Vegapunk: Accurate and Fast Decoding for Quantum LDPC Codes with Online Hierarchical Algorithm and Sparse Accelerator

Kaiwen Zhou
Zhejiang University
Hangzhou, China
kaiwenzhou@zju.edu.cn

Liqiang Lu*
ZJU-Ningbo Global Innovation Center
Zhejiang University
Ningbo, China
liqianglu@zju.edu.cn

Debin Xiang
Zhejiang University
Hangzhou, China
db.xiang@zju.edu.cn

Chenning Tao
Zhejiang University
Hangzhou, China
tcn@zju.edu.cn

Anbang Wu
Shanghai Jiao Tong University
Shanghai, China
anbang@cs.sjtu.edu.cn

Jingwen Leng
Shanghai Jiao Tong University
Shanghai, China
leng-jw@sjtu.edu.cn

Fangxin Liu
Shanghai Jiao Tong University
Shanghai, China
liufangxin@sjtu.edu.cn

Mingshuai Chen
Zhejiang University
Hangzhou, China
m.chen@zju.edu.cn

Jianwei Yin*
ZJU-Ningbo Global Innovation Center
Zhejiang University
Ningbo, China
zjuyjw@zju.edu.cn

Abstract

Quantum Low-Density Parity-Check (qLDPC) codes are a promising class of quantum error-correcting codes that exhibit constant-rate encoding and high error thresholds, thereby facilitating scalable fault-tolerant quantum computation. However, real-time decoding of qLDPC codes remains a significant challenge due to the high connectivity of their check matrices, which typically requires solving large-scale linear systems with sparse structures. In particular, off-the-shelf qLDPC decoders are often subject to a tradeoff between accuracy and latency, thus yielding no accurate and real-time decoding. This paper presents Vegapunk, a software-hardware co-design framework that enables real-time qLDPC decoding with high accuracy. To improve decoding accuracy, we design an offline decoupling strategy leveraging Satisfiability Modulo Theories (SMT) optimizations to mitigate quantum degeneracy. To enable fast decoding, we introduce an online hierarchical decoding algorithm employing a greedy strategy. Furthermore, we show that our SMT-optimized strategy suffices to produce decoupled matrices with maximized sparsity, thus admitting a dedicated accelerator to fully exploit the sparsity and parallelism to achieve real-time qLDPC decoding. Experimental results demonstrate that Vegapunk enables real-time decoding ($< 1\mu s$) for the Bivariate Bicycle (BB) code up to [[784,24,24]] while exhibiting logical error rates on par with the state-of-the-art decoder, i.e., BP+OSD.

*Corresponding Author

MICRO '25, Seoul, Republic of Korea

ACM ISBN 979-8-4007-1573-0/25/10
https://doi.org/10.1145/3725843.3756084

CCS Concepts

• **Hardware → Quantum error correction and fault tolerance**; • **Computer systems organization → Real-time system architecture**.

Keywords

Quantum error correction, qLDPC codes, real-time decoding

ACM Reference Format:
Kaiwen Zhou, Liqiang Lu, Debin Xiang, Chenning Tao, Anbang Wu, Jingwen Leng, Fangxin Liu, Mingshuai Chen, and Jianwei Yin. 2025. Vegapunk: Accurate and Fast Decoding for Quantum LDPC Codes with Online Hierarchical Algorithm and Sparse Accelerator. In *58th IEEE/ACM International Symposium on Microarchitecture (MICRO '25), October 18–22, 2025, Seoul, Republic of Korea.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3725843.3756084

1 Introduction

Quantum Error Correction (QEC) is essential for realizing large-scale, fault-tolerant quantum computing [14, 18]. Among various QEC codes, the surface code has been extensively studied and experimentally implemented due to its high noise threshold and locality-friendly structure [1, 2, 16, 17, 25]. However, the surface code suffers from poor scalability: it requires hundreds or even thousands of physical qubits to encode a single logical qubit, resulting in substantial overhead that limits the feasibility of building large quantum systems [20, 29, 44]. Quantum Low-Density Parity-Check (qLDPC) codes provide a promising alternative. The qLDPC code is defined by a sparse check matrix, where each row corresponds to a stabilizer and each column to a data qubit. A '1' in the check matrix indicates that the associated stabilizer acts on that qubit. This sparse check matrix enables an asymptotically constant encoding rate, making qLDPC codes more resource-efficient than the surface code [7, 38]. Notably, IBM recently introduced the Bivariate Bicycle (BB) code [6] – a family of qLDPC codes achieving a high error threshold of 0.7%

and 10× saving in physical qubit overhead compared to the surface code. This demonstrates the potential of qLDPC codes as a scalable architectural solution for superconducting platforms.

An accurate and real-time decoder plays a pivotal role in QEC. QEC involves a tight interplay between quantum computers and classical decoding. On the quantum side, parity qubits periodically extract error syndromes by interacting with nearby data qubits, forming discrete Pauli error signatures at the end of each QEC cycle. On the classical side, a decoder uses these syndromes to infer and correct errors on the data qubits before the next QEC round. The decoder must be accurate, as any incorrect correction can induce logical errors that propagate through the quantum circuit, ultimately leading to computational failure. Meanwhile, the decoder must operate in real-time. If decoding lags behind syndrome generation, the system accumulates a *syndrome backlog*, where unprocessed error data delays subsequent quantum operations [23, 41]. This latency overhead scales exponentially with the circuit depth.

The 2D-grid structure of surface codes allows for various real-time decoders, e.g., [3, 43, 46], based on the efficient Minimum-Weight Perfect Matching (MWPM) algorithm [47]. However, these MWPM-based decoders cannot be directly applied to qLDPC codes due to structural differences. In qLDPC codes, each data qubit is connected to more than two parity qubits, creating a hypergraph structure rather than a simple graph. This increased complexity renders the MWPM algorithm ineffective, as matching in hypergraphs is an NP-hard problem [10]. As a result, decoding algorithms for qLDPC codes differ significantly from those for surface codes. The most representative decoding algorithms for qLDPC codes are Belief Propagation (BP) [34] and Belief Propagation with Ordered Statistics Decoding (BP+OSD) [15]. BP is a message-passing algorithm that iteratively approximates the error probabilities over the Tanner graph [48, 49]. It approximates the error probabilities of data qubits, refining these estimates with each iteration, ideally converging to the most likely error pattern. If BP fails to converge, BP+OSD invokes OSD to post-process the decoding result. OSD first ranks physical qubits based on their error probabilities from BP. It then selects a subset of qubits, assumes a likely error pattern, and solves a constrained linear system to test if the pattern matches the observed syndrome. This process is repeated across many candidate patterns to find the most likely error pattern.

Nevertheless, BP and BP+OSD fail to achieve both high accuracy and low latency. BP is promising for real-time decoding due to its low complexity and high parallelism, making it suitable for hardware such as FPGAs and ASICs [28, 42]. However, BP suffers from low accuracy due to *quantum degeneracy*, where distinct error patterns produce the same syndrome [27, 48]. This issue arises because the number of columns in the check matrix far exceeds the number of rows. BP does not account for this degeneracy, which often converges to incorrect solutions. Although BP+OSD improves accuracy by using OSD, it incurs substantial latency due to sequential operations such as linear system solving. These costly operations make BP+OSD impractical for achieving real-time decoding. The trade-off between accuracy and latency underscores the urgent need for a practical qLDPC decoder to bridge this gap.

In this paper, we present Vegapunk, a software-hardware co-designed decoding framework that achieves both accurate and fast qLDPC decoding. To improve decoding accuracy, we propose an *offline decoupling strategy* that pre-processes the original wide check matrix into smaller submatrices to mitigate quantum degeneracy. The challenge here is to ensure that the decoupling not only reduces computational complexity but also enhances the decoding accuracy. To achieve this, we cast the matrix decoupling as a Satisfiability Modulo Theories (SMT) optimization task. The goal is to find a row- and column-transformed check matrix with a diagonal block structure and a sparse off-diagonal matrix. Importantly, this matrix transformation depends only on the qLDPC code structure and can thus be pre-computed and stored for fast online decoding.

To enable fast decoding, we propose an *online hierarchical decoding algorithm*. While the diagonal blocks are suitable for parallel decoding, the remaining off-diagonal matrix introduces dependency that makes direct parallelism difficult. To address this, we treat the decoding problem as the sum of two parts: the *left error*, associated with the diagonal blocks, and the *right error*, linked to the off-diagonal matrix. By handling the right error first and then iteratively identifying the left error using a greedy decoding method, we reduce interdependencies and enable fast convergence. This process continues until an optimal solution is found or the maximum iteration limit is reached.

To achieve real-time decoding of qLDPC codes, we design the Vegapunk *hardware accelerator* to fully exploit the sparsity and parallelism inherent in our decoding algorithm. Given the sparse and parallel nature of our online hierarchical decoding, we design the syndrome incremental update unit to effectively compute the residual syndrome by skipping redundant operations. Furthermore, we introduce the greedy decoding cores to decode the left error in parallel. Each greedy decoding core comprises the syndrome incremental update unit, the log-likelihood ratio compute unit, and a comparator tree to efficiently identify the most probable errors.

Our main contributions are as follows:

- We present Vegapunk, a software-hardware co-design framework that enables accurate and real-time qLDPC decoding.
- We introduce an offline decoupling strategy to improve decoding accuracy by decomposing the original check matrix into smaller matrices. By formulating this as an SMT problem, we maximize the sparsity of decoupled matrices, thereby facilitating efficient hardware acceleration.
- We propose an online hierarchical decoding algorithm to realize fast decoding, leveraging a greedy strategy to decode the error in parallel. We further design the Vegapunk accelerator to exploit the sparsity and parallelism of our decoding algorithm, yielding real-time qLDPC decoding.

Experimental results show that Vegapunk achieves a worst-case decoding latency of 840*ns* for the BB code [[784,24,24]], maintaining logical error rates comparable to the state-of-the-art decoder, BP+OSD. The low overheads of Vegapunk demonstrate its practicality for fault-tolerant quantum computing.

2 Background

2.1 Quantum Error Correction Process

Quantum Error Correction (QEC) encodes a logical qubit using multiple physical qubits, including data qubits and parity qubits. The data qubits store the quantum information, while parity qubits detect X and Z errors on the data qubits [8, 13, 40]. Figure 1 (a)

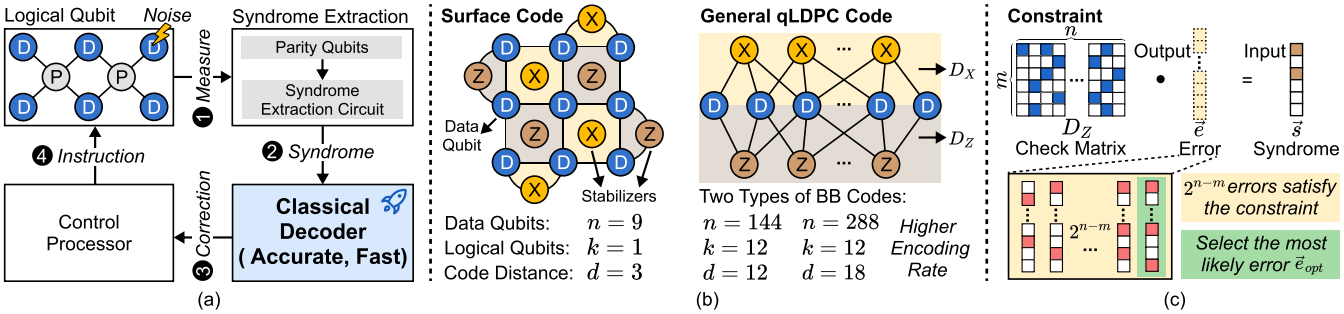


Figure 1: (a) Overview of the quantum error correction process. (b) The code parameters $[[n, k, d]]$ and structure of the surface code and general qLDPC code, respectively. (c) The check constraint of decoding qLDPC codes. The error pattern should satisfy the constraint given the input syndrome. We select the most likely error as the output.

illustrates the QEC process, which mainly consists of four steps that are executed in cycles to protect the logical qubit from errors caused by noise: ❶ The syndrome extraction circuit is executed on parity qubits to generate syndrome. The parity qubits interact with their neighboring data qubits to extract their error information. After the syndrome is obtained, a bitstring where '1' indicates a failed parity check, implying an error in the neighboring data qubit. ❷ The syndrome is sent to a classical decoder, which identifies likely error locations on the data qubits. A good decoder must be fast and accurate, preventing error accumulation or backlog and precisely identifying all errors [11]. ❸ The decoder computes a logical correction and sends it to the control processor. ❹ The control processor receives the correction, updates its operations, and sends QEC instructions to correct logical errors. After that, it initiates the next round of syndrome extraction.

2.2 Basics of qLDPC Code

Like other QEC codes, qLDPC codes are characterized by three key parameters: (1) the number of data qubits n in a logical block, (2) the number of encoded logical qubits k, and (3) the code distance d, which quantifies the minimum number of physical qubit errors required to cause a logical error. These three properties can be denoted as $[[n, k, d]]$ to describe a QEC code. As illustrated on the left side of Figure 1 (b), the surface code $[[9, 1, 3]]$ encodes one logical qubit using nine data qubits, with a code distance of 3. A larger code distance d indicates better error tolerance. At least $n-k$ ancilla qubits are required to perform parity checks on data qubits.

To better understand the characteristics of qLDPC codes, we compare them with surface codes regarding connectivity, encoding rate, and decoding complexity: **(1) Connectivity.** Surface codes, defined by the 2D lattice structure, exhibit low and local connectivity (degree-4 for each stabilizer). As depicted in Figure 1 (b), each stabilizer checks a small, fixed number of nearby qubits (typically four), which often simplifies the decoding process. In contrast, qLDPC codes feature higher and non-local connectivity. In this work, we focus on CSS qLDPC codes, which are defined by two check matrices D_X and D_Z, representing the connections between data qubits and X-type, Z-type stabilizers, respectively. The non-local connectivity in qLDPC codes enables high encoding rates and large code distances but complicates decoding, since decoders must manage

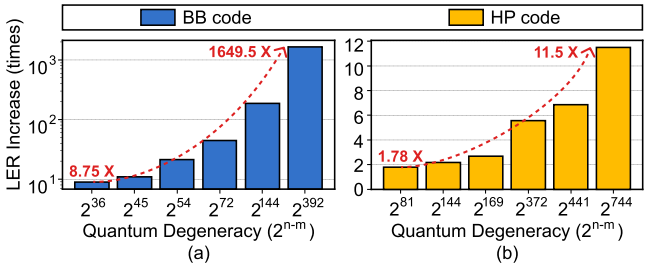


Figure 2: LER increase due to quantum degeneracy, under 0.1% circuit-level noise. (a) BB codes, from [[72,12,6]] to [[784,24,24]]. (b) HP codes, from [[162,2,4]] to [[1488,30,7]].

information propagation over long-range connections; **(2) Encoding Rate.** The encoding rate is defined as k/n, with a higher value indicating fewer physical qubits required for the same number of logical qubits. Surface codes typically exhibit a low encoding rate (e.g., $1/n$), making them resource-intensive. For instance, the surface code $[[1452,12,11]]$ requires 1452 data qubits to encode 12 logical qubits. In contrast, the Bivariate Bicycle (BB) code [6] $[[144,12,12]]$, a representative qLDPC code, achieves the same number of logical qubits using only 144 data qubits; **(3) Decoding Complexity.** The 2D-grid structure of surface codes enables efficient decoding using polynomial-time algorithms such as MWPM [47]. In contrast, the sparse yet irregular connectivity of qLDPC codes leads to decoding being NP-hard, since even the minimum weight decoding problem is computationally intractable in general [4, 24].

2.3 qLDPC Decoding

In a CSS qLDPC code, the check matrix D is composed of two parts: D_X, which represents X-type parity checks, and D_Z, which represents Z-type parity checks [31]. These two sets of stabilizers must commute, which indicates the constraint $D_X \cdot D_Z^T = 0$. The X and Z errors can be decoded separately using D_Z and D_X, respectively. Both D_Z and D_X consist of m parity checks, where each parity check corresponds to an n-length vector, where n is the number of data qubits. This vector indicates the specific data qubits to which the parity qubit is connected. Due to the high encoding rate of qLDPC codes, the number of columns n significantly exceeds the number of rows m in the check matrix, with n being at least twice the magnitude of m.

The decoding problem for qLDPC codes can be formulated as a minimum weight decoding problem, which is NP-hard [24]. The objective is to find the most likely error pattern $\vec{e}_{opt}$ given the syndrome $\vec{s}$. The following constraint and objective function define the problem:

$$\vec{e}_{opt} = \arg\min_{\vec{e}} \sum_j w_j e_j$$
$$s.t.\ D_Z \cdot \vec{e} = \vec{s}$$

where $\vec{s} = (s_1, s_2, ..., s_m)^T$ represents the syndrome, corresponding to the measurement results of each parity qubit, $\vec{e} = (e_1, e_2, ..., e_n)^T$ is the error pattern, representing the errors in the data qubits. The weights $w_j = log\frac{1-p_j}{p_j}$ are the log-likelihood ratio (LLR) based on the prior probability p_j. In terms of decoding, a significant challenge is the phenomenon of *quantum degeneracy*, illustrated in Figure 1 (c). This refers to the existence of 2^{n-m} possible error patterns satisfying the same syndrome, which increases decoding complexity. For qLDPC codes, this issue is exacerbated due to their wider and sparser check matrices, leading to a much larger degenerate subspace and more ambiguous error candidates. To quantitatively explain the impact of quantum degeneracy on the Logical Error Rate (LER), we decode varying BB codes and HP codes using BP and BP+OSD (Section 2.4) and compute the increase in LER, as shown in Figure 2. Notably, BP does not consider degeneracy, whereas BP+OSD does. Due to the impact of degeneracy, the LER increases by an average of 320.3× and 5.1× on BB codes and HP codes, respectively. Furthermore, as $n - m$ increases (indicating more severe degeneracy), the LER increase becomes more significant.

2.4 BP and BP+OSD Decoder

Here, we introduce two most representative decoders for qLDPC codes, Belief Propagation (BP) [34] and BP with Ordered Statistics Decoding (BP+OSD) [15]. BP is an iterative algorithm that exchanges messages between nodes on a Tanner graph [26]. Each node updates its outgoing messages based on incoming messages, aiming to find the most probable error pattern given the syndrome. However, due to quantum degeneracy, BP often fails to converge, necessitating the execution of OSD. OSD first sorts the columns of the check matrix D based on the error probabilities of data qubits from BP. Then, it performs Gaussian elimination with a pivot selection rule that prioritizes the most reliable qubit. Finally, OSD searches for a solution by flipping combinations of the least reliable bits. Since iterating over exponential combinations is impractical, BP+OSD-0 and BP+OSD-CS(t) [39] are proposed to balance time complexity and accuracy. BP+OSD-0 outputs the hard-decision solution after Gaussian elimination, achieving suboptimal accuracy. BP+OSD-CS(t) tries all 1- and 2-bit flips among the t least reliable bits, achieving improved accuracy at the cost of increased latency.

3 Motivation

In this section, we identify the limitations of current qLDPC decoders, specifically BP [34] and BP+OSD [15], by evaluating their Logical Error Rate (LER) and latency. These limitations highlight the need to achieve accurate and real-time qLDPC decoding.

We evaluate the performance of BP and BP+OSD on Bivariate Bicycle (BB) codes [6] of varying sizes, from [[72,12,6]] to [[784,24,24]], using a circuit-level noise model with a physical error rate $p = 0.001$. BP is implemented on an Xilinx Alveo U50 FPGA at 250 MHz, utilizing the hardware architecture from [42], where each iteration takes two cycles. For BP+OSD, we choose BP+OSD-CS(7) version to achieve significantly improved decoding accuracy. We execute BP+OSD-CS(7) on an AMD EPYC 9554 64-core CPU, using the implementation from [39]. We now discuss the two key challenges uncovered by our experiments:

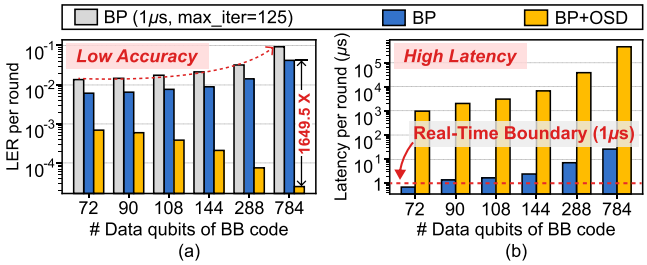


Figure 3: Motivation of Vegapunk. (a) LER (per round) of BP (1μs), BP, and BP+OSD, respectively. (b) Decoding latency (per round) of BP and BP+OSD, respectively. BP stops as soon as it converges.

Challenge 1: Low Accuracy. A good decoder should show a decrease in LER as code distance increases, which is achieved by BP+OSD as shown in Figure 3 (a). However, we find that the LER of BP gradually increases, indicating that its decoding is completely ineffective in practical scenarios [1]. Specifically, the LER of BP is 1649.5× higher than that of BP+OSD when decoding BB code [[784,24,24]]. Furthermore, when we limit the number of BP iterations to 125 to satisfy the 1μs time constraint, its LER increases further due to insufficient convergence.

Challenge 2: High Latency. As shown in Figure 3 (b), BP achieves real-time decoding only for the small BB code [[72,12,6]]; for larger codes, the decoding time exceeds the 1μs boundary. While BP can be parallelized and has a time complexity of $O(n)$, its high memory bandwidth requirements and sequential iterations lead to excessive latency when scaling to larger code sizes. BP+OSD, on the other hand, operates when BP fails to converge and requires additional costly steps such as sorting and matrix inversion. Even for the small BB code [[72,12,6]], BP+OSD still needs around $10^3\mu s$. Given the complex and sequential nature of OSD, it is unrealistic to design a dedicated accelerator for BP+OSD that meets the real-time latency requirement using current VLSI technologies [42].

Thus, these two challenges motivate the development of Vegapunk, a qLDPC decoder that strikes a balance between hardware efficiency and decoding accuracy. This design choice reflects a practical trade-off: Vegapunk can meet the real-time requirement (1μs) while achieving decoding accuracy comparable to BP+OSD, making it well-suited for superconducting platforms.

4 Vegapunk Algorithm

We propose an offline SMT-optimized decoupling to address the accuracy challenge (**Challenge 1**) caused by quantum degeneracy. By decomposing the check matrix D into smaller diagonal block matrices, we mitigate the quantum degeneracy that arises from a wide check matrix. To tackle the latency challenge (**Challenge 2**),

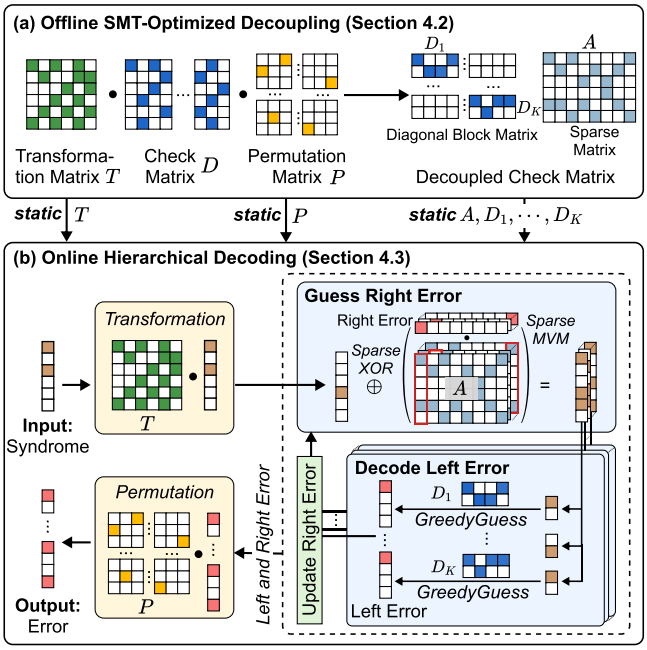


Figure 4: Overview of Vegapunk algorithm. (a) Applying the transformation and the permutation matrix to decouple the wide check matrix. (b) Utilizing the online hierarchical decoding to identify the error pattern given the input syndrome.

we introduce an online hierarchical decoding strategy that maximizes parallelism and exploits sparsity. Instead of directly applying the quantum maximum likelihood decoding constraints for qLDPC, we reformulate the constraints equation in a hardware-friendly way for better efficiency. We also present a new online hierarchical decoding algorithm with lower complexity than BP-based methods.

4.1 Algorithm Workflow

The algorithm workflow is shown in Figure 4. For each qLDPC code, the ideal decomposition aims to transform the check matrix into a diagonal block structure. This decomposition would allow the matrix to be treated as smaller, independent blocks, each of which could be solved in parallel. However, achieving this ideal decomposition directly is impractical. Instead, we opt for a partially diagonal-blocked matrix, with a sparse matrix remaining at the end. To achieve this, we apply a transformation matrix T and a permutation matrix P to the check matrix D. Finding the optimal T and P is a significant challenge, which we address by reformulating the decomposition problem as an SMT problem. This allows us to use an SMT solver to find the optimal transformation. We define a set of constraints to ensure the decomposed matrices adhere to the desired structure, with the goal of maximizing sparsity. The transformation and permutation matrices are represented as Boolean variables, and the resulting matrices are then stored for use in our online hierarchical decoding algorithm.

The online hierarchical decoding algorithm takes the syndrome $\vec{s}$ as input and applies the transformation matrix T. The original constraint $D \cdot \vec{e} = \vec{s}$ is reformulated to create a set of diagonal matrices that enable parallel decoding. This is achieved by splitting the

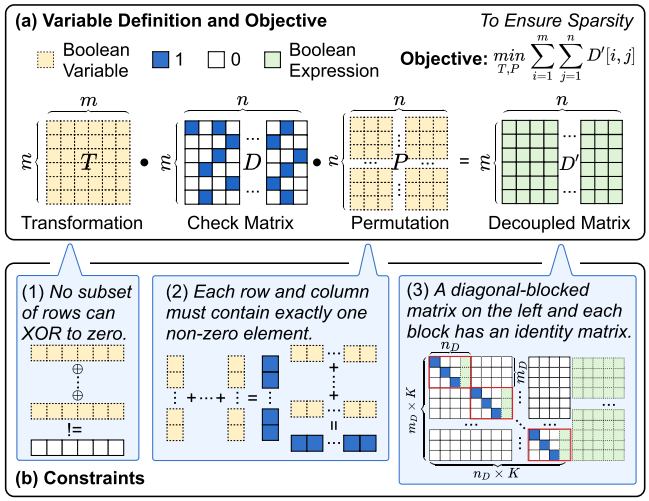


Figure 5: Formulation of check matrix decoupling.

error vector $\vec{e}$ into two parts: the left errors $\vec{l}$ corresponding to the diagonal block matrices $[diag(D_1, D_2, \ldots, D_K)]$, and the right error $\vec{r}$ corresponding to the sparse matrix A. By isolating the right error on the right side, we obtain a fully diagonal matrix for the left errors and a residual syndrome on the right, yielding the formulation $[diag(D_1, D_2, \ldots, D_K)] \cdot \vec{l} = \vec{s} \oplus A \cdot \vec{r}$. During decoding, we start by guessing the right error, gradually increasing the number of ones in each iteration. The right error is then used to compute the residual syndrome, which is used to decode the left error. The left error is decoded by *GreedyDecode* function. The diagonal block matrices enable parallel decoding for each matrix. The best left error result is selected and used to update the right error. After reaching the maximum number of iterations, the right and left errors are concatenated and multiplied by the permutation matrix P to obtain the final error.

4.2 Offline SMT-Optimized Decoupling

The decoupling of the check matrix must preserve the relationship between error patterns and the syndrome in Equation (1). In other words, we aim to find a reversible transformation of Equation (1). In linear algebra, only multiplication with a full-rank square matrix is reversible. Thus, we employ the following transformation to the original $m \times n$ check matrix D:

$$D' = T \cdot D \cdot P \tag{1}$$

where T is the $m \times m$ full-rank transformation matrix, P is the $n \times n$ permutation matrix, and D' is the resulting check matrix. Under this transformation, the maximum likelihood decoding in Equation (1) becomes:

$$\begin{aligned} &\vec{e'}_{opt} = \arg\min_{\vec{e'}} \sum_j w_j \vec{e'}_j \\ &s.t.\ D' \cdot \vec{e'} = \vec{s'} \end{aligned} \tag{2}$$

where $\vec{e'} = P^{-1} \cdot \vec{e}$ is the permutated error (P^{-1} is the inverse matrix of P) and $\vec{s'} = T \cdot \vec{s}$ is transformed syndrome. To recover the optimal original error, we decode Equation (2) to obtain the

optimal permuted error $\vec{e'}_{opt}$ and recover the optimal original error by $\vec{e}_{opt} = P \cdot \vec{e'}_{opt}$.

For the resulting check matrix D', we aim to transform it into the following form:

$$D' = (diag(D_1, D_2, \cdots, D_K), A) \tag{3}$$

where $diag(D_1, D_2, \cdots, D_K)$ is a diagonal block matrix and A is an arbitrary sparse matrix. To simplify our online decoding algorithm, we further require that each D_i contains an identity matrix on the left. Specifically,

$$D_i = (I, B) \tag{4}$$

where I is the identity matrix, and B is an arbitrary matrix. The notation $(\cdot, \cdot)$ is the horizontal concatenation of two matrices. Next, we explain how to search for the appropriate transformation matrix T and permutation matrix P to achieve this partial block-diagonal structure in D'.

Variables. We set the transformation matrix T and the permutation matrix P as the Boolean variables, as shown in Figure 5 (a). By Equation (1), we can represent the elements of D' by variable T and P,

$$D'[i, j] = \sum_{q=1}^{n} \sum_{k=1}^{m} T[i, k] D[k, q] P[q, j] \tag{5}$$

Constraints. The constraints for the decoupling process are summarized in Figure 5 (b).

(1) **Transformation Matrix.** The transformation matrix T must be a full-rank matrix, meaning its rows are linearly independent. Specifically, no subset of the rows can XOR to a zero vector:

$$\sum_{j=1}^{m} \oplus_{i \in \Delta} T[i, j] > 0, \ \forall \Delta \subseteq \{1, \cdots, m\} / \emptyset \tag{6}$$

where Δ is the non-empty subset of all row indexes.

(2) **Permutation Matrix.** The permutation matrix must satisfy that each row and column must contain exactly one non-zero element.

$$\begin{aligned} \sum_{j=1}^{n} P[i, j] = 1, \ \forall i \in \{1, \cdots, n\} \\ \sum_{i=1}^{n} P[i, j] = 1, \ \forall j \in \{1, \cdots, n\} \end{aligned} \tag{7}$$

(3) **Decoupled Matrix.** The decoupled matrix D' must satisfy the specific shape we defined. The constrains on D' are as follows:

- Each block in the decoupled matrix must have the same shape, denoted $m_D \times n_D$. The total number of rows and columns in D' must satisfy the following relationship:

$$m_D \cdot K = m, m \leq n_D \cdot K \leq n \tag{8}$$

where K is the number of blocks, and m and n are the dimensions of the original matrix.

- All elements outside the blocks, except for the right part A, must be zero.

$$\begin{aligned} D'[i \cdot m_D + t, j \cdot n_D + k] = 0, \forall i, j (i \neq j) \in \{1, \cdots, K\} \\ \forall t \in \{1, \cdots, m_D\}, \forall k \in \{1, \cdots, n_D\} \end{aligned} \tag{9}$$

- Each block in the decoupled matrix must have an identity matrix on the left side as shown in Equation (4).

$$\begin{aligned} D'[i \cdot m_D + t, i \cdot n_D + t] = 1, \forall i \in \{1, \cdots, K\}, \forall t \in \{1, \cdots, m_D\} \\ D'[i \cdot m_D + t, i \cdot n_D + k] = 0, \forall i \in \{1, \cdots, K\}, \\ \forall t, k (t \neq k) \in \{1, \cdots, m_D\} \end{aligned} \tag{10}$$

Objective Function. We aim to make the new check matrix D' sparse to facilitate the hardware design, meaning that we seek to minimize the total number of nonzero elements in D'.

$$\min_{T,P} \sum_{i=1}^{m} \sum_{j=1}^{n} D'[i, j] \tag{11}$$

The constraints in Equation (5~10) and the objective in Equation (11) are then passed to the SMT solver [12] to find solutions for the transformation matrix T and permutation matrix P.

Caveats of Check Matrix Decomposition. The decomposition only needs to be performed once per check matrix, and is executed entirely offline by an SMT solver. As shown in Equation (1), this decomposition relies on mathematically equivalent transformations, thus inducing no approximations. In principle, any check matrix can be transformed into the form of Equation (3), ensuring that a valid block diagonalization is always possible. The number of diagonal blocks K is determined by enumerating all possible factors of the number of rows m; the latter is upper-bounded by m/S, where S represents the column sparsity (i.e., the maximum number of non-zero elements across all columns). This upper bound exists because, mathematically, the number of columns m/K in each decomposed block matrix is at least the original column sparsity S, i.e., $m/K \geq S$. For instance, given a check matrix with 36 rows and column sparsity 6, the candidate value for K can be 6, 4, 3, or 2. We iteratively invoke the SMT solver against increasingly smaller K (i.e., starting with $K = 6$), and the first successful solution determines both the value of K and its corresponding block structure.

We further analyze the relationship between check matrix decomposition and the code structure on specific qLDPC codes. For BB codes, the X-type check matrix, denoted as $H = (f(x, y), g(x, y))$, is defined by two polynomials $f(x, y)$ and $g(x, y)$. Here, $x = S_l \otimes I_m$ and $y = I_l \otimes S_m$, where I_l and S_l are the $l \times l$ identity and the cyclic shift matrices, respectively. Through this structure, we find that the number of block matrices can be $K = \max(\min(l, m), \frac{l \times m}{S})$. For HP codes, the X-type check matrix can be denoted as $(H_1 \otimes I_g, I_t \otimes H_2^T)$, where H_1, H_2 are check matrices of two classical codes, and I_t is a $t \times t$ identity matrix. We can see that $I_t \otimes H_2^T$ is a diagnal block matrix since $I_t \otimes H_2^T = \text{diag}(H_2^T, H_2^T, \cdots, H_2^T)$. Then, we can set the off-diagonal matrix A as $H_1 \otimes I_g$ and each block matrix D_i as H_2^T. Thus, the number of block matrices K is t.

4.3 Online Hierarchical Decoding

Problem Formulation. Our goal is to decompose the original constraint into a set of smaller constraints that can be solved in parallel. This is straightforward when the transformed matrices D' are diagonal block matrices. However, in our case, the decoupled matrices include a sparse matrix A on the right, which requires additional processing. To handle this, we split the error $\vec{e}'$ into two parts: $\vec{l}$ and $\vec{r}$. The $\vec{l}$ corresponds to the set of diagonal block matrices

Algorithm 1: Hierarchical Decoding

Input : Syndrome vector $\vec{s}$, Transformation matrix T, Permutation matrix P, Decoupled check matrices $A, D_1, D_2, \cdots, D_K$, the number of columns in A n_A, the maximum iteration $\mathtt{M}$.
Output : Identified error pattern.

1 $\vec{s}' \leftarrow T \cdot \vec{s}$;
2 $\vec{r}_{best}, d_{min} \leftarrow [0, 0, \cdots, 0]$ (length is n_A), ∞;
3 **for** $k = 1$ *to* $\mathtt{M}$ **do**
4 **for** $i = 1$ *to* n_A **do**
5 $\vec{r} \leftarrow \vec{r}_{best}$; $\vec{r}[i] \leftarrow 1$; $\vec{s}_l \leftarrow \vec{s}' \oplus A\vec{r}$;
6 $\{\vec{s}_{l_1}, \cdots, \vec{s}_{l_K}\} \leftarrow$ Split $\vec{s}_l$ by the rows of $\{D_1, \cdots, D_K\}$;
7 $\vec{l} \leftarrow [GreedyGuess(D_j, \vec{s}_{l_j})\ \forall j \in \{1, \cdots, K\}]$;
8 $d \leftarrow \sum_{j=1}^{n} w_j(\vec{l}, \vec{r})_j$;
9 $\mathbf{d}[i], \mathbf{l}[i], \mathbf{r}[i] \leftarrow d, \vec{l}, \vec{r}$;
10 $i_{best} \leftarrow \text{argmin}_i\ \mathbf{d}[i]$;
11 **if** $\mathbf{d}[i_{best}] < d_{min}$ **then**
12 $d_{min}, \vec{l}_{best}, \vec{r}_{best} \leftarrow \mathbf{d}[i_{best}], \mathbf{l}[i_{best}], \mathbf{r}[i_{best}]$;
13 **else**
14 **Break**;
15 **Return** $P \cdot [\vec{l}_{best}, \vec{r}_{best}]$;

$diag(D_1, D_2, \cdots, D_K)$, and $\vec{r}$ corresponds to the sparse matrix A. This transforms the constraint into the following form:

$$diag(D_1, D_2, \cdots, D_K) \cdot \vec{l} \oplus A\vec{r} = \vec{s}' \tag{12}$$

The symbol $\oplus$ denotes a bit-wise XOR operation between two vectors, since the decoding is performed within the binary field $\{0, 1\}$. Here, we name $A\vec{r}$ as the right-part syndrome $\vec{s}_r$. Then, we move $\vec{s}_r$ to the right-hand side and define the left-part syndrome $\vec{s}_l = \vec{s}' \oplus \vec{s}_r$, resulting in:

$$diag(D_1, D_2, \cdots, D_K) \cdot \vec{l} = \vec{s}_l \tag{13}$$

This reformulated decoding constraint involves the check matrices $diag(D_1, D_2, \cdots, D_K)$ and the left-part syndrome $\vec{s}_l$. We can split $\vec{s}_l$ into smaller components $\{\vec{s}_{l_1}, \cdots, \vec{s}_{l_K}\}$ according to the block matrices. The problem is transformed into a set of smaller, independent constraint problems, where each constraint corresponds to one of the smaller check matrices. These constraints are decoupled from each other and can be solved in parallel:

$$D_j\vec{l}_j = \vec{s}_{l_j},\ \forall j \in \{1, \cdots, K\} \tag{14}$$

Hierarchical Decoding Algorithm. Based on our new problem formulation, we propose a two-step online hierarchical decoding algorithm, as outlined in Algorithm 1. The algorithm begins by initializing the transformed syndrome vector (line 1). The initial guess for the right error $\vec{r}$ is set to an all-zero vector, with the initial likelihood of the error vector set to infinity (line 2). Next, the decoding process proceeds by gradually increasing the number of ones in right error $\vec{r}$ until the maximum iteration limit $\mathtt{M}$ is reached (line 3). In each iteration, we apply the two-step decoding strategy: first, we guess the right error $\vec{r}$, and then decode the left error $\vec{l}$. Specifically, we explore the right error $\vec{r}$ by flipping one of its bits, and then calculate the left-part syndrome $\vec{s}_l$ (line 5). The left-part syndrome $\vec{s}_l$ is then split according to the rows of $D_1, \ldots, D_K$ (line 6). We proceed by decoding the left error $\vec{l}$ using the *GreedyGuess* function (line 7), and then computing the weighted summation of the left error $\vec{l}$ and right error $\vec{r}$ (line 8). At this point, we check if the current objective is smaller than the best solution found so far (line 10). If it is, we update the best solution for the next iteration (line 12). If not, we terminate the process (line 14) and return the best solution found up to that point (line 15).

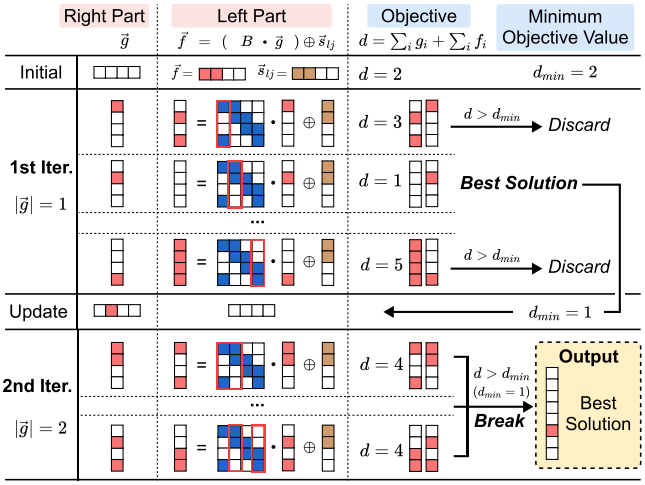


Figure 6: An execution example of *GreedyGuess*.

GreedyGuess is introduced to decode the left error $\vec{l}$ under the constraint specified in Equation (14). Leveraging the decoupled structure of the check matrix, $D_i = [I, B]$, where I is an identity matrix, we reformulate the error vector into two parts: $\vec{f}$, corresponding to I, and $\vec{g}$, corresponding to B. The constraint can then be rewritten as:

$$I\vec{f} \oplus B\vec{g} = \vec{s}_{l_j} \Rightarrow \vec{f} = B\vec{g} \oplus \vec{s}_{l_j} \tag{15}$$

Our goal is to minimize the objective function $d = \sum_j w_j\vec{l}_j$, and each data qubit j has its own weight w_j. To achieve this, we adopt a guessing strategy for the number of ones in $\vec{g}$, starting with the least number and gradually increasing it. This search process follows a similar approach to Algorithm 1.

Figure 6 provides an example illustrating the *GreedyGuess* function. In this example, we assume that all data qubits have the same prior probabilities for simplicity, so the objective is to minimize the number of ones in the left error $\vec{l}$. Initially, the right part $\vec{g}$ is set to an all-zero vector, and from Equation (15), the left part $\vec{f}$ is equal to the syndrome vector $\vec{s}_i$. Thus, the current minimal objective for the full error vector is $d_{min} = 2$. In the first iteration, we flip one bit in $\vec{g}$ and compute the corresponding left part $\vec{f}$. We discard that solution if the total objective exceeds d_{min}. For example, flipping $\vec{g}$ to $[1, 0, 0, 0]$ or $[0, 0, 0, 1]$ results in worse objectives, while $[0, 1, 0, 0]$ minimizes the objective to 1. We then update $\vec{g}$ and set $d_{min} = 1$, proceeding to the next iteration. In the second iteration, we flip another bit of $\vec{g}$ except for the second bit. Since this iteration does not improve the objective, we stop the process and return the best solution found: $\vec{g} = [0, 1, 0, 0]$ and $\vec{f} = [0, 0, 0, 0]$.

Table 1: Complexity of Vegapunk and prior works.

Method	Serial (Limited P)	Parallel (Sufficient P)
BP [34]	$O(\mathtt{M}_{\text{bp}} \cdot \frac{n}{\mathtt{P}})$	$O(\mathtt{M}_{\text{bp}})$
BP+LSD [22]	$O(\mathtt{M}_{\text{bp}} \cdot \frac{n}{\mathtt{P}} + (\text{polylog}(n) + \kappa^3) \cdot \frac{n/\kappa}{\mathtt{P}})$	$O(\mathtt{M}_{\text{bp}} + \text{polylog}(n) + \kappa^3)$
BPGD [48]	$O(n\mathtt{M}_{\text{bp}} \cdot \frac{n}{\mathtt{P}})$	$O(n\mathtt{M}_{\text{bp}})$
Vegapunk	$O(\frac{n}{\mathtt{P}} \log n + \frac{n \cdot K}{P} \cdot S)$	$O(\log n + S)$

$\mathtt{M}_{\text{bp}}$ is the number of iterations of BP. $\log n \le \mathtt{M}_{\text{bp}} \le n$.

$\kappa \le \frac{1+p}{1+p-Sp}$ is the maximum cluster size in BP+LSD and p is the physical error rate.

4.4 Complexity Analysis

Here, we analyze the time complexity of our online decoding algorithm in Vegapunk. We consider decoding a $m \times n$ check matrix, where m is the length of the syndrome vector and n is the length of the error vector. S is the maximum number of non-zero elements across all columns in the check matrix. $\mathtt{P}$ denotes the number of available parallel processing units.

Complexity of Vegapunk. Line 1 of Algorithm 1 initializes the transformed syndrome vector via a matrix-vector multiplication followed by a binary-tree XOR reduction, with parallel complexity $O(\frac{m}{\mathtt{P}} \log m)$. Lines 3–14 iterate $\mathtt{M}$ times (a preset constant). Within each iteration, lines 4–9 perform $O(n)$ parallel guesses for the right error $\vec{r}$. Each guess requires computing the left-part syndrome $\vec{s}_l$ in $O(S)$ time, exploiting incremental updates and matrix sparsity (line 5). The subsequent parallel decoding of the left error (lines 6–9) has complexity $O(K \cdot \mathtt{M} \cdot S)$, resulting in a parallel complexity of $O(\frac{n}{\mathtt{P}} \cdot S + \frac{n \cdot K}{\mathtt{P}} \cdot \mathtt{M} \cdot S) = O(\frac{n \cdot K}{\mathtt{P}} \cdot S)$ for lines 4–9. The update step in lines 10–14 uses a binary-tree comparison, contributing $O(\frac{n}{\mathtt{P}} \log n)$. Hence, lines 3–14 have total complexity $O(\frac{n}{\mathtt{P}} \log n + \frac{n \cdot K}{\mathtt{P}} \cdot S)$. The permutation in line 15 takes $O(\frac{n}{\mathtt{P}})$. Thus, the overall complexity of Vegapunk is $O(\frac{n}{\mathtt{P}} \log n + \frac{n \cdot K}{\mathtt{P}} \cdot S)$ since m is linear to n and $m < n$. If sufficient parallel units are available (i.e., $\mathtt{P} > n \cdot K$), the complexity reduces to $O(\log n + S)$. In this case, the logarithmic scaling of decoding latency makes Vegapunk insensitive to code size.

Comparison with Prior Decoders. Table 1 gives the serial complexity and parallel complexity of prior works and Vegapunk. Since prior works rely on the results of BP, we use $\mathtt{M}_{\text{bp}}$ to denote the iteration numbers of BP, which is $O(\log n)$ in the best case and $O(n)$ in the worst case [34]. We can see that Vegapunk has the lowest complexity $O(\log n + S)$, which outperforms BP since S is a small number due to the low-density properties of qLDPC codes. Note that the BP+LSD states that it only needs to run BP for 30 iterations [22], so the complexity can be simplified as $O(\text{polylog}(n) + \kappa^3)$, which is still higher than Vegapunk.

5 Vegapunk Accelerator

To meet the real-time decoding requirement in supercomputing quantum systems, we design a custom FPGA-based accelerator to support our hierarchical decoding algorithm. The accelerator design is directly derived from the algorithm's structure, with each major operation mapped to dedicated logic to minimize latency. Compared to general-purpose processors, FPGAs are better suited for this task for two main reasons: (1) Our hierarchical decoding algorithm involves fine-grained, bit-level operations such as left-part syndrome computation. These operations do not align well with the wide-vector execution units in CPUs or GPUs, leading to low utilization and memory access inefficiencies; (2) FPGA-based accelerators integrate naturally with superconducting quantum systems, where the syndrome is typically generated and streamed by FPGA-based readout modules. A tightly coupled decoder implemented on the same device allows direct data access, avoiding the delay of transferring data to external processors [23, 37, 43].

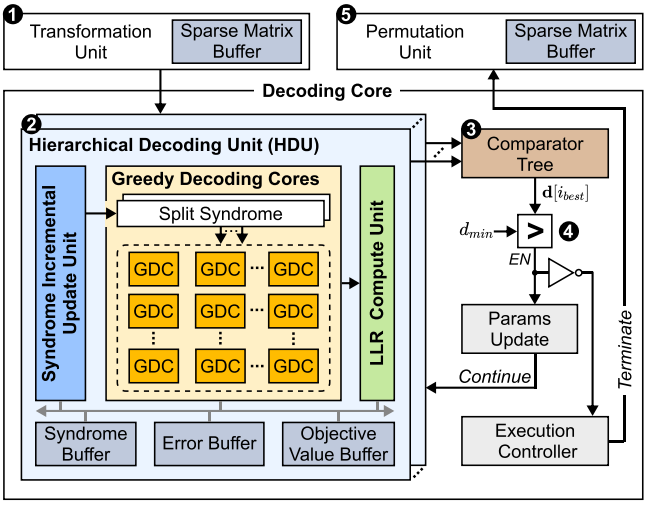


Figure 7: Overview of Vegapunk accelerator architecture.

As shown in Figure 7, the Vegapunk accelerator comprises three main components to fully explore the algorithm's parallelism and sparsity: a transformation unit, a decoding core, and a permutation unit. The transformation unit prepares the input syndrome for decoding by aligning it with the decoupled check matrix. The decoding core performs the core error decoding procedure using a parallel greedy strategy. The permutation unit reconstructs the final error from the decoding output. The decoding core integrates key submodules, including the Hierarchical Decoding Unit (HDU), the comparator tree, and the update and control logic, which will be detailed in the following sections.

5.1 Accelerator Dataflow

The decoding dataflow in each round consists of five main steps, as demonstrated in Figure 7:

❶ **Transformation:** The transformation unit takes the original syndrome $\vec{s}$ and multiplies the transformation matrix T through sparse Matrix-Vector Multiplication (MVM) to produce the transformed syndrome $\vec{s}'$. The transformation matrix T is pre-loaded into the sparse matrix buffer.

❷ **Objective Value Calculation:** The transformed syndrome $\vec{s}'$ is fed into multiple HDUs, each responsible for calculating the objective values of candidate error guesses $\mathbf{d}[i]$. Each HDU first triggers the syndrome incremental update unit to compute the left-part syndrome $\vec{s}_l$, by performing sparse MVM and XOR. Then, the left-part syndrome $\vec{s}_l$ is split into a set of K partial syndromes $\vec{s}_{l_1}, \ldots, \vec{s}_{l_K}$ and sent to K Greedy Decoding Cores (GDCs). Each GDC processes its corresponding block D_j with $\vec{s}_{l_j}$ and outputs a partial left error $\vec{l}_j = [\vec{f}, \vec{g}]$. The outputs from all GDCs are concatenated into the left error $\vec{l}$. The Log-Likelihood Ratio (LLR) compute unit

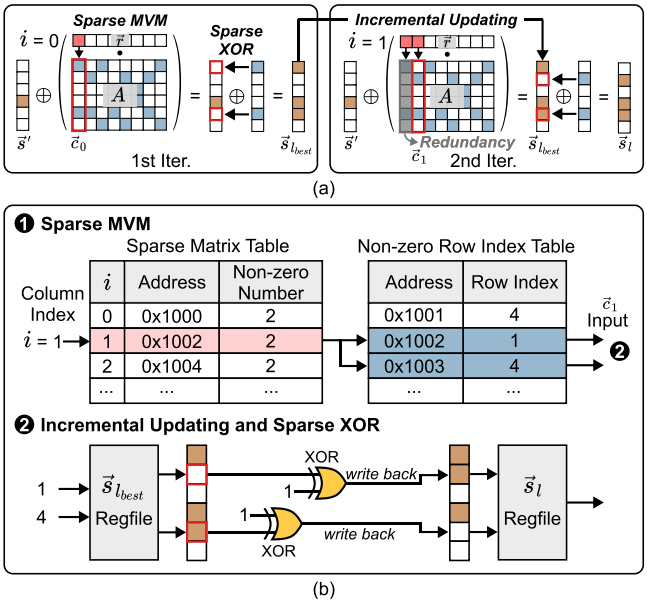


Figure 8: (a) The insight of designing the syndrome incremental update unit, including the sparse opportunity and redundancy in the computation of the left-part syndrome. (b) The details of the syndrome incremental update unit.

then calculates the final objective value using the complete error $\vec{e} = [\vec{l}, \vec{r}]$ and weights w.

❸ **Comparison:** The comparator tree receives the objective values $\mathbf{d}$ from the HDUs and compares them in parallel to identify the minimum-weight solution. The output is the best solution $\mathbf{d}[i_{\text{best}}]$, corresponding to the minimum-weight error guess.

❹ **Parameter Update:** The best solution $\mathbf{d}[i_{best}]$ from the comparator tree is compared with the current minimum value d_{min}. If the new solution reduces the minimum, the params update unit is triggered, and the new values for d_{min}, $\vec{l}_{best}$, and $\vec{r}_{best}$ are stored. The execution controller monitors the enable signal to decide whether decoding should continue or terminate.

❺ **Permutation:** The permutation unit receives the error $\vec{e}' = [\vec{l}_{\text{best}}, \vec{r}_{\text{best}}]$ and multiplies the permutation matrix P through sparse MVM to compute the final error $\vec{e}$. The permutation matrix P is also pre-loaded into the sparse matrix buffer.

5.2 Syndrome Incremental Update Unit

As illustrated in Figure 8 (a), to efficiently compute the left-part syndrome $\vec{s}_l = \vec{s}' \oplus A\vec{r}$ during online hierarchical decoding, we proposed the syndrome incremental update unit, which has two features: (1) Sparse MVM and XOR. By exploiting the sparsity of the right error $\vec{r}$, we transform the complex MVM process into simple row selection. Since the column of the decoupled check matrix A is also sparse, we only select the non-zero elements to perform XOR with the syndrome $\vec{s}'$, thus accelerating the decoding process; (2) Incremental Updating. By storing and reusing the left-part syndrome $\vec{s}_{l_{best}}$ corresponding to the identified error bit i_{best} from previous iterations, we avoid recalculating the entire syndrome, thus significantly reducing redundant computations.

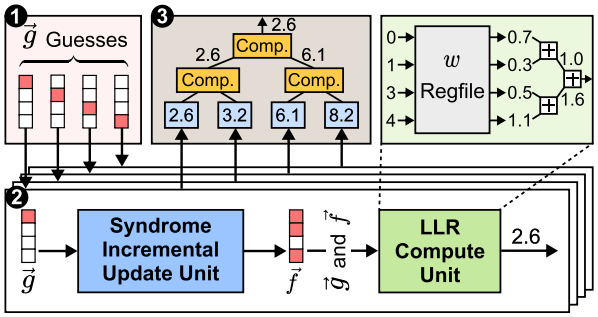


Figure 9: Detailed architecture of Greedy Decoding Core (GDC) to effectively support the *GreedyGuess* procedure.

Figure 8 (b) shows an example of computing the second iteration using the syndrome incremental update unit. First, we apply the ❶ sparse MVM to compute $\vec{c}_1$. The sparse matrix A is stored using our proposed compressed format, which includes a sparse matrix table and a non-zero row index table. The sparse matrix table stores the column addresses and the number of non-zero elements in each column, while the non-zero row index table records the row indices corresponding to these non-zero elements. In the example, the input column index $i = 1$, so we directly fetch the 1st column of A, and extract the row indices of '1's in the 1st column (i.e., 1 and 4) from the non-zero row index table. This non-zero extraction is prepared for the subsequent sparse XOR. Since any value XORed with zero remains unchanged, we only need to apply XOR on these non-zero elements. Next, we perform ❷ incremental updating and sparse XOR to compute the left-part syndrome $\vec{s}_l$. We retrieve the values at the selected row indices (i.e., 1 and 4) from the $\vec{s}_{l_{best}}$ regfile, XOR them with 1, and write the results back to the $\vec{s}_l$ regfile. After all syndrome incremental update units have completed, we select $\vec{s}_l$ corresponding to the identified error bit i_{best} in the current iteration, and store it in the $\vec{s}_{l_{best}}$ regfile. This enables the data reuse of the left-part syndrome computation in the following iteration.

5.3 Greedy Decoding Core

We design the Greedy Decoding Core (GDC) to effectively support the *GreedyGuess* procedure. Figure 9 illustrates the detailed architecture of the GDC. Each GDC consists of three main components: the syndrome incremental update unit, the LLR compute unit, and the comparator tree. Each GDC takes the partial syndrome $\vec{s}_{l_j}$ and all the candidate guesses of $\vec{g}$ as input. Multiple syndrome incremental update units compute $\vec{f} = (B \cdot \vec{g}) \oplus \vec{s}_{l_j}$ in parallel, taking advantage of the sparsity in both the matrix B and right part $\vec{g}$. Once the left part $\vec{f}$ is obtained, it is concatenated with $\vec{g}$ to form the left error $\vec{l}_j = [\vec{f}, \vec{g}]$. To evaluate the candidate left error $\vec{l}$, we compute the objective value $\sum_i w_i g_i + \sum_j w_j f_j$. The LLR compute unit performs this task efficiently by focusing on the non-zero elements of $\vec{f}$ and $\vec{g}$, using their indices to retrieve the corresponding weights from the w-regfile, and then applying an adder tree to compute the objective value d in parallel. Finally, all the computed objective values are sent to the comparator tree, which identifies the minimum value. The left error corresponding to this minimum value is considered the best solution for the current iteration.

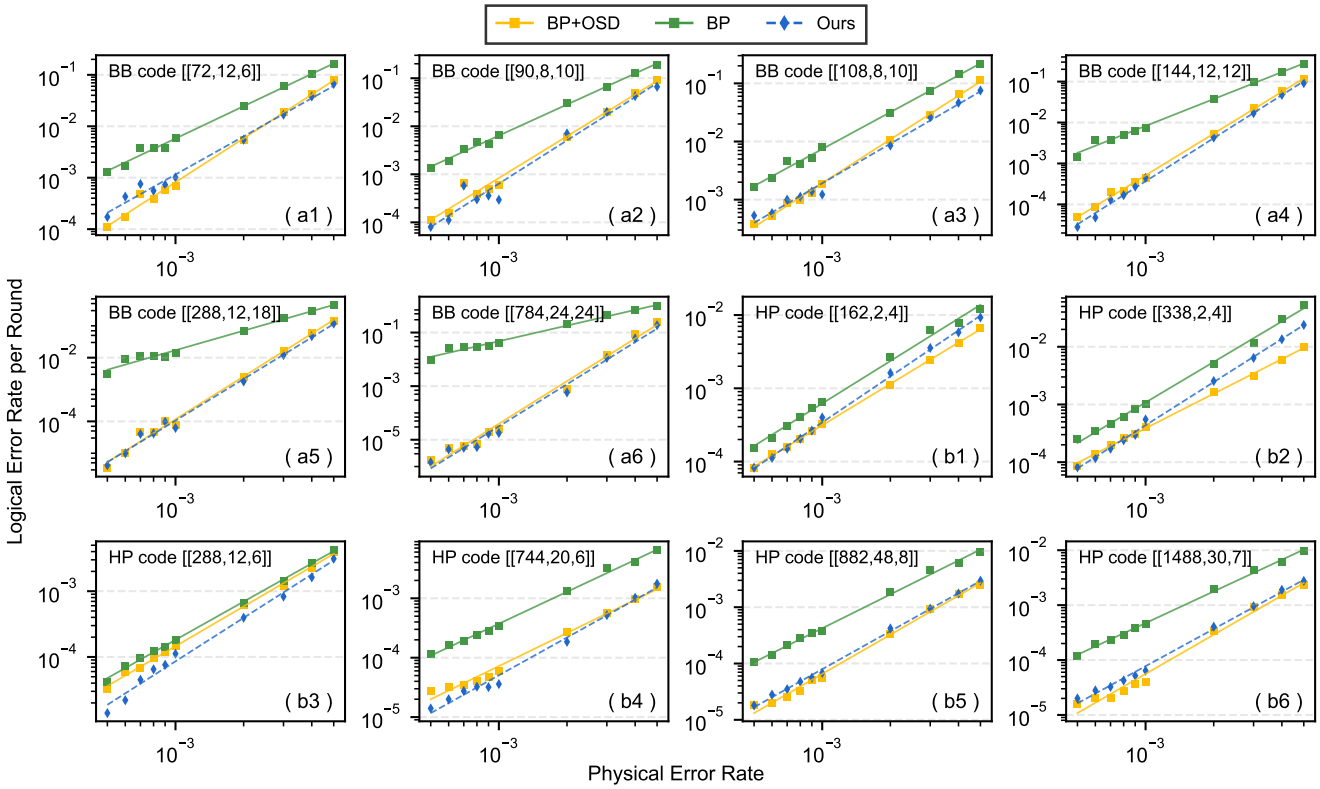


Figure 10: The LER for BP [34], BP+OSD-CS(7) [15], and Vegapunk. The physical error rate p ranges from 5×10^{-4} to 5×10^{-3}.

6 Evaluation

6.1 Evaluation Setup

Implementation: For offline SMT-optimized decoupling, we use Z3 solver [12] to solve the decoupling problem. For online hierarchical decoding, we develop Vegapunk accelerator using Xilinx High-Level Synthesis (HLS) C++ and implement it with Vitis 2023.1. We set the maximum iteration `M` of Vegapunk to 3. We evaluate its hardware performance on the Xilinx Alveo U50 FPGA at 250 MHz. We also implement CPU and GPU versions of Vegapunk, and run them on the AMD EPYC 9554 64-core CPU and NVIDIA GeForce RTX 4070 Ti GPU, respectively.

Baselines: We compare Vegapunk with state-of-the-art qLDPC decoders, including BP [34], BP+OSD [15], BP+LSD [22] and BPGD [48]. For BP, we set the maximum iterations to the number of data qubits n in qLDPC codes and choose the min-sum algorithm as the node information update method. We implement BP on the Xilinx Alveo U50 FPGA at 250MHz, consistent with Vegapunk. For BP+OSD, we choose the BP+OSD-CS(7) version since it offers a high decoding accuracy [39]. The maximum iteration of BP is set to n. We run the OSD part on the AMD EPYC 9554 64-core CPU due to its sequential nature. For BP+LSD, we set the maximum iteration of BP to 30 and the order to 0. For BPGD, we set the maximum rounds to n and the number of BP iterations per round to 100.

Benchmarks: The benchmarks include two types of qLDPC codes: (1) Bivariate Bicycle (BB) codes [6] and (2) Hypergraph Product (HP) codes [5], as listed in Table 2. We choose six BB codes, with the size ranging from [[72,12,6]] to [[784,24,24]]. For HP codes, we use two ring codes with distances 9 and 13 to build [[162,2,4]] and [[338,2,4]] codes, respectively. Additionally, four other HP codes are derived from [33], constructed using two bicycle codes.

Noise Model: For BB codes, we consider a circuit-level error model and leverage the syndrome measurement circuit from [6]. The physical error rate p ranges from 5×10^{-4} to 5×10^{-3}. This error range encompasses the noise levels typically observed in current superconducting quantum devices, thus effectively evaluating the decoder's performance under realistic conditions. The circuit-level noise is sampled from (1) depolarizing errors (X, Y, Z) on data qubits at the beginning of each round, (2) depolarizing errors on data and parity qubits after syndrome extraction operations, (3) measurement errors, and (4) reset errors. For HP codes, we consider a phenomenological noise model with data qubit errors and measurement errors.

Metrics: The performance of Vegapunk is evaluated using three key metrics: (1) Logical Error Rate (LER), (2) latency, and (3) accuracy threshold. We assume Vegapunk receives a syndrome every $1\mu s$ (corresponds to one measurement round). After receiving d rounds of syndromes, Vegapunk applies a correction operation to the logical qubit and measures its state. A logical error occurs if the logical measurement does not match the initially prepared state. We repeat the decoding process multiple times under the noise model, with each syndrome sampled and generated by the Stim package [19]. The overall LER P_L is calculated as the ratio of successful decodings to the total number of trials. The LER per round

Table 2: qLDPC codes for benchmarking, corresponding decoupled check matrices, and decoding performance of decoders.

Code Type	Code Notation	Check Matrix Shape	Decoupled Check Matrices			Accuracy Threshold (%)			Latency per Round (0.5% circuit-level noise)				
			A shape (Spars.*)	D_i shape (Spars.*)	K	BP	BP+OSD-CS(7)	Vegapunk	BP†	BP+OSD-CS(7)	Vegapunk (worst case)		
											CPU	GPU	FPGA
BB	[[72,12,6]]	[36, 360]	[36,108] (6)	[6, 42] (3)	6	0.020	0.112	0.091	694ns	0.98ms	39.3μs	76.8μs	720ns
	[[90,8,10]]	[45, 450]	[45,135] (6)	[9, 63] (3)	5	0.020	0.110	0.125	1104ns	2.02ms	46.4μs	84.9μs	732ns
	[[108,8,10]]	[54, 540]	[54,162] (6)	[6, 42] (3)	9	0.016	0.065	0.060	1400ns	3.03ms	52.5μs	92.5μs	732ns
	[[144,12,12]]	[72, 720]	[72,216] (6)	[6, 42] (3)	12	0.017	0.132	0.149	2387ns	6.77ms	100.3μs	99.2μs	732ns
	[[288,12,18]]	[144, 1440]	[144,432] (6)	[12, 84] (3)	12	0.006	0.186	0.196	7009ns	38.6ms	115.5μs	104.1μs	780ns
	[[784,24,24]]	[392, 3920]	[392,1176] (6)	[28, 196] (3)	14	0.002	0.213	0.227	25881ns	449ms	547.8μs	116.1μs	840ns
Average	-	-	-	-	-	0.013	0.136	0.141	6412ns	83.4ms	150.3μs	95.6μs	756ns
HP	[[162,2,4]]	[81, 243]	[81,81] (2)	[9, 18] (2)	9	0.167	0.377	0.268	151ns	0.18ms	17.9μs	69.3μs	264ns
	[[338,2,4]]	[169, 507]	[169,169] (2)	[13, 26] (2)	13	0.094	0.261	0.175	282ns	0.43ms	37.4μs	81.1μs	276ns
	[[288,12,6]]	[144, 432]	[144,144] (4)	[12, 24] (4)	12	0.622	0.674	0.777	72ns	0.19ms	30.3μs	85.3μs	432ns
	[[744,20,6]]	[372, 1116]	[372,372] (4)	[31, 62] (4)	12	0.350	2.176	1.455	578ns	1.93ms	80.8μs	86.5μs	480ns
	[[882,48,8]]	[441, 1323]	[441,441] (5)	[63, 126] (3)	7	0.237	0.798	0.768	735ns	6.66ms	98.1μs	93.6μs	526ns
	[[1488,30,7]]	[744, 2232]	[744,744] (4)	[31, 62] (4)	24	0.228	0.819	0.775	1875ns	11.7ms	150.7μs	94.9μs	492ns
Average	-	-	-	-	-	0.283	0.851	0.703	616ns	3.52ms	69.2μs	85.1μs	412ns

* Spars.: The maximum number of ones in the columns of the matrix.
† The hardware-accelerated architecture of BP is derived from [42].

p_L is defined as [21]:

$$p_L = 1 - (1 - P_L)^{(1/d)} \tag{16}$$

where d is the code distance. The accuracy threshold p_t is the physical error rate when error correction becomes effective. Specifically, when the physical error rate p is lower than p_t, the LER per round p_L becomes lower than p. A higher accuracy threshold indicates greater tolerance of the decoder to physical error rates. The theoretical relationship between p_L and p follows the equation:

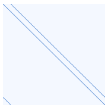
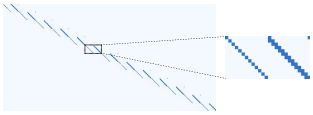
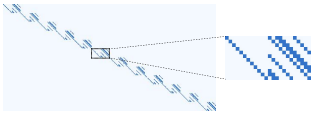
$$\ln p_L = k \ln p + (1 - k) \ln p_t \tag{17}$$

where k and p_t are the const parameters. We can get the accuracy threshold p_t by fitting the experimental data of LER per round p_L under different physical error rates p.

6.2 Decoding Performance

Logical Error Rate. As shown in Figure 10, Vegapunk with maximum iteration M = 3 achieves LER comparable to the state-of-the-art qLDPC decoder BP+OSD-CS(7), while consistently outperforming the BP decoder. Specifically, Vegapunk delivers lower LER than BP+OSD-CS(7) for certain BB codes [[90,8,10]] (a2) and [[144,12,12]] (a4), HP codes [[288,12,6]] (b3) and [[744,20,6]] (b4). Across these codes, Vegapunk yields average improvements of 1.33× for BB codes and 1.51× for HP codes. While the LER of Vegapunk for BB codes [[72,12,6]] (a1) and [[108,8,10]] (a3) is higher than BP+OSD-CS(7) at physical error rates below 10^{-3}, it shows the advantage at higher physical error rates, achieving an average 1.34× improvement at a physical error rate of 5×10^{-3}. For larger codes such as BB codes [[288,12,18]] and [[784,24,24]], both BP+OSD-CS(7) and Vegapunk exhibit a remarkable LER of 10^{-5} at a physical error rate of 5×10^{-4}. In contrast, the BP decoder fails to correct errors in these codes, resulting in consistently higher LER across all physical error rates. The comparable decoding accuracy of Vegapunk to BP+OSD-CS(7) can be attributed to two key reasons. First, most error patterns have low Hamming weight, since the probability of an error pattern occurring decreases exponentially with the number of flipped bits [43]. Therefore, by setting the maximum number of guesses (e.g., M=3), Vegapunk is able to cover the vast majority of likely error patterns. Second, our decoupling strategy significantly reduces the search space for error patterns, increasing the likelihood of successful decoding even with a greedy algorithm.

Table 3: Visual examples of decoupled check matrices.

Code		Off-diagonal Matirx A	Diagonal Block Matrix D
BB code	[[72,12,6]]		
	[[108,8,10]]		
HP code	[[338,2,4]]		
	[[288,12,6]]		

Accuracy Threshold. As shown in Table 2, Vegapunk achieves an accuracy threshold that is 0.93× that of the BP+OSD-CS(7) decoder and 15.8× higher than that of the BP decoder. For large codes such as BB codes [[288,12,18]], [[784,24,24]] and HP codes [[882,48,8]], [[1488,30,7]], Vegapunk achieves a threshold comparable to BP+OSD-CS(7), ranging from 0.95× to 1.13×. This high

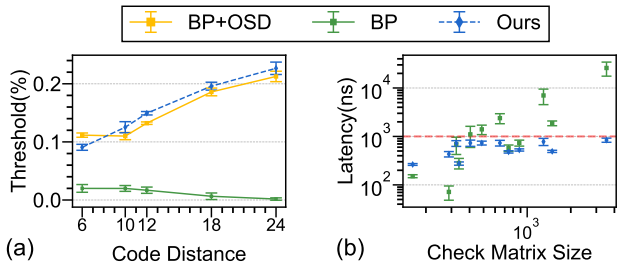


Figure 11: Scalability of Vegapunk, in terms of (a) accuracy threshold and (b) decoding latency.

fault tolerance stems from our offline decoupling strategy, which decomposes the check matrix into smaller submatrices. In large codes, the number of submatrices increases, resulting in smaller matrix sizes, which in turn benefits decoding accuracy. For small codes such as HP codes [[162,2,4]] and [[338,2,4]], Vegapunk yields 0.67× to 0.71× the threshold of BP+OSD, but still outperforms BP by 1.60× to 2.78×.

Latency. We measure the latency of decoders under 0.5% circuit-level depolarising noise. As detailed in Table 2, Vegapunk achieves a decoding latency under 1μs (FPGA) across different qLDPC codes. Despite the notable speed of BP, it fails to meet the real-time requirement as the code size gradually increases(e.g., BB code [[90,8,10]], 1104ns). The latency of BP+OSD-CS(7) far exceeds 1μs, since its sequential characteristics are difficult to accelerate effectively in hardware. By comparing BP and Vegapunk, we find that the latency of Vegapunk is insensitive to the code size but sensitive to the sparsity of the check matrix. To be specific, while the check matrix column size for BB codes (with a constant column sparsity of 6) increases by around 10×, the latency of Vegapunk only rises from 720ns to 840ns (1.16×). For HP codes (with varying column sparsities of 2, 4, and 5), the latency of Vegapunk ranges from 264ns to 526ns, reflecting the direct influence of different sparsity levels. On the contrary, the latency of the BP decoder is highly sensitive to the code size. For example, from BB code [[72,12,6]] to [[784,24,24]], the latency of BP increases by a factor of 37.3×, while the check matrix column size only increases by 10.9×. This is because, to achieve convergence, the actual number of BP iterations increases with the code size, leading to a linearly increasing latency. Vegapunk, however, leverages the sparsity computation and parallel guessing during online decoding, rendering its latency insensitive to the code size. Since the check matrix of qLDPC codes is inherently sparse, Vegapunk is more suited for fast decoding of these codes.

We also implement Vegapunk on both CPU and GPU, with average latencies of 150.3μs and 95.6μs on BB codes, 69.2μs and 85.1μs on HP codes, both slower than FPGA. This is because our FPGA design efficiently handles bit-level sparse operations through the custom decoding pipeline, while CPU and GPU suffer from low utilization due to wide-vector execution and irregular memory access. For HP codes, GPU has a slightly higher average latency (85.1μs) than CPU (69.2μs), mainly due to the overhead of decoding kernel launch and limited parallelism in small problem sizes.

Scalability. Figure 11 (a) shows the accuracy threshold of BB codes with different code distances. Error bars represent the uncertainty of fitting experimental data. Both Vegapunk and BP+OSD exhibit increasing thresholds as the code distance increases, indicating their ability to exploit the error correction capacity of qLDPC codes. In contrast, BP shows a decreasing trend, highlighting its inability to effectively leverage the error correction potential of large codes due to its low decoding accuracy. Figure 11 (b) shows decoding latency for varying check matrix sizes of Vegapunk and BP. Error bars represent the standard deviation of latency across different physical error rates. Vegapunk scales logarithmically with matrix size, while BP grows linearly and exceeds 1μs at a size of 5×10^2. This is due to BP's iterative message passing with linear time complexity, whereas Vegapunk leverages matrix sparsity and parallelism for faster decoding. Furthermore, the latency of Vegapunk is less sensitive to physical error rates than BP, with a lower standard deviation (62.6 vs. 1080.8 for BP).

Table 4: FPGA utilization of Vegapunk

BB code			HP code		
Code	FFs	LUTs	Code	FFs	LUTs
[[72, 12, 6]]	13388(0.77%)	37496(4.30%)	[[162,2,4]]	11944(0.69%)	20726(2.38%)
[[90, 8, 10]]	14661(0.84%)	42118(4.83%)	[[338,2,4]]	24242(1.39%)	41192(4.72%)
[[108, 8, 10]]	15589(0.89%)	51023(5.85%)	[[288,12,6]]	14990(0.86%)	56517(6.48%)
[[144, 12, 12]]	16870(0.97%)	62953(7.22%)	[[744,20,6]]	20145(1.16%)	103999(11.93%)
[[288, 12, 18]]	23020(1.32%)	102277(11.73%)	[[882,48,8]]	24507(1.41%)	135777(15.57%)
[[784, 24, 24]]	40952(2.35%)	272618(31.26%)	[[1488,30,7]]	26385(1.51%)	177954(20.41%)

6.3 FPGA Utilization

Table 4 shows the FPGA utilization of Vegapunk for six BB and six HP codes. For example, decoding the medium BB code [[144,12,12]] uses under 10% of Look-Up Table (LUTs) and 1% of Flip-Flop (FFs), while the large BB code [[784,24,24]] requires only 31.26% of LUTs and 2.35% of FFs. Such low demands allow seamless integration with the FPGA control processor, making Vegapunk well-suited for real-time qLDPC decoding. Furthermore, LUT usage scales linearly with the check matrix column size. Fitting the data shows that 100% LUT utilization occurs only when the matrix column size exceeds 1.26×10^4, corresponding to about 2,511 data qubits—the maximum capacity for implementing Vegapunk on a Xilinx Alveo U50 FPGA.

6.4 Ablation Study

Offline Decoupling Strategy. Figure 12 shows the impact of the offline decoupling strategy on decoding accuracy, using three types of BB codes. With the decoupling strategy, Vegapunk achieves accuracy improvements of 17.9×, 26.1×, and 31.1× compared to the version of Vegapunk without this strategy. This improvement stems from the decoupling strategy's ability to reduce the number of columns in the check matrix, mitigating the issue of *quantum degeneracy*. Additionally, decoding on smaller check matrices enhances the effectiveness of the online decoding algorithm, as it reduces the search space for *GreedyGuess*, leading to more accurate error guesses and, consequently, improved decoding accuracy.

Maximum Iteration. Figure 13 illustrates the sensitivity of decoding performance to the maximum iteration `M` for BB code [[288,12,18]] (Figure 13 (a)) and HP code [[288,12,6]] (Figure 13 (b)). As `M` increases, the decoding latency grows linearly. However, this growth slows noticeably beyond `M` = 5, due to the early-stopping

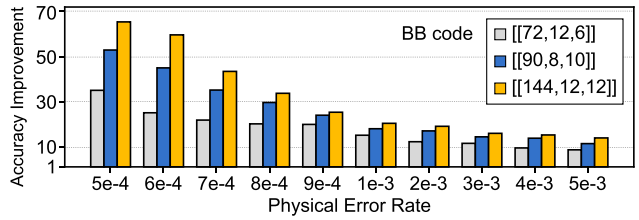


Figure 12: Ablation study of the offline decoupling strategy.

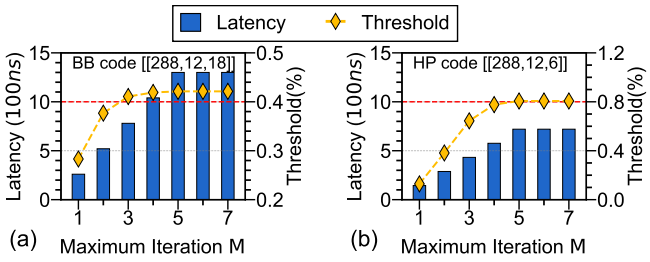


Figure 13: Ablation study of the maximum iteration M on decoding latency and accuracy threshold.

mechanism in Vegapunk, which terminates the process if no better solution is found in a given iteration. In these two codes, the algorithm stops after five iterations. For the accuracy threshold, the benefit of increasing M diminishes rapidly. For instance, from the first to the second iteration, the threshold improves significantly—by 34% for BB codes and 195% for HP codes. However, from the third to the fourth iteration, the gain drops sharply to only 0.5% and 3.8%, respectively. Moreover, running four iterations on the BB code [[288,12,18]] results in a latency exceeding $1\mu s$, leading us to set $M = 3$ to balance the decoding accuracy and latency.

6.5 Comparison with BP+LSD and BPGD

To demonstrate the efficacy of our proposed decoding algorithm, we compare Vegapunk with the state-of-the-art decoders, BP+LSD [22] and BPGD [48]. We run these three decoders serially on the AMD EPYC 9554 64-core CPU to decode six BB codes from [[72,12,6]] to [[784,24,24]], under circuit-level noise with physical error rates from 5×10^{-4} to 5×10^{-3}. Figure 14 (a) depicts the decoding latency per round of three decoders. Vegapunk outperforms both BP+LSD and BPGD, achieving average speedups of 147.6× and 13.9×, respectively. This is because Vegapunk employs a greedy decoding strategy and effectively leverages the sparsity in decoupled check matrices. In contrast, BP+LSD necessitates costly matrix inversion, and BPGD requires multiple serial iterations. Furthermore, as the physical error rate increases, the latency of Vegapunk grows significantly slower than that of BPLSD and BPGD. This is because the latency of Vegapunk is mainly determined by the sparsity of the decoupled matrices and the maximum number of iterations M, making it less sensitive to the physical error rate. Figure 14 (b) presents the accuracy threshold of three decoders. Vegapunk realizes average improvements of 2.53× and 7.11× compared to BP+LSD and BPGD, respectively. The higher accuracy threshold of Vegapunk stems from its offline decoupling strategy, which significantly reduces the search space for greedy guessing, thereby enhancing precision.

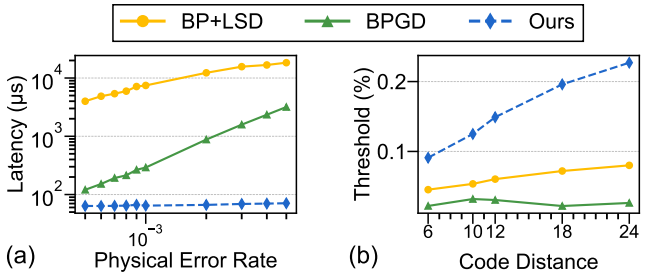


Figure 14: Comparison with BP+LSD [22] and BPGD [48], in terms of (a) decoding latency at different physical error rates and (b) accuracy threshold at varying BB codes.

7 Related Work

Belief Propagation (BP) [34] is one of the most widely used decoding algorithms for qLDPC codes due to its low complexity, but it suffers from convergence issues caused by quantum degeneracy [35, 36]. Various methods have been proposed to enhance the BP process, including neural BP [30], adaptive BP with memory [27], generalized BP [32], trapping set dynamics [9], BP guided decimation (BPGD) [48], and symmetry break [50]. However, many of these methods have been evaluated under simplified noise models that do not fully capture the complexities of real-world quantum hardware. Additionally, some of these approaches involve complex algorithmic structures that are difficult to implement within the strict time constraints of quantum hardware platforms.

To mitigate the non-convergence issue of BP, Ordered Statistics Decoding (OSD) [15] has been proposed as a post-process technique. BP+OSD significantly improves decoding accuracy over BP by solving a linear system, but it suffers from high latency due to costly operations such as matrix inversion. Several methods have been proposed to enhance the efficiency of BP+OSD. For example, BP+GDG [21] introduces a sliding window decoder based on BPGD under circuit-level noise. AC [45] divides the syndrome data into clusters that can be decoded independently and achieves a latency of $135\mu s$ per round when decoding the 144-qubit Gross code. However, despite these advances, these decoders still fail to meet the real-time requirement of superconducting platforms ($1\mu s$), even though they achieve accuracy comparable to BP+OSD.

8 Conclusion

In this paper, we propose Vegapunk, a software-hardware co-design framework that enables accurate and real-time decoding of qLDPC codes. We introduce an offline SMT-optimized decoupling strategy and an online hierarchical decoding algorithm to address the trade-off between decoding accuracy and latency. We further design a dedicated accelerator to efficiently exploit the sparsity and parallelism of our decoding algorithm. Experimental results show that Vegapunk achieves real-time decoding within $1\mu s$ for BB codes up to [[784, 24, 24]] with logical error rates comparable to BP+OSD.

Acknowledgments

This work was supported by the National Natural Science Foundation of China (No.62472374) and the Zhejiang Provincial Natural Science Foundation of China under Grant (No.LR25F020002).

References

[1] 2023. Suppressing quantum errors by scaling a surface code logical qubit. *Nature* 614, 7949 (2023), 676–681.

[2] Google Quantum AI et al. 2024. Quantum error correction below the surface code threshold. *Nature* 638, 8052 (2024), 920.

[3] Narges Alavisamani, Suhas Vittal, Ramin Ayanzadeh, Poulami Das, and Moinuddin Qureshi. 2024. Promatch: Extending the Reach of Real-Time Quantum Error Correction with Adaptive Predecoding. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. 818–833.

[4] Elwyn Berlekamp, Robert McEliece, and Henk Van Tilborg. 2003. On the inherent intractability of certain coding problems (corresp.). *IEEE Transactions on Information theory* 24, 3 (2003), 384–386.

[5] Yonatan Bilu and Shlomo Hoory. 2004. On codes from hypergraphs. *European Journal of Combinatorics* 25, 3 (2004), 339–354.

[6] Sergey Bravyi, Andrew W Cross, Jay M Gambetta, Dmitri Maslov, Patrick Rall, and Theodore J Yoder. 2024. High-threshold and low-overhead fault-tolerant quantum memory. *Nature* 627, 8005 (2024), 778–782.

[7] Nikolas P Breuckmann and Jens Niklas Eberhardt. 2021. Quantum low-density parity-check codes. *Prx Quantum* 2, 4 (2021), 040101.

[8] A Robert Calderbank and Peter W Shor. 1996. Good quantum error-correcting codes exist. *Physical Review A* 54, 2 (1996), 1098.

[9] Dimitris Chytas, Michele Pacenti, Nithin Raveendran, Mark F Flanagan, and Bane Vasić. 2024. Enhanced message-passing decoding of degenerate quantum codes utilizing trapping set dynamics. *IEEE Communications Letters* 28, 3 (2024), 444–448.

[10] Gergely Csáji. 2022. On the complexity of stable hypergraph matching, stable multicommodity flow and related problems. *Theoretical Computer Science* 931 (2022), 1–16. https://doi.org/10.1016/j.tcs.2022.07.025

[11] Poulami Das, Christopher A Pattison, Srilatha Manne, Douglas M Carmean, Krysta M Svore, Moinuddin Qureshi, and Nicolas Delfosse. 2022. Afs: Accurate, fast, and scalable error-decoding for fault-tolerant quantum computers. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 259–273.

[12] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

[13] Eric Dennis, Alexei Kitaev, Andrew Landahl, and John Preskill. 2002. Topological quantum memory. *J. Math. Phys.* 43, 9 (2002), 4452–4505.

[14] Laird Egan, Dripto M Debroy, Crystal Noel, Andrew Risinger, Daiwei Zhu, Debopriyo Biswas, Michael Newman, Muyuan Li, Kenneth R Brown, Marko Cetina, et al. 2020. Fault-tolerant operation of a quantum error-correction code. *arXiv preprint arXiv:2009.11482* (2020).

[15] Marc PC Fossorier and Shu Lin. 2002. Soft-decision decoding of linear block codes based on ordered statistics. *IEEE Transactions on information Theory* 41, 5 (2002), 1379–1396.

[16] Austin G Fowler, Matteo Mariantoni, John M Martinis, and Andrew N Cleland. 2012. Surface codes: Towards practical large-scale quantum computation. *Physical Review A—Atomic, Molecular, and Optical Physics* 86, 3 (2012), 032324.

[17] Austin G Fowler, Ashley M Stephens, and Peter Groszkowski. 2009. High-threshold universal quantum computation on the surface code. *Physical Review A—Atomic, Molecular, and Optical Physics* 80, 5 (2009), 052312.

[18] Kosuke Fukui, Akihisa Tomita, Atsushi Okamoto, and Keisuke Fujii. 2018. High-threshold fault-tolerant quantum computation with analog quantum error correction. *Physical review X* 8, 2 (2018), 021054.

[19] Craig Gidney. 2021. Stim: a fast stabilizer circuit simulator. *Quantum* 5 (2021), 497.

[20] Craig Gidney and Martin Ekerå. 2021. How to factor 2048 bit RSA integers in 8 hours using 20 million noisy qubits. *Quantum* 5 (2021), 433.

[21] Anqi Gong, Sebastian Cammerer, and Joseph M Renes. 2024. Toward low-latency iterative decoding of QLDPC codes under circuit-level noise. *arXiv preprint arXiv:2403.18901* (2024).

[22] Timo Hillmann, Lucas Berent, Armanda O Quintavalle, Jens Eisert, Robert Wille, and Joschka Roffe. 2024. Localized statistics decoding: A parallel decoding algorithm for quantum low-density parity-check codes. *arXiv preprint arXiv:2406.18655* (2024).

[23] Adam Holmes, Mohammad Reza Jokar, Ghasem Pasandi, Yongshan Ding, Massoud Pedram, and Frederic T Chong. 2020. NISQ+: Boosting quantum computing power by approximating quantum error correction. In *2020 ACM/IEEE 47th annual international symposium on computer architecture (ISCA)*. IEEE, 556–569.

[24] Pavithran Iyer and David Poulin. 2015. Hardness of decoding quantum stabilizer codes. *IEEE Transactions on Information Theory* 61, 9 (2015), 5209–5223.

[25] Sebastian Krinner, Nathan Lacroix, Ants Remm, Agustin Di Paolo, Elie Genois, Catherine Leroux, Christoph Hellings, Stefania Lazar, Francois Swiadek, Johannes Herrmann, et al. 2022. Realizing repeated quantum error correction in a distance-three surface code. *Nature* 605, 7911 (2022), 669–674.

[26] Frank R Kschischang, Brendan J Frey, and H-A Loeliger. 2002. Factor graphs and the sum-product algorithm. *IEEE Transactions on information theory* 47, 2 (2002), 498–519.

[27] Kao-Yueh Kuo and Ching-Yi Lai. 2022. Exploiting degeneracy in belief propagation decoding of quantum codes. *npj Quantum Information* 8, 1 (2022), 111.

[28] Chia-Kai Liang, Chao-Chung Cheng, Yen-Chieh Lai, Liang-Gee Chen, and Homer H Chen. 2011. Hardware-efficient belief propagation. *IEEE Transactions on Circuits and Systems for Video Technology* 21, 5 (2011), 525–537.

[29] Daniel Litinski. 2019. A game of surface codes: Large-scale quantum computing with lattice surgery. *Quantum* 3 (2019), 128.

[30] Ye-Hua Liu and David Poulin. 2019. Neural belief-propagation decoders for quantum error-correcting codes. *Physical review letters* 122, 20 (2019), 200501.

[31] Mohammadreza Noormandipour and Tobias Haug. 2024. MaxSAT decoders for arbitrary CSS codes. *arXiv preprint arXiv:2410.01673* (2024).

[32] Josias Old and Manuel Rispler. 2023. Generalized belief propagation algorithms for decoding of surface codes. *Quantum* 7 (2023), 1037.

[33] Pavel Panteleev and Gleb Kalachev. 2021. Degenerate quantum LDPC codes with good finite length performance. *Quantum* 5 (2021), 585.

[34] Judea Pearl. 2022. Reverend Bayes on inference engines: A distributed hierarchical approach. In *Probabilistic and causal inference: the works of Judea Pearl*. 129–138.

[35] David Poulin and Yeojin Chung. 2008. On the iterative decoding of sparse quantum codes. *arXiv preprint arXiv:0801.1241* (2008).

[36] Nithin Raveendran and Bane Vasić. 2021. Trapping sets of quantum LDPC codes. *Quantum* 5 (2021), 562.

[37] Gokul Subramanian Ravi, Jonathan M Baker, Arash Fayyazi, Sophia Fuhui Lin, Ali Javadi-Abhari, Massoud Pedram, and Frederic T Chong. 2023. Better than worst-case decoding for quantum error correction. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 88–102.

[38] Joschka Roffe, David R White, Simon Burton, and Earl Campbell. 2020. Decoding across the quantum low-density parity-check code landscape. *Physical Review Research* 2, 4 (2020), 043423.

[39] Joschka Roffe, David R White, Simon Burton, and Earl Campbell. 2020. Decoding across the quantum low-density parity-check code landscape. *Physical Review Research* 2, 4 (2020), 043423.

[40] Peter W Shor. 1995. Scheme for reducing decoherence in quantum computer memory. *Physical review A* 52, 4 (1995), R2493.

[41] Barbara M Terhal. 2015. Quantum error correction for quantum memories. *Reviews of Modern Physics* 87, 2 (2015), 307–346.

[42] Javier Valls, Francisco Garcia-Herrero, Nithin Raveendran, and Bane Vasić. 2021. Syndrome-based min-sum vs OSD-0 decoders: FPGA implementation and analysis for quantum LDPC codes. *IEEE Access* 9 (2021), 138734–138743.

[43] Suhas Vittal, Poulami Das, and Moinuddin Qureshi. 2023. Astrea: Accurate quantum error-decoding via practical minimum-weight perfect-matching. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*. 1–16.

[44] Suhas Vittal, Ali Javadi-Abhari, Andrew W Cross, Lev S Bishop, and Moinuddin Qureshi. 2024. Flag-Proxy Networks: Overcoming the Architectural, Scheduling and Decoding Obstacles of Quantum LDPC Codes. In *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 718–734.

[45] Stasiu Wolanski and Ben Barber. 2024. Ambiguity Clustering: an accurate and efficient decoder for qLDPC codes. *arXiv preprint arXiv:2406.14527* (2024).

[46] Yue Wu, Namitha Liyanage, and Lin Zhong. 2025. Micro Blossom: Accelerated Minimum-Weight Perfect Matching Decoding for Quantum Error Correction. *arXiv preprint arXiv:2502.14787* (2025).

[47] Yue Wu and Lin Zhong. 2023. Fusion blossom: Fast mwpm decoders for qec. In *2023 IEEE International Conference on Quantum Computing and Engineering (QCE)*, Vol. 1. IEEE, 928–938.

[48] Hanwen Yao, Waleed Abu Laban, Christian Häger, Alexandre Graell i Amat, and Henry D Pfister. 2024. Belief propagation decoding of quantum LDPC codes with guided decimation. In *2024 IEEE International Symposium on Information Theory (ISIT)*. IEEE, 2478–2483.

[49] Jonathan S Yedidia, William T Freeman, Yair Weiss, et al. 2003. Understanding belief propagation and its generalizations. *Exploring artificial intelligence in the new millennium* 8, 236–239 (2003), 0018–9448.

[50] Keyi Yin, Xiang Fang, Jixuan Ruan, Hezi Zhang, Dean Tullsen, Andrew Sornborger, Chenxu Liu, Ang Li, Travis Humble, and Yufei Ding. 2024. SymBreak: Mitigating Quantum Degeneracy Issues in QLDPC Code Decoders by Breaking Symmetry. *arXiv preprint arXiv:2412.02885* (2024).