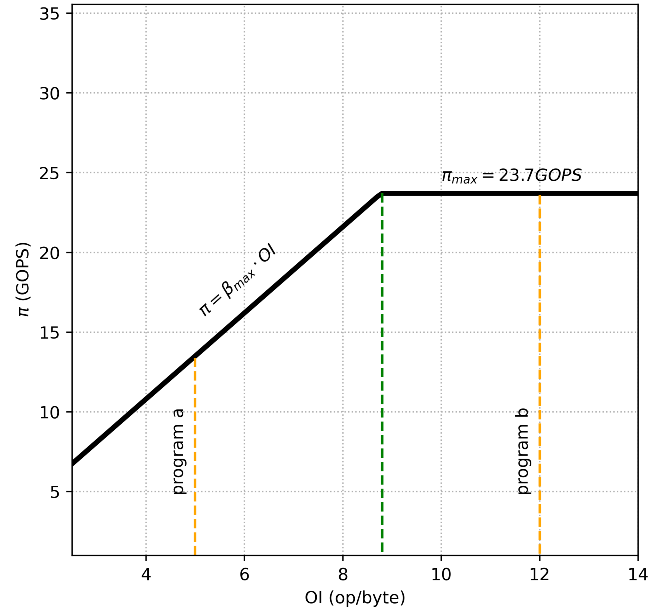

Fig. 3. Modified roofline model for FPGA (Cyclone V 5CSEBA6 [38]). π: computational power measured in Giga finite field operations per second (GOPS). β_{max}: the maximum achievable memory bandwidth. $op/byte$: number of finite field operations per byte data transferred. The roofline model shows the achievable computational throughput with respect to the operational intensity when the platform (FPGA) is used as an accelerator for the BATS code. It is bounded by the logic resources available on the FPGA

Proposition 1: (Attainable Performance) For a particular algorithm, the attainable performance or the attainable computational power (in FLOPS), $\pi_{attainable}$, on a particular hardware with maximum computational power and maximum memory bandwidth denoted by π_{max} and β_{max} respectively, is given by

$$\pi_{attainable} = min(\pi_{max}, \beta_{max} * OI),$$

where *OI* is the operational intensity of this algorithm.

Proposition 2: (Optimality Achievability) The x-coordinate of the ridge point stands for the minimum *OI* required for an algorithm to reach the peak performance on the current platform. It also indicates the level of difficulty for the designer to achieve the best performance on current platform.

Proposition 3: (Performance Bottleneck) If the *OI* of an algorithm lies on the left of the ridge point, then this algorithm is memory bounded. Otherwise it is computation bounded.

With minor modifications, the Roofline model can be applied to the design of the BATS hardware acccelerator, and enables both the platform and implementation to be evaluated. It also indicates which particular optimizations should be prioritised. However, the original roofline model in [37] was designed for floating point algorithms on a general computer system and requires modification before it can be applied to BATS.

A. Finite Field Multiplication

Like LDPC [39] and other error-correcting codes, most implementations of BATS code to date use $GF(2^n)$. In this paper we also use $GF(2^n)$ with polynomial basis [40] and most implementations have chosen $n = 8$ because $GF(2^8)$ provides enough randomness for the code in the sense that it is less probable to have a rank deficiency when generating random matrices. Specifically, the probability for an $r \times m$ randomly generated matrix over $GF(q) = GF(2^n)$ to be full-rank is given by,

$$\xi_r^m = \begin{cases} (1-q^{-m})(1-q^{-m+1})...(1-q^{-m+r-1}) & 0 < r \leq m \\ 1 & r = 0, \end{cases}$$

[29], [19].

Evidently, $\xi_r^m \to 1$ as $q \to \infty$, which means random matrices of full rank will be generated with a very high probability when the field size is large. Also, choosing $n = 8$ leads to a simpler implementation due to byte alignment.

However, in the later sections of this paper, we will show that $GF(2^8)$ is a suboptimal field choice sometimes in terms of performance on hardware. Nevertheless, we continue to use $GF(2^8)$ in this section for ease of discussion. As addition and subtraction in a binary extension field are simply exclusive-OR (XOR) operations, the computational requirements of BATS are dominated by multiplications (we will discuss finite field inversion in Section-IV). Therefore, the finite field multiplier is a crucial component for accelerating BATS.

Bit-parallel finite field multipliers can be analyzed in terms of space and time complexity. The space complexity is usually measured by the number of AND and XOR gates required for its implementation. In [41], it is shown that schoolbook finite field multiplication algorithms have at least a quadratic space complexity, i.e. $O(n^2)$ where n is the extension field degree, and this technique is generally faster (smaller in time complexity) than other algorithms with sub-quadratic space complexity [41]. As our field size is relatively small, we choose to design a bit-parallel multiplier based on the Russian-Peasant Multiplication algorithm [42] (a schoolbook algorithm) with quadratic space complexity which has a faster run time compared with algorithms with sub-quadratic space complexity.

The Russian-Peasant Multiplication algorithm is given in Algorithm 8.

As shown in Fig. 4, this algorithm can be realized using two 8-bit AND gates, two 8-bit XOR gates, three shift registers and three registers. Such an implementation takes 8 clock cycles to obtain the result c. We can also compute c in a single clock cycle using 8 copies of the circuit.

B. Modified Roofline Analysis

The original roofline model [37] only considers accesses that result from a memory system cache miss. In the case of BATS accelerators, performance in most cases is limited by off-chip memory accesses as we assume that on-chip memory parallelism is not a performance bottleneck. A major difference between our construction of the roofline model and the original one in [37] is that our computational power is measured by OPS instead of FLOPS (floating point operations per second), with OPS referring to the number of finite field multiplication per second. Unlike the maximum computational power in a CPU, which is determined by the clock frequency and number of floating point execution units, we need to fit as many Compute Units (CUs), i.e. finite field multipliers, as possible on the given area of FPGA. However, the modified finite field roofline model depends on the field size. For a given FPGA size or chip area, a smaller CU design will result in a higher π_{max} with an increase in the total number of operations as a trade-off.

To show the process of constructing a modified roofline model, we consider two example FPGA platforms, the Intel Cyclone V SoC (5CSEBA6) [38] and Intel Arria 10 SoC (10AS066) [43]. We implemented the multiplier introduced in the previous sections using high-level synthesis tools [44] and the resource consumption is summarized in Table II. Due to their different architectures, the number of logic gates used are not exactly the same.

Other than the resource consumption by multipliers in Table II, extra Logic Elements (LE) are needed for controlling, and

Algorithm 8: Multiplication in $GF(2^8)$ using Ruassian Peasant algorithm

```
Data: Two 8-bit input: a, b
Result: 8-bit output: c
c = 0;
for i ← 1 to 8 do
    if b & 1 then
        c ← c ⊕ a
    end
    if a & 0x80 then
        a ← (a≪ 1) ⊕ 0x11B
    else
        a ← (a≪ 1)
    end
    b ← (b≫ 1)
end
```

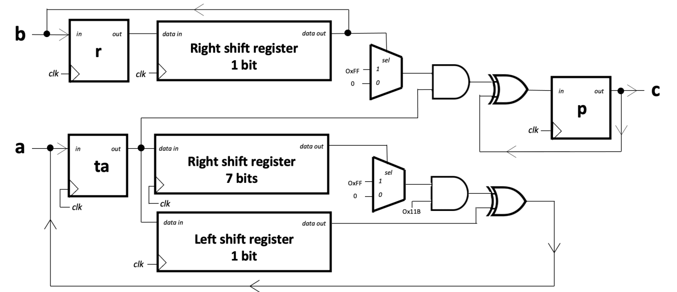

Fig. 4. Circuit design for a $GF(2^8)$ finite field multiplier based on the Ruassian-Peasant Multiplication algorithm [42], [41], where a, b is the input and c is the output. The final result is available after eight clock cycles. Primitive polynomial $x^8 + x^4 + x^3 + x + 1$ is used which is shown as 0x11B in hexadecimal. All data paths are 8 bits.

TABLE II
RESOURCE CONSUMPTION FOR AN 8-BIT MULTIPLIER IN $GF(2^8)$ SHOWN AS THE NUMBER OF ADDITION, EXCLUSIVE OR AND SUBTRACTION OPERATIONS NEEDED AND TOTAL LE CONSUMED.

	AND	XOR	SUB	LE	LE Available
Arria10	9	14	1	171	660K
CycloneV	22	14	7	153	110K

TABLE III
THE MAXIMUM ATTAINABLE OPS (OPERATIONS PER SECOND) FOR FPGA SHOWN IN THE MAGNITUDE OF $\times 10^9$

	Multipliers fitted	Freq. (MHz)	GOPS
Arria10	1157	240	277
CycloneV	215	110	23.7

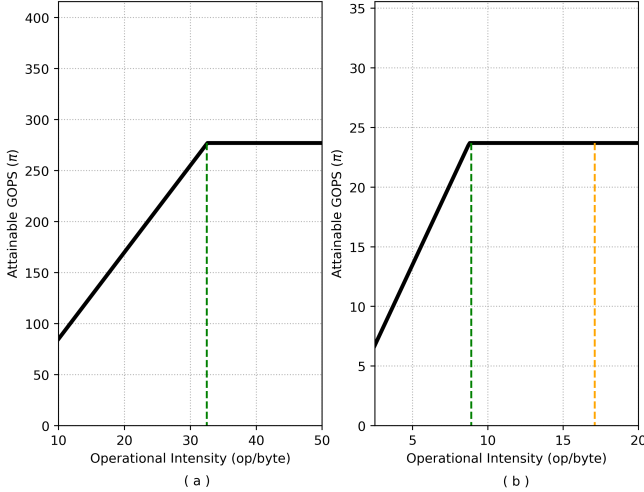

Fig. 5. Finite field modified roofline model based on $GF(2^8)$ for (a) Arria 10 10AS066 on the left and (b) Cyclone V 5CSEBA6 on the right. Arria has a ridge point at OI = 32.5 op/byte and maximum performance 277 GOPS. Cyclone V has a ridge point at OI = 8.9 op/byte and maximum performance 23.7 GOPS. The Arria 10 has much more logic resources than Cyclone V, therefore (a) has higher maximum attainable GOPS than (b), even though the same finite field is used.

building the interconnects and registers which consume the majority of the available resources. We assume that 30% of available LEs are devoted to the multipliers[3]. Based on this, we can now approximate the total number of multipliers that can fit on the device by dividing the total available LEs (minus the extra logic) by the LE consumption for a single multiplier. Given the clock frequency, the maximum finite field operations per second which is defined as the processing rate when all multipliers are working in parallel can be calculated for each platform. The results are summarized in Table III.

Cyclone V and Arria 10 have an external SDRAM interface with a dedicated on-FPGA hard memory controller, through which the FPGA accesses large-size data. Through benchmarking, the maximum external memory bandwidth is 8528 MB/s and 2700 MB/s for Arria 10 and Cyclone V, respectively. Then the modified roofline model can be constructed as in Fig. 5. Even though the Arria 10 is a much larger device than Cyclone and hence supports more CUs, the memory bandwidth is not increased as much as the computational power, which results in a ridge point that is harder to achieve.

Before proceeding to the next section, we note the following:

Proposition 4: The maximum performance of a BATS code on a hardware platform is determined by the finite field modified roofline model of this platform and the total number of operations due to BATS, where the computational power is measured in OPS.

[3]The 30% usage is an estimation obtained from synthesis reports of multiple BATS code implementations with different parameter configurations. A similar estimation can also be obtained with the analytical tool in [45].

IV. ROOFLINE ANALYSIS OF BATS

A. OI of Encoding and Recoding

Encoding and recoding computations of BATS can be divided into two steps: random number generation and matrix multiplication. Random number generation is the step where the generator matrices, batch degrees, etc are generated. Pseudorandom number generation algorithms like [46] [47] incur little overhead in most CPUs, but FPGA/ASIC implementation requires additional chip area and on-chip memory [48][49]. Matrix multiplication on CPUs is usually implemented using nested loops with blocking for better data locality [50]. Other techniques utilise SIMD operations [51] to compute multiple finite field operations that are normally computed serially. However, most CPUs available in the market do not have SIMD support for finite field multiplication, and a lookup table based implementation is used instead. Nevertheless, matrix multiplication is a well-studied problem for FPGAs. Designers can use highly parallel architectures in FPGAs, for example the systolic array [52] combined with tiling for better data locality to achieve a very efficient implementation.

Based on the above analysis, we propose to implement the randomness generation and the matrix multiplication separately on the CPU and FPGA respectively. This way, the FPGA will serve as a powerful accelerator for finite field matrix multiplication leading to a simple and easy-to-optimise hardware design. Reference [53] utilized the same design philosophy for high-throughput tensor processing unit where the hardware design is functionally simple but compact and highly paralleled. Using a System on Chip (SoC) [54], where the CPU and the FPGA are closely coupled on the same fabric, the communication between the CPU and the FPGA can be very efficient through dedicated interconnects as shown in Fig. 2. Large data transfers can be even more efficient through a direct connection between FPGA and the external memory.

To assess the computational power that can be achieved by a BATS code, we need to calculate the OI. In the following section, we derive an algebraic expression for OI in terms of the BATS parameters.

Consider a file of size F bits. In $GF(2^n)$ with 2^n finite field symbols per element, each symbol is represented by n bits. A file can be represented by $\varphi = F/n$ symbols. To encode this file and ensure the decodability, the following constraint is necessary:

$$\varphi_{coded} \geq \varphi, \tag{1}$$

where the total number of coded symbols to be generated is given by φ_{coded}.

Notice that the outer code rate $R_{out} = \varphi/\varphi_{coded}$. Suppose φ_{coded} are divided into K batches with M coded packets. Each packet is of size pk with unit of finite field symbol. Then we have

$$\varphi_{coded} = K \cdot M \cdot pk. \tag{2}$$

Notice that K is given by,

$$K = \frac{\varphi \cdot (1+\epsilon_o)}{pkM} \quad (3a)$$
$$= \frac{F \cdot (1+\epsilon_o)}{npkM} \quad (3b)$$

where $\epsilon_o \in \{x \in \mathbb{R} : x \geq 0\}$ is the encoding overhead in order to satisfy the constraint in (1). Since the file contains φ/pk packets, we have

$$dg_i \in [1, \varphi/pk]. \quad (4)$$

Denote the total number of finite field multiplication by α. Recall that the encoding of a batch can be simplified as a finite field matrix multiplication: $X_i = B_i \times G_i$, where $B_i \in \mathbb{F}_q^{pk\times dg_i}, G_i \in \mathbb{F}_q^{dg_i\times M}$ and $X_i \in \mathbb{F}_q^{pk\times M}$. Then, the total number of multiplications involved in encoding K batches is given by

$$\alpha_{encoding} = \sum_{i=1}^{K}(pk \cdot dg_i \cdot M). \quad (5)$$

Since $dg_i \in [1, \varphi/pk]$, we can write the average degree,

$$\frac{1}{K}\sum_{i=1}^{K} dg_i = \epsilon\frac{\varphi}{pk}, \quad (6)$$

where $\epsilon \in$ (0,1]. Then (5) can be simplified as

$$\alpha_{encoding} = \epsilon \cdot \varphi \cdot M \cdot K. \quad (7)$$

Now we can calculate the maximum achievable OI for encoding. Denote the memory transfer size by γ which has the unit of finite field symbol. The least amount of data to be transferred is the sum of the file size, the sizes of the generator matrices involved in the matrix multiplication and the size of the output matrix, given by

$$\gamma_{encoding} = \varphi + \frac{\epsilon\varphi}{pk}MK + pk \cdot MK. \quad (8)$$

The maximum OI is calculated by dividing the number of operations with the data transfer size. Notice that we regard the OI thus obtained as the maximum achievable OI because the memory transfer γ used in the formulation is the least memory transfer required. Depending on the implementation on FPGA, more memory transfer may be required, thus incurring a smaller OI. Substituting (3a) into (7) and (8), we obtain

$$\begin{aligned} OI_{max}^{encoding} &= \alpha_{encoding}/\gamma_{encoding} \\ &= \epsilon\varphi MK \times \frac{1}{\varphi + \frac{\epsilon\varphi}{pk}MK + pk \cdot MK} \\ &= \frac{pk \cdot \varphi}{\varphi + pk^2\left(\frac{1}{\epsilon} + \frac{1}{\epsilon(1+\epsilon_o)}\right)}, \end{aligned} \quad (9)$$

where OI_{max} has the unit of operation per bit. Similarly, for recoding, $\alpha_{recoding}$ and $\gamma_{recoding}$ can be calculated as

$$\alpha_{recoding} = pkM^2K, \quad (10)$$

$$\gamma_{recoding} = (2pk \cdot M + M^2)K. \quad (11)$$

Then,

$$\begin{aligned} OI_{max}^{recoding} &= \alpha_{recoding}/\gamma_{recoding} \\ &= \frac{pkM^2K}{(2pkM + M^2)K} \\ &= \frac{M \cdot pk}{M + 2pk}. \end{aligned} \quad (12)$$

From (9) and (12), we first draw the following conclusions:

- For encoding, OI_{max} depends only on pk and φ, where φ is a function of the file size and the finite field size, degree distribution and encoding overhead which are controlled by ϵ and $(1+\epsilon_o)$ respectively.
- For recoding, OI_{max} depends only on the batch size M and the packet size pk.

To see how to find the attainable computational power, let us take one of the most common BATS configuration from [29] as an example. Consider encoding a file of size $F = 64$ Kbytes with $n = 8$, $pk = 1024$, $\epsilon = 0.5$ and $\epsilon_o = 0.2$. We can calculate and plot $OI_{encoding} = 17.16$ op/byte in the roofline model. Then we can see in Fig. 5 (the orange dash line) that the corresponding attainable computational power is equal to the maximum computational power, i.e. $\pi = \pi_{max} = 23.7$ GOPS. Therefore, the BATS encoder implemented as described is computational bounded. While neither the computational power nor the OI directly represents the total time spent on the coding, they provide important information for determining the optimal design.

Equations (9) and (12) directly connect the hardware performance with the parameters in the BATS code design space. The OI_{max} for recoding is relatively simple as given in (12), which monotonically increases with either M or pk. This result is also intuitively correct because the inner code is designed to be simple so that it does not incur too much complexity at the intermediate nodes.

When the performance is memory bandwidth bounded, the closer OI is to the OI of the ridge point, the better, which means that we would want to increase OI in most cases. For example, in recoding, OI clearly increases with M and pk. However, we notice that the total number of operations $\alpha_{recoding}$ increases with M quadratically and linearly with pk. The best way to increase the OI while minimising the execution time for recoding is to use a large pk instead of a larger M. For encoding, the answer is not so obvious.

B. OI of Decoding

BP decoding is more complicated than encoding and recoding because it involves many logical decisions and data abstraction. In this paper, we only consider naïve BP decoding without inactivation decoding [55]. We proposed an FPGA accelerated algorithm for decoding in Algorithm 7. In the algorithm, the computations implemented on FPGA are specified.

Roughly speaking, BP decoding for BATS can be divided into the following steps: rank calculation, coefficient inversion, batch decoding and decoded packet propagation. Rank calculation and coefficient inversion are usually implemented with the classical Gaussian Elimination (GE). It has been shown that the GE is much more efficient in CPU than in FPGA

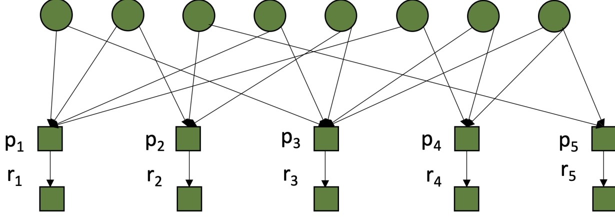

Fig. 6. A Tanner graph is a minimum decodable graph if all variable nodes can be decoded and there is no unused check node. Each batch is associated with two important parameters that determine its decodability, p: the number of variable nodes propagated to this check node. r: rank of this check node.

[56]. Therefore, we perform rank calculation and coefficient inversion in the CPU. Even though such a design incurs some communication cost, it is compensated by the ease of managing a Tanner graph for BP decoding on a CPU.

Algorithm 9: FPGA Accelerated Decoding with coalescing

```
Input: H + Y
Output: B
Collector
while exist undecoded pkts in B do
    for i = 1 to n do
        if batch_i is not decoded then
            if rank(G_i H_i) == rank(B_i) then
                Collector{(G_i H_i)^{-1}, i};
                mark batch_i as decoded;
            end
        end
    end
end
B = FPGA{Collector, Y}
```

Based on the proposed architecture, Algorithm 7 can be further improved to Algorithm 9 which reduces the FPGA-CPU communication cost. In Algorithm 9, inverted matrices are collected and only rank calculation and coefficient vector inversions are performed along the decoding process. When the BP decoding stops, all the inverted matrices and received packets will be transferred to the FPGA for computation. This way, less interruptions and invocations are needed between the FPGA and the CPU. We now analyze the OI of BP decoding.

Definition 2: (Decodable Graph) Define a decodable Tanner graph as the Tanner graph with sufficient information for all variable nodes to be decoded.

The information in Definition 2 refers to the received packets, coefficient vectors, and generator matrices.

Assume we want to decode a Tanner graph with K check nodes constructed in the encoding stage. Let r_i be the rank of the i-th batch, and p_i be the number of packets propagated to the i-th batch. Recall that a batch is solvable if and only if $r_i + p_i \geq dg_i$. For a minimum decodable graph, we have

$$r_i + p_i = dg_i. \tag{13}$$

For example, the Tanner graph in Fig. 6 is a minimum decodable graph if and only if (13) holds for all check nodes.

To decode the i-th batch, we first need to propagate the relevant decoded packets to it (*propagation stage*) and then multiply it with the corresponding $(\boldsymbol{G_i H_i})^{-1}$ (*decoding stage*). The propagation stage is essentially a matrix subtraction and a multiplication of size $(pk, r_i) - (pk, p_i) \times (p_i, r_i)$. The decoding stage is a matrix multiplication of size $(pk, r_i) \times (r_i, r_i)$ and requires a total number of finite field multiplications equal to

$$\sum_{i=1}^{K} \alpha_i = pk \sum_{i=1}^{K} (r_i^2 + p_i r_i), \tag{14}$$

where α_i is the total number of finite field operations needed to decode the i-th batch. Substituting (13) into (14), we have

$$\sum_{i=1}^{K} \alpha_i = pk \sum_{i=1}^{K} dg_i r_i. \tag{15}$$

Now, we analyze the memory transfer requirements. The minimum memory transfer for decoding a file with φ finite field symbols is given by $\sum_{i=1}^{K} \gamma_i$, where γ_i is the memory transfer required to decode the i-th batch. The minimum memory transfer consists of the transfer of all linearly independent received packets, the corresponding coefficient vectors and the final decoded file, i.e.,

$$\begin{aligned}\sum_{i=1}^{K} \gamma_i &= \sum_{i=1}^{K} (r_i pk + dg_i r_i + \varphi_i) \\ &= \sum_{i=1}^{K} (r_i pk + dg_i r_i) + \varphi,\end{aligned} \tag{16}$$

where φ_i is the portion of file decoded from the i-th batch and $\varphi = \sum_{i=1}^{K} \varphi_i$. Finally, OI_{max} is calculated as

$$\begin{aligned}OI_{max}^{decoding} &= \frac{\sum_{i=1}^{K} \alpha_i}{\sum_{i=1}^{K} \gamma_i} \\ &= \frac{pk \sum_{i=1}^{K} dg_i r_i}{\sum_{i=1}^{K} (r_i pk + dg_i r_i) + \varphi}.\end{aligned} \tag{17}$$

We use the same trick as in (6) to remove the summation. Notice that $dg_i \in [1, \frac{\varphi}{pk}], r_i \in [1, M]$ and $dg_i r_i \in [1, \frac{\varphi}{pk} M]$. Therefore, we can write

$$\sum_{i=1}^{K} dg_i r_i = \epsilon_1 \frac{\varphi}{pk} MK, \tag{18}$$

$$\sum_{i=1}^{K} r_i = \epsilon_2 MK, \tag{19}$$

where $\epsilon_1, \epsilon_2 \in (0, 1]$. Notice that ϵ_1 and ϵ_2 are parameters depending on dg_i and r_i. Substituting (3a), (18) and (19) into (17), we obtain

$$\begin{aligned}OI_{max}^{decoding} &= \frac{pk \sum_{i=1}^{K} dg_i r_i}{\sum_{i=1}^{K} (r_i pk + dg_i r_i) + \varphi} \\ &= \frac{pk \cdot \epsilon_1 \frac{\varphi}{pk} \cdot M \frac{\varphi(1+\epsilon_o)}{pkM}}{pk \cdot \epsilon_2 M \frac{\varphi(1+\epsilon_o)}{pkM} + \epsilon_1 \frac{\varphi}{pk} \cdot M \frac{\varphi(1+\epsilon_o)}{pkM} + \varphi} \\ &= \frac{\epsilon_1 (1+\epsilon_o) \cdot pk \cdot \varphi}{\epsilon_2 (1+\epsilon_o) \cdot pk^2 + \epsilon_1 (1+\epsilon_o) \varphi + pk^2} \\ &= \frac{pk \cdot \varphi}{\varphi + pk^2 \left(\frac{1}{\epsilon_1 (1+\epsilon_o)} + \frac{\epsilon_2}{\epsilon_1} \right)}.\end{aligned} \tag{20}$$

Observe that (20) has a form similar to that of (9). When $\epsilon_2 = 1$, the two equations are exactly the same. Due to this similarity, the optimal design for encoding and decoding are similar, even though there may be some slight mismatch due to the value chosen for ϵ_2, which largely depends on the network status. In practice, this discrepancy is not significant. We also observe from (9) and (20) that they have the same number of operations when $\epsilon_2 = 1$. According to our roofline model, the upper bound of the computational power is determined only by the OI of the algorithm when the hardware platform is fixed, which means that the encoding and decoding will share a similar upper bound for computational power. Therefore, we can draw the following conclusion from this observation, which will be clearer in (25) derived in the next section.

Proposition 5: (Shared Optimality) The optimal choice of parameters for encoding in terms of hardware performance would be approximately the same as the optimal parameters for decoding.

V. Maximum Throughput Analysis and the Design Paradigm

When a program is bandwidth bounded, higher OI leads to higher attainable computational power. But as one may notice, increasing the OI by changing the BATS parameters, e.g., pk, may also increase the total number of operations. The appropriate metric to use is the total execution time for processing a given amount of data or coding throughput subject to the given hardware resources. Therefore, in this section, we will characterize the execution time and coding throughput in terms of the BATS parameters on a hardware platform.

Suppose the attainable computational power for a BATS code is $\hat{\pi}_{attainable}$ OPS. It can be attained from the roofline model given the OI_{max} of the current BATS code design. Recall that the number of batches is given by (3b). Substituting this into (7), (10) and (15), we obtain

$$\alpha_{encoding} = \epsilon(1+\epsilon_o)\frac{F^2}{n^2pk}, \tag{21}$$

$$\alpha_{recoding} = (1+\epsilon_o)\frac{FM}{n}, \tag{22}$$

$$\alpha_{decoding} = \epsilon_1(1+\epsilon_o)\frac{F^2}{n^2pk} \tag{23}$$

as the total number of finite field operations for encoding, recoding and decoding respectively.

Notice that we assume the worst-case scenario for decoding, namely that all the batches are needed to recover the file and thus (21) and (23) are actually equivalent. We also observe the same relation for encoding and decoding in (9) and (20). Additionally, we can see that the computational complexity of recoding is linear in n, while the finite field multiplier is of quadratic space complexity (as discussed in Section III-A). Due to the difference in computational complexity and space complexity of finite field multiplier for FPGA implementation, more data can be recoded as the finite field size decreases, assuming that the total hardware resources are fixed.

Following the above, the minimum execution time is obtained by dividing the number of operations by the attainable computational power, i.e.,

$$t_{min} = \frac{\alpha}{\hat{\pi}_{attainable}}. \tag{24}$$

The maximum throughput of encoding and decoding is obtained by dividing the file size by the minimum execution time. For recoding, the maximum throughput is obtained by dividing the amount of data transferred by the minimum execution time. The maximum throughput denoted by T is given in (25) and (26) for encoding, recoding and decoding.

$$T^{encoding}_{max} = T^{decoding}_{max} = \frac{F}{t_{min}} \tag{25}$$

$$T^{recoding}_{max} = \frac{\gamma_{recoding}}{t_{min}} \tag{26}$$

Continuing with the previous example in Section IV-A of encoding a file of size $F = 64$ Kbytes with $n = 8$, $pk = 1024$, $\epsilon = 0.5$ and $\epsilon_o = 0.2$, we obtain $OI = 17.16$ op/byte that corresponds to $\hat{\pi}_{attainable} = 23.7$ GOPS. Using (21) and (24), we can calculate that $t_{min} = 0.106$ ms. If we change pk to 2048, we will have $OI = 8.69$ op/byte that corresponds to $\hat{\pi}_{attainable} = 18$ GOPS. Then the minimum execution time is $t_{min} = 0.053$ ms. We can also convert the minimum execution time into maximum throughput by dividing the file size by the execution time, which are 4.9Gbps and 9.9Gbps respectively for this example. In Section VI, we will show that these theoretical performance bounds are practical and achievable as supported by the measurements from the FPGA implementations.

In the design space of a BATS code, the file size F and the batch size M are mostly related to the network condition. A network with a large amount of data requires a large F and M for better efficiency. On the other hand, if the data rate in the network is too small while using a large F and M, the BATS code would introduce a significant delay to the network. To balance all these considerations, we propose Algorithm 10 to determine the most suitable design with minimum implementing and testing.

Algorithm 10: BATS for hardware design paradigm

Input: application constraints ζ_1
Input: network conditions ζ_2
Input: design space S
Output: Optimal BATS design $\hat{d}$

1: Determine $F \leftarrow \zeta_1, \zeta_2$;
2: Search S and calculate theoretical bounds;
3: Find designs with highest bounds;
4: Refine choices according to ζ_1, ζ_2;
5: $\hat{d} \leftarrow$ implementing and testing;

As indicated in Algorithm 10, we first need to consider the application constraints and network conditions to design a suitable BATS code. For example, the application constraint in vehicular communications could be low-latency, limited computation power and memory. The network condition could be defined as high packet loss due to challenging channel

Algorithm 11: Naïve Matrix Multiplication in $GF(2^n)$

```
Input: Matrix A: M × K, Matrix B: K × N
Output: Matrix C: M × N
Cache A, B for m ← 0 : M do
    for n ← 0 : N do
        acc ← 0;
        for k ← 0 : M do
            acc ← acc ⊕ A[m, k] × B[k, n];
        end
        C[m, n] ← acc;
    end
end
```

Algorithm 12: Two-level blocked version [57]

```
Input: Matrix A: M × K, Matrix B: K × N
Output: Matrix C: M × N
tileA, tileB[TS][TS];
regA, regB[TS]; /* TS:Tile size                */
for m ← 0 : M/TS do
    for n ← 0 : N/TS do
        acc[TS][TS] ← 0;
        for k ← 0 : M/TS do
            Load tileA, tileB ← A, B;
            /* register level tiling           */
            for t_k ← 0 : TS do
                Cache regA, regB ← tileA, tileB;
                acc ← acc ⊕ regA × regB;
            end
        end
        C ← acc;
    end
end
```

conditions. In Step 1 of Algorithm 10, we choose a small F, e.g., $F = 2$ KBytes, for low latency and low buffer size. Then we build up a design space S with parameters of interest, such as $pk, M, \epsilon, \epsilon_o$, and n. In Step 2 we calculate the theoretical throughput of all design combinations in S using the analytical tool proposed, after which we can select ones with high throughput in Step 3. In Step 4, we refine the selected designs under the application and network conditions. In this example, we choose a smaller M which reduces the buffering cost in both time and memory (ζ_1). Additionally, we need a higher coding overhead ϵ_o due to the network condition (ζ_2). Therefore, we can exclude designs from those selected in the last step with criteria defined by ζ_1, ζ_2. Finally, we can implement and test the remaining designs for the best solution. Notice that choosing F in Step 1 is optional since it can also be included in the design space as a variable. Nonetheless, in many applications, F is limited by the network buffer size and closely related to the network latency. Thus determining F prior to Step 2 is often desirable when the latency is prioritized over throughput.

Introducing the constraint variables ζ_1, ζ_2 in our design paradigm aids the searches for suitable designs within throughput constraints while keeping enough flexibility to meet the application requirement. In an actual design process, more complicated requirements, such as the network communication protocol, can be included in ζ. We refer readers to [58] for more detailed discussions on the design of network communication protocol from the perspective of the BATS code.

Exhaustively searching the design space S becomes intractable. Restricting the design space exploration via Algorithm 10 to a subset with the highest theoretical throughput allows a good solution to be found with minimal effort.

VI. Achievability of the Performance Bound

This section will show that our proposed theoretical bound is achievable by improving the hardware design. We first implement the encoder and recoder using a naïve matrix multiplication given in Algorithm 11, which we call the "naïve" implementation. It is a simple baseline implementation that requires further optimization.

For an implementation, plotting the achieved performance in its corresponding roofline model can help to decide how the current implementation can be optimized. As shown in Fig. 7, the naïve implementation is far from the theoretical roof, so the easiest way to optimize this implementation is to increase data locality, which reduces the data transfer time and practically moves the implementation up in the roofline model. The optimized version is called the "blocked" implementation, which is explained in Algorithm 12, a two-level blocked matrix multiplication scheme in $GF(2^n)$. The first level blocking creates tiles in the BRAMs on the FPGA and stores data from the off-chip memory, which reduces the expensive off-chip memory access. The second level blocking is register blocking. Since the BRAM has limited read-write ports, which stalls the FPGA pipeline, we read part of the data in the BRAM to the registers and then perform the computation. Because we can create multiple read-write ports in the register-based memory blocks, pipeline stalls are avoided. Notice that if the naïve implementation is already close to the roof, the most efficient optimization will be moving the performance diagonally in the roofline model, which means increasing the data locality while increasing the OI.

Following the setup of the previous example, we use a fixed $\epsilon = 0.5$ and $\epsilon_o = 0.2$ and vary the value for n, pk, M and F in our experiment. Table V summarizes the theoretical bounds and the implementation testing results for selected designs. Fig. 7 plots the implementations in the corresponding roofline models, showing the performance increase from the naïve implementation to the blocked implementation. The resource consumption of eight basic FPGA implementations for the naïve version and the blocked version is shown in Table IV. Notice that the blocked implementations consume more logic resources (LUTs and FFs) than the naïve implementation but consume fewer block RAMs. It is due to the fact that the blocked implementation has more pipeline stages, so more registers need to be configured to accommodate the tiles. The blocked implementation is a compact and efficient design with more parallelism achieved. Since the naïve implementation caches the two input matrices on the FPGA before computation, it consumes more block RAMs than the blocked implementation, which caches the tiles dynamically. Also, we notice that the GOPS/Slice increases as the field size decreases in all implementations. It is expected as explained in the previous discussions, that multipliers in smaller field sizes have a better area/time product. Four BATS designs in different finite fields are tested for encoder and recoder

TABLE IV

RESOURCE CONSUMPTION FOR THE TWO IMPLEMENTATIONS IN DIFFERENT FINITE FIELDS ($n = 1, 2, 4, 8$) ON CYCLONE V 5CSEBA6. UNIT OF RAM: KBYTE. UNIT OF F_{max}: MHZ. GOPS/SLICE: AVERAGE GOPS ACHIEVED BY CORRESPONDING DESIGNS IN TABLE V DIVIDED BY THE TOTAL NUMBER OF SLICES USED. THE BEST DESIGN IN TERMS OF GOPS/SLICE IS HIGHLIGHTED IN BOLD.

design	LUT	FF	RAM	Slice	GOPS/Slice	Fmax	design	LUT	FF	RAM	Slice	GOPS/Slice	Fmax
naïve,n=8	17,022	38,446	302.6	20,974	4.76×10^{-9}	129	blocked,n=8	49,402	68,787	178.8	39,645	1.026×10^{-4}	106.6
naïve,n=4	16,911	37,444	293.6	20,522	4.54×10^{-9}	116.1	blocked,n=4	26,871	43,020	109.2	24,222	1.745×10^{-4}	118.8
naïve,n=2	16,778	37,813	289.1	20,648	7.65×10^{-9}	108.8	blocked,n=2	22,924	36,104	88.1	20,453	2.780×10^{-4}	115.6
naïve,n=1	17,064	38,361	302.6	20,938	21.77×10^{-9}	121.8	**blocked,n=1**	**30,609**	**43,012**	**101.9**	**24,352**	$\mathbf{10.512 \times 10^{-4}}$	**112.1**

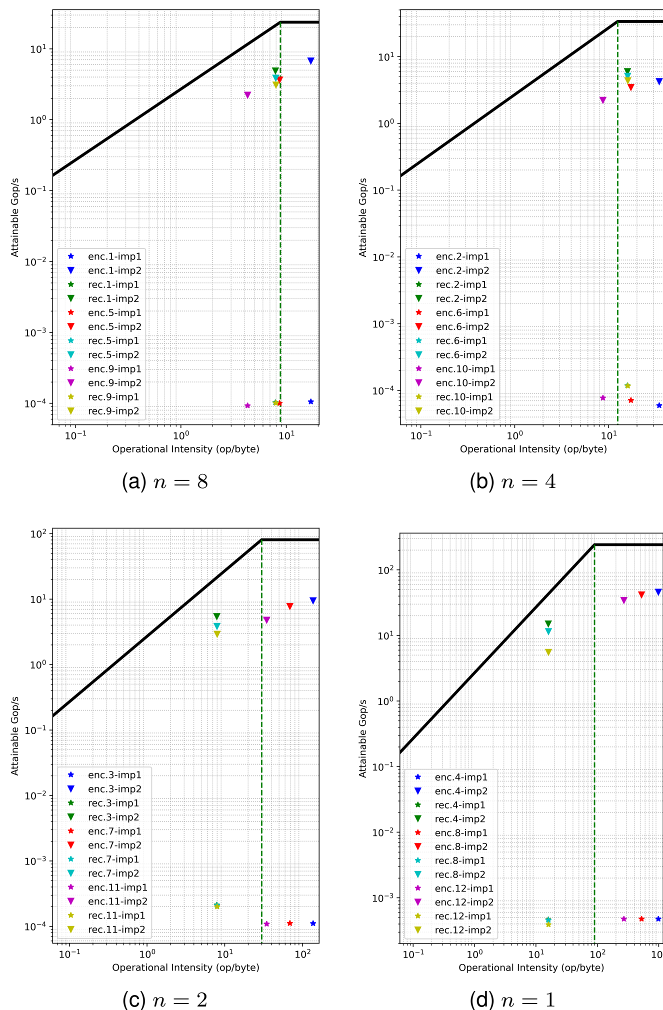

Fig. 7. Roofline models based on Cyclone V 5CSEBA6 for $n = 8, 4, 2, 1$. The achieved performance of designs corresponding to Table V are shown in the roofline model. The naïve implementations are indicated by '$\star$' and the blocked implementation are indicated by '$\bigtriangledown$'.

TABLE V

BATS DESIGNS WITH THEORETICAL AND IMPLEMENTATION PERFORMANCE. *THEORETICAL. ^{1}NAÏVE IMPLEMENTATION. ^{2}BLOCKED IMPLEMENTATION. t: LATENCY (MS). $Thr.$: THROUGHPUT (MBPS). TARGET PLATFORM: CYCLONE 5CSEBA6.

design	n	pk	M	$t^{\star}/Thr.^{\star}$	$t^{1}/Thr.^{1}$	$t^{2}/Thr.^{2}$
enc.1	8	1024	16	0.11/4.5k	24.8/20.1	0.388/1.28k
enc.2	4	2048	16	0.156/3.2k	88.4/5.65	1.24/403
enc.3	2	2048	4	0.253/1.97k	185/2.7	2.16/231
enc.4	1	1024	4	0.667/749	339/1.47	3.53/141
rec.1	8	1024	16	0.061/9.8k	12.8/46.8	0.268/2.23k
rec.2	4	2048	16	0.078/7.6k	22.2/27	0.438/1.36k
rec.3	2	2048	4	0.0592/10.1k	6.01/99	0.236/2.54k
rec.4	1	1024	4	0.0585/10.2k	5.44/110	0.168/3.57k

(a) $F = 64$ Kbytes

design	n	pk	M	$t^{\star}/Thr.^{\star}$	$t^{1}/Thr.^{1}$	$t^{2}/Thr.^{2}$
enc.5	8	1024	16	0.033/7.4k	7.83/31.9	0.217/1.15k
enc.6	4	2048	16	0.046/5.4k	22.2/11.2	0.453/551
enc.7	2	2048	4	0.065/3.8k	47.1/5.3	0.677/369
enc.8	1	1024	4	0.166/1.4k	84.8/2.9	0.97/257
rec.5	8	1024	16	0.036/8.3k	7.74/38.7	0.204/1.47k
rec.6	4	2048	16	0.046/6.4k	13.4/22.3	0.309/970
rec.7	2	2048	4	0.03/10k	3.13/95.8	0.171/1.75k
rec.8	1	1024	4	0.029/10.3k	2.77/108	0.11/2.72k

(b) $F = 32$ Kbytes

design	n	pk	M	$t^{\star}/Thr.^{\star}$	$t^{1}/Thr.^{1}$	$t^{2}/Thr.^{2}$
enc.9	8	1024	16	0.022/5.6k	2.82/44.3	0.118/1.05k
enc.10	4	2048	16	0.022/5.6k	6.8/18.3	0.236/529
enc.11	2	2048	4	0.016/7.7k	12.1/10.3	0.275/454
enc.12	1	1024	4	0.042/2.9k	21.5/5.81	0.299/418
rec.9	8	1024	16	0.024/6.1k	5.19/28.9	0.17/882
rec.10	4	2048	16	0.031/4.8k	8.97/16.7	0.24/625
rec.11	2	2048	4	0.015/10k	1.64/91.4	0.113/1.32k
rec.12	1	1024	4	0.014/10.1k	1.64/91.4	0.115/1.30k

(c) $F = 16$ Kbytes

respectively, under various file sizes. We can see that under all conditions, the performance is significantly improved from the naïve implementation to the blocked implementation. The performance of the latter is comparably close to the theoretical bound, which indicates that the theoretical performance bound is practical and can be approached by improving the hardware design. The optimization priority is also suggested by the roofline model as discussed above.

Notice that most designs using blocked implementations are within a factor of 10 away from the roof, yet, many more optimization techniques can be applied. For example, we can exploit the full memory bandwidth using SIMD or a better pipeline with a systolic-array structure [53]. This shows that the design paradigm proposed in Section V indeed provides useful guidance for achieving good hardware designs. Since the proposed theoretical bound is practical and achievable, it is justified to use the theoretical bound as a selection criterion in the design paradigm.

VII. CONCLUSION AND DISCUSSION

In this paper, we studied implementations of BATS with consideration of the finite field multiplier using a roofline model. We proposed a finite field modified roofline model

which was then used to analyze the maximum throughput and minimum latency of a BATS code on FPGA. Finally we connected important BATS code design parameters to hardware performance through Operational Intensity and the roofline model. Three equations are derived, (9), (12) and (20), which quantify the attainable hardware performance in terms of the BATS code parameters. We also propose a design paradigm for finding the best BATS code implementation on FPGA based on our equations without intensively searching the design space. Results from FPGA implementations were used to verify our theory. The proposed design paradigm can be easily extended to any hardware platform and it can even be adapted to other linear codes.

Even though the roofline model provides insightful guidance for optimization, it is not perfect. When calculating the attainable computational power of the platform, we assume that matrix multiplication is perfectly scalable. However, in actual implementations this is not true in general. Nevertheless, this simplified model provides strong insight into how a hardware implementation of BATS should be optimised.

The actual implementation of the decoding is omitted in Table V for two reasons. As asserted in Proposition 5, decoding shares the same optimality with encoding. Therefore, the most suitable design for encoding would also be the optimal design for decoding. Secondly, due to the difficulty in analysing complexity of the decoding algorithm, how to find efficient and highly parallel architectures for decoding is still an open problem. Even though hardware designs for BP decoding for other applications like the polar code [59] and LDPC code [60] have been proposed, their direct application to BATS is neither obvious nor optimal.

To the best of our knowledge, this is the first paper that investigates the BATS code from a hardware perspective. We believe that as applications of the BATS code are emerging, efficient hardware implementations will be crucial for the next generation of wireless communication infrastructure.

References

[1] D. C. Salyers, A. D. Striegel, and C. Poellabauer, “Wireless reliability: Rethinking 802.11 packet loss,” in *2008 International Symposium on a World of Wireless, Mobile and Multimedia Networks*. IEEE, 2008, pp. 1–4.

[2] E. Tanghe, W. Joseph, L. Verloock, and L. Martens, “Evaluation of vehicle penetration loss at wireless communication frequencies,” *IEEE transactions on vehicular technology*, vol. 57, no. 4, pp. 2036–2041, 2008.

[3] Y. Tian, K. Xu, and N. Ansari, “TCP in wireless environments: problems and solutions,” *IEEE Communications Magazine*, vol. 43, no. 3, pp. S27–S32, 2005.

[4] S. Yang and R. W. Yeung, “Batched sparse codes,” *IEEE Transactions on Information Theory*, vol. 60, no. 9, pp. 5322–5346, 2014.

[5] R. Ahlswede, N. Cai, S.-Y. R. Li, and R. W. Yeung, “Network information flow,” *IEEE Trans. Inform. Theory*, vol. 46, no. 4, pp. 1204–1216, Jul. 2000.

[6] S.-Y. R. Li, R. W. Yeung, and N. Cai, “Linear network coding,” *IEEE Trans. Inform. Theory*, vol. 49, no. 2, pp. 371–381, Feb. 2003.

[7] A. F. Dana, R. Gowaikar, R. Palanki, B. Hassibi, and M. Effros, “Capacity of wireless erasure networks,” *IEEE Trans. Inform. Theory*, vol. 52, no. 3, pp. 789–804, Mar. 2006.

[8] D. S. Lun, M. Médard, R. Koetter, and M. Effros, “On coding for reliable communication over packet networks,” *Physical Communication*, vol. 1, no. 1, pp. 3–20, Mar. 2008.

[9] T. Kanematsu, K. Sanada, Z. Li, T. Pei, Y.-J. Choi, K. Nguyen, and H. Sekiya, “Throughput and delay analysis for ieee 802.11 multi-hop networks considering data rate,” *International Journal of Distributed Sensor Networks*, vol. 16, no. 9, p. 1550147720959262, 2020.

[10] X. Zhang, H. Jiang, L. Zhang, C. Zhang, Z. Wang, and X. Chen, “An energy-efficient asic for wireless body sensor networks in medical applications,” *IEEE transactions on biomedical circuits and systems*, vol. 4, no. 1, pp. 11–18, 2009.

[11] F. Karray, M. W. Jmal, M. Abid, M. S. BenSaleh, and A. M. Obeid, “A review on wireless sensor node architectures,” in *2014 9th International Symposium on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC)*. IEEE, 2014, pp. 1–8.

[12] I. Zhang, J. Liu, A. Austin, M. L. Roberts, and A. Badam, “I'm not dead yet! the role of the operating system in a kernel-bypass era,” in *Proceedings of the Workshop on Hot Topics in Operating Systems*, 2019, pp. 73–80.

[13] R. Chen and G. Sun, “A survey of kernel-bypass techniques in network stack,” in *Proceedings of the 2018 2nd International Conference on Computer Science and Artificial Intelligence*, 2018, pp. 474–477.

[14] V. Chamola, S. Patra, N. Kumar, and M. Guizani, “Fpga for 5g: Re-configurable hardware for next generation communication,” *IEEE Wireless Communications*, vol. 27, no. 3, pp. 140–147, 2020.

[15] J. Nadal and A. Baghdadi, “Parallel and flexible 5g ldpc decoder architecture targeting fpga,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 29, no. 6, pp. 1141–1151, 2021.

[16] M. Vestias and H. Neto, “Trends of cpu, gpu and fpga for high-performance computing,” in *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2014, pp. 1–6.

[17] M. Huang, D. Wu, C. H. Yu, Z. Fang, M. Interlandi, T. Condie, and J. Cong, “Programming and runtime support to blaze fpga accelerator deployment at datacenter scale,” in *Proceedings of the Seventh ACM Symposium on Cloud Computing*, 2016, pp. 456–469.

[18] S. Yang and R. W. Yeung, “Coding for a network coded fountain,” in *2011 IEEE International Symposium on Information Theory Proceedings*. IEEE, 2011, pp. 2647–2651.

[19] S. Yang and Q. Zhou, “Tree analysis of bats codes,” *IEEE Communications Letters*, vol. 20, no. 1, pp. 37–40, 2015.

[20] S. Yang, T.-C. Ng, and R. W. Yeung, “Finite-length analysis of bats codes,” *IEEE Transactions on Information Theory*, vol. 64, no. 1, pp. 322–348, 2017.

[21] X. Xu, Y. Zeng, Y. L. Guan, and L. Yuan, “Expanding-window bats code for scalable video multicasting over erasure networks,” *IEEE Transactions on Multimedia*, vol. 20, no. 2, pp. 271–281, 2017.

[22] J. Yang, Z. Shi, and J. Ji, “Design of improved expanding-window bats codes,” *IEEE Transactions on Vehicular Technology*, vol. 71, no. 3, pp. 2874–2886, 2021.

[23] X. Xu, Y. Zeng, Y. L. Guan, and L. Yuan, “Bats code with unequal error protection,” in *2016 IEEE International Conference on Communication Systems (ICCS)*. IEEE, 2016, pp. 1–6.

[24] H. H. Yin, S. Yang, Q. Zhou, and L. M. Yung, “Adaptive recoding for bats codes,” in *2016 IEEE International Symposium on Information Theory (ISIT)*. IEEE, 2016, pp. 2349–2353.

[25] Z. Zhou, C. Li, S. Yang, and X. Guang, “Practical inner codes for bats codes in multi-hop wireless networks,” *IEEE Transactions on Vehicular Technology*, vol. 68, no. 3, pp. 2751–2762, 2019.

[26] X. Xu, Y. L. Guan, Y. Zeng, and C.-C. Chui, “Quasi-universal bats code,” *IEEE Transactions on Vehicular Technology*, vol. 66, no. 4, pp. 3497–3501, 2016.

[27] T. Ho, M. Médard, R. Koetter, D. R. Karger, M. Effros, J. Shi, and B. Leong, “A random linear network coding approach to multicast,” *IEEE Transactions on Information Theory*, vol. 52, no. 10, pp. 4413–4430, 2006.

[28] D. J. MacKay, “Fountain codes,” *IEE Proceedings-Communications*, vol. 152, no. 6, pp. 1062–1068, 2005.

[29] S. Yang and R. W. Yeung, “BATS codes: Theory and practice,” *Synthesis Lectures on Communication Networks*, vol. 10, no. 2, pp. 1–226, 2017.

[30] R. Tanner, “A recursive approach to low complexity codes,” *IEEE Transactions on information theory*, vol. 27, no. 5, pp. 533–547, 1981.

[31] X. Xu, M. S. G. P. Kumar, Y. L. Guan, and P. H. J. Chong, “Two-phase cooperative broadcasting based on batched network code,” *IEEE Transactions on Communications*, vol. 64, no. 2, pp. 706–714, 2015.

[32] X. Xu, Y. L. Guan, and Y. Zeng, “Batched network coding with adaptive recoding for multi-hop erasure channels with memory,” *IEEE Transactions on Communications*, vol. 66, no. 3, pp. 1042–1052, 2017.

[33] H. Zhao, S. Yang, and G. Feng, "Fast degree-distribution optimization for BATS codes," *Science China Information Sciences*, vol. 60, no. 10, pp. 1–15, 2017.

[34] Z. Zhou, C. Li, S. Yang, and X. Guang, "Practical inner codes for bats codes in multi-hop wireless networks," *IEEE Transactions on Vehicular Technology*, vol. 68, no. 3, pp. 2751–2762, 2019.

[35] F. Restuccia, A. Biondi, M. Marinoni, G. Cicero, and G. Buttazzo, "Axi hyperconnect: a predictable, hypervisor-level interconnect for hardware accelerators in fpga soc," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2020, pp. 1–6.

[36] J. Vasiljevic, R. Wittig, P. Schumacher, J. Fifield, F. M. Vallina, H. Styles, and P. Chow, "Opencl library of stream memory components targeting fpgas," in *2015 international conference on field programmable technology (FPT)*. IEEE, 2015, pp. 104–111.

[37] S. Williams, A. Waterman, and D. Patterson, "Roofline: an insightful visual performance model for multicore architectures," *Communications of the ACM*, vol. 52, no. 4, pp. 65–76, 2009.

[38] *Cyclone V Device Datasheet*, 2019th ed., Intel, 11 2019.

[39] R. Gallager, "Low-density parity-check codes," *IRE Transactions on information theory*, vol. 8, no. 1, pp. 21–28, 1962.

[40] G. L. Mullen and D. Panario, *Handbook of finite fields*. CRC Press Boca Raton, 2013, vol. 17.

[41] H. Fan and M. A. Hasan, "A survey of some recent bit-parallel GF(2^n) multipliers," *Finite Fields and Their Applications*, vol. 32, pp. 5–43, 2015.

[42] D. Wells, *The Penguin Dictionary of Curious and Interesting Numbers*, 1st ed. Penguin Books, 1986.

[43] *Arria 10 Device Datasheet*, 2020th ed., Intel, 06 2020.

[44] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, "High-level synthesis for fpgas: From prototyping to deployment," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 4, pp. 473–491, 2011.

[45] C. Y. Lin, H. K.-H. So, and P. H. Leong, "A model for matrix multiplication performance on fpgas," in *2011 21st International Conference on Field Programmable Logic and Applications*. IEEE, 2011, pp. 305–310.

[46] J. Håstad, R. Impagliazzo, L. A. Levin, and M. Luby, "A pseudorandom generator from any one-way function," *SIAM Journal on Computing*, vol. 28, no. 4, pp. 1364–1396, 1999.

[47] L. Blum, M. Blum, and M. Shub, "A simple unpredictable pseudo-random number generator," *SIAM Journal on computing*, vol. 15, no. 2, pp. 364–383, 1986.

[48] K. H. Tsoi, K. H. Leung, and P. H. W. Leong, "Compact fpga-based true and pseudo random number generators," in *11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, 2003. FCCM 2003.* IEEE, 2003, pp. 51–61.

[49] M. Garcia-Bosque, A. Pérez-Resa, C. Sánchez-Azqueta, C. Aldea, and S. Celma, "Chaos-based bitwise dynamical pseudorandom number generator on fpga," *IEEE Transactions on Instrumentation and Measurement*, vol. 68, no. 1, pp. 291–293, 2018.

[50] M. D. Lam, E. E. Rothberg, and M. E. Wolf, "The cache performance and optimizations of blocked algorithms," *ACM SIGOPS Operating Systems Review*, vol. 25, no. Special Issue, pp. 63–74, 1991.

[51] H. Li and Q. Huan-yan, "Parallelized network coding with simd instruction sets," in *2008 International Symposium on Computer Science and Computational Technology*, vol. 1. IEEE, 2008, pp. 364–369.

[52] J. Fowers, K. Ovtcharov, M. Papamichael, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, L. Adams, M. Ghandi *et al.*, "A configurable cloud-scale dnn processor for real-time ai," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2018, pp. 1–14.

[53] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the 44th annual international symposium on computer architecture*, 2017, pp. 1–12.

[54] J. Nurmi, *Processor design: system-on-chip computing for ASICs and FPGAs*. Springer Science & Business Media, 2007.

[55] F. Lázaro, G. Liva, and G. Bauch, "Inactivation decoding of lt and raptor codes: Analysis and code design," *IEEE Transactions on Communications*, vol. 65, no. 10, pp. 4114–4127, 2017.

[56] S. Che, J. Li, J. W. Sheaffer, K. Skadron, and J. Lach, "Accelerating compute-intensive applications with gpus and fpgas," in *2008 Symposium on Application Specific Processors*. IEEE, 2008, pp. 101–107.

[57] C. Nugteren, "Clblast: A tuned opencl blas library," in *Proceedings of the International Workshop on OpenCL*, 2018, pp. 1–10.

[58] S. Yang and R. W. Yeung, "Network communication protocol design from the perspective of batched network coding," *IEEE Communications Magazine*, vol. 60, no. 1, pp. 89–93, 2022.

[59] A. Pamuk, "An fpga implementation architecture for decoding of polar codes," in *2011 8th International symposium on wireless communication systems*. IEEE, 2011, pp. 437–441.

[60] S. Mhaske, H. Kee, T. Ly, A. Aziz, and P. Spasojevic, "High-throughput fpga-based qc-ldpc decoder architecture," in *2015 IEEE 82nd Vehicular Technology Conference (VTC2015-Fall)*. IEEE, 2015, pp. 1–5.

Jiaxin Qing received a B.E. degree in electrical engineering from the University of Sydney, Australia, in 2020. He is currently working toward a Ph.D. degree in information engineering at the Chinese University of Hong Kong, China. His research interests include communication system, deep learning, and reconfigurable computing.

Philip H. W. Leong (Senior Member, IEEE) Biography text here.

Raymond W. Yeung (Fellow, IEEE) Biography text here.