RECURRENCE STRIKES BACK: EFFICIENT NEURAL DECODING BEYOND TRANSFORMERS

Anonymous authors
Paper under double-blind review

ABSTRACT

Recent work has made attention-based neural decoders substantially more efficient, yet it leaves a more fundamental question open: is attention the right computational primitive for error correction at all? Error correction is naturally expressed as repeated constraint resolution over a sparse code graph, suggesting recurrence rather than token interaction. We introduce **Neural Syndrome Flow (NSF)**, a recurrent neural decoder that replaces attention with sparse message passing over the Tanner graph. NSF maintains persistent variable- and check-node states and repeatedly applies a shared, state-conditioned transition across decoding iterations. Lightweight edge gating, syndrome feedback, and adaptive solver control further refine this recurrent process. We evaluate NSF across a range of LDPC codes, including MacKay constructions, with particular emphasis on long blocks up to length 2304. NSF outperforms EfficientMPT, the previous state-of-the-art attention-based neural decoder for long LDPC codes; at 4 dB, it reduces BER by $2.77\times$ on the $(1920, 1600)$ code and by $2.74\times$ on the $(2304, 1920)$ code. Across the long-code suite, NSF uses 2.20–$2.98\times$ fewer FLOPs and 3.63–$6.84\times$ lower peak inference memory than EfficientMPT even at the full step budget, while syndrome-based early stopping at 4 dB increases the FLOP reduction to 8.01–$11.00\times$. A length-invariant shared-initialization variant uses only 152k trainable parameters, $12.0\times$ fewer than EfficientMPT, while retaining most of the decoding gain. These results show that structured recurrence can simultaneously improve decoding accuracy and computational efficiency for long-block LDPC neural decoding.

1 INTRODUCTION

Decoding a linear code is, at its core, a repeated attempt to satisfy a known set of constraints. The parity-check matrix specifies which variables are coupled, while the syndrome identifies the constraints that remain violated by the current estimate. Classical iterative decoders exploit exactly this structure: they revisit the same sparse graph, exchange information between variable and check nodes, and progressively drive the received word toward parity consistency (Gallager, 1962; Tanner, 1981). In this sense, error correction is naturally a recurrent computation.

Early neural decoders largely preserved this view. Learned belief-propagation methods introduced trainable components into Tanner-graph updates, while recurrent formulations shared parameters across decoding iterations (Nachmani et al., 2016; 2018). Syndrome-based neural decoding further demonstrated that recurrent networks can learn powerful decoding rules directly from channel reliabilities and parity information (Bennatan et al., 2018). More recently, graph neural decoders have continued this line by learning message-passing functions directly on the code graph (Cammerer et al., 2022). As Transformer-based models became increasingly successful, however, the strongest model-free neural decoders moved toward attention-based architectures.

ECCT established masked self-attention as an effective mechanism for neural decoding by deriving its connectivity from the parity-check matrix (Choukroun & Wolf, 2022). Subsequent work improved generalization across codes and refined how code structure enters the attention mechanism. CrossMPT separated magnitude and syndrome representations and iteratively exchanged information between them through masked cross-attention (Park et al., 2025). EfficientMPT pushed this direction further: instead of constructing conventional attention maps, it compresses the query into

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

a global vector and uses lightweight element-wise interactions, substantially reducing memory and computation while retaining strong decoding performance on long codes (Park et al., 2026). Thus, the recent progress in neural decoding has not merely made Transformers more accurate; it has also made their attention mechanisms considerably more efficient.

This progress motivates a different question. If attention can already be made efficient, *is attention the right computational primitive for decoding in the first place?* Unlike a generic sequence-modeling problem, decoding comes with an explicit sparse interaction graph and an observable residual signal: after every recurrent update, the syndrome reveals which parity constraints remain unsatisfied. The decoder therefore need not repeatedly rediscover arbitrary interactions between positions. It can instead maintain a state, propagate information only through valid code connections, and use the evolving constraint violations to determine how that state should change.

We take this perspective with **Neural Syndrome Flow (NSF)**, a neural decoder formulated as a learned recurrent solver. Rather than applying a sequence of attention blocks, NSF maintains evolving states on the Tanner graph and repeatedly applies a learned transition that moves the decoding state toward parity consistency. Crucially, the recurrent state is carried exclusively by the variable and check nodes: Tanner edges mediate transient information exchange but do not maintain persistent edge-wise messages across decoding iterations. Information is exchanged only along Tanner-graph edges through sparse, edge-gated message passing, while the current syndrome is fed back into the recurrent computation so that every iteration is conditioned on the residual errors left by the preceding ones. The same learned dynamics are reused throughout the solving process, decoupling the number of decoding iterations from the number of distinct transformation blocks.

This formulation changes the role of the code structure. In attention-based decoders, the parity-check matrix constrains, masks, or mediates interactions inside an attention mechanism. In NSF, the Tanner graph is the computational substrate itself. Recurrence is therefore not introduced merely as a way to reduce model size: it defines how the decoder reasons about unresolved parity constraints over time. We additionally develop a shared-initialization formulation in which the recurrent solver no longer requires code-length-dependent state initialization, yielding a compact architecture whose trainable parameterization does not grow with the block length.

The resulting picture reverses the recent architectural trajectory of neural decoding. Attention displaced recurrence as the dominant design choice, and increasingly specialized attention mechanisms then made Transformer decoders substantially more efficient. NSF shows that recurrence becomes competitive again when it is aligned with the sparse constraint structure of the decoding problem. Across LDPC-family codes, with particular emphasis on long LDPC blocks, NSF achieves higher decoding accuracy than EfficientMPT while being markedly smaller and cheaper to execute. NSF contains only about 170k trainable parameters on short codes and remains below 0.5M parameters on the longest codes we consider. Moreover, a single recurrent NSF iteration requires less computation than a single EfficientMPT block. The gain therefore does not come from exchanging accuracy for efficiency: NSF improves decoding performance while simultaneously reducing model size, FLOPs, and peak memory usage. In this sense, the recurrent solver shifts the performance–efficiency frontier rather than merely moving along it. Our primary focus is on LDPC-family codes, particularly long sparse codes whose Tanner-graph structure naturally supports local recurrent message passing; BCH and Polar codes are included as additional cross-family evaluations in Appendix A.

Our contributions are:

- **Neural decoding as learned constraint solving.** We formulate neural decoding as a recurrent dynamical process that repeatedly updates a latent decoding state toward parity consistency, using the residual syndrome as an explicit signal of which constraints remain unresolved.
- **Sparse Tanner-graph recurrence without attention.** We introduce NSF, which combines edge-gated message passing, recurrent bit and check states, and dynamic syndrome feedback while reusing the same learned transition across decoding iterations. The resulting computation follows the sparse structure of the code rather than an attention map.
- **Higher accuracy at substantially lower computational cost.** Across long-code benchmarks, NSF achieves lower BER than EfficientMPT through the waterfall region, including up to a $2.77\times$ reduction at 4 dB, while using 2.20–$2.98\times$ fewer FLOPs and 3.63–$6.84\times$ less peak inference memory at the full 12-step budget. NSF uses only about 170k trainable

parameters on short codes and remains below 0.5M even on the longest evaluated blocks, yielding an increasingly favorable performance–efficiency trade-off as code length grows.

- **A compact, length-invariant recurrent solver.** We further introduce a shared-initialization variant that removes the remaining length-dependent initialization parameters. This yields a code-length-invariant NSF with approximately 152k trainable parameters, while preserving most of the decoding advantage of the full model.

2 Related Work

Learned iterative and graph-based decoding. Learning-based channel decoders have long exploited the iterative structure of classical decoding rather than replacing it with a generic feed-forward model. Neural belief-propagation methods introduced trainable parameters into conventional Tanner-graph updates and showed that sharing these parameters across iterations yields compact recurrent decoders (Nachmani et al., 2016; 2018). Syndrome-based neural decoding further demonstrated that recurrent models can effectively combine channel reliabilities with parity information (Bennatan et al., 2018). A complementary line of work replaces the hand-designed BP rules themselves with learned graph computation. Cammerer et al. Cammerer et al. (2022) formulate decoding as a fully learned message-passing neural network on the Tanner graph, using shared node and edge update functions and a parameterization that naturally scales across block lengths. Tian et al. Tian et al. (2023) introduce an edge-weighted GNN in which message weights are inferred from node and edge reliabilities, further demonstrating transfer across code lengths and rates. Most recently, Qian et al. Qian et al. (2026) follow the BP-iteration framework and replace each BP iteration with a GAT–GRU block, combining learned graph-attention aggregation with recurrent node updates and demonstrating transfer across LDPC lengths and rates.

These works establish learned message passing and recurrence as powerful inductive biases for decoding, but they leave a nontrivial efficiency problem. General neural edge updates must be evaluated throughout the graph, and adding increasingly expressive attention and recurrent modules further increases the cost of each iteration. Cammerer et al. explicitly identify per-edge neural inference as a remaining complexity bottleneck, while GAT–GRU improves decoding by adding both graph attention and gated recurrent computation. Their scaling experiments primarily establish transfer to larger Tanner graphs rather than jointly targeting decoding performance and computational efficiency in the long-code regime. NSF takes a different direction: it treats recurrence as the decoding algorithm itself, confines persistent state to the variable and check nodes, and communicates projected node states through lightweight scalar edge gates rather than learned vector-valued edge messages, with explicit syndrome feedback closing the recurrent loop. This sparse solver design is evaluated directly for both decoding performance and inference efficiency on long LDPC blocks.

Transformer-based neural decoding. A second line of work moved model-free neural decoding toward Transformer architectures. ECCT (Choukroun & Wolf, 2022) introduced parity-check-aware masked self-attention and demonstrated that Transformer representations can substantially improve neural decoding. Subsequent work has progressively incorporated more of the algebraic and iterative structure of error-correcting codes into this framework. Foundation ECCT (Choukroun & Wolf, 2024) develops position-, length-, and code-invariant components that allow a single decoder to operate across multiple codes. CrossMPT (Park et al., 2025) separates magnitude and syndrome representations and updates them iteratively through masked cross-attention, bringing Transformer decoding closer to conventional message passing.

EfficientMPT (Park et al., 2026) substantially narrows the efficiency gap of this family. Rather than constructing conventional attention maps, its EEC attention compresses the query into a global representation and combines it with code-aware key features through lightweight vector operations. This removes the dominant dense attention interaction and makes the attention operation scale linearly with block length. Consequently, the motivation for NSF is not simply that standard attention is computationally expensive: recent Transformer decoders have already made attention considerably more efficient. Our question is more fundamental — whether attention is needed as the central computational primitive for decoding at all. NSF instead represents decoding as recurrent constraint solving on the Tanner graph, using the residual syndrome to guide how the state evolves from one iteration to the next.

Attention-free scalable neural decoding. Recent work has also begun to challenge attention-based decoding directly. MMPD (Gusev et al., 2026) combines local message passing along Tanner-graph edges with bidirectional Mamba state-space blocks (Gu & Dao, 2024), providing long-range information exchange without dense attention. This demonstrates that competitive neural decoding does not require a Transformer and motivates attention-free architectures that scale more favorably with block length. NSF shares this broad objective but follows a different computational principle. MMPD augments local graph interactions with a separate state-space sequence model for global mixing; NSF keeps the entire decoding process within a sparse recurrent solver. Bit and check states evolve through Tanner-graph message passing, while the syndrome is recomputed and fed back at every step as an explicit measure of the remaining constraint violations. The resulting architecture requires neither attention nor a separate global sequence model: global progress emerges from repeated local propagation under shared recurrent dynamics.

Taken together, prior work has explored three increasingly structured ways to learn decoding: parameterizing classical iterative updates, learning message passing on the code graph, and embedding code structure inside increasingly efficient sequence models. NSF combines the scalability of graph-structured computation with the statefulness of recurrent decoding, but uses the evolving parity constraints themselves to organize the computation. This learned-solver view distinguishes NSF from both attention-based decoders and existing recurrent or state-space message-passing architectures.

3 Neural Syndrome Flow

3.1 Problem Formulation

We consider a binary linear block code $\mathcal{C} \subseteq \{0,1\}^n$ of dimension k, specified by a parity-check matrix $\mathbf{H} \in \{0,1\}^{m\times n}$, where typically $m = n-k$. A vector $\mathbf{c} \in \{0,1\}^n$ is a valid codeword when $\mathbf{Hc} = \mathbf{0} \pmod 2$. Under BPSK modulation, $\mathbf{c}$ is mapped to $\mathbf{x} = \mathbf{1} - 2\mathbf{c}$ and transmitted over an additive white Gaussian noise (AWGN) channel, $\mathbf{y} = \mathbf{x} + \boldsymbol{\nu}, \boldsymbol{\nu} \sim \mathcal{N}(\mathbf{0}, \sigma^2\mathbf{I})$.

From the received vector $\mathbf{y}$, we obtain the binary hard-decision vector $\hat{\mathbf{y}}$ with entries $\hat{y}_i = \mathbf{1}_{y_i<0}$ and the reliability vector $\mathbf{r} = |\mathbf{y}|$. The corresponding syndrome is computed as $\mathbf{s}_0 = \mathbf{H}\hat{\mathbf{y}} \pmod 2$, where $\mathbf{s}_0 \neq \mathbf{0}$ indicates that the hard decision violates at least one parity constraint. We also define the signed syndrome as $\tilde{\mathbf{s}}_0 = \mathbf{1} - 2\mathbf{s}_0 \in \{-1,+1\}^m$.

Let $\mathbf{e} = \hat{\mathbf{y}} \oplus \mathbf{c}$ denote the binary error pattern induced by the channel. Since every valid codeword satisfies the parity constraints,

$$\mathbf{s}_0 = \mathbf{H}\hat{\mathbf{y}} = \mathbf{H}(\mathbf{c} \oplus \mathbf{e}) = \mathbf{He} \pmod 2. \tag{1}$$

Thus, decoding can be expressed directly as inference of an unknown error pattern whose parity structure is specified by the observed syndrome. Once an estimate $\hat{\mathbf{e}}$ is obtained, the decoded codeword is simply $\hat{\mathbf{c}} = \hat{\mathbf{y}} \oplus \hat{\mathbf{e}}$.

Equation 1 is the starting point of NSF. The reliabilities $\mathbf{r}$ provide local evidence about which hard decisions are likely to be incorrect, while $\mathbf{s}_0$ specifies the global parity constraints that the error estimate must satisfy. The parity-check matrix additionally induces a bipartite Tanner graph $\mathcal{G} = (\mathcal{V}, \mathcal{C}, \mathcal{E})$, with one variable node $v_i \in \mathcal{V}$ for each codeword bit and one check node $c_j \in \mathcal{C}$ for each row of $\mathbf{H}$. An edge $(v_i, c_j) \in \mathcal{E}$ exists exactly when $H_{j,i} = 1$.

3.2 NSF as a Recurrent Constraint Solver

NSF treats decoding as a learned iterative constraint-solving process. At iteration t, the solver maintains error logits $\mathbf{z}^{(t)} \in \mathbb{R}^n$, variable-node states $\mathbf{U}_v^{(t)} \in \mathbb{R}^{n\times d}$, and check-node states $\mathbf{U}_c^{(t)} \in \mathbb{R}^{m\times d}$, where d is the hidden-state dimension. Positive $z_i^{(t)}$ indicates that bit i is currently predicted to be erroneous. All learnable parameters of the recurrent transition are shared across iterations up to $T_{\max}$; only the solver state evolves with t. Thus, increasing the number of solving steps does not require adding new decoder blocks. Moreover, every intermediate state already defines a complete error hypothesis and can therefore be checked directly against the code constraints.

Initialization. Using the reliability vector $\mathbf{r}$ and the signed observed syndrome $\tilde{\mathbf{s}}_0$ defined above, NSF initializes an error logit for each variable node as $z_i^{(0)} = b_0 - |a_0| r_i, \quad p_i^{(0)} = \sigma(z_i^{(0)})$, where

$p_i^{(0)}$ is the initial estimated error probability and $a_0, b_0 \in \mathbb{R}$ are learned scalars. $\mathbf{u}_{v,i}^{(0)} = r_i \boldsymbol{\eta}_{v,i} + \boldsymbol{\beta}_{v,i} + \phi_v^{\text{init}}\left([r_i, p_i^{(0)}]\right)$, $\mathbf{u}_{c,j}^{(0)} = \tilde{s}_{0,j}\boldsymbol{\eta}_{c,j} + \boldsymbol{\beta}_{c,j} + \phi_c^{\text{init}}(\tilde{s}_{0,j})$. Here, $\boldsymbol{\eta}_{v,i}, \boldsymbol{\beta}_{v,i}, \boldsymbol{\eta}_{c,j}, \boldsymbol{\beta}_{c,j} \in \mathbb{R}^d$ are learned position-specific initialization vectors, while $\phi_v^{\text{init}} : \mathbb{R}^2 \to \mathbb{R}^d$ and $\phi_c^{\text{init}} : \mathbb{R} \to \mathbb{R}^d$ are two-layer MLPs with GELU activations.

Dynamic syndrome feedback. The key feedback signal in NSF is the agreement between the current error hypothesis and the observed parity constraints. The error logits are mapped to soft spins, from which NSF computes a differentiable syndrome prediction and residual:

$$q_i^{(t)} = \tanh\left(-\frac{z_i^{(t)}}{\tau_s}\right), \qquad \pi_j^{(t)} = \prod_{i \in \mathcal{N}_G(j)} q_i^{(t)}, \qquad \rho_j^{(t)} = \tilde{s}_{0,j}\pi_j^{(t)}. \tag{2}$$

Here, $\mathcal{N}_G(j) \subseteq \mathcal{V}$ denotes the variable nodes participating in check j, and $\tau_s > 0$ is the soft-parity temperature. The quantity $\rho_j^{(t)}$ is a signed parity-consistency score: values $\rho_j^{(t)} \approx 1$ indicate that the current error hypothesis explains the observed parity of check j, whereas $\rho_j^{(t)} \approx -1$ indicates an unresolved constraint. Equivalently, matching the observed syndrome means that applying the current error hypothesis to the hard decision would satisfy check j; for an exact error estimate, the syndrome of the resulting decoded codeword is therefore zero. Because $\boldsymbol{\rho}^{(t)}$ is recomputed after every update of $\mathbf{z}^{(t)}$, parity consistency becomes an evolving feedback signal, while the observed syndrome $\tilde{\mathbf{s}}_0$ remains fixed. This recurrent circulation of parity information motivates the name *Neural Syndrome Flow*.

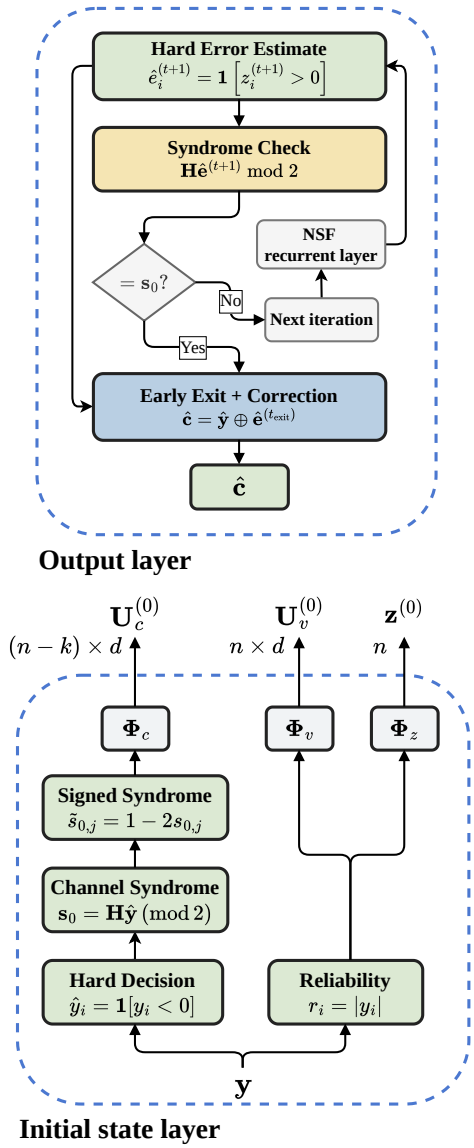


Figure 1: **Overview of NSF.**

Edge-gated Tanner-graph message passing. NSF propagates information only along edges defined by the parity-check matrix. For each existing Tanner edge $(u, v) \in \mathcal{E}$, a lightweight gate network predicts an edge-specific, state-dependent weight $g_{uv} \in (0, 1)$. Messages arriving at a target node u are aggregated as

$$\mathbf{m}_u = \mathbf{W}_o \frac{\sum_{v \in \mathcal{N}_G(u)} g_{uv} \mathbf{W}_m \mathbf{u}_v}{\max\left(\sum_{v \in \mathcal{N}_G(u)} g_{uv}, \epsilon\right)}, \qquad g_{uv} = \sigma\left(\frac{f_g(\mathbf{u}_u, \mathbf{u}_v, \xi_u, \xi_v)}{\tau_g}\right). \tag{3}$$

Here, $\epsilon = 10^{-6}$, $\tau_g > 0$ is the gate temperature, and $\mathbf{W}_m, \mathbf{W}_o \in \mathbb{R}^{d \times d}$ are learned message/output projections. The gate uses low-rank projections $\bar{\mathbf{u}}_u = \mathbf{P}_t \mathbf{u}_u$ and $\bar{\mathbf{u}}_v = \mathbf{P}_s \mathbf{u}_v$: $f_g(\mathbf{u}_u, \mathbf{u}_v, \xi_u, \xi_v) = \text{MLP}_g([\bar{\mathbf{u}}_u \odot \bar{\mathbf{u}}_v,\ \bar{\mathbf{u}}_u - \bar{\mathbf{u}}_v,\ \xi_u, \xi_v])$, where $\mathbf{P}_t, \mathbf{P}_s : \mathbb{R}^d \to \mathbb{R}^r$ are learned projections, r is the edge-gating rank, and MLP_g is a lightweight two-layer gate network producing the scalar score f_g, which is mapped through the sigmoid in equation 3 to obtain the gate g_{uv}. The scalar gate features also depend on the propagation direction. For variable-to-check propagation from variable i to check j, $\xi_u = \rho_j^{(t)}$ and $\xi_v = 2p_i^{(t)} - 1$. The transformation $2p_i^{(t)} - 1$ follows the error-logit sign convention: positive values indicate that bit

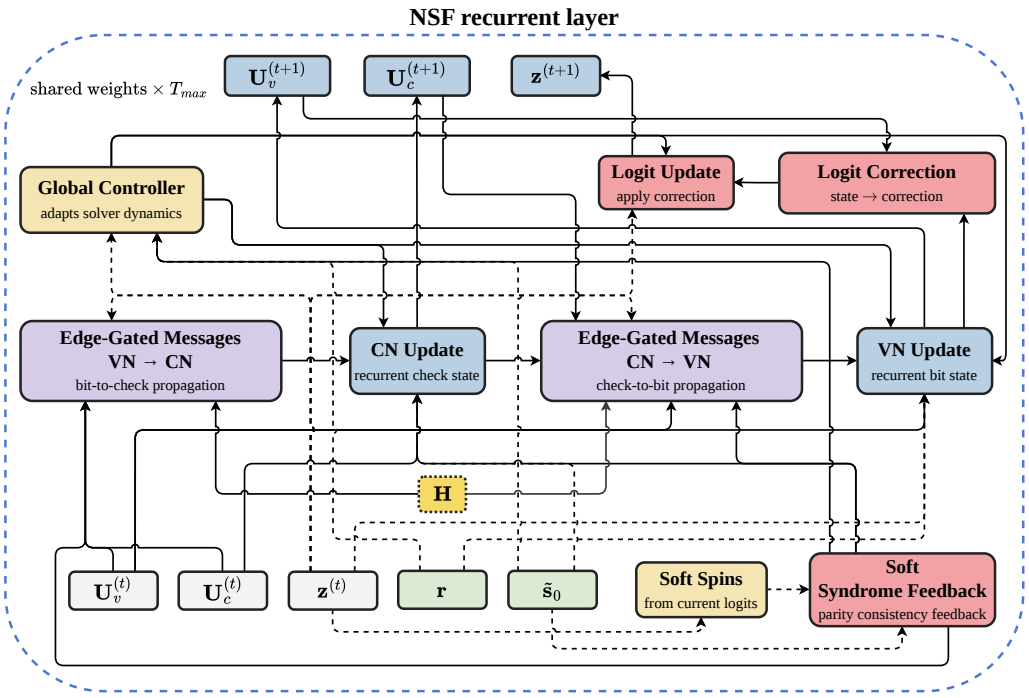


Figure 2: **One recurrent solving step of NSF.** Variable and check states exchange information only along Tanner-graph edges through sparse edge-gated message passing. The current error logits induce soft bit spins and a soft syndrome residual, which is fed back into the recurrent update. A lightweight global controller modulates message scale, memory mixing, damping, and logit step size. The resulting variable states produce a correction to the error logits. The same transition is shared across all decoding iterations.

i is more likely to be erroneous. For check-to-variable propagation from check j to variable i, $\xi_u = 2p_i^{(t)} - 1$ and $\xi_v = -\rho_j^{(t)}$. Unlike attention, the gates are evaluated only on existing Tanner edges and are independent sigmoids rather than a softmax over all positions.

The variable-to-check messages are augmented with an explicit representation of the current parity state. In particular, $\mathbf{f}_{c,j}^{(t)} = \phi_c\Big(\big[\tilde{s}_{0,j}, \pi_j^{(t)}, \rho_j^{(t)}, \mathbf{1}_{\{\rho_j^{(t)}<0\}}\big]\Big)$. After the check update, information is propagated back to the variable nodes. Writing $p_i^{(t)} = \sigma(z_i^{(t)})$ and $\kappa_i^{(t)} = 2|p_i^{(t)} - \frac{1}{2}|$, the bit-side features are $\mathbf{f}_{v,i}^{(t)} = \phi_v([r_i, \tanh(z_i^{(t)}), p_i^{(t)}, \kappa_i^{(t)}])$. Here, $\phi_c, \phi_v : \mathbb{R}^4 \to \mathbb{R}^d$ are learned GELU projections. Thus, each step combines persistent states with channel, error, and syndrome features.

The two node updates are performed sequentially within each recurrent iteration. NSF first updates the check states using $\mathbf{U}_v^{(t)}$ and $\mathbf{U}_c^{(t)}$. The resulting $\mathbf{U}_c^{(t+1)}$ is then immediately used for check-to-variable propagation before updating $\mathbf{U}_v^{(t+1)}$. Both node types are updated by compact recurrent units:

$$\begin{aligned}\mathbf{U}_c^{(t+1)} &= \mathcal{R}_c\Big(\mathbf{U}_c^{(t)}, \lambda_m^{(t)}\mathbf{M}_c^{(t)}, \mathbf{F}_c^{(t)}; \mu^{(t)}\Big),\\ \mathbf{U}_v^{(t+1)} &= \mathcal{R}_v\Big(\mathbf{U}_v^{(t)}, \lambda_m^{(t)}\mathbf{M}_v^{(t)}, \mathbf{F}_v^{(t)}; \mu^{(t)}\Big),\end{aligned} \tag{4}$$

where $\mathbf{M}_c^{(t)}$ and $\mathbf{M}_v^{(t)}$ are the aggregated messages and $\mathbf{F}_c^{(t)}$ and $\mathbf{F}_v^{(t)}$ collect the corresponding local features defined above. Each $\mathcal{R}$ consists of a GRU update, controller-dependent interpolation with the previous state, and a residual feed-forward refinement. The recurrent memories preserve information across solving steps, while the Tanner graph determines where this information can propagate.

Table 1: Comparison of decoding performance at three SNR values (4, 5, 6 dB) for EfficientMPT, CrossMPT, MMPD, and NSF. Results are reported as the negative natural logarithm of BER ($-\ln(\mathrm{BER})\uparrow$). We additionally report the average number of recurrent iterations used by NSF before termination, denoted by $\overline{T}_{\mathrm{NSF}}$.

Codes	(n,k)	**EfficientMPT** $1.09M$			**CrossMPT** $1.2M$			**MMPD** $1.2M$			**NSF** $0.17M, T_{\max}=12$			$\overline{T}_{\mathrm{NSF}}$		
		4 dB	5 dB	6 dB	4 dB	5 dB	6 dB	4 dB	5 dB	6 dB	4 dB	5 dB	6 dB	4 dB	5 dB	6 dB
LDPC	(49,24)	6.78	9.70	13.68	6.68	9.52	13.19	6.78	9.67	13.68	**7.01**	**9.97**	**13.82**	1.67	1.20	1.05
	(121,60)	5.62	9.04	14.73	5.74	9.26	14.78	5.84	9.62	15.15	**6.19**	**9.98**	**15.76**	2.70	1.52	1.12
	(121,70)	6.96	11.31	16.97	7.06	11.39	17.52	7.24	11.57	17.72	**7.68**	**12.15**	**17.86**	2.04	1.29	1.05
	(121,80)	7.95	12.40	17.90	7.99	12.75	18.15	8.07	12.83	18.17	**8.62**	**13.36**	**18.67**	1.75	1.19	1.03
MacKay	(96,48)	7.91	11.46	15.44	7.97	11.77	15.52	8.25	12.01	15.85	**8.78**	**12.52**	**15.95**	2.03	1.40	1.11

Adaptive solver control and logit refinement. Received words may differ substantially in reliability and decoding difficulty, so NSF does not use a fixed update strength. A lightweight two-layer MLP controller summarizes the current solver state using the mean, standard deviation, and minimum channel reliability, the fraction of negative entries in the observed signed syndrome, the mean and maximum absolute error-logit magnitudes, the mean soft parity-consistency score, and the normalized iteration index. It outputs four sample-dependent controls: message scale $\lambda_m^{(t)}$, recurrent memory mixing $\mu^{(t)}$, logit step size $\alpha^{(t)}$, and damping $\delta^{(t)}$.

After the variable-state update, a small readout predicts a scalar correction for every error logit:

$$\Delta z_i^{(t)} = f_\Delta\left(\mathbf{u}_{v,i}^{(t+1)}\right), \qquad z_i^{(t+1)} = z_i^{(t)} + \alpha^{(t)}\gamma_\Delta \Delta z_i^{(t)} - 0.01\,\delta^{(t)} z_i^{(t)}. \tag{5}$$

Here, $f_\Delta : \mathbb{R}^d \to \mathbb{R}$ is a shared node-wise readout network and $\gamma_\Delta \in \mathbb{R}$ is a learned global correction scale, initialized to 0.1. The updated logits define the next error hypothesis, from which a new soft syndrome residual is computed, closing the recurrent loop shown in Figure 2.

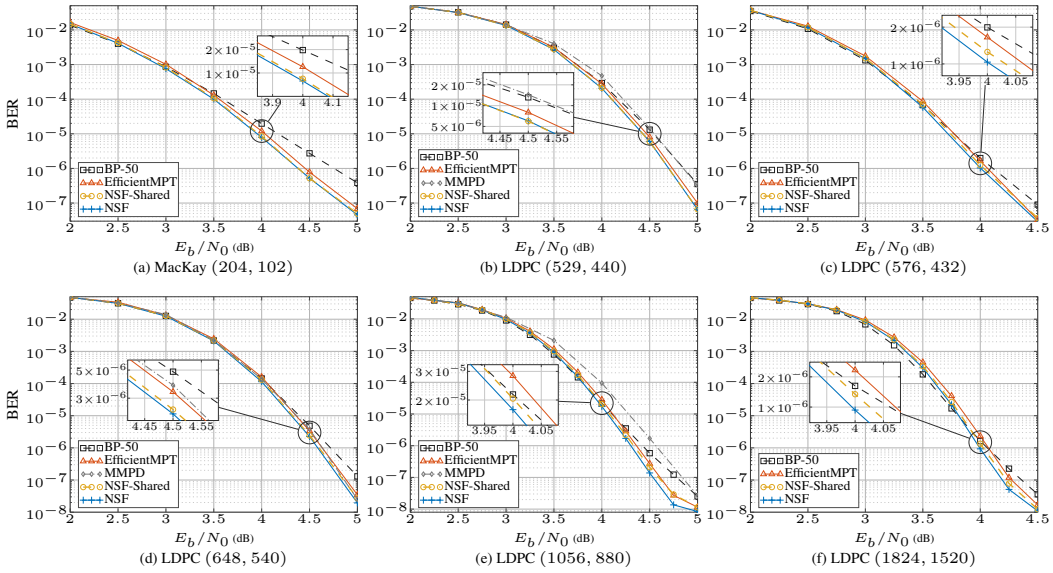
Syndrome-based early stopping. Every recurrent step produces a complete candidate error pattern, $\hat{\varepsilon}_i^{(t)} = \mathbf{1}_{\{z_i^{(t)}>0\}}$. Inference terminates as soon as the candidate satisfies the observed syndrome constraint, $\mathbf{H}\hat{\boldsymbol{\varepsilon}}^{(t)} = \mathbf{s}_0 \pmod 2$, or when $T_{\max}$ iterations are reached. The final codeword is obtained as $\hat{\mathbf{c}} = \hat{\mathbf{y}} \oplus \hat{\boldsymbol{\varepsilon}}$. Thus, NSF does not require a separately learned stopping classifier: the same parity constraints that define the decoding problem provide an exact parity-consistency test for the current solution.

Shared initialization. All learned transition functions are shared across node positions and recurrent iterations. Variable-to-check and check-to-variable propagation use separate message-passing parameters, and variable and check nodes use separate recurrent units. In the default NSF, the initialization embeddings and biases remain position-specific. The shared-initialization variant ties $\boldsymbol{\eta}_{v,i}$ and $\boldsymbol{\beta}_{v,i}$ across all variable nodes and ties $\boldsymbol{\eta}_{c,j}$ and $\boldsymbol{\beta}_{c,j}$ across all check nodes. This removes the remaining dependence of the trainable parameter count on the block length while leaving the recurrent solver itself unchanged. We refer to this length-invariant model as *NSF-Shared* (*NSF-S* in tables).

Training objective. Let $\ell_{\mathrm{BCE}}(\mathbf{z},\mathbf{e})$ denote binary cross-entropy with the true channel-induced error pattern $\mathbf{e}$. During training, NSF is unrolled for all $T = T_{\max}$ recurrent steps. We apply deep supervision to every recurrent iteration, give the final iterate an additional unit-weight BCE term to emphasize terminal decoding accuracy, and additionally encourage parity consistency: $\mathcal{L} = \sum_{t=1}^{T} w_t \ell_{\mathrm{BCE}}(\mathbf{z}^{(t)},\mathbf{e}) + \ell_{\mathrm{BCE}}(\mathbf{z}^{(T)},\mathbf{e}) + \frac{\lambda_s}{Tm}\sum_{t=1}^{T}\sum_{j=1}^{m}\mathrm{softplus}\left(-2\rho_j^{(t)}\right), w_t = \frac{\gamma^{T-t}}{\sum_{\ell=1}^{T}\gamma^{T-\ell}}$. We use $\gamma = 0.9$, $\lambda_s = 0.02$, $\tau_g = 0.7$, and $\tau_s = 1$ in all NSF experiments.

4 Experimental Results

Training protocol. Unless stated otherwise, NSF uses a hidden dimension of $d = 64$, an edge-gating rank of $r = 32$, and a maximum of $T_{\max} = 12$ recurrent decoding iterations. Both the recurrent update MLPs and the global controller use an expansion factor of 2, and dropout is set to zero. The recurrent transition is shared across iterations, and inference terminates early as soon



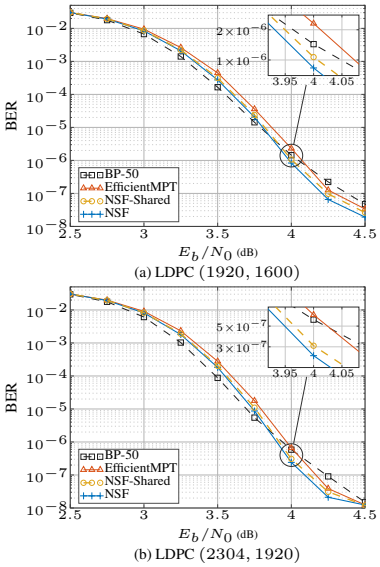
(a) MacKay $(204, 102)$ (b) LDPC $(529, 440)$ (c) LDPC $(576, 432)$

(d) LDPC $(648, 540)$ (e) LDPC $(1056, 880)$ (f) LDPC $(1824, 1520)$

Figure 3: BER comparison on six long-code benchmarks. NSF and NSF-Shared are compared with EfficientMPT and BP-50; MMPD is shown where available.

as the current error estimate satisfies all parity constraints. NSF is trained for 1000 epochs on the short-code benchmarks, matching the standard protocol used in prior neural decoding work. For long-code experiments, all models trained in this work use a common budget of 2000 epochs, with 1000 minibatches per epoch and a batch size of 128. Training samples are generated online over an AWGN channel, with E_b/N_0 sampled uniformly from 3 to 7 dB. We use AdamW with an initial learning rate of 10^{-4}, cosine decay to 10^{-6}, and weight decay 10^{-4}. We use the longer common budget on long codes to reduce sensitivity to architecture-dependent convergence speed.

Evaluation protocol. We exploit the symmetry of binary linear codes over the BPSK–AWGN channel and evaluate the all-zero codeword. For NSF, this symmetry is reflected directly in its inputs: by equation 1, the observed syndrome depends only on the error pattern, while $|\mathbf{y}|$ is codeword-invariant in distribution. At each reported SNR, Monte Carlo evaluation continues until at least 10^5 codewords and 500 frame errors have been observed, with a hard cap of 10^9 codewords for very low error rates. BER is computed over all n codeword bits of all evaluated codewords. NSF uses the exact syndrome test described above for early termination, and we additionally report its mean number of recurrent iterations.



(a) LDPC $(1920, 1600)$

(b) LDPC $(2304, 1920)$

Figure 4: BER comparison on the two longest LDPC codes.

Baselines. We compare the proposed NSF and NSF-Shared with EfficientMPT (Park et al., 2026), MMPD (Gusev et al., 2026), and also report CrossMPT on the short-code benchmarks. For short codes, we use published results of EfficientMPT, CrossMPT, and MMPD when available; for configurations not reported in the original works, we train the corresponding model following the authors' training protocol. For long codes, EfficientMPT is trained for 2000 epochs using $h = 8$, $d = 128$, and $N = 10$, while MMPD is retrained for 2000 epochs using the code-specific configurations reported in the original work. BP-50 denotes standard flooding sum-product belief propagation.

Table 1 shows a consistent advantage on the short LDPC-family suite: NSF achieves the lowest BER at all 15 operating points across five codes, outperforming EfficientMPT, CrossMPT, and MMPD.

Table 2: Computational efficiency. Percentages are relative to EfficientMPT (100%) for the same code. Memory denotes the model-related peak inference footprint. The NSF / NSF-S columns at 4 dB use the measured mean stopping time; NSF max uses all $T_{\max} = 12$ iterations.

Codes	Parameter	Inference memory [MiB]		FLOPs [G]			# parameters [M]	
		EffMPT / MMPD	NSF / NSF-S	EffMPT / MMPD	NSF / NSF-S @ 4 dB	NSF $T_{\max}$	EffMPT / MMPD	NSF / NSF-S
LDPC	$(49, 24)$	4.49 / 5.85 (100% / 130%)	**0.78 / –** (17% / –)	0.172 / 0.138 (100% / 80%)	**0.019 / –** (11% / –)	**0.136** (79%)	1.093 / 1.208 (100% / 110%)	**0.162 / –** (15% / –)
MacKay	$(96, 48)$	4.80 / 6.01 (100% / 125%)	**0.90 / –** (19% / –)	0.329 / 0.240 (100% / 73%)	**0.044 / –** (13% / –)	**0.251** (76%)	1.093 / 1.216 (100% / 111%)	**0.171 / –** (16% / –)
LDPC	$(121, 60)$	4.97 / 6.50 (100% / 131%)	**1.07 / –** (22% / –)	0.434 / 0.394 (100% / 91%)	**0.079 / –** (18% / –)	**0.345** (80%)	1.093 / 1.222 (100% / 112%)	**0.176 / –** (16% / –)
LDPC	$(121, 70)$	4.96 / 6.37 (100% / 128%)	**1.02 / –** (21% / –)	0.406 / 0.352 (100% / 87%)	**0.056 / –** (14% / –)	**0.321** (79%)	1.093 / 1.220 (100% / 112%)	**0.175 / –** (16% / –)
LDPC	$(121, 80)$	4.94 / 6.24 (100% / 126%)	**0.97 / –** (20% / –)	0.377 / 0.311 (100% / 83%)	**0.045 / –** (12% / –)	**0.297** (79%)	1.093 / 1.219 (100% / 112%)	**0.173 / –** (16% / –)
MacKay	$(204, 102)$	8.32 / 6.51 (100% / 78%)	**1.22 / 1.07** (15% / 13%)	1.223 / 0.511 (100% / 42%)	**0.111 / 0.111** (9% / 9%)	**0.532** (44%)	1.821 / 1.237 (100% / 68%)	**0.191 / 0.152** (11% / 8%)
LDPC	$(529, 440)$	10.32 / 8.40 (100% / 81%)	**2.04 / 1.74** (20% / 17%)	2.517 / 1.239 (100% / 49%)	**0.291 / 0.291** (12% / 12%)	**1.143** (45%)	1.821 / 1.277 (100% / 70%)	**0.232 / 0.152** (13% / 8%)
LDPC	$(576, 432)$	10.78 / 9.59 (100% / 89%)	**2.19 / 1.84** (20% / 17%)	3.060 / 1.519 (100% / 50%)	**0.309 / 0.317** (10% / 10%)	**1.302** (43%)	1.821 / 1.290 (100% / 71%)	**0.244 / 0.152** (13% / 8%)
LDPC	$(648, 540)$	11.12 / 8.82 (100% / 79%)	**2.34 / 1.97** (21% / 18%)	3.121 / 1.487 (100% / 48%)	**0.386 / 0.394** (12% / 13%)	**1.381** (44%)	1.821 / 1.295 (100% / 71%)	**0.249 / 0.152** (14% / 8%)
LDPC	$(1056, 880)$	14.02 / 10.80 (100% / 77%)	**3.39 / 2.79** (24% / 20%)	5.473 / 2.382 (100% / 44%)	**0.683 / 0.704** (12% / 13%)	**2.229** (41%)	1.821 / 1.356 (100% / 74%)	**0.310 / 0.152** (17% / 8%)
LDPC	$(1824, 1520)$	20.05 / 14.99 (100% / 75%)	**5.41 / 4.37** (27% / 22%)	10.708 / 4.115 (100% / 38%)	**1.284 / 1.319** (12% / 12%)	**3.851** (36%)	1.821 / 1.470 (100% / 81%)	**0.425 / 0.152** (23% / 8%)
LDPC	$(1920, 1600)$	20.86 / 15.54 (100% / 74%)	**5.66 / 4.57** (27% / 22%)	11.437 / 4.331 (100% / 38%)	**1.365 / 1.404** (12% / 12%)	**4.053** (35%)	1.821 / 1.485 (100% / 82%)	**0.439 / 0.152** (24% / 8%)
LDPC	$(2304, 1920)$	24.21 / 17.73 (100% / 73%)	**6.67 / 5.36** (28% / 22%)	14.517 / 5.198 (100% / 36%)	**1.690 / 1.726** (12% / 12%)	**4.864** (34%)	1.821 / 1.542 (100% / 85%)	**0.496 / 0.152** (27% / 8%)

It uses about 0.17M trainable parameters versus roughly 1.09–1.2M for the neural baselines and requires only a few recurrent iterations on average.

Figures 3 and 4 show that the advantage of NSF becomes more pronounced in the waterfall region. At 4 dB, NSF achieves lower BER than EfficientMPT on every long-code benchmark shown, including reductions of $2.77\times$ on LDPC $(1920, 1600)$ and $2.74\times$ on LDPC $(2304, 1920)$. At the same 4 dB operating point, NSF also outperforms BP-50 on all eight long-code benchmarks shown. Notably, NSF-Shared keeps the parameter count fixed at 152k across block lengths while remaining close to standard NSF in BER, showing that the long-code gains do not rely on length-specific initialization parameters.

5 Computational Efficiency

Table 2 compares the inference cost of the configurations used in the decoding experiments, except for MMPD, for which we use the fixed configuration from the short-code experiments. FLOPs are recomputed under a common analytical convention; for NSF, we report both the full recurrent budget and the effective cost at 4 dB using the measured mean stopping time. Peak inference memory is measured for batch-size-one FP32 inference. Complete calculation details are given in Appendix D. Even with all 12 recurrent steps, NSF uses 2.20–$2.98\times$ fewer FLOPs than EfficientMPT across the long-code suite. With exact-syndrome early stopping at 4 dB, the reduction grows to 8.01–$11.00\times$, while peak inference memory is 3.63–$6.84\times$ lower (4.52–$7.79\times$ for NSF-Shared). NSF-Shared additionally remains fixed at 0.152M parameters, compared with 1.821M for long-code EfficientMPT.

6 Conclusion

We introduced Neural Syndrome Flow, a compact recurrent decoder that propagates state-dependent messages over the Tanner graph and feeds the evolving syndrome back into the decoding state. Across LDPC codes up to length 2304, NSF improves waterfall-region decoding while substantially reducing FLOPs and peak inference memory relative to EfficientMPT; NSF-Shared further reduces the model to 152k parameters. Ablations show that the gain persists without the BP-inspired leave-one-out approximation evaluated here, supporting sparse, syndrome-conditioned recurrence as an efficient alternative to attention for long-block neural decoding.

AI USE STATEMENT

Generative AI tools were used solely to improve the readability of the manuscript and to correct grammatical errors. They were not used for research ideation, methodology, experimental design, or interpretation of results. All AI-assisted edits were reviewed by the authors, who take full responsibility for the final content.

REPRODUCIBILITY STATEMENT

The supplementary material includes the source code for the proposed NSF model, as well as the training code used in our experiments. The model architecture, training and evaluation protocols, stopping rule, and computational accounting are specified in the main paper and appendices. All experiments reported in this work were run on NVIDIA A100 GPUs. If the paper is accepted, we will make the full code publicly available and include a link to the corresponding GitHub repository in the final version of the paper.

REFERENCES

Amir Bennatan, Yoni Choukroun, and Pavel Kisilev. Deep learning for decoding of linear codes-a syndrome-based approach. In *2018 IEEE International Symposium on Information Theory (ISIT)*, pp. 1595–1599. IEEE, 2018.

Sebastian Cammerer, Jakob Hoydis, Fayçal Aït Aoudia, and Alexander Keller. Graph neural networks for channel decoding. In *2022 IEEE Globecom Workshops (GC Wkshps)*, pp. 486–491. IEEE, 2022.

Yoni Choukroun and Lior Wolf. Error correction code transformer. *Advances in Neural Information Processing Systems*, 35:38695–38705, 2022.

Yoni Choukroun and Lior Wolf. A foundation model for error correction codes. In *International Conference on Learning Representations*, volume 2024, pp. 2153–2171, 2024.

Robert Gallager. Low-density parity-check codes. *IRE Transactions on information theory*, 8(1): 21–28, 1962.

Albert Gu and Tri Dao. Mamba: Linear-time sequence modeling with selective state spaces. 2024. URL https://openreview.net/forum?id=tEYskw1VY2.

Rostislav Gusev, Nikita Aleksandrov, Artem Solomkin, and Dmitry Artemasov. Scalable mamba-based message-passing neural decoder for error-correcting codes, 2026. URL https://arxiv.org/abs/2605.10681.

Eliya Nachmani, Yair Be'Ery, and David Burshtein. Learning to decode linear codes using deep learning. In *2016 54th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, pp. 341–346. IEEE, 2016.

Eliya Nachmani, Elad Marciano, Loren Lugosch, Warren J Gross, David Burshtein, and Yair Be'ery. Deep learning methods for improved decoding of linear codes. *IEEE Journal of Selected Topics in Signal Processing*, 12(1):119–131, 2018.

Seong-Joon Park, Hee-Youl Kwak, Sang-Hyo Kim, Yongjune Kim, and Jong-Seon No. Crossmpt: Cross-attention message-passing transformer for error correcting codes. In Y. Yue, A. Garg, N. Peng, F. Sha, and R. Yu (eds.), *International Conference on Learning Representations*, volume 2025, pp. 56710–56728, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/file/8f0942c43fcfba4cc66a859b9fcb1bba-Paper-Conference.pdf.

Seong-Joon Park, Taewoo Park, Hee-Youl Kwak, Sang-Hyo Kim, Yongjune Kim, and Jong-Seon No. Efficient message-passing transformer for error correcting codes. In C. Vondrick, B. Hariharan, C. Raffel, L. Pinto, D. Yang, and A. Faust (eds.), *International Conference on Learning Representations*, volume 2026, pp. 16773–16787,

2026. URL https://proceedings.iclr.cc/paper_files/paper/2026/file/1c2fc6d9343ae8c16aff3934b09bb040-Paper-Conference.pdf.

Lixin Qian, Yang Lu, Xuefeng Wang, Wei Chen, and Bo Ai. Learning to decode: Graph attention network-gated recurrent unit as channel decoder. *IEEE Wireless Communications Letters*, 15: 2929–2933, 01 2026. doi: 10.1109/LWC.2026.3688747.

R Tanner. A recursive approach to low complexity codes. *IEEE Transactions on information theory*, 27(5):533–547, 1981.

Kou Tian, Chentao Yue, Changyang She, Yonghui Li, and Branka Vucetic. A scalable graph neural network decoder for short block codes. In *ICC 2023-IEEE International Conference on Communications*, pp. 1268–1273. IEEE, 2023.

A Additional Short-Code Results

We additionally evaluate NSF on the broader set of short-code benchmarks reported by EfficientMPT (Park et al., 2026). Following the original comparison, we report performance at $E_b/N_0 = \{4, 5, 6\}$ dB and use $-\ln(\mathrm{BER})$ throughout, with larger values indicating better decoding performance. EfficientMPT and CrossMPT results are converted from the BER values reported in the original work.

These cross-family results, reported in Table 3, also clarify the scope of NSF. NSF is competitive on several BCH settings but is not uniformly strongest, while the attention-based baselines remain stronger on some Polar codes. Our main claims therefore concern sparse LDPC-family codes, where NSF's recurrent Tanner-graph inductive bias is best aligned with the decoding structure; extending the same advantage consistently across other code families remains an open direction.

B Component Ablations

We isolate three architectural components of NSF on the LDPC $(121, 60)$ code: state-dependent edge gating, dynamic syndrome feedback, and adaptive solver control. Each ablation modifies a single component of NSF while preserving the remaining architecture and training protocol.

Removing edge gates. In NSF, a message from a neighboring node v to a target node u is modulated by a state-dependent scalar gate $g_{uv} \in (0, 1)$. The resulting aggregation has the form

$$\mathbf{m}_u = \mathbf{W}_o \frac{\sum_{v \in \mathcal{N}_G(u)} g_{uv} \mathbf{W}_m \mathbf{u}_v}{\max\left(\sum_{v \in \mathcal{N}_G(u)} g_{uv}, \epsilon\right)}, \qquad g_{uv} = \sigma(f_g(\mathbf{u}_u, \mathbf{u}_v, \xi_u, \xi_v)/\tau_g)\,, \tag{6}$$

where ξ_u and ξ_v contain the current scalar solver features at the two endpoints. In the *No edge gates* ablation, all valid Tanner edges receive unit weight, $g_{uv} = 1$. Message passing therefore reduces to degree-normalized averaging,

$$\mathbf{m}_u = \mathbf{W}_o \frac{1}{|\mathcal{N}_G(u)|} \sum_{v \in \mathcal{N}_G(u)} \mathbf{W}_m \mathbf{u}_v, \tag{7}$$

where $|\mathcal{N}_G(u)|$ is the degree of node u. This preserves the Tanner-graph topology and recurrent updates while removing the ability to selectively amplify or suppress individual edges as the decoding state evolves.

Removing dynamic syndrome feedback. NSF converts its current error logits into soft bit spins and recomputes a predicted parity state after every recurrent step,

$$q_i^{(t)} = \tanh\left(-z_i^{(t)}/\tau_s\right), \qquad \pi_j^{(t)} = \prod_{i \in \mathcal{N}_G(j)} q_i^{(t)}, \qquad \rho_j^{(t)} = \tilde{s}_{0,j} \pi_j^{(t)}. \tag{8}$$

The predicted syndrome $\pi^{(t)}$ and residual $\rho^{(t)}$ are then used by the check-node features, edge gates, and adaptive controller at the next solver step. In the *No dynamic syndrome feedback* ablation, we

Table 3: Additional short-code decoding results at three SNR values (4, 5, and 6 dB). Results are reported as $-\ln(\mathrm{BER})\uparrow$. EfficientMPT and CrossMPT values are converted from the BER results reported by Park et al. (2026).

Codes	(n,k)	**EfficientMPT** $1.09M$			**CrossMPT** $1.20M$			**NSF** $0.17M$		
		4 dB	5 dB	6 dB	4 dB	5 dB	6 dB	4 dB	5 dB	6 dB
BCH	(31,16)	**7.00**	**9.38**	**12.54**	6.98	9.25	12.48	6.03	8.10	10.72
	(63,36)	4.90	6.64	9.23	**5.03**	**6.91**	9.37	4.86	6.77	**9.60**
	(63,45)	5.73	8.05	11.11	**5.90**	**8.20**	**11.61**	5.37	7.57	10.63
Polar	(64,32)	7.44	**10.10**	**13.37**	**7.50**	9.97	13.31	5.72	7.77	10.16
	(64,48)	6.36	8.58	11.02	**6.51**	**8.70**	**11.31**	6.40	8.45	10.88
	(128,64)	7.05	10.69	**14.94**	**7.52**	**11.21**	14.76	5.89	9.11	13.56

Table 4: Component ablations on LDPC $(121, 60)$. Results are reported as $-\ln(\mathrm{BER})\uparrow$. Each variant changes a single component of NSF while retaining the same training and evaluation protocol.

Variant	**4 dB**	**5 dB**	**6 dB**
NSF	6.19	9.98	15.76
No edge gates	6.00	9.78	15.45
No dynamic syndrome feedback	6.09	9.85	15.49
Static controller	6.08	9.86	15.44

set $\pi^{(t)} = \mathbf{0}$ and $\rho^{(t)} = \mathbf{0}$ inside the recurrent transition for every t. Importantly, the observed channel syndrome $\widetilde{\mathbf{s}}_0$ is still provided to the model, so this ablation removes only feedback derived from the model's *current prediction* rather than removing syndrome information altogether. The auxiliary soft-syndrome term in the training objective is also left unchanged, isolating the architectural feedback pathway from syndrome-based regularization.

Removing adaptive solver control. The default controller adapts the recurrent dynamics separately for every received word and solver iteration. Its input is an eight-dimensional statistical feature representation

$$\mathbf{c}^{(t)} = \big[\,\mathrm{mean}(\mathbf{r}), \mathrm{std}(\mathbf{r}), \min(\mathbf{r}), w_s, \mathrm{mean}\,|\mathbf{z}^{(t)}|, \max|\mathbf{z}^{(t)}|, \mathrm{mean}(\boldsymbol{\rho}^{(t)}), t/T_{\max}\big], \tag{9}$$

where w_s is the fraction of violated checks in the observed syndrome. An MLP maps $\mathbf{c}^{(t)}$ to four raw controller outputs $(a_m^{(t)}, a_\alpha^{(t)}, a_\delta^{(t)}, a_\mu^{(t)})$, which determine the message scale, logit-update scale, damping, and recurrent-memory mixing through

$$\lambda_m^{(t)} = 0.5 + \sigma(a_m^{(t)}), \quad \alpha^{(t)} = 0.05 + 0.95\,\sigma(a_\alpha^{(t)}), \quad \delta^{(t)} = \sigma(a_\delta^{(t)}), \quad \mu^{(t)} = \sigma(a_\mu^{(t)}). \tag{10}$$

The *Static controller* ablation replaces the input-dependent MLP with four learned scalars shared across all samples and all iterations. The same output transformations and ranges are retained. Thus, this ablation distinguishes the benefit of sample- and state-dependent control from the simpler benefit of learning four global update constants.

Table 4 shows that each single-component ablation causes only a modest degradation, while the full NSF achieves the strongest result at all three reported SNRs. This suggests that the recurrent Tanner-graph solver provides the dominant inductive bias, with edge gating, dynamic syndrome feedback, and adaptive solver control contributing complementary incremental improvements rather than any single component accounting for the overall gain.

C Alternative Message-Passing Rules

We further study whether the performance of NSF depends specifically on its default state-dependent edge-gated aggregation rule. A key distinction from belief propagation is that recurrence in NSF is

carried by the variable- and check-node states rather than by persistent edge-wise messages; Tanner edges mediate state-dependent information exchange but do not retain messages across solver steps. We therefore use the following variants to test whether NSF's gains can be attributed to BP-style extrinsic exclusion or to a particular aggregation mechanism. To isolate the message-passing mechanism from the rest of the recurrent solver, we keep the variable/check recurrent states, dynamic syndrome feedback, adaptive controller, logit refinement, training objective, and stopping rule unchanged, and replace only the aggregation operator used for variable-to-check and check-to-variable propagation.

Extrinsic aggregation. The default NSF aggregation is inclusive: all incoming Tanner-edge messages contribute directly to the target-node update. Writing $\mathbf{x}_{uv} = \mathbf{W}_m \mathbf{u}_v$ for the projected message on edge (u, v) and g_{uv} for its learned gate, the default aggregation is

$$\mathbf{m}_u = \mathbf{W}_o \frac{\sum_{v \in \mathcal{N}_G(u)} g_{uv} \mathbf{x}_{uv}}{\max\left(\sum_{v \in \mathcal{N}_G(u)} g_{uv}, \epsilon\right)}. \tag{11}$$

The *Extrinsic* variant introduces a BP-inspired leave-one-out construction without adding persistent edge states or trainable parameters. For each incoming edge (u, v), we first form a leave-one-out aggregate that excludes that edge,

$$\mathbf{c}_{u \setminus v} = \frac{\sum_{j \in \mathcal{N}_G(u) \setminus \{v\}} g_{uj} \mathbf{x}_{uj}}{\max\left(\sum_{j \in \mathcal{N}_G(u) \setminus \{v\}} g_{uj}, \epsilon\right)}, \tag{12}$$

with degree-one leave-one-out aggregates defined as zero. This produces one edge-specific aggregate for each incoming edge. Because NSF maintains node states rather than persistent edge states, these edge-specific aggregates are then pooled back into a single target-node message using the original learned gates,

$$\mathbf{m}_u^{\text{ext}} = \mathbf{W}_o \frac{\sum_{v \in \mathcal{N}_G(u)} g_{uv} \mathbf{c}_{u \setminus v}}{\max\left(\sum_{v \in \mathcal{N}_G(u)} g_{uv}, \epsilon\right)}. \tag{13}$$

Thus, the Extrinsic variant preserves the same state-dependent gates and parameter count as NSF while introducing a BP-inspired leave-one-out construction. Because NSF does not maintain persistent edge-wise states, this variant should be viewed as a node-state approximation to extrinsic aggregation rather than a full reproduction of BP-style edge-message dynamics.

EEC-style aggregation. As a second alternative, we replace the edge-gated operator with an EfficientMPT-style EEC interaction (Park et al., 2026). This ablation retains the recurrent NSF solver but removes the learned edge-specific gates and scalar endpoint features from message aggregation. Let $\mathbf{X} \in \mathbb{R}^{N_s \times d}$ denote the source-node states for one propagation direction. Here, $N_s = n$ for variable-to-check propagation and $N_s = m$ for check-to-variable propagation. For each head $a \in \{1, \ldots, h\}$, with $h = 8$ and $d_h = d/h$, we form query and key projections using $\mathbf{W}_Q^{(a)}, \mathbf{W}_K^{(a)} \in \mathbb{R}^{d \times d_h}$:

$$\mathbf{Q}^{(a)} = \mathbf{X}\mathbf{W}_Q^{(a)}, \qquad \mathbf{K}^{(a)} = \mathbf{X}\mathbf{W}_K^{(a)}, \tag{14}$$

and obtain a global query by summing across source positions and applying a softmax within the head,

$$\bar{\mathbf{q}}^{(a)} = \mathrm{softmax}\left(\sum_i \mathbf{Q}_i^{(a)}\right). \tag{15}$$

The Tanner-graph connectivity is then used as a sparse binary parity-check multiplication,

$$\widetilde{\mathbf{K}}^{(a)} = \mathbf{A}\mathbf{K}^{(a)}, \tag{16}$$

where $\mathbf{A} = \mathbf{H}$ for variable-to-check propagation and $\mathbf{A} = \mathbf{H}^\top$ for check-to-variable propagation. The resulting message is

$$\mathbf{m} = \mathrm{concat}_{a=1}^{h}\left(\bar{\mathbf{q}}^{(a)} \odot \widetilde{\mathbf{K}}^{(a)}\right)\mathbf{W}_o, \tag{17}$$

Table 5: Comparison of message-passing rules on short codes. Results are reported as $-\ln(\mathrm{BER})\uparrow$ after 1000 training epochs. All variants share the same recurrent NSF solver and differ only in the message aggregation rule.

Family	(n,k)	**NSF**			**Extrinsic**			**EEC-style**		
		4 dB	5 dB	6 dB	4 dB	5 dB	6 dB	4 dB	5 dB	6 dB
BCH	$(63,36)$	4.86	6.77	9.60	4.82	6.72	9.50	3.60	4.64	6.13
Polar	$(128,64)$	5.89	9.11	13.56	5.90	9.08	13.57	3.81	5.27	7.18
LDPC	$(121,60)$	6.19	9.98	15.76	6.16	9.95	15.67	4.16	6.26	9.55

Table 6: Message-passing variants on the long LDPC $(576,432)$ code. Results are $-\ln(\mathrm{BER})\uparrow$.

Variant	3 dB	3.5 dB	4 dB	4.5 dB
NSF	6.52	9.75	13.78	17.36
Extrinsic	6.50	9.74	13.70	17.34
EEC-style	4.27	5.59	7.33	9.28

This operator uses no value projection, no target-node features, and no edge-specific scalar gates. The sparse implementation is algebraically equivalent to multiplying the key features by the binary PCM while avoiding a dense matrix multiplication. Therefore, this experiment tests whether the recurrent NSF scaffold remains effective when its local state-conditioned propagation rule is replaced by an EEC-style global query–key interaction.

Table 5 shows that the Extrinsic variant closely tracks the default NSF, remaining within 0.1 in $-\ln(\mathrm{BER})$ across all nine operating points. In contrast, the EEC-style replacement degrades performance at every operating point, with the gap reaching 3.47–6.38 in $-\ln(\mathrm{BER})$ at 6 dB. These results indicate that NSF is not sensitive to this node-state leave-one-out modification; rather, preserving local, state-conditioned propagation over the Tanner graph is substantially more important than the precise aggregation rule.

Table 6 shows that the Extrinsic variant closely tracks the default NSF across the reported SNRs, differing by at most 0.08 in $-\ln(\mathrm{BER})$. In contrast, replacing the NSF message-passing rule with the EEC-style interaction leads to a substantial degradation at every operating point. The gap relative to NSF increases from 2.25 at 3 dB to 8.08 at 4.5 dB. These results indicate that the BP-inspired leave-one-out node-state approximation evaluated here does not improve over the default NSF, while NSF's recurrent Tanner-graph message-passing mechanism remains important for strong decoding performance on long codes.

D Computational Complexity Accounting

This appendix provides the computational accounting used in Table 2. We use a common arithmetic convention for NSF, EfficientMPT, and MMPD and derive all FLOP counts directly from the corresponding computation graphs. This avoids differences between profiler-specific conventions and, in particular, allows sparse Tanner-graph operations to be treated explicitly.

D.1 Evaluation Configurations

For NSF, all complexity results use the same configuration as in the main experiments: hidden dimension $d = 64$, edge-gating rank $r = 32$, and a maximum recurrent budget of $T_{\max} = 12$. The recurrent transition is shared across iterations.

For EfficientMPT (Park et al., 2026), we use embedding dimension $d_e = 128$ and $h = 8$ attention heads. The number of decoder layers is set to $N = 6$ for the short-code suite and $N = 10$ for the long-code suite. This distinction is important because decoding complexity scales with the number of EfficientMPT layers.

Let

$$\mathbf{H} \in \{0,1\}^{m \times n} \tag{18}$$

denote the parity-check matrix of a code, where n is the block length and m is the number of parity checks. We denote by

$$E = \|\mathbf{H}\|_0 \tag{19}$$

the number of nonzero entries in $\mathbf{H}$, equivalently the number of edges in the corresponding Tanner graph. All graph-dependent NSF costs below are evaluated using the actual value of E for each code.

D.2 FLOP Convention

We count each floating-point multiplication, addition, subtraction, or division as one FLOP. Consequently, an inner product of length q requires

$$q + (q-1) = 2q - 1 \tag{20}$$

FLOPs. In particular, one multiply–accumulate operation corresponds to two FLOPs under this convention.

Non-polynomial elementary functions — including sigmoid, tanh, GELU, exponential, and reciprocal square root — are tracked separately and are not assigned an arbitrary hardware-dependent FLOP equivalent. Similarly, integer operations used for the exact parity-consistency test are not included in the headline floating-point counts.

All entries for NSF, EfficientMPT, and MMPD in Table 2 are recomputed under this same convention.

D.3 Primitive Arithmetic Costs

We first define several primitive costs used throughout the derivation.

Linear projection. For L vectors of input dimension a projected to output dimension b, the cost is

$$C_{\mathrm{lin}}(L, a, b; \beta) = Lb\,(2a - 1 + \beta)\,, \tag{21}$$

where $\beta = 1$ when a bias term is present and $\beta = 0$ otherwise.

Matrix multiplication. For the product of a $p \times q$ matrix and a $q \times r$ matrix,

$$C_{\mathrm{mm}}(p, q, r) = pr(2q - 1). \tag{22}$$

Mean reduction. Computing the mean of a length-L vector requires $L - 1$ additions and one division, giving

$$C_{\mathrm{mean}}(L) = L. \tag{23}$$

Layer normalization. Excluding the reciprocal-square-root operation itself, LayerNorm over L vectors of dimension d costs

$$C_{\mathrm{LN}}(L, d) = L(8d + 1). \tag{24}$$

This includes mean and variance computation, centering, normalization, and the final affine transformation.

GRU update. For a GRU with input dimension a and hidden dimension d, applied to L states, the arithmetic cost is

$$C_{\mathrm{GRU}}(L, a, d) = C_{\mathrm{lin}}(L, a, 3d; 1) + C_{\mathrm{lin}}(L, d, 3d; 1) + 8Ld. \tag{25}$$

The sigmoid and tanh evaluations of the GRU gates are excluded from the headline FLOP count according to the convention above.

D.4 NSF COMPLEXITY

NSF consists of an initialization stage followed by a shared recurrent step. We therefore write the total cost after T recurrent updates as

$$C_{\mathrm{NSF}}(T) = C_{\mathrm{init}} + TC_{\mathrm{step}}. \tag{26}$$

The following expressions correspond to the default normalized sigmoid edge gating used in all main experiments.

D.4.1 EDGE-GATED TANNER AGGREGATION

Consider one message-passing direction with N_s source nodes, N_t target nodes, and E Tanner edges. NSF first forms rank-r representations of source and target states and then evaluates one scalar gate for every existing Tanner edge.

The arithmetic cost of one directional aggregation is

$$\begin{aligned} C_{\mathrm{agg}}(N_t, N_s, E) = {} & \underbrace{\begin{array}{l} C_{\mathrm{lin}}(N_t, d, r; 0) + C_{\mathrm{lin}}(N_s, d, r; 0) \\ +C_{\mathrm{lin}}(N_t, r, r; 0) + C_{\mathrm{lin}}(N_s, r, r; 0) \\ +C_{\mathrm{lin}}(N_t, 1, r; 0) + C_{\mathrm{lin}}(N_s, 1, r; 0) \end{array}}_{\text{endpoint gate features}} \\ & + \underbrace{\begin{array}{c} Er + C_{\mathrm{lin}}(E, r, r; 1) + 4Er \\ +C_{\mathrm{lin}}(E, r, 1; 1) + E \end{array}}_{\text{edge-gate evaluation}} \\ & + \underbrace{E + 2Ed + N_t d}_{\text{gated sparse aggregation}} \\ & + \underbrace{C_{\mathrm{lin}}(N_t, d, d; 0)}_{\text{output projection}}. \end{aligned} \tag{27}$$

The first six linear projections construct the low-rank source and target features. The E-dependent terms account for edge scoring, gating, message weighting, sparse accumulation, and normalization, while the final terms perform the target-node output projection. Importantly, the dominant graph-dependent cost scales with the number of existing Tanner edges rather than with the dense product nm.

For bounded-degree LDPC families, $E = \mathcal{O}(n)$, so the graph propagation component scales linearly with block length.

D.4.2 INITIALIZATION

The initial error logits and variable/check memories are constructed from the channel magnitudes and observed syndrome. The arithmetic cost is

$$\begin{aligned} C_{\mathrm{init}} = {} & 2n + 2nd + C_{\mathrm{lin}}(n, 2, d; 1) + C_{\mathrm{lin}}(n, d, d; 1) + nd \\ & + 2md + C_{\mathrm{lin}}(m, 1, d; 1) + C_{\mathrm{lin}}(m, d, d; 1) + md \\ & + 5n + m. \end{aligned} \tag{28}$$

The final term accounts for the static controller statistics computed from the channel magnitudes and observed syndrome. The associated sigmoid, GELU, and square-root evaluations are excluded according to Section D.2.

D.4.3 ADAPTIVE CONTROLLER

At each recurrent iteration, the controller summarizes the current error logits and syndrome residual together with the static channel statistics. Let

$$d_c = 2d = 128 \tag{29}$$

denote the controller hidden dimension. Its arithmetic cost per recurrent step is

$$\begin{aligned} C_{\mathrm{ctrl}} = {} & C_{\mathrm{mean}}(n) + C_{\mathrm{mean}}(m) \\ & + C_{\mathrm{lin}}(1, 8, d_c; 1) + C_{\mathrm{lin}}(1, d_c, 4; 1) + 3. \end{aligned} \tag{30}$$

The four controller sigmoid evaluations and the GELU activation are again excluded from the reported floating-point total.

D.4.4 Recurrent Node Update

After message aggregation, each variable or check state is updated by a GRU followed by controller-dependent memory interpolation, LayerNorm, and a residual feed-forward refinement.

Let

$$d_f = 2d \tag{31}$$

be the feed-forward hidden dimension used in the recurrent state update. For L nodes, the recurrent update cost after the local feature projection is

$$\begin{aligned} C_{\mathrm{rec}}(L) = {} & Ld + C_{\mathrm{GRU}}(L, 2d, d) + (3Ld + 1) \\ & + C_{\mathrm{LN}}(L, d) + C_{\mathrm{lin}}(L, d, d_f; 1) \\ & + C_{\mathrm{lin}}(L, d_f, d; 1) + Ld. \end{aligned} \tag{32}$$

The first Ld term applies the controller message scale, while $3Ld + 1$ accounts for memory interpolation and the final Ld term for the residual feed-forward addition.

D.4.5 Logit Refinement

After the variable-node update, NSF predicts one correction per error logit. Its cost is

$$\begin{aligned} C_{\mathrm{head}} = {} & C_{\mathrm{LN}}(n, d) + C_{\mathrm{lin}}(n, d, d; 1) \\ & + C_{\mathrm{lin}}(n, d, 1; 1) + 6n. \end{aligned} \tag{33}$$

The final $6n$ operations account for the sample-dependent logit step, damping, and residual update.

D.4.6 One Recurrent Step

The soft syndrome prediction first maps the current logits to soft spins and multiplies them along each parity check. Excluding the tanh evaluations, its arithmetic cost is

$$C_{\mathrm{syn}} = n + (E - m) + m = n + E. \tag{34}$$

The scalar bit features additionally require $4n$ arithmetic operations.

Combining all components, one complete NSF recurrent iteration costs

$$\begin{aligned} C_{\mathrm{step}} = {} & \underbrace{C_{\mathrm{syn}} + C_{\mathrm{ctrl}} + 4n}_{\text{feedback, control, and bit features}} \\ & + \underbrace{C_{\mathrm{agg}}(m, n, E) + C_{\mathrm{lin}}(m, 4, d; 1) + C_{\mathrm{rec}}(m)}_{\text{variable-to-check update}} \\ & + \underbrace{C_{\mathrm{agg}}(n, m, E) + C_{\mathrm{lin}}(n, 4, d; 1) + C_{\mathrm{rec}}(n)}_{\text{check-to-variable update}} \\ & + \underbrace{C_{\mathrm{head}}}_{\text{logit refinement}}. \end{aligned} \tag{35}$$

Thus, both directed message-passing phases explicitly scale with the actual Tanner-graph edge count E.

The exact stopping criterion additionally requires thresholding the n logits, accumulating parity over the E Tanner edges, and checking the m parity constraints. We count these separately as approximately

$$C_{\text{stop}}^{\text{int}} = n + E + 2m \tag{36}$$

integer/index operations per recurrent iteration. They are not included in the floating-point totals in Table 2.

D.4.7 Worst-Case and Effective Decode Cost

The maximum NSF cost is obtained when every received word uses the full recurrent budget:

$$C_{\text{NSF}}^{\max} = C_{\text{init}} + T_{\max} C_{\text{step}}, \qquad T_{\max} = 12. \tag{37}$$

In practice, decoding terminates immediately once the predicted error pattern satisfies the observed syndrome. We therefore additionally report the expected arithmetic cost at 4 dB:

$$C_{\text{NSF}}^{\text{eff}}(4\,\text{dB}) = C_{\text{init}} + \overline{T}_{4\text{dB}} C_{\text{step}}, \tag{38}$$

where $\overline{T}_{4\text{dB}}$ is the empirical mean number of recurrent iterations measured during the same Monte Carlo decoding evaluation used to obtain BER.

NSF and NSF-Shared perform the same arithmetic for a fixed number of iterations. Their effective FLOPs can nevertheless differ because their measured stopping times $\overline{T}$ need not be identical. The corresponding stopping times are reported in Table 7.

D.5 EfficientMPT Complexity

EfficientMPT alternates updates between the magnitude and syndrome domains. We denote the target sequence length of one directional update by N_t and the source sequence length by N_s. Thus, $(N_t, N_s) = (m, n)$ for the magnitude-to-syndrome update and $(N_t, N_s) = (n, m)$ for the syndrome-to-magnitude update.

For EfficientMPT, let $d_e = 128$ and let $h = 8$ denote the number of heads, with per-head dimension

$$d_k = \frac{d_e}{h}. \tag{39}$$

The arithmetic cost of one directional EfficientMPT update is

$$\begin{aligned} C_{\text{dir}}(N_t, N_s) = {} & \underbrace{\begin{array}{c} C_{\text{LN}}(N_t, d_e) + 2C_{\text{lin}}(N_s, d_e, d_e; 1) \\ + d_e N_s \end{array}}_{\text{source preparation}} \\ & + \underbrace{C_{\text{mm}}(d_e, N_s, N_t) + d_e + h(2d_k - 1) + N_t d_e}_{\text{PCM-domain EEC interaction}} \\ & + \underbrace{C_{\text{lin}}(N_t, d_e, d_e; 1) + N_t d_e}_{\text{attention output and residual}} \\ & + \underbrace{\begin{array}{c} C_{\text{LN}}(N_t, d_e) + C_{\text{lin}}(N_t, d_e, 4d_e; 1) \\ + C_{\text{lin}}(N_t, 4d_e, d_e; 1) + N_t d_e \end{array}}_{\text{feed-forward refinement}}. \end{aligned} \tag{40}$$

The first three groups comprise the Efficient Error Correcting attention update: source preparation, the PCM-domain interaction, and the output projection with its residual connection. The final group is the LayerNorm–feed-forward–residual sublayer.

Softmax exponential evaluations and GELU activations are excluded from the headline count according to the common convention in Section D.2.

Each EfficientMPT layer performs one magnitude-to-syndrome update and one syndrome-to-magnitude update. The complete decoder cost is therefore

$$\begin{aligned} C_{\mathrm{EffMPT}} = \; & \underbrace{(n+m)d_e}_{\text{input embeddings}} \\ & + \underbrace{N\left[C_{\mathrm{dir}}(n,m) + C_{\mathrm{dir}}(m,n)\right]}_{\text{alternating decoder layers}} \\ & + \underbrace{\mathbf{1}_{N>1}\left[C_{\mathrm{LN}}(n,d_e) + C_{\mathrm{LN}}(m,d_e)\right]}_{\text{midpoint normalization}} \\ & + \underbrace{\begin{array}{c} C_{\mathrm{LN}}(n,d_e) + C_{\mathrm{LN}}(m,d_e) \\ + C_{\mathrm{mm}}(d_e,m,n) + nd_e + C_{\mathrm{lin}}(n,d_e,1;1) \end{array}}_{\text{final normalization and readout}}. \end{aligned} \tag{41}$$

We evaluate equation 41 with

$$N = \begin{cases} 6, & \text{short-code suite,} \\ 10, & \text{long-code suite.} \end{cases} \tag{42}$$

Unlike NSF, EfficientMPT uses a fixed computational depth and therefore has a single decode FLOP count for a given code and architecture.

D.6 MMPD Complexity

For MMPD (Gusev et al., 2026), we use the same configuration as in the short-code experiments:

$$N_M = 7, \qquad d_M = 64, \qquad r_M = 32, \qquad d_s = 28, \qquad k_M = 4, \tag{43}$$

with Mamba expansion factor $\gamma_M = 1$, gated-update parameters $g_n = 2$ and $g_k = 1$, and feed-forward multiplier $\mu = 1$. Bidirectional Mamba refinement is applied to both variable- and check-node states. The same configuration is used for all codes in Table 2.

MMPD alternates local Tanner-graph aggregation, gated node updates, and bidirectional Mamba refinement. All components are counted under the common arithmetic convention of Section D.2.

Unlike the edge-list aggregation used by NSF, the MMPD pairwise aggregator evaluates fixed-width padded neighborhoods. We therefore define

$$K_v = \max_{v_i \in \mathcal{V}} |\mathcal{N}_G(v_i)|, \qquad K_c = \max_{c_j \in \mathcal{C}} |\mathcal{N}_G(c_j)|, \tag{44}$$

so that the check-to-variable and variable-to-check updates operate on nK_v and mK_c neighbor slots, respectively.

D.6.1 Pairwise Tanner Aggregation

Consider one propagation direction with N_t target nodes and padded neighborhood width K, and let

$$P = N_t K \tag{45}$$

denote the number of processed target–neighbor pairs. For hidden dimension d_M and pairwise rank r_M, one directional pairwise aggregation costs

$$
\begin{aligned}
C_{\mathrm{pair}}(N_t, K) = & \underbrace{\begin{array}{c} C_{\mathrm{lin}}(P, d_M, d_M; 0) + C_{\mathrm{lin}}(N_t, d_M, r_M; 0) \\ +C_{\mathrm{lin}}(P, d_M, r_M; 0) \end{array}}_{\text{pair representations}} \\
& + \underbrace{\begin{array}{c} 2Pr_M + C_{\mathrm{lin}}(P, 2r_M, r_M; 1) \\ +C_{\mathrm{lin}}(P, r_M, 1; 1) \end{array}}_{\text{pairwise compatibility scoring}} \\
& + \underbrace{N_t(2K-1) + Pd_M + N_t d_M(K-1)}_{\text{padded-neighborhood aggregation}} \\
& + \underbrace{C_{\mathrm{lin}}(N_t, d_M, d_M; 0)}_{\text{output projection}}.
\end{aligned}
\tag{46}
$$

The first three projections construct the message representation and low-rank representations of the two endpoints. The $2Pr_M$ term forms the element-wise product and difference used by the pairwise scorer, after which a two-layer network produces one compatibility score for each neighbor slot. The remaining terms account for neighborhood normalization, weighted message aggregation, and the output projection. SiLU and exponential evaluations are excluded from the headline FLOP count according to Section D.2.

D.6.2 Gated Node Update

Following aggregation, MMPD applies a gated residual transition to each node state. The hidden dimension of the gate network is

$$
d_g = \left\lfloor \frac{d_M}{g_n} \right\rfloor g_k, \tag{47}
$$

which gives $d_g = d_M/2$ for the configuration used here.

For L node states, the gated update costs

$$
\begin{aligned}
C_{\mathrm{upd}}(L) = & \, 2C_{\mathrm{LN}}(L, d_M) + C_{\mathrm{lin}}(L, 2d_M, d_g; 1) \\
& + 2C_{\mathrm{lin}}(L, d_g, d_M; 1) + 2Ld_M \\
& + C_{\mathrm{LN}}(L, d_M) + C_{\mathrm{lin}}(L, d_M, 2d_M; 1) \\
& + C_{\mathrm{lin}}(L, 2d_M, d_M; 1) + Ld_M.
\end{aligned}
\tag{48}
$$

The first two LayerNorm operations normalize the current state and aggregated message. Their concatenation is projected to the gate representation, from which separate projections produce the interpolation gate and residual update. The final LayerNorm and two-layer feed-forward network refine the resulting node state. GELU and sigmoid evaluations are excluded according to the common FLOP convention.

D.6.3 Bidirectional Mamba Refinement

Each node stream is subsequently refined by a bidirectional Mamba block. Let $d_i = \gamma_M d_M$ denote the Mamba inner dimension and let r_δ denote the rank of the low-rank time-step (Δ) projection. Following the default `dt_rank="auto"` rule of the reference Mamba-v1 implementation (Gu & Dao, 2024),

$$
r_\delta = \left\lceil \frac{d_M}{16} \right\rceil. \tag{49}
$$

For our configuration, $d_M = 64$, yielding $r_\delta = 4$. The state-space dimension is d_s and the depthwise convolution width is k_M.

For a sequence of L node states, one directional Mamba transformation has arithmetic cost

$$
\begin{aligned}
C_{\mathrm{Mamba}}(L) = & \underbrace{C_{\mathrm{lin}}(L, d_M, 2d_i; 0) + 2Ld_i k_M}_{\text{input projection and convolution}} \\
& + \underbrace{\begin{array}{c} C_{\mathrm{lin}}(L, d_i, r_\delta + 2d_s; 0) \\ +C_{\mathrm{lin}}(L, r_\delta, d_i; 0) \end{array}}_{\text{selective-parameter projections}} \\
& + \underbrace{7Ld_i d_s + 3Ld_i}_{\text{selective state-space recurrence}} \\
& + \underbrace{C_{\mathrm{lin}}(L, d_i, d_M; 0)}_{\text{output projection}}.
\end{aligned}
\tag{50}
$$

The terms correspond to the input projection, depthwise causal convolution, generation of the selective state-space parameters, selective state update and readout, and output projection. In particular, the $7Ld_i d_s + 3Ld_i$ term accounts for the arithmetic of the selective recurrence, including state discretization and update, state-to-output projection, skip connection, and output gating. Exponential, softplus, and SiLU evaluations are tracked separately and are not assigned a hardware-dependent FLOP equivalent.

MMPD processes each node sequence in both directions and averages the two Mamba outputs. Including the surrounding LayerNorm operations, residual connections, and feed-forward refinement, one bidirectional Mamba block therefore costs

$$
\begin{aligned}
C_{\mathrm{BiM}}(L) = & \, 2C_{\mathrm{LN}}(L, d_M) + 2C_{\mathrm{Mamba}}(L) + 4Ld_M \\
& + C_{\mathrm{lin}}(L, d_M, \mu d_M; 1) + C_{\mathrm{lin}}(L, \mu d_M, d_M; 1) + Ld_M.
\end{aligned}
\tag{51}
$$

Here, the $4Ld_M$ term accounts for combining the forward and backward Mamba outputs and the subsequent scaled residual connection, while the final Ld_M term is the feed-forward residual addition.

D.6.4 One MMPD Layer

A complete MMPD layer first propagates information from check nodes to variable nodes and updates the variable-node states. It then propagates the updated variable-node information back to the check nodes and updates the check-node states. Both node streams are additionally refined by bidirectional Mamba blocks. The cost of one MMPD layer is therefore

$$
\begin{aligned}
C_{\text{MMPD-layer}} = & \underbrace{C_{\mathrm{pair}}(n, K_v) + C_{\mathrm{upd}}(n) + C_{\mathrm{BiM}}(n)}_{\text{variable-state update}} \\
& + \underbrace{C_{\mathrm{pair}}(m, K_c) + C_{\mathrm{upd}}(m) + C_{\mathrm{BiM}}(m)}_{\text{check-state update}}.
\end{aligned}
\tag{52}
$$

The graph-dependent cost of the pairwise component is thus determined by the padded neighborhood sizes nK_v and mK_c. This differs from NSF, whose sparse Tanner aggregation scales directly with the number of existing edges E.

D.6.5 Total Decode Cost

The input magnitude and syndrome embeddings require

$$
C_{\mathrm{MMPD,init}} = 2(n + m)d_M \tag{53}
$$

arithmetic operations.

After N_M MMPD layers, the final variable-node states are normalized and projected to one output logit per codeword position. The complete decode cost is therefore

$$
\begin{aligned}
C_{\mathrm{MMPD}} = & \underbrace{2(n+m)d_M}_{\text{input embeddings}} \\
& + \underbrace{N_M C_{\text{MMPD-layer}}}_{\text{stacked MMPD layers}} \\
& + \underbrace{C_{\mathrm{LN}}(n, d_M) + C_{\mathrm{lin}}(n, d_M, 1; 1)}_{\text{final readout}}.
\end{aligned} \tag{54}
$$

MMPD uses a fixed computational depth and therefore has a single decode FLOP count for a given code and architecture.

D.7 Trainable Parameter Counts

Parameter counts in Table 2 denote the exact number of trainable scalar parameters.

For NSF, it is useful to separate the recurrent solver parameters from the position-dependent initialization parameters. Let P_{core} denote all parameters shared across nodes and iterations. Under the configuration used in this work,

$$P_{\mathrm{core}} = 152{,}138. \tag{55}$$

The default NSF assigns separate initialization embeddings and biases to all n variable nodes and m check nodes. Hence,

$$P_{\mathrm{NSF}} = P_{\mathrm{core}} + 2d(n+m). \tag{56}$$

NSF-Shared instead uses one variable-node embedding/bias pair and one check-node embedding/bias pair, giving

$$P_{\text{NSF-S}} = P_{\mathrm{core}} + 4d. \tag{57}$$

With $d = 64$,

$$P_{\text{NSF-S}} = 152{,}394, \tag{58}$$

independently of block length.

For EfficientMPT, one decoder layer with embedding dimension d_e contains

$$P_{\mathrm{layer}} = 11d_e^2 + 14d_e \tag{59}$$

trainable parameters. Including the input embeddings, intermediate and final normalizations, and output projection, the total for $N > 1$ is

$$P_{\mathrm{EffMPT}} = N(11d_e^2 + 14d_e) + 7d_e + 1. \tag{60}$$

For $d_e = 128$, this gives

$$
\begin{aligned}
P_{\mathrm{EffMPT}}(N = 6) &= 1{,}092{,}993, \\
P_{\mathrm{EffMPT}}(N = 10) &= 1{,}821{,}057.
\end{aligned} \tag{61}
$$

These values correspond to the 1.093M and 1.821M entries reported in Table 2.

For MMPD, the trainable parameter count consists of the parameters of the compact pairwise aggregators, gated node updates, bidirectional Mamba blocks, position-specific node initialization, and the final output head.

For the compact pairwise aggregator, the parameter count is

$$P_{\mathrm{pair}} = 2d_M^2 + 2d_M r_M + 2r_M^2 + 2r_M + 1. \tag{62}$$

Let

$$d_g = \left\lfloor \frac{d_M}{g_n} \right\rfloor g_k \tag{63}$$

denote the hidden dimension of the gated node update. Its parameter count is

$$P_{\mathrm{upd}} = 4d_M^2 + 4d_M d_g + d_g + 11d_M. \tag{64}$$

For the Mamba component, let

$$d_i = \gamma_M d_M, \qquad r_\delta = \left\lceil \frac{d_M}{16} \right\rceil. \tag{65}$$

One-directional Mamba module contains

$$P_{\text{Mamba}} = d_i \left(3d_M + k_M + 2r_\delta + 3d_s + 3\right) \tag{66}$$

trainable parameters. A bidirectional Mamba block contains two such modules together with two LayerNorms, a learnable residual scale, and a feed-forward network, giving

$$P_{\text{BiM}} = 2P_{\text{Mamba}} + 2\mu d_M^2 + (\mu + 5)d_M + 1. \tag{67}$$

Since each MMPD layer contains two pairwise aggregators, two gated node updates, and bidirectional Mamba refinement for both variable and check nodes, its parameter count is

$$P_{\text{MMPD-layer}} = 2\left(P_{\text{pair}} + P_{\text{upd}} + P_{\text{BiM}}\right). \tag{68}$$

Under the configuration used in this work,

$$d_M = 64, \quad r_M = 32, \quad d_s = 28, \quad k_M = 4, \quad \gamma_M = 1, \quad \mu = 1, \quad g_n = 2, \quad g_k = 1,$$

which gives

$$P_{\text{MMPD-layer}} = 171{,}076. \tag{69}$$

MMPD uses separate initialization embeddings and biases for all variable and check nodes. These contribute

$$2d_M(n + m)$$

parameters. The two final LayerNorms and the variable-node output projection contribute an additional $5d_M + 1$ parameters. Hence, for N_M MMPD layers,

$$P_{\text{MMPD}} = N_M P_{\text{MMPD-layer}} + 2d_M(n + m) + 5d_M + 1. \tag{70}$$

With $N_M = 7$ and $d_M = 64$, this simplifies to

$$P_{\text{MMPD}} = 1{,}197{,}853 + 128(n + m). \tag{71}$$

Thus, the graph-dependent neighborhood widths affect MMPD's computational cost but not its trainable parameter count; the remaining code-length dependence arises only from the position-specific variable- and check-node initialization parameters.

D.8 Peak Inference Memory

FLOPs quantify arithmetic work but do not capture the memory required to store model parameters, graph structure, recurrent states, and temporary activations. We therefore measure inference memory separately.

All memory measurements use batch size one and FP32 arithmetic on the same CUDA device. Each decoder is warmed up before measurement to remove one-time initialization effects.

We decompose the architecture-dependent inference footprint into four parts:

$$M_{\text{param}}, \qquad M_{\text{buffer}}, \qquad M_{\text{input}}, \qquad M_{\text{temp}},$$

where M_{param} is the storage occupied by trainable parameters, M_{buffer} contains persistent decoder and code-structure tensors, M_{input} contains the input tensors, and M_{temp} is the peak transient allocation required by the forward pass.

Let M_0 denote the allocated CUDA memory immediately before the measured forward pass after warm-up, and let M_1 denote the maximum allocated memory observed during that pass. We define

$$M_{\text{temp}} = M_1 - M_0. \tag{72}$$

Table 7: Code-dependent quantities used in the computational-efficiency analysis. Here, m is the number of parity checks, $E = \|\mathbf{H}\|_0$ is the number of Tanner-graph edges, and the last two columns report the mean recurrent stopping times at 4 dB for NSF and NSF-Shared.

Code	(n, k)	m	E	E/n	$\overline{T}_{4\mathrm{dB}}^{\mathrm{NSF}}$	$\overline{T}_{4\mathrm{dB}}^{\mathrm{NSF\text{-}S}}$
LDPC	$(49, 24)$	28	196	4.00	1.67	–
MacKay	$(96, 48)$	48	288	3.00	2.03	–
LDPC	$(121, 60)$	66	726	6.00	2.70	–
LDPC	$(121, 70)$	55	605	5.00	2.04	–
LDPC	$(121, 80)$	44	484	4.00	1.75	–
MacKay	$(204, 102)$	102	612	3.00	2.46	2.45
LDPC	$(529, 440)$	92	2116	4.00	3.01	3.01
LDPC	$(576, 432)$	144	2112	3.67	2.81	2.88
LDPC	$(648, 540)$	108	2376	3.67	3.31	3.38
LDPC	$(1056, 880)$	176	3520	3.33	3.64	3.75
LDPC	$(1824, 1520)$	304	6080	3.33	3.96	4.07
LDPC	$(1920, 1600)$	320	6400	3.33	4.00	4.12
LDPC	$(2304, 1920)$	384	7680	3.33	4.13	4.22

The value reported in Table 2 is then

$$M_{\mathrm{inf}} = M_{\mathrm{param}} + M_{\mathrm{buffer}} + M_{\mathrm{input}} + M_{\mathrm{temp}}. \tag{73}$$

This definition retains all memory whose size is determined by the decoder while excluding persistent CUDA runtime state that is unrelated to the architecture being evaluated. Memory values are converted to MiB using $1\,\mathrm{MiB} = 2^{20}$ bytes.

Peak memory is measured for a complete recurrent step. Syndrome-based early termination does not reduce this quantity because all received words are active at the first decoding iteration, where the maximum working state is already instantiated.

Unlike the parameter count, the total inference footprint of NSF-Shared does not become independent of block length. Sharing initialization parameters removes the position-dependent learned tensors, while the variable states, check states, graph buffers, and recurrent activations continue to scale with n, m, and E.

D.9 Code-Dependent Quantities

Table 7 lists the code-dependent quantities used to evaluate equations 28–41. The final two columns give the measured mean NSF stopping times at 4 dB used in equation 38.

The Tanner graphs remain sparse throughout the long-code suite: E/n stays between approximately 3 and 4. Consequently, the edge-dependent part of NSF grows approximately linearly with block length. At the same time, the observed stopping time increases only gradually, from approximately 2.46 iterations for MacKay $(204, 102)$ to 4.13 iterations for LDPC $(2304, 1920)$ at 4 dB, remaining far below the maximum recurrent budget of $T_{\max} = 12$.

This distinction explains the two complementary NSF complexity numbers reported in Table 2: the full-budget cost characterizes the worst case independent of channel realization, whereas the effective cost captures the adaptive computation actually required at the operating SNR.

E Average Recurrent Iterations

Table 8 reports the mean number of recurrent iterations used by NSF and NSF-S across the short- and long-code benchmarks. For the short BCH and Polar codes, we report the operating points used in the short-code comparison, while the long-code suite is reported over the available operating points in the 2–5 dB range. NSF-S is reported for the long-code suite.

Table 8: Mean number of recurrent iterations $\overline{T}$ for NSF and NSF-S.

(a) Short BCH and Polar codes

Codes	(n, k)	**Method**	**4 dB**	**5 dB**	**6 dB**
BCH	$(31, 16)$	NSF	1.88	1.33	1.10
	$(63, 36)$	NSF	3.71	1.98	1.27
	$(63, 45)$	NSF	2.85	1.51	1.10
Polar	$(64, 32)$	NSF	5.19	3.36	2.20
	$(64, 48)$	NSF	1.95	1.26	1.05
	$(128, 64)$	NSF	4.06	2.27	1.46

(b) Long-code suite

Codes	(n, k)	**Method**	E_b/N_0 **(dB)**						
			2.00	2.50	3.00	3.50	4.00	4.50	5.00
MacKay	$(204, 102)$	NSF	7.08	5.28	3.94	3.05	2.46	2.02	1.68
		NSF-S	7.05	5.27	3.93	3.05	2.46	2.02	1.68
LDPC	$(529, 440)$	NSF	11.90	11.14	8.57	5.15	3.01	2.04	1.51
		NSF-S	11.90	11.17	8.57	5.14	3.01	2.04	1.52
	$(576, 432)$	NSF	11.10	8.62	5.63	3.79	2.81	2.20	1.76
		NSF-S	11.19	8.82	5.82	3.91	2.88	2.24	1.79
	$(648, 540)$	NSF	11.95	11.39	9.01	5.52	3.31	2.28	1.70
		NSF-S	11.95	11.42	9.10	5.61	3.38	2.32	1.73
	$(1056, 880)$	NSF	11.97	11.70	9.67	5.90	3.64	2.56	–
		NSF-S	11.97	11.70	9.86	6.11	3.75	2.63	–
	$(1824, 1520)$	NSF	12.00	11.95	10.47	6.34	3.96	2.83	–
		NSF-S	12.00	11.96	10.61	6.55	4.07	2.91	–
	$(1920, 1600)$	NSF	12.00	11.96	10.54	6.40	4.00	2.86	–
		NSF-S	12.00	11.97	10.68	6.60	4.12	2.94	–
	$(2304, 1920)$	NSF	12.00	11.98	10.80	6.54	4.13	2.95	–
		NSF-S	12.00	11.98	10.87	6.69	4.22	3.02	–