Algorithm-Architecture Co-Design for Domain-Specific Accelerators in Communication and Artificial Intelligence

by

Yaoyu Tao

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Electrical and Computer Engineering)
in The University of Michigan
2022

Yaoyu Tao

taoyaoyu@umich.edu

ORCID iD 0000-0001-7500-5250

To all that have loved and supported me in my life

ACKNOWLEDGEMENTS

First and foremost, I would like to thank my advisor, Professor Zhengya Zhang, for his tremendous support and guidance throughout my graduate work. It has been an honor to do research under his supervision from my undergraduate years and later started doing Ph.D with him. His tremendous effort, valuable ideas, and warm encouragement helped me make my Ph.D experience productive and stimulating. I am especially thankful for the insightful discussions and ideas from Professor Zhengya Zhang. I would also like to thank Professor Michael Flynn, Professor Scott Malhke and Professor Hun-Seok Kim for participating in my dissertation committee and evaluating my research proposal and reviewing the dissertation.

I have been very fortunate to work with many talented individuals at the University of Michigan and Stanford University. I am grateful for all the inspiring discussions and valuable help from my peer graduate students. In particular, I would like to thank Dr. Youn-Sung Park for the great cooperation on LDPC projects. I still remember the time we worked together and the constructive discussions we had in the lab. I would like to thank Dr. Sung-Gun Cho for the substantial support on my polar decoder projects and the cooperation in Arvon projects. I learnt a lot in FPGA design and VLSI physical design from him. I would like to thank all my lab mates, Chia-Hsiang Chen, Wei Tang, Jerry Zhu, Shuanghong Sun, Jungkuk Kim, Phil Knag, Alex Lee, Jacob Botimer, Reid Pinkham, Thomas Chen, Shiming Song, Chester Liu, Hao Nan, Supreet Jeloka, as well as all the students I got to know from other groups in the EECS departments for their friendship and support.

I would also like to thank my collaborators from industry, including but not limited to Dr. Joyce Kwong at Apple Hardware, Dr. Rashmi Nanda and Dr. Xiaoru Zhang at Qualcomm Wireless Research and Development, Dr. Manish Goel at Texas Instruments Kilby Lab, Dr. Farhana Sheikh at Intel Labs, Dr. Pascal Urard at ST Microelectronics, Dr. Engling Yeo at Marvell Semiconductor and Tim Hoang at Intel PSG, for their valuable feedback and help in chip design and testing.

Last but not least, I would like to thank my family for their love and support along the way. I couldn't have made it through this challenging but fruitful journey without their never ending encouragement.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

LDPC Low Density Parity Check

DNC Differentiable Neural Computer

NODE Neural Ordinary Differential Equation

5G Fifth Generation

SC Successive-Cancellation

SCL Successive-Cancellation List

NB-LDPC Nonbinary Low Density Parity Check

GF Galois Field

ML Machine Learning

NoC Network on Chip

DNC-D Distributed Differentiable Neural Computer

F-DNC Flip Differentiable Neural Computer

FV-DNC Flip Validate Differentiable Neural Computer

DNC-SCLF DNC-Aided SCL Flip Decoding

AI Artificial Intelligence

CPU Central Processing Unit

GPU Graphics Processing Unit

SNR Signal to Noise Ratio

FER Frame Error Rate

BER Bit Error Rate

ECC	Error Correction Code
DL	Deep Learning
NN	Neural Network
CV	Computer Vision
NLP	Natural Language Processing
FNN	Feed Forward Neural Network
RNN	Recurrent Neural Network
ResNet	Residual Neural Network
NTM	Neural Turing Machine
MANN	Memory Augmented Neural Network
eMBB	Enhanced Mobile Broadband
6G	Sixth Generation
3GPP	The 3rd Generation Partnership Project
LTE	Long-Term Evolution
WiFi	Wireless Fidelity
DVB	Digital Satellite Broadcast
BP	Belief Propagation
CRC	Cyclic Redundancy Check
CN	Check Node
VN	Variable Node
TS	Trapping Set
ETS	Elementary Trapping Set
BCH	Bose-Chaudhuri-Hocquenghem
QC	Quasi-Cyclic
LLRV	Log-Likelihood Ratio Vector
LSTM	Long-Short-Term Memory
GRU	Gated Recurrent Units

QA	Question Answering
DMN	Dynamic Memory Network
ODE	Ordinary Differential Equations
RK	Runge-Kutta
FD-SOI	Fully Depleted Silicon on Insulator
LUT	Look-Up Table
MAC	Multiply-And-Accumulate

ABSTRACT

The past decade has witnessed an explosive growth of data and the needs for high-speed data communications and processing. The needs continue to drive the development of new hardware for transmitting more data reliably and processing more data to obtain a higher level of intelligence. This thesis work explores algorithm-architecture co-design approaches to derive efficient solutions for domain-specific communication and machine learning accelerators. It focuses on advanced and most compute-intensive accelerator designs for: 1) channel coding for data transmission, including polar codes and low-density parity-check (LDPC) codes, and 2) neural networks for machine learning, including differentiable neural computer (DNC) and neural ordinary differential equations (NODE). It also covers an interdisciplinary area of AI-aided communication, exploring DNC-aided flip decoding for polar codes.

This work introduces a split-tree successive-cancellation list (SCL) decoder that works by dividing a polar code's decoding tree to sub-trees following a split-tree algorithm. Through algorithm-architecture co-optimizations, a 0.64mm^2 40nm test chip implements a split-4, list-2, 8-frame-interleaved decoder that supports configurable code lengths up to 1024 bit and variable code rates. At 0.9V and room temperature, the chip achieves 3.25Gb/s with 42.8mW power, or 13.2pJ/b, and demonstrates competitive error-correction performance.

This work advances LDPC codes in two aspects: 1) improve the error-correcting performance at the low error rate regime by a post-processing algorithm, and 2) demonstrate the feasibility of a Gb/s nonbinary LDPC (NB-LDPC) decoder. In designing the post-processor, we take the inspiration from simulated annealing to generalize the post-processor design using three methods: quenching, extended heating, and focused heating, each of which targets a different decoding error structure. The resulting post-processor is demonstrated to lower the error rate by two orders of magnitudes. NB-LDPC is another approach to improve error-correction performance. In this work, we present a fully parallel decoder for a (160, 80) regular-(2, 4) NB-LDPC code over Galois field GF(64) in 65nm CMOS. The decoder employs fine-grained dynamic clock gating and decoder early termination to achieve a throughput of 1.22Gb/s and an energy efficiency of 3.03nJ/b, or 259pJ/b/iteration, at 1.0V and 700MHz.

This work contributes to the hardware acceleration of DNC. We present HiMA, a tiled, history-based memory access engine with distributed memories in tiles. HiMA incorporates a traffic-aware multi-mode network-on-chip (NoC), an optimal submatrix-based memory partition and a two-stage usage sort method leveraging distributed tiles. We create a distributed DNC (DNC-D) to allow almost all memory operations to be applied to local memories. In a 40nm design, HiMA running DNC and DNC-D demonstrates 6.47× and 39.1× higher speed, 22.8× and 164.3× better area efficiency, and 6.1× and 61.2× better power efficiency over the state-of-the-art accelerators.

This work contributes to the hardware acceleration of NODE for improved modeling capability of continuous-time events. We carry out algorithm-architecture co-design: 1) we propose adaptive neural activation sparsity for up to 80% complexity reduction while maintaining excellent training accuracy. 2) we develop a multi-mode PE design for NODE compute kernels with configurable interconnects between PEs to handle a variety of numerical ODE solvers. The hardware efficiency can be further enhanced by exploring hardware reuse and hierarchical memory.

Lastly, this work investigates an interdisciplinary area of AI-aided communication, applying DNC for flip decoding of polar codes. New state and action encoding are developed for better DNC training and inference efficiency. The proposed method consists of two phases: i) a flip DNC (F-DNC) is exploited to rank the most likely flip positions for multi-bit flipping; ii) if decoding still fails, a flip-validate DNC (FV-DNC) is used to re-select error bit positions for successive flip decoding trials. Simulation results show that proposed DNC-aided SCL-Flip (DNC-SCLF) decoding demonstrates up to 0.34dB coding gain improvement or 54.2% reduction in average number of decoding attempts compared to prior works.

The five pieces of work presented in this thesis tackle hardware acceleration challenges in domain-specific computing using algorithm-architecture co-design techniques. The results of this work contribute to the developments of hardware accelerators in channel coding and deep learning, enabling next-generation communication and artificial intelligence from theory to efficient hardware.

CHAPTER I

Introduction

We live in an exciting time. Information is booming by orders of magnitudes for the past decades. The exploding demands to efficiently transmit massive amount of information and learn useful knowledge from them have stimulated domain-specific breakthroughs like fifth-generation (5G) wireless communication or artificial intelligence (AI). The powers granted by these advancements have fundamentally changed the way we live and are also drastically shifting the dynamics of industries and business around the world.

However, the path to achieving high-efficiency domain-specific hardware systems is challenging. On one hand, new paradigms in communication or AI algorithms are computationally expensive, resulting in a low efficiency when running on commercial CPUs and GPUs. On the other hand, the hardware performance requirements (e.g. power, performance and cost) are key factors that are hindering mass commercial deployment. Despite the obstacles ahead, these algorithmic advancements have revealed rich opportunities for algorithm-architecture co-design, driving new innovations within the research field of domain-specific accelerators.

Before diving into domain-specific algorithm and architecture developments, we will take a few steps back to revisit the fundamentals in two domains: communication and AI. The goal of efficient communication is to transmit the most information using

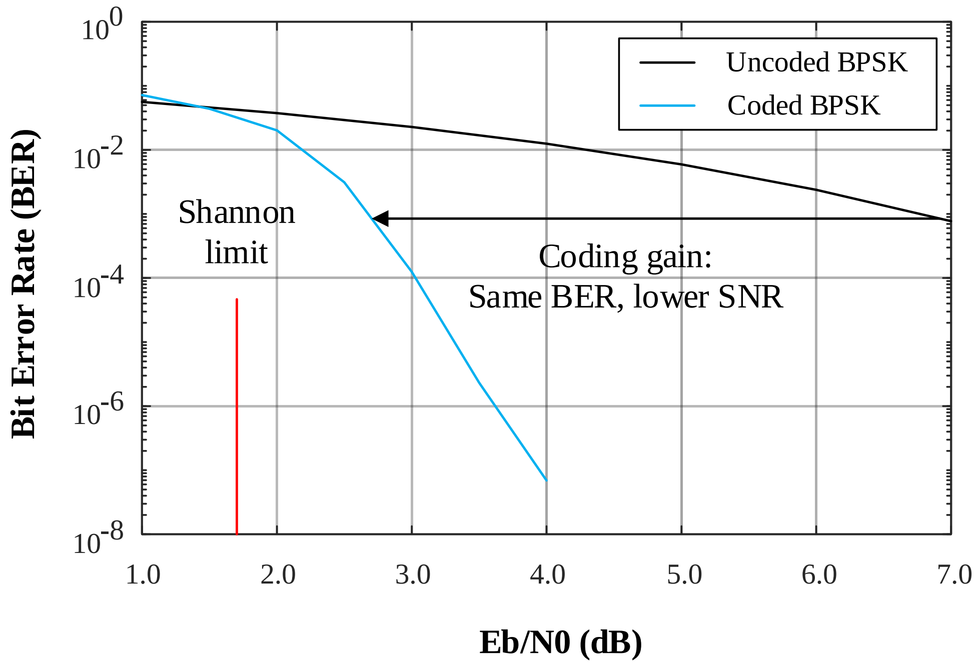


Figure 1.1: Bit error rate (BER) comparison between uncoded and encoded systems.

the least energy. The ultimate theoretical limit of efficient communication is defined by the Shannon capacity, which captures the least transmit energy, or signal-to-noise ratio (SNR), needed for reliable transmission. For a given information reliability measured in terms of frame error rate (FER) or bit error rate (BER), a system with weak or no error correction codes (ECC) will require a high SNR, whereas a system with a strong ECC will be able to reduce the necessary SNR and the transmit energy. Figure 1.1 illustrates the difference between a coded system versus an uncoded system. State-of-the-art ECCs include low-density parity-check (LDPC) [1], nonbinary LDPC (NB-LDPC) [2], and polar codes [3].

On the other hand, AI techniques, particularly deep learning (DL) algorithms based on neural networks (NN) have shown superior performance in tackling the learning problems in the emerging fields of computer vision (CV) [4] and natural language processing (NLP) [5]. Deep learning NNs typically consist of multiple layers of artificial neuron cells. The cells can be categorized as input cells, output cells, hidden cells, recurrent cells or memory cells according to the their functionalities and the

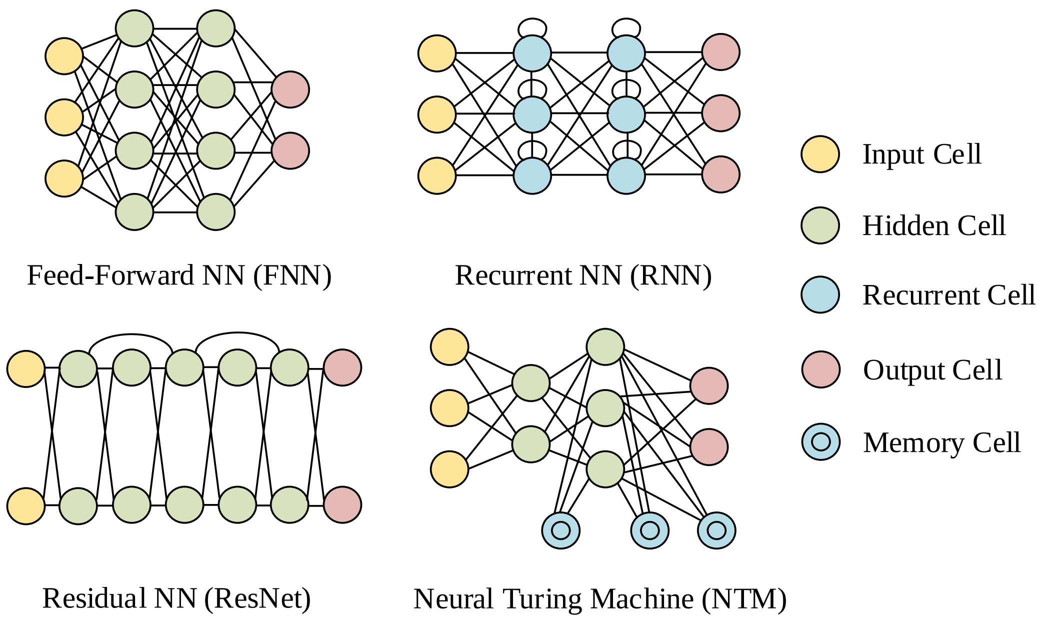


Figure 1.2: Conventional neural network topologies.

NN topology. Figure 1.2 demonstrates some conventional NN topologies, including feed-forward NN (FNN), recurrent NN (RNN) [6, 7], residual NN (ResNet) [8] and neural Turing machine (NTM) [9]. However, they are still lacking the capability in handling more complex problems like learning long-term dependencies or modeling continuous-time events. Recent developments in NN introduce new paradigms such as differentiable neural computer (DNC) [10] (latest variant of memory-augmented neural networks, or MANNs) and neural ordinary differential equations (NODE) [11] to help NN tackling a variety of complex applications.

1.1 Domain I: Channel Coding

Polar code and LDPC code are the two representative ECCs for channel coding in state-of-the-art communication systems and they are still evolving rapidly. Polar code was invented by Erdal Arikan [3] in 2009 with capacity achieving capability. Soon after, they have been adopted in fifth-generation (5G) enhanced mobile broadband (eMBB) control channels and are one of the most promising candidates for sixth-generation (6G) wireless standard in the near future. LDPC code is a class of near-capacity channel codes and was rediscovered from mid 2000s. They have been widely adopted in commercial applications, such as 3GPP-LTE, WiFi-6, and digital satellite broadcast (DVB), etc. This section reviews LDPC, nonbinary LDPC and polar codes, their state-of-the-art algorithms and decoder designs, and major design challenges.

1.1.1 Polar Codes

Polar codes exhibit a polarization effect [3] when transmitted over certain types of channels and decoded using the SC algorithm. The polarization effect refers to that certain bits become highly reliable and the other bits become highly unreliable. To use polar codes, information is conveyed over the reliable bits and the unreliable ones are frozen to predetermined values.

An $(N,\ K)$ polar code has a code length N and code rate K/N, where $N = 2^n$, $n \in \mathbb{Z}^+$. Let $u_0^{N-1} = (u_0, u_1, ..., u_{N-1})$ denotes the vector of input bits to the encoder. The $N - K$ least reliable bits in u_0^{N-1}, called frozen bits, are typically set to 0, while the remaining K bits, called free bits, are used to carry information a_0^K. An encoder encodes u_0^{N-1} to the codeword $x_0^{N-1} = (x_0, x_1, ..., x_{N-1})$. The systematic encoder is mathematically described by (1.1).

$$x_0^{N-1} = u_0^{N-1} G_N = u_0^{N-1} B_N F^{\otimes n} \text{ for } n \geq 1, \tag{1.1}$$

where $G_N = B_N F^{\otimes n}$ is the $N \times N$ generator matrix, B_N is called the bit-reversal permutation matrix, and $\otimes n$ denotes the n-th Kronecker power:

$$F^{\otimes n} = F \otimes F^{\otimes (n-1)},\ F = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} \text{ and } F^{\otimes 0} = 1 \tag{1.2}$$

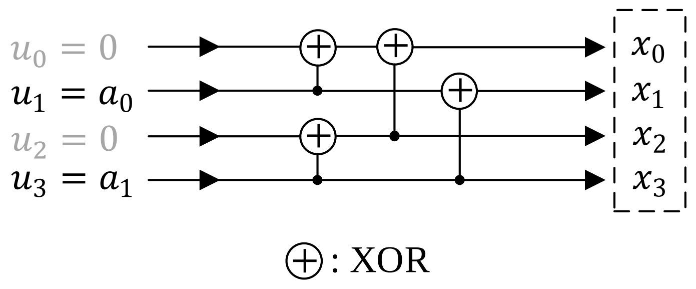


Figure 1.3: Encoding graph and SC decoding trellis of a (4,2) polar code.

A polar codeword is produced by first setting the frozen bit locations to 0. For example, a vector of 4 input bits ($N = 4$, $n = 2$) is set to $u_0^3 = \begin{bmatrix} 0 & a_0 & 0 & a_1 \end{bmatrix}$, where the bit 0 and bit 2 are the frozen bits and are set to 0, and bit 1 and 3 are used to carry information bits a_0 and a_1. For simplicity of understanding, assume B_4 is identity, then the encoder performs the following mathematical operation:

$$\begin{aligned} x_0^3 = u_0^3 F^{\otimes 2} &= \begin{bmatrix} 0 & a_0 & 0 & a_1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix} \\ &= \begin{bmatrix} a_0 \oplus a_1 & a_0 \oplus a_1 & a_1 & a_1 \end{bmatrix}, \end{aligned} \tag{1.3}$$

where modulo-2 operations are used and $\oplus$ represents modulo-2 addition, or XOR. The encoder produces a (4,2) code with a block length of 4 and 2 information bits. The mathematical operation can be illustrated in the encoding graph shown in Figure 1.3. Notice the regular placement of XORs between every pair of bits in the first stage, and every 2-bit blocks in the second stage. Encoding is done by propagating the bit vector through the graph from left to right.

Although Arikan proved that polar code achieves capacity as the block length N approaches infinity, the error-correction performance of polar codes of finite block

Table 1.1: Summary of state-of-the-art polar decoder implementations

	arXiv'18 [18]	TCAS-II'19 [19]	ASSCC'18 [20]	TCAS-I'19 [21]
Design	silicon	layout	silicon	silicon
Code	up to 2^{15}b variable rate	1024b rate-1/2	1024b rate-1/2	1024b rate-1/2
Decoding Algorithm	SCL	SC	SC	BP
Process Technology	16nm FinFet	180nm CMOS	180nm CMOS	40nm CMOS
Decoder Area (mm^2)	2.27	1.95	3.17	0.704
Supply (V)	0.9	1.8	1.8	0.9
Frequency (MHz)	1000	447	382	500
Throughput (Gb/s)	3.24	0.30	0.66	7.61
Power (mW)	-	1073	-	422.7
Area Eff. (Gb/s/mm^2)	1.43	0.154	0.208	10.81
Energy Eff. (pJ/b)	-	7994	-	55.58

lengths are still away from the Shannon limit. Polar codes can be decoded by belief propagation (BP) [12] or successive cancellation list (SCL) [3, 13] decoding algorithms. There are handful of pre-silicon SCL decoder designs published in literature recently as shown in Table 1.1. For simplicity and a fair comparison, we will cite the performance using a common configuration of 1024-bit code length, 1/2 code rate, a list size of $L = 2$ and 65nm CMOS for area. The direct mapping of an LLR-based CRC-aided SCL decoder design [14] was estimated to achieve a 334 Mb/s decoding throughput at a 847 MHz clock frequency. A 4-bit grouping approach was proposed in [15] to decode neighboring bits together for a higher throughput, and the decoder design was estimated to achieve 395 Mb/s at 500 MHz. Other speedup techniques like partial look-ahead decoder [16] and multi-mode decoder designs [17] were shown in simulation to achieve 537Mb/s and 1.21 Gb/s, respectively.

Area saving techniques have been developed in [22, 23]. [22] introduced an area saving strategy based on interpolation-based code construction and layered decoding scheme, showing up to 50.7% area saving over a conventional SCL decoder. [23] presented another area saving approach using a partial-sum network that efficiently computes the list candidates, resulting in up to 70% area reduction based on synthesis.

Performance-enhanced SCL decoders proposed recently include symbol-decision SCL decoder [24] and sphere SCL decoder designs [25] that were shown in simulation to achieve 398 Mb/s at 500 MHz and 1.23 Gb/s at 1 GHz, respectively. These designs however incur a high area overhead.

Complexity reduction techniques have also been developed, including simplified SCL (SSCL) and fast-SSCL-SPC [26], where the SCL tree is pruned and a single parity check (SPC) is introduced to maintain the error-correction performance. A fast-SSCL-SPC decoder design was estimated to achieve 1.86 Gb/s at 885 MHz in a 1.048 mm^2 area . A follow-up rate-flexible fast-SSCL decoder design [27] improved the results to an estimated 1.22 Gb/s at 955MHz in a 1.45 mm^2 area. An early stopping criterion was introduced [28] to improve the fast-SSCL decoder further to a projected throughput of 2.05 Gb/s at 650 MHz. However, these complexity reduction techniques tend to hurt the error-correction performance.

When a high-speed decoder is implemented in silicon, we can expect that the measured performance is worse, the area is larger, and the power is higher than the estimates, because it is difficult to fully account for all the overhead of wiring, clocking, and memory by simulation and synthesis. It is therefore more reliable to check silicon measurements to make comparisons. For a simple and fair comparison of silicon decoders, we will report the measured results without applying any process or voltage normalization because popular scaling formulas are no longer reflective of the reality of scaling in advanced processes. The latest SC decoder [20] was designed in 180 nm CMOS, occupying 3.17 mm^2 silicon area. The decoder delivered 655 Mb/s at 382 MHz for 1024-bit codewords. The first SCL decoder [29] was designed in a 28 nm FD-SOI technology, occupying 0.44 mm^2. The decoder achieved 614 Mb/s

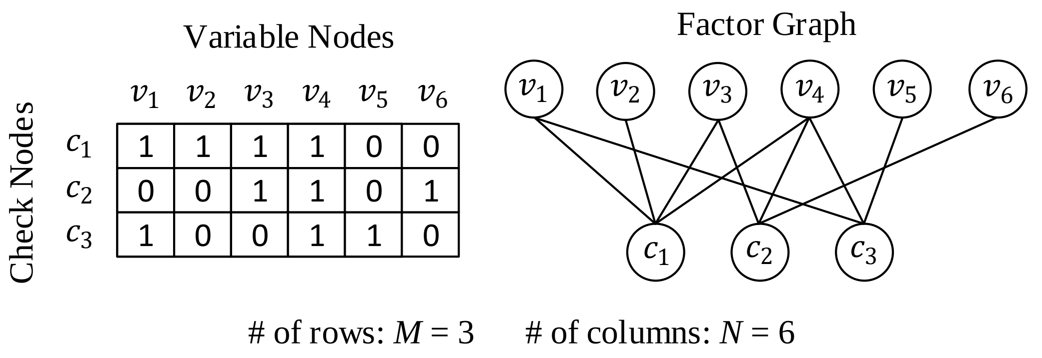


Figure 1.4: An example H matrix and factor graph representation of an LDPC code.

in throughput, 3.34 μs in latency, and 209 pJ/b in energy at 1.3 V for a 1024-bit code length and $L = 4$. The latest SCL decoder [18] was designed in 16 nm FinFET, occupying 2.27 mm^2. The chip demonstrated a 3.24 Gb/s throughput for 1024-bit codes, but no power measurements were reported. Clearly, it is still challenging to meet a multi-Gb/s throughput, sub-μs latency, sub-10 pJ/b energy and compact silicon area all at the same time.

1.1.2 Low-Density Parity-Check Codes

An LDPC code is a block code defined by an $M \times N$ parity-check matrix H, where M is the block length (number of bits in the codeword) and N is the number of parity checks. The element of the matrix $H(i, j)$ are either 0 or 1 to represent whether bit j of the codeword is part of parity check i. An H matrix can be represented using a factor graph composed of two sets of nodes: a variable node (VN) for each column of the H matrix and a check node for each row. An edge is drawn between VN(j) and CN(i) if H(i, j) = 1. An example H matrix with its corresponding factor graph is shown in Figure 1.4.

A LDPC codeword is produced by the mod 2 multiplication of the source bit sequence and a generator matrix G, which can be obtained by putting the parity-check matrix H into this form of $[I_k—P]$. For the example shown in Figure 1.4, the generator matrix and the codeword for the source bit sequence '101' can be obtained as (1.4), where $\odot$ is the symbol for mod 2 multiplication.

$$[1\ 0\ 1] \odot G = [1\ 0\ 1] \odot \begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 & 0 \end{bmatrix} = [1\ 0\ 1\ 0\ 1\ 1] \tag{1.4}$$

LDPC codes can be efficiently decoded by the iterative belief propagation (BP) decoding algorithm or its simplifications such as min-sum (MS) algorithm [30]. The MS algorithm enables simple processing nodes that are easily implemented in hardware. Therefore the decoder complexity can be kept low while achieving good error-correction performance.

The coding gain of an LDPC code is captured by a waterfall curve featuring a steep reduction in BER (and FER) with increasing SNR. Popular LDPC codes of block length up to 2 Kb or 4 Kb for wireless [31, 32, 33] and wireline applications [34] have demonstrated excellent waterfall performance down to a BER level of 10^{-7} to 10^{-10}, below which the curve flattens in a phenomenon called error floor [35]. The presence of an error floor degrades the achievable BER performance. With future communication and storage systems demanding data rates at multiple Gb/s or higher, error floors will worsen the quality of service. To prevent BER degradation due to error floors, SNR needs to be raised excessively, moving away from the capacity that defines the optimal performance.

Over the last decade, solving the error floor problem has been one research focus in coding theory and decoder design communities. Past experiments have shown that error floors can be caused by practical decoder implementation [36]. Improved algorithm implementation and better numerical quantization can suppress these effects [36]. However, error floors are fundamentally attributed to noncodeword trapping sets (TS), especially elementary trapping sets (ETS), associated with LDPC codes

[37, 35, 38]. A TS refers to a set of bits in a codeword, when received incorrectly, cause the belief propagation (BP) decoding algorithm to be trapped in a local minimum [35].

Much work has been done on lowering the error floor by improving code construction using methods such as selective cycle avoidance [39], improved progressive edge growth [40], code doping [41], and cyclic lifting [42]. These methods are effective, reporting up to 2 orders of magnitudes lower error floor, but they may produce unstructured codes that are not amenable to efficient decoder implementation. The irregular parity check matrices of these techniques complicate the encoder/decoder design, introducing significant overheads in latency, throughput and hardware area. Since theoretical approaches require complete redesign of codes, they are not applicable to the current deployed LDPC systems.

Code concatenation is another approach to lower error floors. With appropriately designed outer codes, such as Bose-Chaudhuri-Hocquenghem (BCH) codes [43, 44] or Reed-Solomon codes [45], error floors of the concatenated codes can be lowered by up to 2 orders of magnitude. However, the addition of an outer code increases the system complexity, power, cost, and decoding latency.

Alternatively, improvements can be made to decoding using methods such as scaling [46], averaging [38], and reordering steps [47] in BP decoding, but the effectiveness of these methods is often limited, usually 1 to 2 orders of magnitude, and some require extra steps that are incompatible with BP decoding, leading to a higher decoder complexity and longer latency. A backtracking approach was proposed in [48] to use a trial and error strategy to flip bits that are likely to be incorrect, and rerun decoding to check if the trial is successful. The approach does not rely on any prior knowledge of trapping sets, but its implementation can be costly in terms of memory and latency. Schedule diversity [49] was proposed to make multiple decoding attempts using different decoding schedules to reduce the probability of falling into a trapping

Table 1.2: Comparison of Techniques for Lowering LDPC Error Floors

Techniques	Cycle avoidance [39], PEG [40], cyclic lift [42], code doping [41]	LDPC+ BCH [43]	LDPC+ RS [45]	Scaled BP [46]	Reordered BP [47]	Bi-mode [50]
LDPC code	irregular	(504,252) regular	(1944,972) regular	(2640,1320) regular	(2016,1512) regular	(1984,1240) regular
Outer code	—	(756,696,6) BCH code	(155,75) RS code	—	—	—
Error floor lower by	$\sim$10$\times$	$\sim$100$\times$	—	10$\sim$100$\times$	$\sim$10$\times$	100$\sim$1000$\times$
Latency	high (due to code irregularity)	high (due to outer code)	high (due to outer code)	negligible	moderate	moderate
Code rate loss	no loss	0.75$\rightarrow$0.69	0.62$\rightarrow$0.30	no loss	no loss	negligible
Hardware cost	high (due to code irregularity)	high (extra BCH block)	high (extra RS block)	negligible	negligible	low
Throughput loss	high (due to code irregularity)	negligible	negligible	negligible	negligible	negligible
Back compatibility	no	no	no	yes	yes	yes

set. The approach is another form of trial and error, and can be costly in latency.

In theory, a more effective approach is to add a post-processing step if a decoding error is detected, and the post-processing is done in a targeted manner without having to rely on trial and error. An example of post-processing is the bi-mode syndrome-erasure decoding algorithm [50, 51]. One drawback of the post-processing approach is that it is usually limited to specific codes and it is not known whether it is generally applicable. A redecoding approach based on attenuating a predetermined set of bits [52] was proposed for quasi-cyclic (QC) LDPC codes. The approach involves an offline search and it may only be applied to QC-LDPC codes. Table 1.2 summarizes the qualitative features of techniques for lowering LDPC error floors.

1.1.3 Nonbinary Low-Density Parity-Check Codes

Binary LDPC code can be extended to nonbinary LDPC (NB-LDPC) codes [53] defined over Galois field GF(q). NB-LDPC codes offer better coding gain than binary LDPC codes. The main difference between an LDPC and an NB-LDPC is that an NB-LDPC code is formed by grouping multiple bits to symbols using GF elements, as illustrated in an example in Figure 1.5. In the example, two bits are grouped together

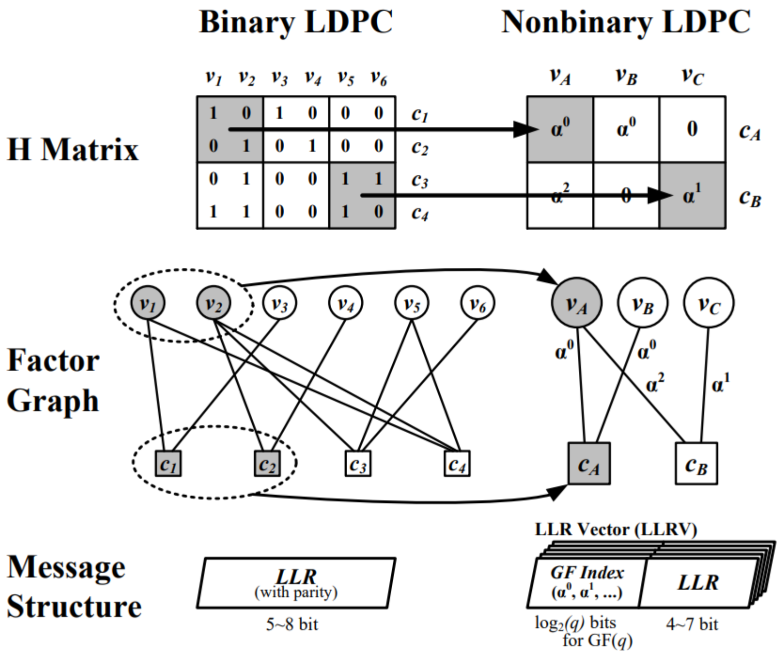


Figure 1.5: Comparison of a binary LDPC code and an NB-LDPC code.

to a 2-bit symbol using GF(2^2) or GF(4). From the 4×6 binary LDPC H matrix on the left-hand side, 2×2 submatrices are replaced with single GF(4) elements, resulting in the 2×3 GF(4) nonbinary H matrix on the right-hand side. An NB-LDPC code can also be illustrated using a factor graph composed of VNs and CNs. An edge connects VN v_j and CN c_i if the corresponding entry in the H matrix $H(i,j) \neq 0$ in GF(q). The message exchanged between nonbinary CNs and VNs are extended to an log-likelihood-ratio vector (LLRV) of length q.

The decoding of NB-LDPC codes follows the same BP algorithm[53] that is used in the decoding of binary LDPC codes. However, the complexity of an NB-LDPC decoder is notably higher: each message exchanged between processing nodes in an NB-LDPC decoder carries an LLRV; parity check processing follows a forward-backward algorithm; and high-order GF operations require expensive matching and sorting, in contrast to the much simpler addition and compare-select used in binary LDPC decoding. The complex NB-LDPC decoding has prevented any large-scale high-throughput chip implementations in silicon. Only FPGA designs, synthesis and layout have been demonstrated prior to this work [54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65].

The complexity of the NB-LDPC decoder and its error-correcting performance are determined by code construction. Quasi-cyclic LDPC codes have been invented to provide a good error-correcting performance [66, 67, 68], and their regular structures are amenable to efficient decoder architectures. Compared to the quasi-cyclic LDPC codes, the $(2, d_c)$ codes [69] feature a very low variable node degree $d_v = 2$, and a check node degree as low as $d_c = 4$, reducing the processing complexity, the wiring, and the quantization loss. Therefore, the $(2, d_c)$ codes are attractive for practical implementations. A $(2, d_c)$ NB-LDPC code offers a competitive error-correcting performance even at a short block length. The performance can be further improved by increasing q, the order of the GF field, but a higher q increases the size and complexity of the decoder.

The direct implementation of the BP decoding for NB-LDPC codes results in a check node complexity of $O(q^2)$ and a variable node complexity of $O(q)$. A fast Fourier transform (FFT) implementation reduces the check node complexity to $O(q \log q)$, but it requires check node processing in the linear domain and the conversion between linear- and log-domain messages. The extended min-sum (EMS) algorithm [70] in the log domain reduces the check node complexity to using only a small subset of n_m values among an array of LLRs in a message, where $n_m \ll q$. A further simplification of the EMS algorithm truncates the least significant values in a message and keeps only the most significant n_m values in memory [71]. The processing is done entirely using truncated messages, thereby reducing the complexity of the check node to $O(n_m \log n_m)$ and the complexity of the variable node to $O(n_m)$. The truncated EMS algorithm has demonstrated minimal loss in error-correcting performance at low SNR compared with BP, while the performance surpasses BP at high SNR. The truncated EMS algorithm makes it possible to design an NB-LDPC decoder with a

Table 1.3: Comparison of NB-LDPC Decoder Implementations Prior to This Work

	TVLSI'14 [64]	TVLSI'13 [65]	TSP'13 [56]	TCAS-I'12 [59]
Technology	90nm	28nm	90nm	90nm
Design	layout	layout	layout	layout
Code Length (symbols)	837	110	837	248
Code Rate	0.86	0.8	0.87	0.55
Galois Field	GF(32)	GF(256)	GF(32)	GF(32)
Decoding Algorithm	SES-GBFDA	RTBCP	Trellis-based Max-Log-QSPA	Selective-input Min-Max
Core Area (mm^2)	6.6	1.289	46.18	10.33
Utilization (%)	-	75.7	-	-
Gate Count	0.468M (XOR)	2.57M (NAND)	8.51M (NAND)	1.92M (NAND)
Core Supply (V)	-	-	-	-
Clock Frequency (MHz)	277	520	250	260
Iterations	10	10	5	10
Throughput (Mb/s)	716	546	234	47.7
Power (mW)	-	976	893	480
Energy Efficiency (nJ/b)	-	1.78	3.82	10.06
Energy Efficiency (pJ/b/iter)	-	178	765	1006
Area Efficiency (Mb/s/mm^2)	108	424	5.06	4.62

reasonable complexity that is within the range of binary LDPC decoders. A further simplification using the min-max algorithm [72] suffers from a noticeable degradation in the error-correcting performance. Table 1.3 summarizes the NB-LDPC decoder implementations prior to this work.

1.2 Domain II: Neural Networks

Neural networks are emerging to be the most advanced techniques in learning useful knowledge from massive amount of data. They are inspired by mimicking the method in which animals learn and perceive the world. A neural network is essentially an explicit expression of neurons connections, representing features that could be learnt in different dimensions. The term deep neural networks (DNNs) refers to multiple layers of neurons that are connected together, where the data can propagate from one layer to the next.

Typically, there are two major functions that an NN needs to perform: 1) training; and 2) inference. Training is the process where a model neural network fits itself using a given set of input data and mathematically optimizes for a target metric (e.g. minimization of a loss function) that describes how well the NN model is performing with respect to its current trained status. Some training algorithms require labels (i.e. ground truths), referred to as supervised learning, while some do not require labels, referred to as unsupervised learning. On the other hand, the inference process takes a trained NN and passes the input data through the network to obtain the output result.

Conventional NN topologies include deep FNNs, RNNs or ResNets, as shown in Figure1.2. They have shown superior performance in feature extraction or sequence learning problems in CV, NLP, etc. However, they are still lacking the capability in tackling more complex problems like learning long-term dependencies or modeling continuous-time events. In this thesis, we focus on two new variants of NNs for these complex learning problems, DNC [73], the latest version of memory-augmented neural networks (MANNs), and neural ordinary differential equations (NODEs) [11].

1.2.1 Differentiable Neural Computer

The application of neural networks (NNs) have grown extensively to many practical problems such as natural language processing (NLP) [74], speech recognition [75] and computer vision (CV) [76]. In the case of NLP, improvements come from the sequence modeling capabilities of recurrent NNs (RNNs) [77] such as long short-term memory (LSTM) [78] or gated recurrent units (GRU) [79]. However, performance of RNNs are limited by how long memories can persist, because the dynamic states are intrinsically embodied within the network. RNNs become less effective in tasks like question answering (QA) [80] where relevant information for the correct answers could be far away from where the questions are asked. This motivated the development of memory-augmented NNs (MANNs), a fully differentiable model that contains an isolated external memory module that NNs can learn to store to and read from when computing predicted outputs.

Such MANNs include memory network (MemNet and MemN2N) [80, 81], dynamic memory network (DMN) [82], neural Turing machine (NTM) [83] and differentiable neural computer (DNC) [73]. Compared to traditional RNN/LSTM or recently developed Transformer [84], MANNs outperform in tackling long-term dependency problems and found many applications not only in NLP, but also in graph modeling [85, 86], navigation [73, 87] and reinforcement learning [88, 89, 90]. Specifically, DMN uses a GRU as the memory component. MemNet/MemN2N uses an external addressable memory; however, neither memory content nor access history is considered in the addressing. NTM enhances the performance by using content-based soft write and read. An NTM with an LSTM can infer simple algorithms such as copying or sorting. Subsequently, DNC extends NTM by incorporating history-based mechanisms that consider historical events when accessing external memory. This allows DNC to achieve better performance than NTM in handling long-term dependencies [73]. However, the enhanced performance of DNC comes at a much higher computational cost, more complex memory operations, and more specifically history-based attention mechanisms.

NN accelerators [91, 92, 93, 94, 95, 96] cannot run DNC due to the lack of capability to handle elaborate memory operations. Compared to NN accelerators that store weights, perform convolutions and accumulate partial sums, DNC accelerators need to support more complex and diverse workloads, including new primitives like sorting or matrix transpose and a variety of new state memories to store access history that do not exist in NNs. MANN accelerators (for MemNets or NTM) [97, 98, 99, 100] also cannot run DNC due to the lack of support of DNC's unique sorting primitive and new state memories. The only way to run DNC using an existing accelerator is to have it attached to a general-purpose CPU or GPU, which is unlikely to deliver a high efficiency.

NTM accelerators only support content-based attention mechanisms without state memories. Specifically, X-MANN [99] implements external memory using resistive crossbars. The performance gain relies on emerging devices that are not widely available. The recently developed Manna [100] proposed a network-on-chip (NoC) architecture for NTM. Manna's distributed architecture provides more memory bandwidth and compute parallelism, but its H-tree NoC still incurs a traffic bottleneck when running DNC's history-based attention mechanisms. DNC accelerators have been developed recently [101, 102]. In [102], the efficiency enhancements mainly come from analog-based processing elements such as analog-to-digital converters (ADCs), which are more sensitive to variations and noise, and less portable between process technologies. These designs also followed a centralized architecture for memory access and compute, which could lead to poor scalability when memory size increases. Operations may need to be serialized, degrading both speed and latency.

We provide a brief overview of relevant concepts, mathematical descriptions of DNC and review MANN accelerators.

1.2.1.1 Memory-Augmented Neural Networks

Recurrent neural networks (RNNs) [103] such as LSTMs [104] extend feedforward DNNs by introducing recurrent connections, thereby allowing the network to store dynamic states across iterations of inputs. Suppose x_t be the input and y_t be the output at time stamp t, LSTM is composed of a chain of LSTM cells, denoted by A, as shown in Figure 1.6. The introduction of dynamic state has benefited domains which requires remembering of event sequences, such as QA in NLP. However, the

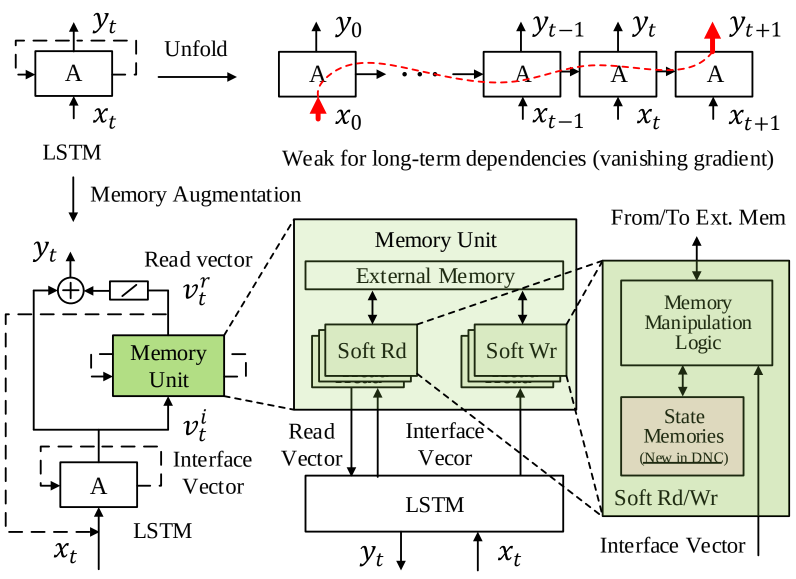
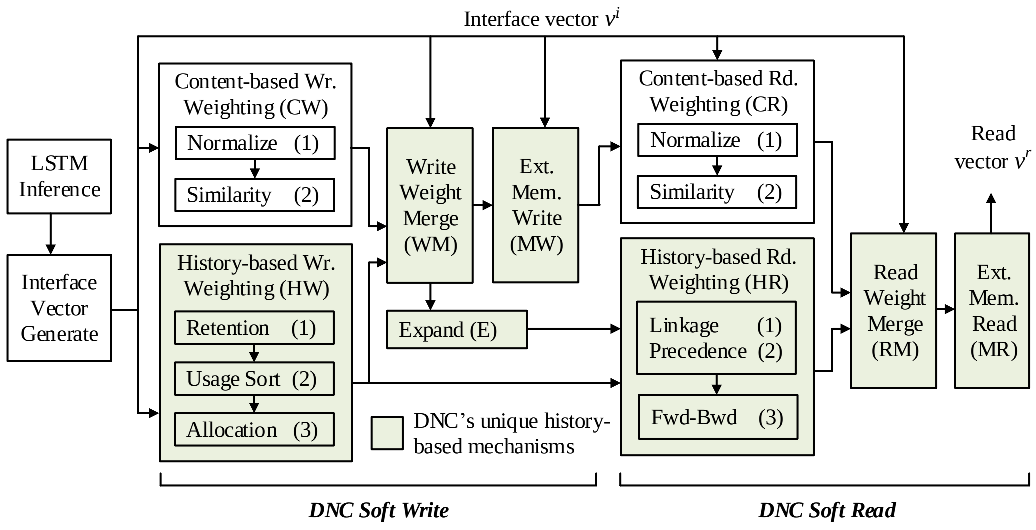


Figure 1.6: LSTM and memory-augmented neural network.

amount of information that can be stored in the network is bounded by the size of the underlying network. Therefore LSTM lacks the scalability to handle complicated sequence events with long-term dependencies.

To address the scaling problem of LSTM, MANNs have been proposed as shown in Figure 1.6. A memory unit is connected to an NN (typically an LSTM) and the external memory[1] can be accessed by attention-based mechanisms through soft read and write heads. Specifically, at time t the LSTM sends an *interface vector* v_t^i to the memory unit and receives a *read vector* v_t^r from the memory unit. In this way, the dynamic state can be explicitly decoupled from the neural network. NTM [83] is a MANN that outperforms LSTM by employing content-based soft write and read to access the memory, a form of attention mechanism. However, memory access history is completely discarded in memory slot selection and weighting. DNC [73] extends NTM and the memory is accessed based on both memory content and memory access history. History-based attention mechanisms allow DNC to achieve much better performance than NTM, but also introduce complex memory manipulations and a

[1]In the context of MANN, an external memory refers to a memory external to the NN (e.g. LSTM) that stores data.



CW.(1): $\|\mathrm{M}[i,\cdot]\| \leftarrow \mathrm{M}[i,\cdot] \quad \|k^w\| \leftarrow k^w$	CR.(1): $\|\mathrm{M}[i,\cdot]\| \leftarrow \mathrm{M}[i,\cdot] \quad \|k^r\| \leftarrow k^r$
CW.(2): $w^u \leftarrow \mathrm{softmax}(s^w \|\mathrm{M}[i,\cdot]\| \cdot \|k^w\|)$	CR.(2): $r^u \leftarrow \mathrm{softmax}(s^r \|\mathrm{M}[i,\cdot]\| \cdot \|k^r\|)$
HW.(1): $\psi[i]=\prod_{r=1}^{R}(1-\mathrm{g}^f[r]w^r[i,r])$	HR.(1): $\mathrm{L}=\{(E\text{-}w\text{-}w^{\mathrm{T}})\circ\mathrm{L}+w^w p^{\mathrm{T}}\}\circ(E-I)$
HW.(2): $\mathrm{I_s} \leftarrow \mathrm{sort}\,(u+w^w\text{-}u\circ w^w)\circ\psi$	HR.(2): $p=(1\text{-}\sum_{i=1}^{N} w^w[i])p+w^w$
HW.(3): $w^a[\mathrm{I}_s^i]=(1\text{-}u[\mathrm{I}_s^i])\prod u[\mathrm{I}_s^{1\to i}]$	HR.(3): $f^r=\mathrm{L}w^r \quad b^r=\mathrm{L}^{\mathrm{T}}w^r$
WM: $w^w=\mathrm{g}^w(\mathrm{g}^a w^a+(1\text{-}\mathrm{g}^a w^u))$	RM: $w^r=m_1^r b^r+m_2^r r^u+m_3^r f^r$
E: $w[\cdot,i]=w^w$	MR: $v^r=\mathrm{M}^{\mathrm{T}}w^r$
MW: $\mathrm{M}\leftarrow\mathrm{M}\circ(E\text{-}w^w(v^e)^{\mathrm{T}})+w^w(v^w)^{\mathrm{T}}$	

Figure 1.7: DNC inference dataflow: history-based and content-based memory access in DNC memory unit.

variety of new state memories as highlighte in Figure 1.6.

1.2.1.2 DNC Memory Unit

In this work, we focus on DNC memory unit where elaborate memory operations take place. The DNC inference dataflow and mathematical descriptions of the operations are shown in Figure 1.7. Compared to other variants of MANNs such as NTM or MemNet, DNC is the only model that incorporates history-based memory access. The NN (e.g. LSTM) sends an input to the memory unit, known as the *interface vector* v^i, including the necessary access information such as write key k^w, read key k^r or write vector v^w. The memory unit returns the output *read vector* v^r. Suppose the memory M is modeled as a $N \times W$ matrix ($N > W$) and the number of read heads (i.e., number of parallel reads) is R, we provide a brief operational explanation of the DNC soft write and read.

Soft Write: As illustrated in Figure 1.7, soft write is done in two steps: 1) compute write weighting w^w, and 2) memory write, i.e., apply the weighting w^w to the values (write vector v^w and erase vector v^e) and write them to memory. In DNC, the write weighting is a combination of *content-based weighting* and *history-based weighting*. The content-based weighting is inherited from NTM, and it is based on the similarity to the write key. Mathematically, the memory entries $M[i, \cdot]$ and the write key k^w are first normalized, and the similarity of the two are computed as the content-based write weighting w^u.

The history-based weighting is brand new in DNC. DNC enhances the selection of memory cells by biasing towards those that are most recently read from (based on read weighting w^r from the previous time step), least recently written to (based on write weighting w^w from the previous time step), or deemed inconsequential (based on free gate g^f). The history-based weighting is computed in three steps: 1) the retention vector ψ is first calculated based on the free gate g^f and the read weighting w^r; 2) the *usage vector* u is updated based on the retention vector and the write weighting w^w, and then sorted; and 3) the history-based write weighting w^a is computed by the accumulation of product of the sorted usage. The content-based weighting w^u and the history-based weighting w^a are combined to obtain the write weighting w^w.

Soft Read: Soft read is done in two steps: 1) compute read weighting w^r, and 2) memory read, i.e., apply the weighting w^r to memory M to obtain the read vector v^r. Similar to soft write, soft read combines both content-based weighting and history-based weighting. The content-based read weighting r^u is computed in the same way as the content-based write weighting.

The history-based read weighting is computed in three steps: 1) the write weighting w^w is first expanded to an $N \times N$ matrix to derive linkage matrix L. The *linkage* tracks the order in which memory locations are written to; 2) the precedence vector p is updated to track the degree each memory entry is most recently written to; and 2) a *forward and backward pass* is used to merge the read weighting w^r from the previous time step with the linkage matrix L, as well as the content-based read weighting r^u to update the read weighting w^r.

1.2.1.3 State-of-the-art MANN Accelerators

Conventional NN or matrix multiplication accelerators do not fully support DNC because they lack the primitives including sorting and matrix transpose, and miss a specialized memory unit to support a variety of state memories. MANN accelerators [98, 97, 100, 99, 102, 101] have been proposed for MANN's memory unit. The memory unit receives an input interface vector from an NN accelerator that executes LSTM inference. In performing the inference, the NN accelerator may communicate with off-chip DRAMs. Here we focus on NTM and DNC accelerators that support both soft write and soft read and omit simpler accelerators that do not come with such support. Some NTM and DNC accelerators use a centralized-memory architecture

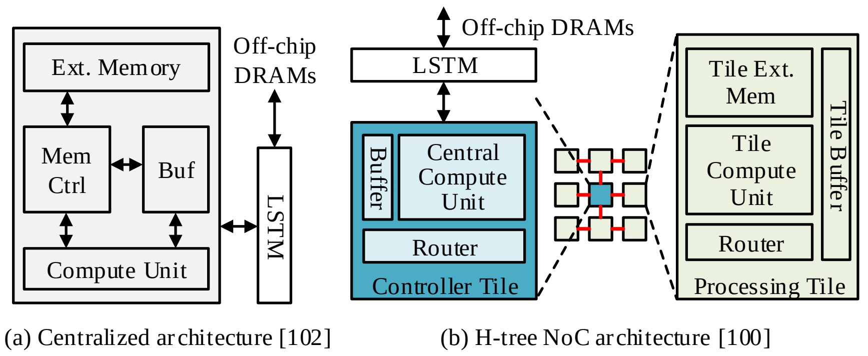


Figure 1.8: Centralized-memory architecture and tiled architecture for accelerating MANN's memory unit.

[99, 102, 101] as shown in Figure 1.8(a). In particular, [99, 102] improve the memory access efficiency by introducing in-memory compute through resistive crossbar or analog operations. However, the centralized memory ultimately limits the bandwidth and parallelism available, and the emerging devices and custom mixed-signal circuits are not yet practical for sufficiently large memory sizes.

Manna [100] introduces the first tiled NoC architecture as shown in Figure 1.8(b) to solve the bandwidth and parallelism limitation. Manna contains two types of tiles: processing tile (PT) which includes external memory sub-banks and the associated compute units, and controller tile (CT) which includes top-level processing units to distribute information to the PTs and collect results from the PTs. Designed for NTM, Manna does not support DNC due to the lack of support for new primitives like sorting and new state memories for maintaining access history. Manna's H-tree NoC also becomes less efficient in carrying inter-tile communication when the PT count increases. The inefficiency is exacerbated by DNC's history-based attention mechanisms that inject complex traffic patterns onto the NoC, limiting the scalability.

1.2.2 Neural Ordinary Differential Equations

NODE [11] is extended from ResNets [8] for enhanced capability on modeling continuous-time events. Conventional ResNet builds complicated transformations by

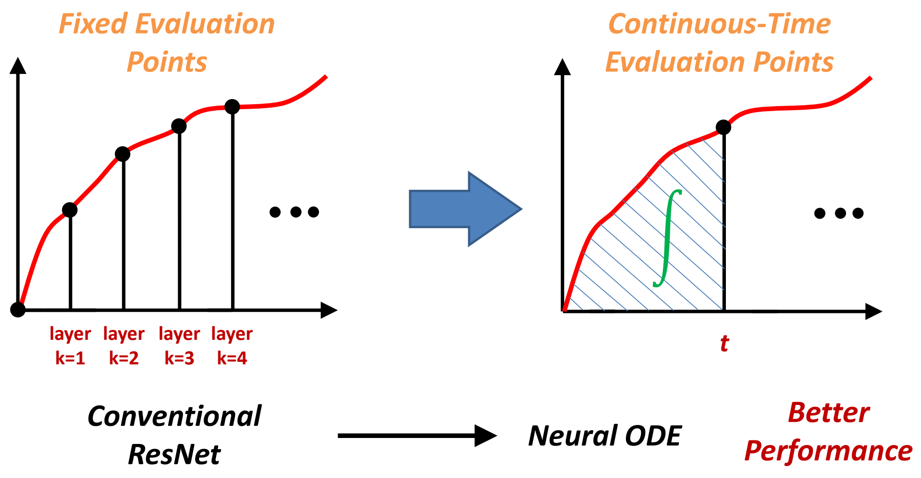


Figure 1.9: Left: A Residual network defines a discrete sequence of finite transformations. Right: A neural ODE network defines continuous transformations of the state.

composing a sequence of transformations to a hidden state as (1.5):

$$h_{k+1} = h_k + f(h_k, \theta_k),\ \ k \in \{0, 1, ..., T\}, \tag{1.5}$$

where k is the layer index, h_k is the discrete hidden state, and f is the network with trainable parameters θ_k at k-th layer. These iterative updates can be seen as an Euler discretization of a continuous transformation. Increasing the number of layers and taking smaller steps in the limit, we can parameterize the continuous dynamics of hidden units using an ODE specified by a neural network as (1.6):

$$\frac{\mathrm{d}h(t)}{\mathrm{d}t} = f(h(t), t, \theta),\ \ h(0) = x,\ \ t \in [0, T] \tag{1.6}$$

where t is a continuous time variable, $h(t)$ is the continuous hidden state, and f is the network with trainable parameters θ. Starting from the input layer $h(0) = x$, we can define the output layer $h(T)$ to be the solution to this ODE initial value problem at some time T. Integration of (1.6) is the mechanism for feed-forward computation of neural ODEs. The input-output relation can be described as (6.2):

$$h(T) = h(0) + \int_{0}^{T} f(h(t), t, \theta) dt \tag{1.7}$$

This value can be computed by a numerical differential equation solver (e.g. RK method), which evaluates the hidden unit dynamics f wherever necessary to determine the solution with the desired accuracy. Figure 1.9 contrasts conventional ResNets and Neural ODEs. Compared to discrete-layer models, the feature map evolves smoothly with NODE's continuous depths. The continuous transformations improve the modeling capability especially for time series problems or normalizing flows; however, the numerical integration introduces higher computational complexity and new kernels that cannot be accelerated by state-of-the-art DNN accelerators. An efficient acceleration hardware is yet to be researched for Neural ODEs.

1.3 Scope of this Work

In this work, new design techniques are proposed to improve upon the state-of-the-art designs reviewed in the previous sections through the use of algorithm, architecture and circuit techniques that are co-optimized to work with the domain-specific algorithms.

1.3.1 Polar Codes

In this work, we present a 0.64mm^2 configurable SCL decoder chip using a split-tree architecture in 40nm CMOS [105, 106]. The decoding tree is split to 4 subtrees to be decoded by 4 sub-decoders in parallel with decision reconciliation in every stage. The new split-tree architecture improves the throughput and cut the latency by nearly

4×. To maximize utilization, 8 frames are interleaved and decoded simultaneously to increase throughput by another 8× to 3.25Gb/s for code length up to 1024b and variable code rates. Dynamic clock gating reduces the peak power dissipation to 42.8mW at 0.9V, or 13.2pJ/b. The throughput, energy efficiency and area efficiency are 5.3×, 15.9× and 4.0× better, respectively, than the SCL decoder chip in a 28nm FD-SOI process [29]. Compared to the latest SCL decoder chip [18] in a more advanced 16nm process, our test chip achieves a similar throughput and 3× better area efficiency. These results make this chip suitable for 5G applications.

1.3.2 LDPC Codes

In this work, we extend the post-processing method that was first presented in [51] using ideas from simulated annealing (SA). The SA algorithm combines random walk (or heating in annealing terminology) and gradient descent (or cooling) to escape local minima [107, 108, 109]. In post-processing, we use message reweighting or soft bit flipping to perturb, or heat up, local minima, and use BP to cool down for convergence towards a codeword. Compared to well-known approaches above, the cost of implementing post-processing is low: no code change is needed, and the post-processing is entirely based on BP. As post-processing is conditionally invoked, i.e., when a decoder fails to converge at a low BER, the impact on decoding throughput and power is negligible.

We further present two new methods inspired by SA: extended heating and focused heating, aiming at eliminating the vast majority of ETS errors of various structures [110]. We use a rate-0.89 (2209, 1978) array LDPC code [111] that is known for its collection of ETS errors [36] to derive these methods. Finally, we combine extended heating and focused heating into a generalized method that is applicable to LDPC codes with unknown ETS structures. We use a rate-0.83 (1944, 1620) LDPC code for the IEEE 802.11n standard [32] to test the effectiveness of the generalized method.

Experimental results show that post-processing is one of the most practical and efficient solutions in designing low-error-floor LDPC decoders. Table 1.2 summarizes the qualitative features of the proposed approach compared to prior techniques.

1.3.3 NB-LDPC Codes

In this work, we present a 7.04 mm^2 65 nm CMOS NB-LDPC decoder chip for a GF(64) (160, 80) regular-(2, 4) code using the truncated EMS algorithm [112, 113, 61]. We use a fully parallel architecture and scheduling techniques to enhance the throughput to 1.22 Gb/s at 700 MHz. To reduce the power consumption, we design a fine-grained dynamic clock gating based on node-level convergence detection to save 50% power.

1.3.4 Differentiable Neural Computer

In this work, we present HiMA [114], a History-based Memory Access engine to efficiently accelerate DNC memory operations. To the best of our knowledge, HiMA is the *first distributed, tiled architecture* that supports all DNC features. History-based attention mechanisms introduce a variety of new primitives and state memories, require access to various memories concurrently and incur complex traffic profile in an NoC architecture. The design focuses on distributed processing of DNC and optimizing NoC traffic to enhance scalability. We summarize the contributions of this work as follows:

- *Scalable Multi-Mode NoC.* We study the DNC computation and memory access profile, especially history-based attention mechanisms. Based on the analysis, a multi-mode NoC is designed to adapt to DNC's traffic profile, improving both traffic latency and scalability.

- *Optimized Memory Partition.* Conventional row or column-based partition is suboptimal for DNC's new state memories. We consider both content-based and history-based mechanisms and propose a submatrix -based partition to reduce NoC traffic amount.

- *Distributed and Efficiency-Enhanced Kernels.* Both memory and memory operations are distributed to tiles using a new distributed DNC (DNC-D) model. The distributed computational kernels minimize the NoC traffic and provide a higher parallelism. We propose local-global two-stage sort, usage skimming, and softmax approximation to reduce the complexity and improve the computational efficiency.

Prototypes of HiMA are implemented in RTL and synthesized in a 40nm technology. HiMA running DNC and DNC-D demonstrates up to 6.47× and 39.1× improvements in speed, 22.8× and 164.3× improvements in area efficiency, and 6.1× and 61.2× improvements in power efficiency, respectively, over Manna, the state-of-the-art MANN accelerator for NTM. Compared to an Nvidia 3080Ti GPU, HiMA shortens the inference time by up to 437× and 2,646× when running DNC and DNC-D.

1.3.5 Neural Ordinary Differential Equations

Neural ordinary differential equations (NODE) demonstrate superior performance in modeling continuous-time events and normalizing flows. Their higher performance are derived by using numerical ODEs between NN's discrete hidden layers. To run NODE, state-of-the-art NN accelerators need to be attached to general-purpose CPUs or GPUs for elaborate ODE operations, which is unlikely to deliver a high efficiency. To address this, we present a programmable accelerator for training NODE. We study the weight and activation sparsity, the data dependency and reuse patterns, and their impacts on hardware architecture design. The accelerator employs distributed processing elements (PEs) and hierarchical memories that focus on maximizing performance for elaborate NODE operations. The key algorithmic and architectural features are:

- We propose an adaptive sparsity operations during NODE training for reduced computational complexity, while still maintaining excellent training accuracy.
- We develop a multi-mode PE design for NODE compute kernels with configurable interconnects between PEs to handle a variety of ODE solvers.
- We carry out software/hardware co-optimizations to save redundant operations, memory accesses and inter-PE data movements. The hardware efficiency is further enhanced by exploring hardware reuse and memory hierarchy.

1.3.6 DNC-Aided SCL Flip Decoding of Polar Codes

In this work, we propose to use DNC for bit flipping of practical-length polar codes to enhance the accuracy of identifying error bit positions [115]. The main contributions are summarized as follows:

1. A new two-phase decoding is proposed assisted by two DNCs, flip DNC (F-DNC) and flip-validate DNC (FV-DNC), as shown in Figure 7.1. F-DNC ranks mostly likely flip positions for multi-bit flipping. If decoding still fails, FV-DNC is used to re-select flip positions for successive flip decoding trials.
2. We propose new action encoding with soft multi-hot scheme and state encoding considering both PMs and received LLRs for better DNC training and inference efficiency. Training methods are designed accordingly for the two DNCs, where training database are generated based on supervised flip decoding attempts.
3. Simulation results show that the proposed DNC-aided SCL-Flip (DNC-SCLF) decoder outperforms the state-of-the-art techniques by up to 0.34dB in error

correction performance or 54.2% reduction in average number of decoding attempts.

CHAPTER II

Configurable Split-Tree Polar SCL Decoder

In this section, we present a configurable split-tree SC list decoder chip[1] that works by dividing a polar code's decoding tree to sub-trees following a split-tree decoding algorithm.

2.1 Decoding Algorithm

2.1.1 Successive Cancellation (SC) Decoding

The bits of the codeword x_0^3 are modulated and transmitted through a physical channel, where noise is injected. On the receiver side, each "bit" in the codeword is received as a multi-bit "soft" value, known as log-likelihood ratios (LLRs). A polar decoder operates on the LLRs to produce bit estimates. SC is the first and most widely used decoding algorithm for polar codes.

The SC decoding algorithm was proposed in [116]. It can be visualized by a decoding trellis as shown in Figure 2.1. The LLRs are provided on the left hand side and the bit decisions are made on the right hand side. The trellis consists of $n = \log_2 N$ stages of minimum (F) and summation (G) operations. The F function

[1]Special thanks to Dr. Sung-Gun Cho for his help in physical design and chip testing

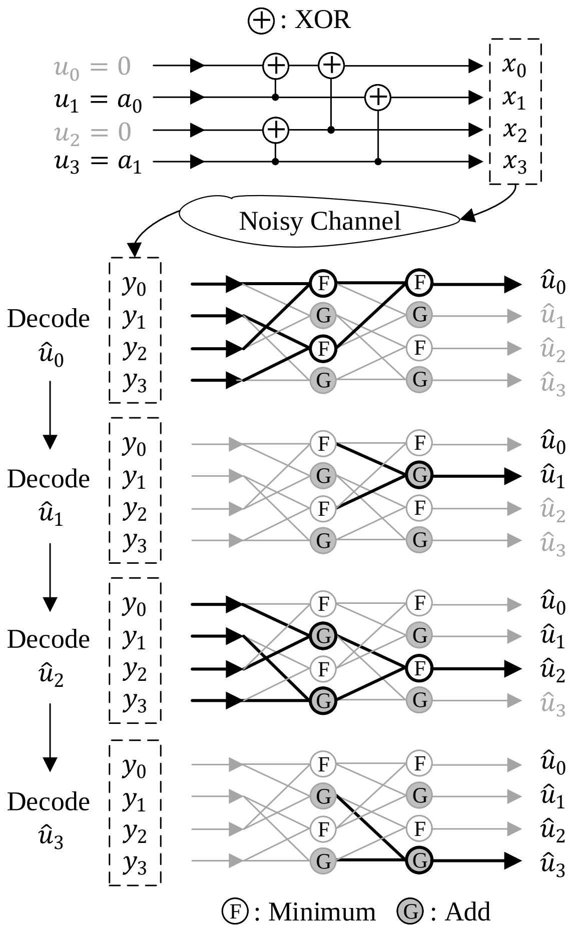


Figure 2.1: Encoding graph (top) and SC decoding trellis (bottom) of a (4,2) polar code.

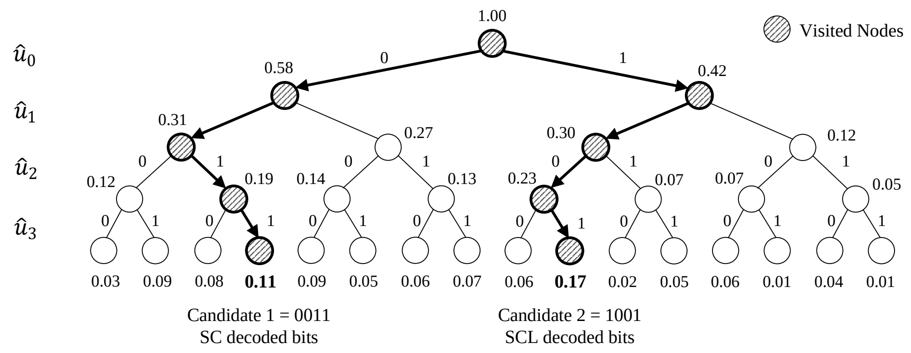


Figure 2.2: SC and SCL ($L = 2$) decoding represented in a binary tree for a (4,4) polar code.

receives two LLRs L_1 and L_2 and finds the minimum as follows:

$$F(L_1, L_2) = \text{sign}(L_1) \cdot \text{sign}(L_2) \cdot \min(|L_1|, |L_2|). \tag{2.1}$$

The G function receives the partial modulo-2 sum of all previously decoded bits, $\hat{u}_s$, (not annotated in Figure 2.1) in addition to the two LLRs, and computes the conditional sum as follows:

$$G(L_1, L_2, \hat{u}_s) = L_1 \cdot (-1)^{\hat{u}_s} + L_2. \tag{2.2}$$

The decoding follows a bit-by-bit sequential order. An example of SC decoding is shown in Figure 2.1, where in each decoding step, the highlighted paths and nodes are active. One can easily see that only a subset of F and G functions at certain stages are active at a time to decode 1 bit, leaving the other functions in the trellis idle. Assume it takes 1 unit time per function (F or G) and the trellis is directly mapped in hardware. The latency to decode a bit is variable ranging from 1 to $\log_2 N$. It can be shown that the latency to decode a N-bit codeword is $2N-2$. For example, decoding a 1024bit polar code requires 2046 time units. If the trellis is directly mapped to hardware, the hardware complexity is approximately $O(N \log_2 N)$.

The SC decoding can also be represented by the depth traversal of an N-level binary tree, as shown in Figure 2.2. For a $N = 4$ polar code, the binary tree consists of 4 levels, where the branching at each level represents the decoding of a bit to either 0 or 1. The sequential decoding starts from the top and descends. At each level i, a branch is taken, corresponding to calculating the likelihood $L(\hat{u}_i)$, based on which $\hat{u}_i$ is decoded.

For the depth traversal, we can compute the probability of the path, or the path metric (PM) $PM(\hat{u}_{i-1})$, for the traversal to reach a node representing the decision of $\hat{u}_{i-1}$. For the next step, the path can branch to one of the child nodes, $\hat{u}_i = 0$ or $\hat{u}_i = 1$. The PM can be updated as follows:

$$PM(\hat{u}_i) = PM(\hat{u}_{i-1}) + \log(1 + e^{(-1)^{\hat{u}_i} L(\hat{u}_i)}) \tag{2.3}$$

The goal of SC decoding is to find the most likely path, or the path of the highest PM. To achieve this goal, SC decoding takes one branch at a level, and keeps the path of the highest PM. The highlighted path on the left in Figure 2.2 represents the survival path in SC decoding. The value associated with each node is the PM for the decoding path from root node to that node. Note that after selecting the survival branch at a level, the other branch and its child nodes will never be considered. In the example, the SC decoder chooses the path $\begin{bmatrix} 0 & 0 & 1 & 1 \end{bmatrix}$ with a PM of 0.11.

2.1.2 Successive Cancellation List (SCL) Decoding

SC decoding makes one hard decision per step and never visits other possible paths. Though SC decoding is simple to implement, it is not guaranteed to find the best global path because the local optimal path may not be part of the global optimal path. In the example shown in Figure 2.2, the global optimal path is $\begin{bmatrix} 1 & 0 & 0 & 1 \end{bmatrix}$ with a PM of 0.17, but SC decoding misses this path because it takes the optimal local decision in step 1.

The SCL decoding algorithm [117] overcomes this drawback by keeping a list of L candidate paths at each level in traversing the binary tree. SC decoding can be viewed as a special case of SCL decoding with $L = 1$. Figure 2.2 shows SCL decoding for $L = 2$. At each level, the two most likely paths are kept. Moving to the next level, the two most likely paths branch to 4 child nodes, leading to 4 candidate paths. The SCL decoder selects the top 2 paths among the 4 to keep. In the end, the best path is selected.

In general, a SCL decoder calculates the PMs of all $2L$ child nodes ($\hat{u}_i = 0$ or $\hat{u}_i = 1$) that are connected to the L survival paths from the previous step based on (2.4). The decoder selects L paths among the $2L$ candidates with the highest PMs to keep. SCL decoding enhances the decoding accuracy with larger L.

$$\begin{aligned} PM_j(\hat{u}_i) &= PM_j(\hat{u}_{i-1}) + \log(1 + e^{(-1)^{\hat{u}_i} L(\hat{u}_i)}) \\ &\qquad 0 \leq j \leq L-1, \hat{u}_i \in \{0,1\}. \end{aligned} \tag{2.4}$$

The decoding accuracy can be further improved by concatenating polar code with cyclic redundancy check (CRC) [118, 119]. In addition to ranking PMs, codewords corresponding to valid paths also need to pass CRC check.

An N-bit SCL decoder can be designed with an N-bit SC decoder core and additional processing logic and memory to sort and track L candidate paths. Assume it takes 1 unit time to sort and track candidate paths after decoding each bit, the SCL decoding latency is $(2N - 2) + N$. However, the sorting overhead is only incurred if a bit is a free bit. Take the 1024b, rate-1/2 polar code selected by the 5G eMBB standard as an example. The code has 512 free bits, so the decoding latency is $(2N - 2) + N/2 = 2558$ time units, or 25% longer than SC decoding.

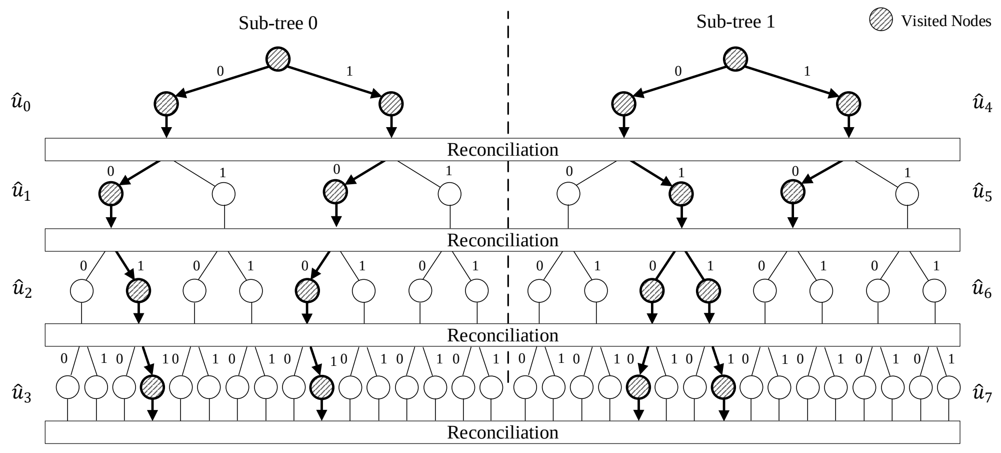


Figure 2.3: Split-tree SCL decoding tree for (8,8) polar code with list size $L = 2$ and split factor $M = 2$.

2.1.3 Split-tree SCL Decoding

To reduce the latency and improve the throughput of SCL decoding, a split-tree SCL (ST-SCL) decoding algorithm [120] was proposed. Conceptually, the N-level decoding tree is split into M subtrees of N/M levels, equivalent to splitting the N-bit code to M N/M-bit subcodes linked by a constraint matrix. An ST-SCL decoder consists of M N/M-bit SC sub-decoders that operate on the subcodes in parallel. Each SC sub-decoder works on a $M\times$ shorter subcode. The hardware complexity of all M sub-decoders is $O(M \cdot N/M \log_2(N/M)) = N \log_2(N/M)$. The theoretical decoding latency is reduced by a factor of M compared to SC or SCL decoding to $O(N/M)$, and the throughput is increased by the same factor. ST-SCL decoding requires an extra reconciliation stage to combine sub-decoders' local decisions.

To illustrate ST-SCL decoding, suppose a $N = 8$ polar code is decoded with a list size of $L = 2$ and a split factor of $M = 2$. The 8-level decoding tree is split to sub-tree 0 of 4 levels and sub-tree 1 of 4 levels, as shown in Figure 2.3, sub-tree 0 for $\hat{u}_0^3$ and sub-tree 1 for $\hat{u}_4^7$. The decoding of each subcode is based on a 4-bit code trellis, similar to Figure 2.1. Frozen bits on each sub-tree are determined by the original 8-level decoding tree. ST-SCL decoding proceeds level by level. At level i, the decoding consists of two stages:

1. Sub-decoding: SC sub-decoder 0 and sub-decoder 1 operate on their code trellises and compute the likelihood of bit $\hat{u}_i$ and $\hat{u}_{4+i}$, respectively.

2. Reconciliation: The PMs of the 4 candidate paths per sub-decoder are computed following (2.4), and the 2 survival paths per sub-decoder (called sub-paths) are selected. The sub-paths are assembled, checked for constraints, and the top 2 global paths are selected. The top global paths are then disassembled and distributed to the two sub-decoders.

The stage 1 above is the same as in SCL decoding, but stage 2, the reconciliation, is new in ST-SCL decoding. If a bit is not a frozen bit, the sub-decoder provides L sub-paths. In decoding a level, if none of the M bits are frozen bits, there could be up to L^M possible global paths made by combinations of sub-paths. The global PMs (GPM) for the global paths are calculated by summing the sub-path PMs.

The complexity of the reconciliation stage is proportional to L^M, limiting the maximum split factor and list size. However, some of the L^M global paths are invalid and can be removed from evaluation. For example, in Figure 2.3, if u_5 is a frozen bit, the paths with $\hat{u}_5 = 1$ are not valid because $\hat{u}_5$ can only be 0. In the best case, if both $\hat{u}_i$ and $\hat{u}_{4+i}$ are frozen bits at level i, reconciliation is bypassed. Only in the case when both $\hat{u}_i$ and $\hat{u}_{4+i}$ are free bits at level i, all L^M GPMs need to be evaluated. Valid global paths are sorted by GPM. The top L global paths are kept and are disassembled into sub-paths and distributed to the sub-decoders.

2.2 ST-SCL Decoder Architecture for High Throughput and Low Latency

In this section, we present a ST-SCL decoder architecture to realize the near-theoretical latency and throughput improvements of ST-SCL decoding by an efficient

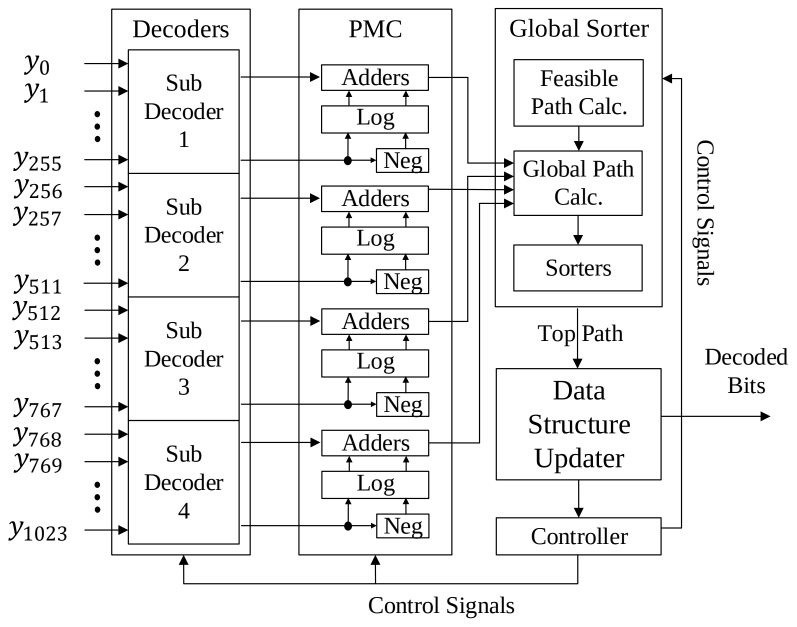


Figure 2.4: Top-level architecture for a 1024b list-2 split-4 configurable SCL decoder.

reconciliation stage and a significantly higher utilization of the SC decoding hardware. A prototype ST-SCL decoder is shown in Figure 2.4 for the list size $L = 2$ and the split factor $M = 4$. The prototype design supports configurable code length up to $N = 1024$ and variable code rates. In decoding a 1024b polar code, the input LLRs are equally split to 4 groups. A group of 256 LLRs is sent to a SC sub-decoder. The 4 SC sub-decoders operate on their decoding trellises to compute the 4 bit likelihoods in parallel.

The reconciliation stage is divided to 3 steps:

1. PM calculation: For each sub-decoder, the PMs of $2L = 4$ candidate paths are calculated following (2.4) and $L = 2$ sub-paths are selected.

2. Enumeration and sorting: Based on frozen bit information, valid combinations from the $L^M = 16$ sub-path combinations are enumerated, and the GPMs are calculated. The GPMs are sorted to select the top $L = 2$ global paths.

3. Update: The top global paths are disassembled and distributed to the 4 sub-

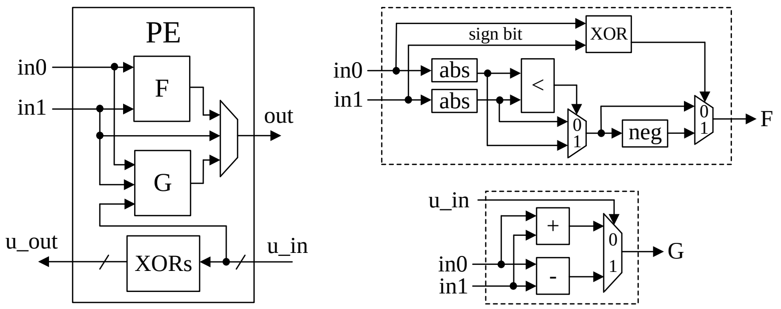


Figure 2.5: Processing element (PE) design.

decoders.

The 3 steps are carried out by PM calculator (PMC), global sorter (GS) and data structure updater (DSU) blocks shown in Figure 2.4. We discuss the details of the sub-decoding stage and the reconciliation stage below.

2.2.1 Sub-Decoder Design

Each SC sub-decoder decodes a polar code of length up to $N = 256$ bits ($n = 8$) by recursively passing through an 8-stage decoder trellis following the sequential order as illustrated in Figure 2.1. We group an F and a G function in a processing element (PE) as shown in Figure 2.5. It consists of a F function described in (2.1), a G function described in (2.2) and XORs to compute partial modulo-2 sums of decoded bits. Back routing is needed to feed decoded bits back to G functions. The back routing is implemented by routers between decoding stages.

A direct mapping of the decoding trellis produces an 8-stage sub-decoder architecture and each stage consists of 128 PEs. However, the hardware utilization of a direct mapped architecture is very low. A well-known pattern in SC decoding is that stage i has at most 2^{n-1-i} PEs active at the same time. For example, in Figure 2.1 for a $N = 4$ bit polar code ($n = 2$), stage 0 has 2 PEs active and stage 1 has 1 PE active in decoding $\hat{u}_0$. Therefore, instead of the direct mapping, we design the sub-decoder by instantiating only 2^{n-1-i} or 2^{7-i} PEs in decoding stage i (named D_i), as shown in

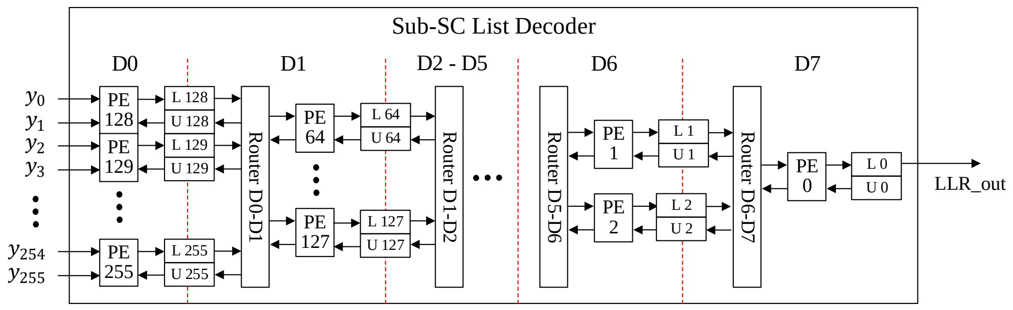


Figure 2.6: 256b SCL sub-decoder design.

Figure 2.6. In total, the 8 stages contain $\sum_{i=0}^{7} 2^{7-i} = 255$ PEs, instead of $128 \times 8 = 1024$ PEs for a direct mapped architecture.

To achieve a high clock rate, the sub-decoder is pipelined to 8 stages, aligned with decoding stages, D0 to D7. The pipeline boundaries are shown as dotted lines in Figure 2.6. The 255 pipeline registers, L0 to L255, store intermediate likelihoods that are propagated forward, and 255 state registers, U0 to U255, store the partial modulo-2 sums of the decoded bits that are routed back.

To support a shorter code length, decoding stages can be bypassed. For example, to support a 512b code, the code is split to 4 128b subcodes to be decoded by the 4 sub-decoders. In each sub-decoder, stage D0 is bypassed by forwarding the 128 input LLRs to the stage D1 PEs. Multiplexers are placed at the inputs to the stage D1 PEs to select either the bypassed input LLRs or the intermediate LLRs from the D0-D1 router. Bypassing stage D0 shortens the latency and increases the throughput by reducing the pipeline depth from 8 to 7 stages. The clock inputs to the bypassed D0-D1 pipeline registers are gated to save power. Similarly, to support a 256b code, in each sub-decoder, stage D0 and D1 are bypassed by forwarding the 64 input LLRs to stage D2 PEs. Both D0-D1 and D1-D2 pipeline registers are clock gated to save power.

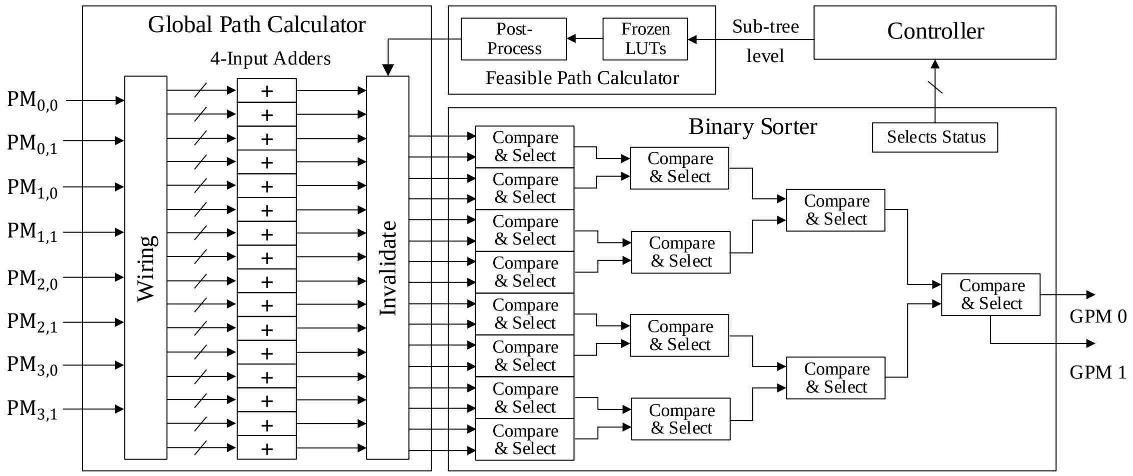


Figure 2.7: Split-tree reconciliation global sorter architecture for $L = 2$ and $M = 4$.

2.2.2 Reconciliation Design

A 3-stage reconciliation is done by PMC, GS and DSU. For each sub-decoder, the PMC takes the soft decision of a bit (LLR of a bit decision being 0) from each sub-decoder output and the L survival paths to compute the PMs of $2L$ candidate paths by (2.4). To reduce the complexity of exponentiation and logarithm evaluations, the PMC employs a piece-wise linear approximation (2.5).

$$\log(1+e^x) \approx \begin{cases} 0 & \text{for } x \leq -1 \\ 0.5(x+1) & \text{for } -1 < x < 1 \\ x & \text{for } x \geq 1 \end{cases} \tag{2.5}$$

The PMC consists of 4 sets of hardware, one set per sub-decoder. A set consists of one negation block to compute the LLR of a bit decision being 1, two log-approximation blocks and 4 adders to compute the PMs of 4 candidate paths. From the 4 candidate paths, the top 2 candidate paths are selected to be passed on to the GS.

The GS is shown in Figure 2.7. It consists of a feasible path calculator, a global path calculator and a binary sorter. The feasible path calculator uses frozen bit lookup tables (LUTs) to generate control signals for selecting only the valid global paths. The LUTs can be reconfigured to support different code lengths and rates. To save area, the LUTs are implemented as 4 copies of length-256 cyclic shift registers to store frozen bit indicators for each subcode of up to 256 bits.

The global path calculator sums up all combinations of sub-paths using L^M 4-input adders. For $M = 4$ and $L = 2$, there are a total of $L^M = 16$ possible global paths. The complexity of wiring to route the local PMs and the number of adders increase exponentially with the split factor M, limiting the practical M to 4. The 16 GPMs are filtered by the feasible path calculator. The GPMs of the invalid paths are set to the minimum value. The filtered GPMs undergo a 4-stage binary sorter to select the top and the second top global paths. The sub-paths that are present in the top and the second top global paths are recorded. Note that the second top global path is approximated by the smaller GPM of the final-stage comparator and this approximation introduces negligible performance loss in mid-to-high SNR regime.

Finally, the DSU disassembles the top L global paths to constituent sub-paths and distributes them to the sub-decoders. The disassembling is done by marking the corresponding local PMs as visited and updating the list state registers in controller. The back-propagation XORs also update the state registers Us in sub-decoders. The worst case DSU delay happens when the newly decoded bits back-propagate through all 8 stages of XOR network inside sub-decoders.

The 3-step reconciliation, including PMC, GS and DSU, occupies only 0.02mm^2 in 40nm CMOS when synthesized at a 500MHz clock frequency.

2.2.3 Pipeline and Hardware Utilization

In the ST-SCL decoder, the 4 sub-decoders operate in parallel and feed to the 3-step reconciliation. The sub-decoder is pipelined to 8 stages, from D0 to D7; and

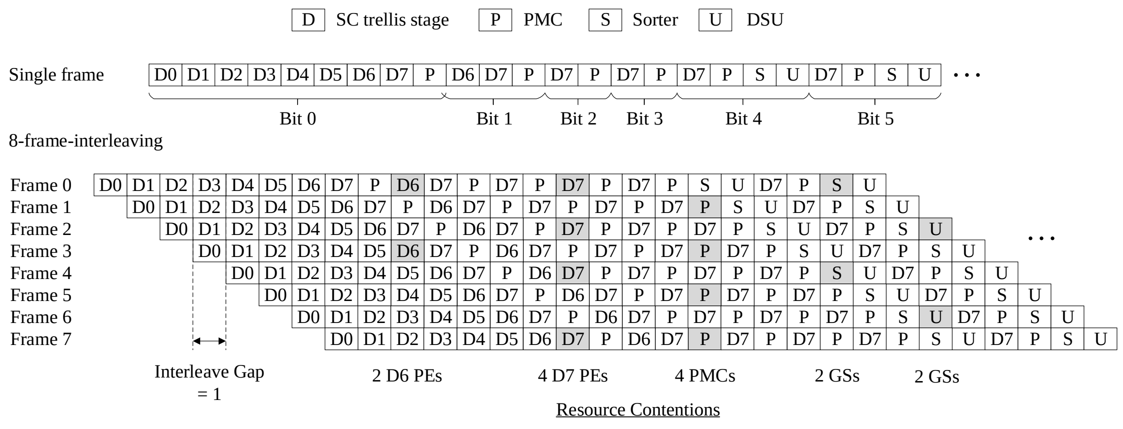


Figure 2.8: Split-tree SCL decoding pipeline for a single frame (top) and 8-frame-interleaving (bottom).

the 3-step reconciliation is pipelined to 3 stages, PMC, GS, and DSU, abbreviated by P, S and U. What complicates the design is that the decoding of any given bit follows a different set of pipeline stages. The irregularity is due to two factors: 1) the variable path through the trellis for decoding a bit, as shown in Figure 2.1, and 2) S and U stages can be bypassed if all four bits at a given level are frozen bits.

An example is illustrated in Figure 2.8. Decoding the 4 bits in level 0 by the 4 sub-decoders requires going through all 8 stages of the trellis, corresponding to D0 to D7 pipeline stages; decoding the 4 bits in level 1 by the 4 sub-decoders requires only the last two stages of the trellis, corresponding to D6 and D7 pipeline stages; and so on. In decoding level 0, all four bits are frozen bits, so S and U pipeline stages are bypassed; similarly in decoding level 1, 2 and 3, S and U stages are also bypassed. The irregularity is handled by routers in the sub-decoders and the bypass switches in reconciliation.

To estimate latency and throughput, we use the 1024b, rate-1/2 polar code selected by the 5G eMBB standard as an example. The code has 512 frozen bits. We split the code into $M = 4$ subcodes to be decoded by 4 256b sub-decoders in parallel. Since the latency of SC decoding is $2N - 2$ for a N-bit code, decoding a 256b subcode requires 510 clock cycles. The reconciliation latency depends on the frozen bit pattern. Among 256 decoding levels, 101 levels involve four bits that are all frozen bits and the S and U stages are bypassed, the remaining 155 levels involve at least one free bit. Therefore, the reconciliation latency is $101 + 155 \times 3 = 566$ clock cycles. In total, ST-SCL decoding requires $510 + 566 = 1076$ cycles to decode a 1024b code, or 47% times faster than SC decoding.

The hardware utilization is low if one frame is processed at a time. During sub-decoding, P, S and U stages are idle; and during reconciliation, D0 to D7 stages are idle. Furthermore, during sub-decoding, only one of D0 to D7 stages is active at a time. If we define utilization as the average fraction of active hardware units at a given clock cycle, sub-decoders' PE utilization is only 1.57%, PMC's utilization is 23.8%, and GS and DSU utilization are 14.4%. There is ample room to increase the utilization for improving the efficiency and the throughput of the hardware.

2.3 Frame-Interleaving to Enhance Throughput and Efficiency

To increase the sub-decoders' PE utilization, a straightforward way is to fold the 8 stages of 255 PEs in Figure 2.6 to 1 stage of 128 PEs. Folding reduces the PE count to approximately half, allowing the PE utilization to be doubled to 3.14%, which is still low. Complex wiring, muxes and control logic have to be added to support PE reuse, costing an estimated 24% extra area and 21% longer clock period based on synthesis.

A better approach is to exploit the pipeline to accommodate decoding of multiple frames at the same time. Through frame interleaving, the same hardware is used to process more workload, improving the hardware utilization and increasing the throughput proportional to the number of frames. However, resource contentions may occur. Suppose 8 frames are launched over 8 consecutive cycles (interleave gap = 1), as shown in the 8-frame-interleave pipeline chart in Figure 2.8. The highlighted parts

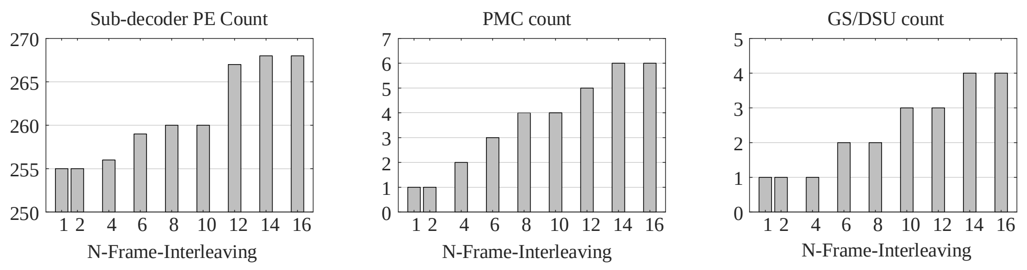


Figure 2.9: Allocation of additional hardware units to support N-frame interleaving.

show contentions for a hardware unit. Resolving the contentions requires multiple copies of hardware units, including PMCs, GSs and DSUs. For example, 2 copies of D6 and 4 copies of D7 are required in each sub-decoder; and 4 copies of PMCs and 2 copies of GSs and DSUs are required in reconciliation.

We studied the optimal number of frames for interleaving by first checking the amount of hardware addition as shown in Figure 2.9. With more frames, more hardware units are needed for the worst-case resource contention scenarios. However, due to the general low utilization of the baseline architecture, resource contentions due to frame interleaving are relatively infrequent. As a result, only a small number of hardware units need to be added to the baseline architecture, making frame interleaving a relatively low-cost approach to increasing both throughput and efficiency. For example, to support 8-frame interleaving, only 5 additional PEs are needed on top of the 255 PEs in each sub-decoder, and 4 copies of PMCs and 2 copies of GSs and DSUs are needed for reconciliation. What is not shown in Figure 2.9 is that N-frame interleaving also requires N copies of state registers in each sub-decoder as well as muxes to select frames.

Besides hardware duplication, frame interleaving also requires extra control and dispatchers, which become more expensive with more frames. Interleaving more frames produces a higher throughput, but the silicon area increases too. We use chip synthesis in a 40nm CMOS technology at room temperature and the nominal

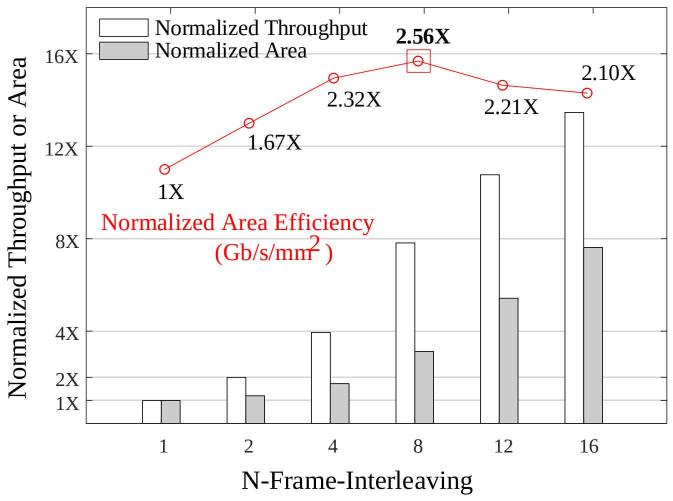


Figure 2.10: Throughput and area of frame-interleaved designs. Silicon area is obtained from chip synthesis in 40nm CMOS at room temperature and the nominal 0.9V supply voltage.

voltage 0.9V to evaluate the area with different number of interleaved frames. As shown in Figure 2.10, the increased throughput initially outpaces the increase in silicon area until it reaches 8 frames. If we use area efficiency, i.e., throughput/area, as the metric, 8-frame interleaving is the optimal, as it increases the throughput by 7.8× and the area by only 3× over the baseline architecture.

As shown in Figure 2.11, with 8-frame interleaving, 2 copies of D6 and 4 copies of D7 are used in each sub-decoder, and the PE utilization increases by 7.8×, from 1.57% to 12.4%. 4 copies of PMCs, 2 copies of GSs and DSUs are also needed, with a utilization of 47.3%, 57.3% and 57.3%, respectively, which are 2×, 4× and 4× higher than the baseline.

Frame interleaving proportionally increases the number of state registers. To estimate the power consumption, the split-4 list-2 8-frame-interleaved ST-SCL decoder was synthesized and placed and routed in a 40nm CMOS process. Figure 2.12(a) shows the power breakdown of the decoder. The switching power of the sequential circuits is the dominant portion, claiming 88% of the total power. Further breakdown of the switching power of sequential circuits in Figure 2.12(b) shows that the switch-

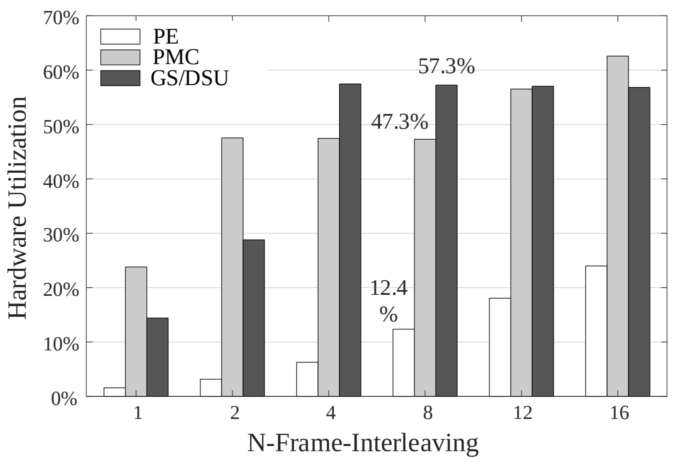


Figure 2.11: Improved hardware utilization with frame interleaving.

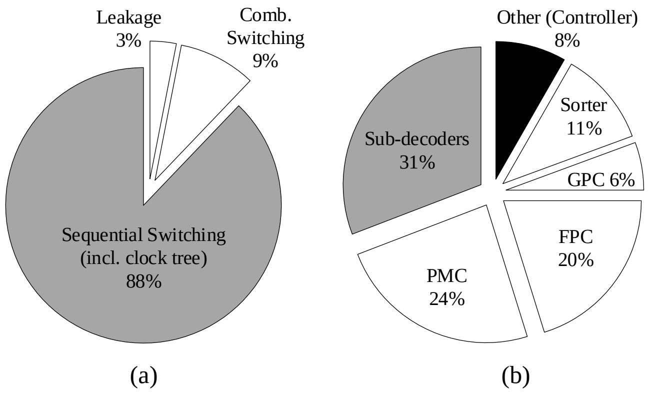


Figure 2.12: (a) Power breakdown of a split-4, list-2, 8-frame-interleaved ST-SCL decoder, and (b) detailed breakdown of the sequential switching power of the ST-SCL decoder. Power is obtained from chip synthesis in 40nm CMOS at room temperature the nominal 0.9V supply voltage.

Table 2.1: Chip design optimization summary based on chip synthesis in 40nm CMOS at room temperature and the nominal 0.9V supply voltage

Decoder Configuration	Max Clock (MHz)	Throughput (Gb/s)	Latency (ms)	Area (mm2)	PE Utilization	Area Efficiency (Gb/s/mm2)
Baseline SCL	720	0.240	0.43	0.138	0.49%	1.74
Split-4 ST-SCL	575	0.547	0.19	0.176	1.57%	3.11
Folded Split-4 ST-SCL	476	0.453	0.23	0.132	3.14%	3.43
Interleave-8 Split-4 ST-SCL	565	4.274	0.19	0.537	12.4%	7.96

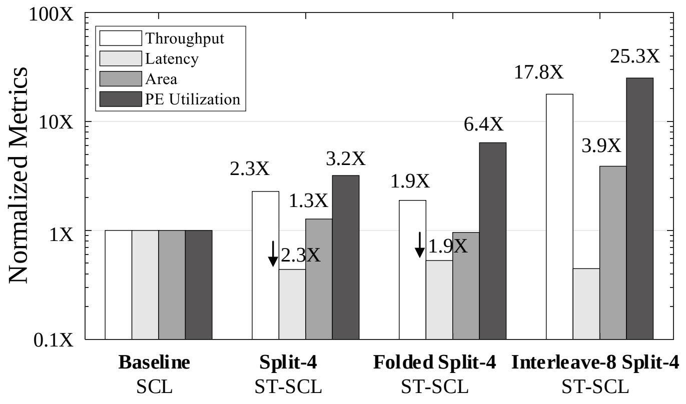


Figure 2.13: Chip design optimization summary based on chip synthesis in 40nm CMOS at room temperature and the nominal 0.9V supply voltage.

ing power of the sub-decoders, the PMCs, and the sorters account for more than 90% of the sequential switching power.

2.4 Summary of Decoder Design Optimization Steps

We summarize the decoder design optimization steps based on 40nm CMOS synthesis for code length of 1024b and list size of 2 in Figure 2.13 and Table 2.1. The conventional SCL decoder is set as the baseline. The baseline runs at maximum clock rate of 720MHz to achieve a 240Mb/s throughput and a 0.43ms latency with a core area of 0.138mm^2. The PE utilization is only 0.49%.

The split-4 ST-SCL decoder enhances the throughput and latency by 2.3× to

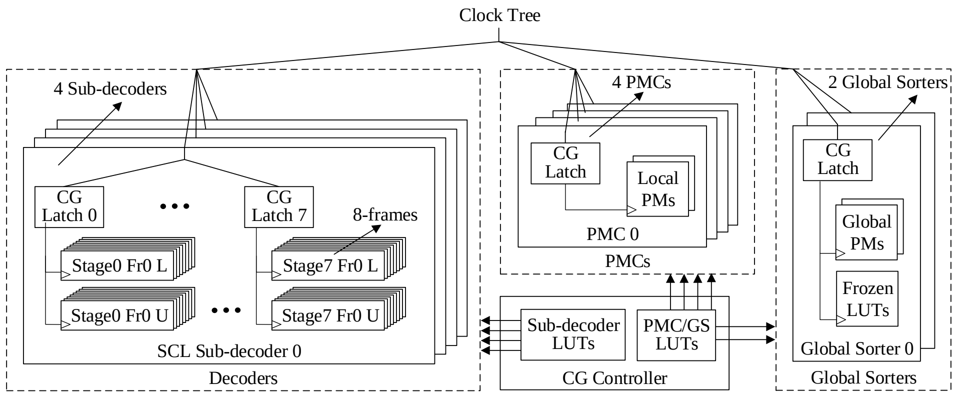


Figure 2.14: Clock gating design for split-4, 8-frame-interleaved ST-SCL polar decoder.

547Mb/s and 0.19ms, respectively, while incurring a 30% area penalty. Compared to the baseline, the ST-SCL decoder increases the PE utilization to 1.57% and the area efficiency to 3.11Gb/s/mm^2. Folding the ST-SCL decoder produces 1.9× higher throughput and lower latency compared to the baseline. Folding also increases the PE utilization to 3.14%.

The split-4 ST-SCL decoder with 8-frame-interleaving boosts the throughput by 17.8× to 4.27Gb/s and shortens the latency by 2.3× compared to the baseline. The area efficiency is 4.57× better than the baseline. The PE utilization is increased to 12.4%.

We apply a per-block clock gating (CG) strategy to reduce the active power consumption of sequential circuits by exploiting the idle cycles. The PE utilization of the sub-decoders is 12.4%, and the utilization of the reconciliation stage is approximately 50%. By systematically gating the clocks to unused hardware units, the active power is reduced proportionally.

Adding per-block CG increases the area by 2.6% for the 8-frame-interleaved split-4 ST-SCL decoder. CG is implemented with a CG controller sending clock enables to sub-decoders, PMCs and GSs as shown in Figure 2.14. Clock enable patterns are determined by code configurations including code length, code rate and frozen bit

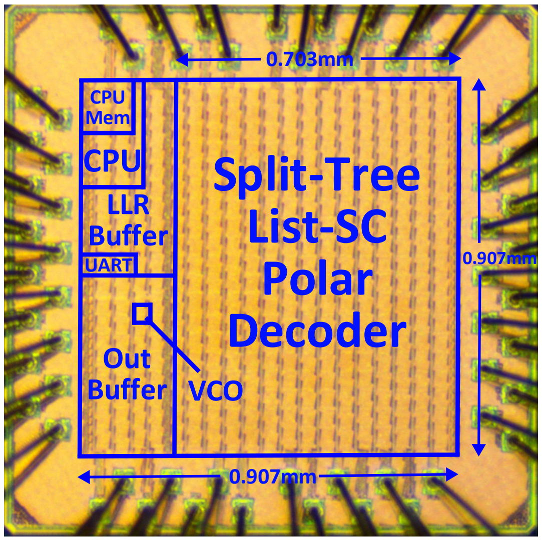


Figure 2.15: Microphotograph of the decoder test chip fabricated in a 40nm CMOS technology.

locations. For each code, clock enable patterns are pre-computed based on stage D0 to D7 active/idle patterns, and they are stored in LUTs inside the CG controller. The decoder top controller sends the code configuration to the CG controller, and the CG controller outputs clock enable signals by reading from the LUTs. In the test chip design, we disable clock input to a sub-decoder stage if the stage will be idle for at least 3 consecutive cycles to avoid frequent off/on switching. For shorter code lengths of 512b and 256b, CG latch 0 and latch 1 inside sub-decoders will switch off the clock inputs to the bypassed stages. The CG controller also stores the number of required PMCs and GSs for each cycle and disables the clock inputs to unused PMCs and GSs to save power.

2.5 Decoder Chip Implementation and Measurements

A test chip for the split-4, list-2, 8-frame-interleaved, configurable polar ST-SCL decoder supporting code length up to 1024b and variable code rates was implemented in 40nm CMOS. The chip microphoto is shown in Figure 2.15. The chip measures 0.91mm×0.91mm, and the decoder core measures 0.70mm×0.91mm, or 0.64mm^2.

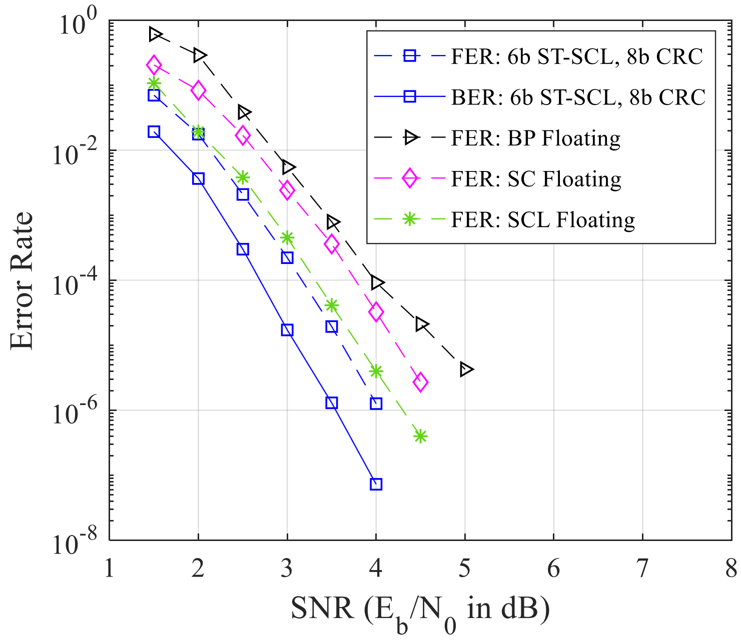


Figure 2.16: Bit error rate and frame error rate performance of 1024b rate-1/2 ST-SCL decoder with split factor 4 and list size 2 using 6-bit quantization and 8-bit CRC.

The chip incorporates input buffers to provide input vectors and output buffers to collect the decoded bits. An on-chip CPU with UART interface enable testing of various code lengths, code rates, number of interleaving frames, and clock gating. It also supports optional post-processing. The chip is verified to be fully functional for code lengths of 512b and 1024b, and code rates of 1/2, 2/3, 3/4 and 5/6.

2.5.1 Measurement Results

The bit error rate (BER) and frame error rate (FER) for decoding a split-4 list-2 1024b rate-1/2 code are plotted in Figure 2.16. The 6-bit quantized decoder uses an 8-bit CRC to assist with final path selection for a better error-correction performance. The design achieves a FER of 10^{-5} at 3.55dB, demonstrating 0.15dB and 0.65dB coding gains over the floating-point SCL ($L = 2$) decoder and the floating-point SC decoder, respectively. Compared to the floating-point belief propagation (BP) decoder, our design provides a 1.1dB coding gain.

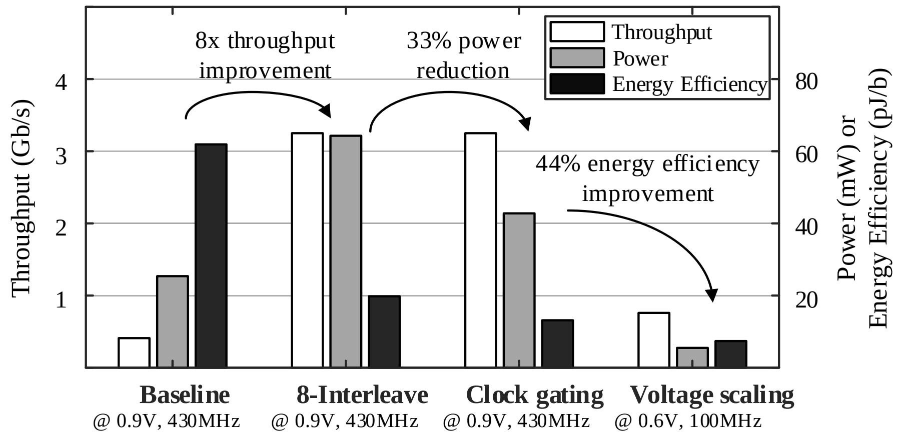


Figure 2.17: Measured throughput, power and energy efficiency of various configurations for the 40nm ST-SCL decoder chip in decoding 1024b rate-1/2 polar codes at room temperature.

The decoder test chip runs at a maximum clock frequency of 430MHz at a 0.9V nominal supply voltage and room temperature when decoding 1024b, rate-1/2 polar codes. Figure 2.17 summarizes the measured throughput and power consumption of the test chip. In the baseline design without frame interleaving and clock gating, the decoder delivers a 407Mb/s throughput. It consumes 25.39mW power, which translates to an energy efficiency of 61.92pJ/b. To achieve a higher throughput, 8-frame interleaving is enabled to provide an 8× throughput to 3.25Gb/s at a power consumption of 64.29mW. The energy efficiency is improved to 19.80pJ/b, due to the efficient sharing and reusing of under-utilized hardware. The power increase from the baseline to the 8-frame-interleaved design is mainly attributed to three factors: 1) the number of state registers (Ls and Us) in the sub-decoders is increased by 8×; 2) PE, PMC and GS/DSU utilization are increased to 12.4%, 47.3% and 57.3%, respectively; and 3) decoder controller and router switching activities are increased to support decoding 8 frames in parallel. Among the three factors, factor 1) contributes the most power increase. The sub-decoders are estimated to consume 31% of the total power, 90% of which is sequential power consumed by state registers. Compared to a

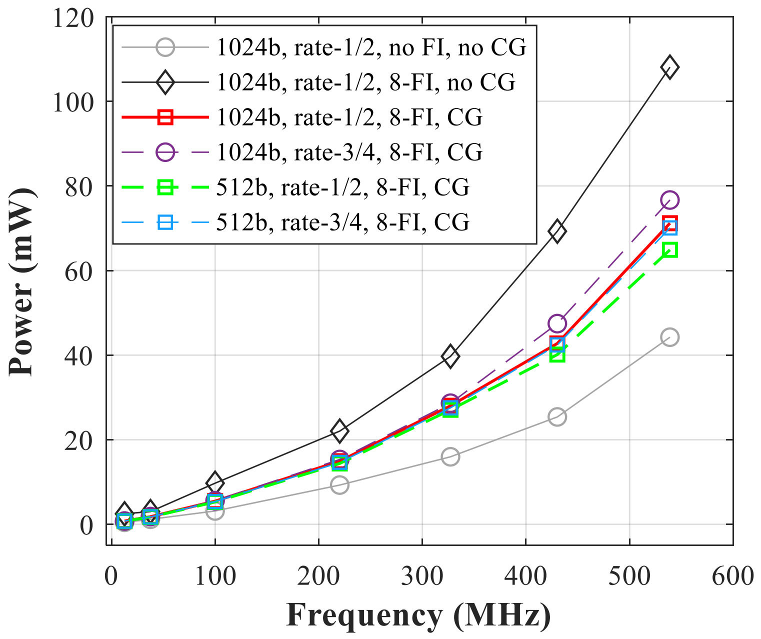


Figure 2.18: Measured power of the 40nm test chip for decoding 512b and 1024b polar codes at room temperature and different supply voltages.

decoder that supports only 1 frame at a time, the 8-frame-interleaved design requires 8 sets of state registers, and the total chip power increases by about 2.2× (calculated from 31%×90%×8) due to the sub-decoder's state register increase. Combining factor 2) and factor 3), the power of the 8-frame interleaved decoder increases by 2.5× over the single-frame decoder.

Clock gating can be enabled to reduce the power consumption to 42.80mW and improve the energy efficiency to 13.17pJ/b. Scaling the supply voltage from 0.9V to 0.6V reduces the maximum clock frequency from 430MHz to 100MHz and further improves the energy efficiency to 7.40pJ/b.

Figure 2.18 shows the power consumption for decoding 1024b and 512b codes of rate 1/2 and 3/4, and the effect of frame interleaving and clock gating. The power was recorded at the lowest operating voltage at each frequency. From the baseline without frame interleaving to 8-frame interleaving, the power increases as expected, but the per-block clock gating effectively lowers the power. For a given code length,

Table 2.2: Comparison of State-of-the-Art Polar Decoders

	This Work	TCAS-I'20 [20]	TVLSI'19 [21]	arXiv'18 [18]	JETCAS'17 [17]	TCAS-II'19 [22]	ASSCC'18 [16]	TCAS-I'19 [23]
Design	silicon	synthesis	synthesis	silicon	silicon	layout	silicon	silicon
Code	up to 1024b variable rate	1024b rate-1/2	1024b rate-1/2	up to 2^{15}b variable rate	1024b rate-1/2	1024b rate-1/2	1024b rate-1/2	1024b rate-1/2
Decoding Algorithm	Split-tree SCL	SCL-flip/ fast-SCL-flip	stack SCL	SCL	SCL	SC	SC	BP
List Size	2	2	2	8	4	1	1	-
Quantization	6b	6b	6b	6b	6b	5b	custom [24]	5b
Process	40nm CMOS	65nm CMOS	90nm CMOS	16nm FinFet	28nm FD-SOI	180nm CMOS	180nm CMOS	40nm CMOS
Decoder Area (mm^2)	0.637	0.56	0.44	2.27	0.44	1.95	3.17	0.704
Supply (V)	0.9	1.0	-	0.9	1.3	1.8	1.8	0.9
Frequency (MHz)	430	-	1077	1000	721	447	382	500
Throughput (Gb/s)	3.25	1.51	0.764	3.24	0.61	0.30	0.66	7.61
Power (mW)	42.80	-	-	-	128.3	1073 @200MHz	-	422.7
Area Efficiency (Gb/s/mm^2)	5.10	2.71	1.74	1.43	1.39	0.154	0.208	10.81
Energy Efficiency (pJ/b)	13.17	-	-	-	209	7994 @200MHz	-	55.58
**Normalized to 40nm, 0.9V*								
Area Efficiency (Gb/s/mm^2)	5.10	11.63	19.82	0.092	0.48	14.03	18.95	10.81
Energy Efficiency (pJ/b)	13.17	-	-	-	143.1	444.1	-	55.58

* General scaling-theory is used to scale area, frequency (and throughput), and power by $1/s^2$, s, and $1/u^2$ respectively, where s is dimension scale factor and u1.07 is voltage scale factor.

decoding a higher code rate (in this case 3/4) consumes slightly higher power, due to more switching activities to process more free bits. For a given code rate, decoding a shorter code length costs less power, due to the sub-decoders' bypassing of trellis stages.

2.5.2 Comparisons

The ST-SCL decoder test chip is compared with the state-of-the-art polar decoder designs in Table 2.2 including both synthesis results (where no test chip was fabricated and power was not reported) and silicon measurements. Compared to the most recent synthesis results of SCL decoders [121, 122], our ST-SCL chip outperforms by more than 2.15× in throughput and 1.88× in area efficiency than [121] before normalization. After technology normalization to 40nm and 0.9V supply voltage, the area efficiency of [121, 122] surpass our design, which is mainly due to three factors: 1) [121, 122] do not support variable code lengths and rates; 2)[121, 122] use simplified or modified SCL decoding algorithms with performance loss; and 3) [121, 122] are synthesis results only without silicon measurements. Only silicon results capture the layout and wiring congestion overheads that can be significant in high-throughput decoder designs.

Compared to the recent fabricated silicon SCL polar decoders [29, 18] in more advanced 28nm and 16nm technology nodes, our design exceeds the throughput reported in [29, 18]. After technology normalization, our design achieves an order of magnitude better area efficiency (in Gb/s/mm^2) and an order of magnitude better energy efficiency (in pJ/b) than [29, 18] (note that [18] did not report power, and the energy efficiency cannot be estimated).

Compared to the much simpler SC decoder designs [20, 19], the ST-SCL decoder delivers a better error-correction performance as shown in Figure 2.16, and the energy efficiency is more than an order of magnitude better after technology normalization. Compared to the most recent belief propagation (BP) decoder synthesis [21], the ST-SCL decoder achieves more significant coding gain as shown in Figure 2.16, and the energy efficiency is still 4.2× better.

2.6 Summary

We present a fabricated test chip in a 40nm CMOS technology that implements a ST-SCL decoder for polar codes. In this design, a given polar code is split into 4 sub-codes and decoded separately with smaller sub-decoders followed by a reconciliation step in every decoding stage. Taking advantage of the under-utilized PEs in the sub-decoders, 8 frames are interleaved and decoded in parallel to achieve a high throughput and area efficiency. The decoder supports variable code lengths up to 1024b and variable code rates by programming the control LUTs. Per-block clock gating is implemented to further reduce the power consumption and improve the energy efficiency. The 0.64mm^2 test chip is measured to achieve a decoding throughput of 3.25Gb/s at 430MHz and the nominal supply voltage of 0.9V, consuming 13.17pJ/b, and it demonstrates a competitive error-correction performance. Voltage and frequency scaling of the chip to 0.6V and 100MHz further improves the energy efficiency to 7.4pJ/b at a reduced throughput of 760Mb/s. The test chip outperforms the state-of-the-art SCL polar decoder chips in throughput, and its normalized energy efficiency and area efficiency are an order of magnitude better than the latest published work.

CHAPTER III

Efficient Post-Processors for LDPC Codes

In this section, we take the inspiration from simulated annealing to generalize the post-processor design using three methods: quenching, extended heating, and focused heating, each of which targets a different error structure. The resulting post-processor[1] is demonstrated to lower the error floors by two orders of magnitude and can be integrated to a belief-propagation decoder with minimal overhead.

3.1 Min-Sum Decoding Algorithm

Assume a binary phase-shift keying (BPSK) modulation and an additive white Gaussian noise (AWGN) channel. The binary values 0 and 1 are mapped to 1 and -1, respectively. The min-sum decoding can be explained using the factor graph. In the first step of decoding, each VN x_i is initialized with the prior log-likelihood ratio (LLR) defined in (3.1) based on the channel output y_i:

$$L^{pr}(x_i) = \log \frac{\Pr(x_i = 0 \mid y_i)}{\Pr(x_i = 1 \mid y_i)} = \frac{2}{\sigma^2} y_i \tag{3.1}$$

where σ^2 represents the channel noise variance.

After initialization, VNs send the prior LLRs to the CNs along the edges defined by the factor graph. The LLRs are recomputed based on parity checks, as in equa-

[1]Special thanks to Dr. Shuanghong Sun for her help in FPGA design and emulations

tion (3.2), and returned to the VNs. Each VN then updates its decision based on the posterior LLR that is computed as the sum of the prior LLR from the channel and the LLRs received from the CNs, as in equation (3.3). One round of message exchange between VNs and CNs completes one iteration of decoding. To start the next iteration, each VN computes the marginalized LLRs, as in equation (3.4), and passes them to the CN.

$$L(r_{ij}) = \min_{i' \in Row[j] \backslash i} |L(q_{i'j})| \prod_{i' \in Row[j] \backslash i} \operatorname{sgn}\left(L(q_{i'j})\right) \tag{3.2}$$

$$L^{ps}(x_i) = \sum_{j' \in Col[i]} L(r_{ij'}) + L^{pr}(x_i) \tag{3.3}$$

$$L(q_{ij}) = L^{ps}(x_i) - L(r_{ij}) \tag{3.4}$$

The LLRs passed between VNs and CNs are known as the variable-to-check message (VC message, $L(q_{ij})$) and check-to-variable message (CV message, $L(r_{ij})$), where i is the VN index and j is the CN index. In representing the connectivity of the factor graph, $Col[i]$ refers to the set of all the CNs connected to the ith VN and $Row[j]$ refers to the set of all the VNs connected to the jth CN.

The magnitude of $L(r_{ij})$ computed using (3.2) is overestimated and correction terms are introduced to reduce the approximation error. The correction is in the form of either an offset or a normalization factor [123].

A hard decision is made in each iteration based on the posterior LLR, as in (3.5). The iterative decoding is allowed to run until the hard decisions satisfy all the parity checks or when an upper limit on the iteration number is reached.

$$\hat{x}_i = \begin{cases} 0 & \text{if } L^{ps}(x_i) \geq 0 \\ 1 & \text{if } L^{ps}(x_i) < 0 \end{cases} \tag{3.5}$$

In a practical decoder implementation, the VC messages and CV messages are quantized to fixed point. We use the notation $Qp.q$ to indicate a two's-complement fixed-point quantization with p bits for integer and q bits for fraction.

3.2 Error Floor and Trapping Set

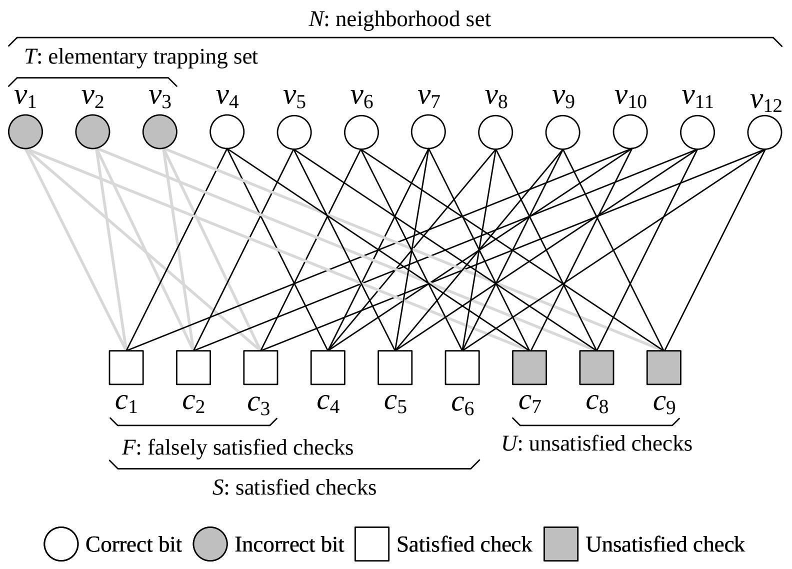
It is known that TS is the fundamental cause of error floor in BP decoding of LDPC codes [35]. We repeat the definition of TS [38] and a special type of TS called elementary TS, or ETS, that is the most dominant in error floors.

Definition 1. Trapping set (TS) and elementary trapping set (ETS)

An (a, b) TS is a configuration of a number of VNs, for which the induced subgraph in G contains $b > 0$ odd-degree CNs with respect to the TS. An (a, b) ETS is a TS for which all CNs in the induced subgraph have either degree 1 or 2 with respect to the TS, and there are exactly b CNs of degree 1 with respect to the TS. The CNs of degree 1 are called degree-1 CNs, and the CNs of degree 2 are called degree-2 CNs.

We will focus the following discussions on ETS as it is the most common type of TS and the most damaging in causing error floors. A factor graph of a small LDPC code is shown in Figure 3.1, which contains a (3,3) ETS $\mathcal{T}$. Each VN in $\mathcal{T}$ is connected to 1 degree-1 CN and 2 degree-2 CNs.

To see how an ETS can cause a decoding error, we use an example of transmitting an all-zero vector of length 12, which is a codeword for the code defined in Figure 3.1. Suppose the received word contains errors in the first three bits. That is, the VNs in $\mathcal{T}$ are initialized to 1 and the remaining VNs are initialized to 0. The received word does not constitute a valid codeword, as the degree-1 CNs labeled $\mathcal{U}$ are not satisfied. Note that among the satisfied CNs labeled $\mathcal{S}$, the degree-2 CNs labeled $\mathcal{F}$ are falsely satisfied, i.e., the ones that are connected to an even number of bits in $\mathcal{T}$. In BP decoding, the CV messages from the degree-1 CNs in $\mathcal{U}$ will attempt to correct



N: neighborhood set
T: elementary trapping set
v_1 v_2 v_3 v_4 v_5 v_6 v_7 v_8 v_9 v_{10} v_{11} v_{12}
c_1 c_2 c_3 c_4 c_5 c_6 c_7 c_8 c_9
F: falsely satisfied checks
U: unsatisfied checks
S: satisfied checks
Correct bit
Incorrect bit
Satisfied check
Unsatisfied check

Figure 3.1: Illustration of a (3,3) ETS.

the wrong bits, but the CV messages from the degree-2 CNs in $\mathcal{F}$ will reinforce the wrong bits. If there is stronger reinforcement than correction of the wrong bits, the decoder is trapped in the non-codeword ETS.

Many LDPC codes contain ETS of lower weight than the minimum distance of the code. As a result, the decoders can be more easily trapped in an ETS at a moderate to high SNR level than converging to a minimum-distance codeword. The presence of ETS results in error floors. Reducing the likelihood of trapping in the local minimum due to ETS is the key to lowering the error floors of LDPC codes.

3.3 Simulated Annealing and Post-processing in BP Decoding

The local minimum problem has been studied extensively in the field of optimization. Notably, the SA algorithm combines gradient descent and random walk to escape local minima [107, 108, 109]. Annealing is a process in metallurgy, where metal is heated to a high temperature and then undergoes controlled cooling to form a low-energy crystalline structure. If metal contains no defects, its energy is at the minimum; otherwise, it will be at a higher energy level. An analogy can be made for decoding: the highest energy occurs in the beginning of decoding when most errors or defects are present. As decoding proceeds, errors are corrected and the energy goes down, just as the cooling process in annealing that removes defects. When the decoding converges to a correct codeword, the energy goes down to the minimum, like metal reaching its defect-free, lowest-energy crystalline state.

The decoder can be trapped in an ETS. The weight of the ETS that induce error floors is often lower than the minimum distance. If an ETS is within the minimum distance away from the correct codeword, a local search algorithm can be applied. SA is such an algorithm that targets local minimum problems.

3.3.1 Neighborhood Identification for Trapping Sets

SA uses heating to perturb the local minimum, making it unstable before breaking away from it. The most efficient way is to heat only the defective points in order to keep the amount of perturbation low and reduce the risk of moving much further away from the closest global minimum. Similarly in LDPC decoding, heating needs to be directed to the error bits in an ETS. The ETS is not known, but the degree-1 CNs are known because they are not satisfied. We can trace the neighboring VNs of the degree-1 CNs, called the neighborhood set $\mathcal{N}$, as labeled in Figure 3.1.

The neighborhood set contains one or more VNs in the ETS, and also VNs outside of the ETS. There is no choice but to apply heating to the entire neighborhood set. As a result, heating will perturb not only the error bits but also the correct bits. In practice, the neighborhood set can be as large as tens or a few hundred bits, therefore heating needs to be carefully adjusted to be effective to resolve the local minimum, but not too much to be pushed to a different codeword.

3.3.2 Heating

Heating is used to perturb the local minimum. In BP decoding, perturbation can be done by reweighting the VC and CV messages [51] or soft bit flipping. In Figure 3.1, the bits in the ETS $\mathcal{T}$ are incorrect. Each VN in $\mathcal{T}$ receives CV messages from 2 degree-2 CNs to reinforce the error and a CV message from 1 degree-1 CN that attempts to correct the error. To perturb this local minimum and possibly escape the local minimum, the CV messages from the satisfied CNs (including degree-2 CNs) are weakened, and the CV message from the degree-1 CN is strengthened. This procedure is called message reweighting. As the magnitude of the messages are changed, noise is injected to the system to achieve a perturbation effect.

Message reweighting applied to the VNs in an ETS helps correct errors, but message reweighting applied to the VNs outside the ETS can possibly introduce more errors. For example, in Figure 3.1, $v_4 \notin \mathcal{T}$, and v_4 is connected to c_1 and c_4 that are both satisfied and c_7 that is degree-1 and unsatisfied. By the reweighting procedure outlined above, the CV messages from c_1 and c_4 to v_4 are weakened, and the CV message from c_7 to v_4 is strengthened, which is likely to cause v_4 to flip to the incorrect value. Therefore, heating needs to be carefully adjusted to avoid perturbing too many correct bits and eventually converge to an undesired global minimum.

3.3.3 Post-Processing Procedure

BP decoding with post-processing follows a two-phase procedure. In the first phase, conventional BP decoding is performed. If BP decoding fails to converge after a set number of iterations, denoted as M, at a moderate to high SNR, the decoding is most likely trapped in a local minimum and it enters the second phase.

In the second phase, post-processing is invoked. Neighborhood set needs to be properly identified for effective heating. The identification can be conveniently done in VN by inspecting the sign of incoming CV messages: if the sign indicates that the parity check is unsatisfied, the VN tags the bit as part of the neighborhood set $\mathcal{N}$. Heating is performed by reweighting the reliability of CV messages, i.e., increasing the reliability of CV messages from the unsatisfied checks to $\mathcal{N}$, or decreasing the reliability of CV messages from the satisfied checks to $\mathcal{N}$, or both. Equation (3.6) describes a way to implement message reweighting that decreases the reliability of the CV messages from the satisfied CNs $\mathcal{S}$ to the VNs in the neighborhood set $\mathcal{N}$ to a low value A_0. The value of A_0 determines the amount of heating, or perturbation injected to the local minimum. Heating can also be done using soft bit flipping to be described in Section 3.4.3.

$$L(r_{ij}) = \prod_{i' \in Row[j] \setminus i} \operatorname{sgn}\left(L(q_{i'j})\right) \cdot \begin{cases} A_0 & \text{if } v_i \in \mathcal{N},\, c_j \in \mathcal{S} \\ \min_{i' \in Row[j] \setminus i} |L(q_{i'j})| & \text{otherwise.} \end{cases} \tag{3.6}$$

After P iterations of heating, N iterations of BP decoding is applied to cool down. The post-processing procedure is summarized in Algorithm 1.

Algorithm 1. Post-Processing Procedure

1. BP decoding: run for M iterations. If there are unsatisfied CNs, continue post-processing.
2. Post-processing:
   (a) Heating: run P iterations of reweighted message passing.
   (b) Cooling: run N iterations of BP decoding.

In Algorithm 1, M is set to ensure that the decoder has been trapped in an ETS, and N is set to ensure that the decoder has enough time to cool down to the global minimum after heating. In this paper, we set $M = N = 20$.

3.3.4 Implementing Post-Processing in Hardware

The primary design goal of post-processing is to lower the error floor with minimal cost of area, power, latency and throughput. An ideal post-processor works likes a "plug-in" feature that can be easily integrated to any standard LDPC decoder.

In a standard min-sum LDPC decoder, a CN is implemented as a comparison tree to find the first and the second minimum. A CN often contains little memory and does not retain states. On the other hand, a VN keeps state and stores prior and posterior information. If post-processing is implemented by reweighting CV messages, a CN needs to be augmented to keep track of all the VNs in the neighborhood set $\mathcal{N}$, which could be costly. Therefore, instead of reweighting CV messages, we devise an alternative by reweighting VC messages. In this alternative approach, a VN is augmented by 1 bit to track whether it belongs to the neighborhood set $\mathcal{N}$. Because the magnitude of VC messages tends to saturate to the maximum value allowed by quantization in a few iterations, the reweighting is implemented in VN by decreasing the magnitude of the VC message from a VN in the neighborhood set to a satisfied CN to a low value A_0. The reweighted (magnitude-reduced) VC message propagates to the satisfied CN, and through the CN's minimum operation becomes reweighted CV message. Equation (3.7) describes post-processing by reweighting VC messages.

$$L(q_{ij}) = \begin{cases} A_0 \cdot sgn(L^{ps}(x_i) - L(r_{ij})) & \text{if } v_i \in \mathcal{N},\ c_j \in \mathcal{S} \\ L^{ps}(x_i) - L(r_{ij}) & \text{otherwise.} \end{cases} \tag{3.7}$$

The choice of A_0 depends on the quantization. Assume VC messages are quantized to $Qp.q$, the possible A_0 values are $\{0, 2^{-q}, 2^{-q+1}, ..., 2^{p-1} - 2^{-q}\}$. The lower the A_0, the more noise is injected to the local minimum. As a result, lower A_0 is more effective in resolving an ETS error, but also highly likely to cause more perturbation to the bits outside the ETS, which may push the decoder to an undesired global minimum. Detailed message reweighting strategies are dependent on the structures of the ETS, which will be elaborated in Section IV.

Post-processing does not require changing the code structure or decoder architecture. Muxes and label bit registers are added for VC message reweighting and neighborhood identification, respectively. A controller monitors the decoding and enables post-processing upon detecting failed CNs after M iterations; therefore post-processing is activated at a rate of approximately the decoding FER and has a negligible impact on the decoding throughput and the average latency.

We demonstrate post-processing implementation based on two commonly used LDPC decoder architectures, the fully-parallel architecture [124] and the row-parallel architecture [125]. A fully-parallel architecture is efficient for short code length and it yields the highest throughput. In a fully-parallel decoder, all CNs, VNs and their interconnections are instantiated in hardware exactly as those in the code's factor graph. Assume a decoder contains Q CNs and K VNs, and the VN degree is d_v. At each VN, a label register is added to indicate whether the VN belongs to the neighborhood set, and a post-processor is added to perform neighborhood labeling and VC message reweighting, as shown in Figure 3.2. The post-processor takes the signs of d_v CV messages (without marginalization) as inputs *sat* to identify whether CNs are satisfied. If post-processing is enabled and at least one incoming CV message indicates that the CN is unsatisfied, the post-processor turns the VN's neighborhood label on with a unary NAND gate. In performing post-processing, the VN's neighborhood label is AND'ed with the *sat* of each CV message to determine whether reweighting is enabled. If reweighting is enabled, a MUX is used to select the reduced magnitude of A_0 for the outgoing VC message $v2c\_pp$.

A row-parallel decoder architecture is the most popular architecture for moderate to long QC LDPC codes, including many that have been used in standards. A row-parallel architecture often employs layered BP decoding. Each iteration is divided into multiple layers of processing. An example decoder architecture is shown in Figure 3.3.

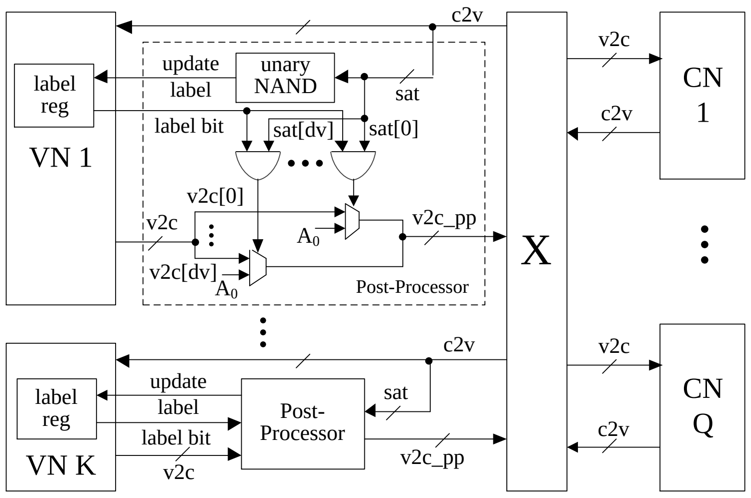


Figure 3.2: Post-processing added to a fully-parallel decoder.

Each layer processing is done by multiple processing elements (PEs), each consisting of a physical VN and memory. The read/write addresses are stored in lookup tables. In the row-parallel architecture, a physical VN is time-multiplexed and it assumes the roles of multiple logical VNs (VNs in the factor graph), one in each layer. A label memory is added to store the neighborhood labels of the logical VNs. A post-processing controller is added to the PE to perform the same labeling and reweighting functions as what the post-processor does in the fully-parallel architecture. The only difference is that the post-processing controller performs the labeling and reweighting serially as the CV messages are received one at a time.

Table 3.1 shows the overhead when post-processing is added to a fully-parallel decoder and a row-parallel decoder for the IEEE 802.11n (648,540) LDPC code and the IEEE 802.11n (1944,1620) LDPC code, respectively. The percentage in the brackets indicate the device utilization. The 100 MHz clock frequency can be kept even after post-processing is added, so the average throughput and latency can be kept constant. Implementing post-processing on the row-parallel decoder uses 4.5% and 8.3% more slice registers and slice LUTs, respectively, compared to the baseline. The

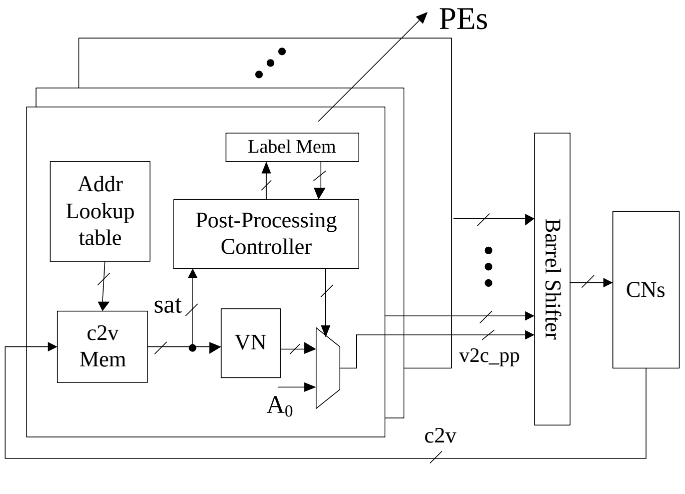


Figure 3.3: Post-processing added to a row-parallel decoder.

cost is even lower when post-processing is added to the fully-parellel decoder.

3.4 Error Structure and Post-Processing Methods

Studying the error floor phenomenon requires fast simulations. FPGA accelerated emulations are particularly useful because software-based simulations often take weeks or months to reach low BER levels. In previous work [126], a library and script based approach was developed to automate the FPGA emulations for LDPC decoders. In this work, we used it to collect errors in the error floor region.

After collecting enough errors in the error floor region, we analyze the ETS structures associated with these errors. The ETS structures are dependent on the code structure. The post-processing method is formulated to be the most effective towards the structures.

3.4.1 Type I ETS and Quenching

The (2048,1723) RS-LDPC code [127] for the IEEE 802.3an standard [34] is a well-studied code for error floor investigation [36]. The H matrix of this regular code

Table 3.1: Evaluation of Post-Processing Implementations of Fully-Parallel and Row-Parallel Decoders (based on Xilinx Virtex-5 XC5VLX155T FPGA)

Design	Fully-par. (648,540) dec.		Row-par. (1944,1620) dec.	
	Baseline	Post-proc added	Baseline	Post-proc added
Slice registers	13,724 (14.24%)	13,901 (14.43%)	4,432 (4.60%)	4,633 (4.81%)
Slice LUTs	39,007 (40.30%)	40,822 (42.17%)	10,066 (10.4%)	10,901 (11.2%)
Occupied slices	10,852 (44.70%)	11,208 (46.17%)	4,782 (19.7%)	4,844 (19.9%)
BRAMs	64 (29.9%)	64 (29.9%)	35 (16.4%)	35 (16.4%)

has a column degree of 6, a row degree of 32, and 64×64 permutation matrices as component submatrices [127]. The code has a girth of at least 6. The code has an error floor below 10^{-10}. It has been shown that the error floor is dominated by (8,8) ETS errors [36].

The (8,8) ETS is illustrated in Figure 3.4 using a simplified representation that only includes VNs in the ETS and degree-1 CNs. The (8,8) ETS consists of 8 VNs, each of which is connected to one degree-1 CN. The degree-2 CNs are shown implicitly in Figure 3.4 as lines connecting pairs of VNs in the ETS. The illustration makes it clear if the bits in the ETS are initialized with incorrect binary values, these VNs will reinforce each other through the degree-2 CNs. As each VN in the ETS neighbors 5 degree-2 CNs and only 1 degree-1 CN, a BP decoder can be easily trapped in this local minimum. The (8,8) ETS is an example of a Type I ETS. **A Type I ETS is one in which each VN is connected to exactly 1 degree-1 CN.**

To resolve a Type I ETS error, Algorithm 1 can be used with $P = 1$, i.e., only one iteration of heating followed by immediate cooling, as proposed by [51]. This post-processing method is named “quenching”. Quenching is effective towards Type I ETS errors, since the neighborhood set traced from the unsatisfied degree-1 CNs

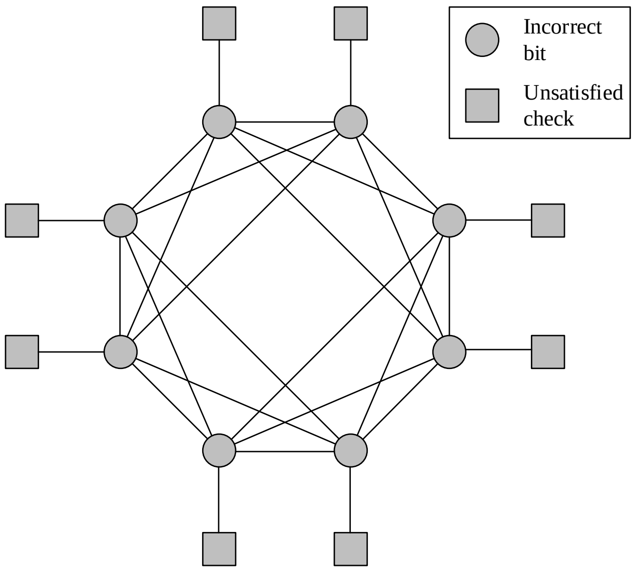


Figure 3.4: An (8,8) ETS of a (2048,1732) RS-LDPC code.

contains the entire ETS. One iteration of heating reaches all VNs in the ETS, and cooling can be applied immediately after to help convergence.

Using the (8,8) ETS illustrated in Figure 3.4 as an example, after the heating step, each VN in the ETS receives 5 weakened CV messages from degree-2 CNs and 1 CV message from a degree-1 CN. The lower the reweighted value A_0 is, the more likely the CV message from the degree-1 CN can overcome the sum of 5 weakened CV messages from the degree-2 CNs.

Previous work showed that over 97% of the ETS errors in the error floor region of the (2048, 1723) RS-LDPC code are corrected using quenching with proper choice of A_0, resulting in nearly two orders of magnitude lower error floor as shown in Figure 3.5 [51]. $A_0 = 1$ is used in this experiment.

3.4.2 Type II ETS and Extended Heating

To extend from previously proposed quenching post-processing [51], we choose a (5,47)-regular rate-0.89 (2209, 1978) array LDPC codes [111] for investigation of

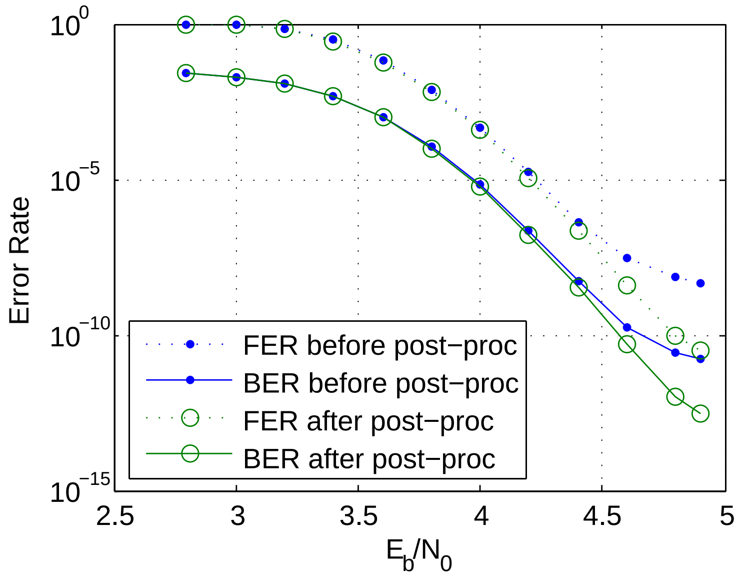


Figure 3.5: Error rate of the (2048, 1723) RS-LDPC code before and after post-processing using quenching.

other types of ETS structures. The H matrix of this code can be partitioned into 5 row groups and 47 columns groups of 47×47 permutation matrices.

We collected 274 errors through FPGA emulation of a Q4.0 (2209, 1978) array LDPC decoder in the error floor region (E_b/N_0 = 5.6 dB, 5.8 dB, and 6.0 dB). 243 out of the total 274 errors are ETS errors, among which there is only 1 Type I ETS error. We then applied quenching to post-process these errors and the results are summarized in Table 3.2. A resolving rate of only 73% indicates that quenching alone is not sufficient to lower the error floor of array code.

From Table 3.2, one can observe that, unlike the RS-LDPC code discussed above, the error floor of array LDPC code is not dominated by only one kind of ETS error, but attributed to several kinds of ETS errors, including (8,6), (9,5), (8,8), and (10,4) ETS errors [36]. An (8,6) ETS is illustrated in Figure 3.6. It is an example of type II ETS. **A Type II ETS is one in which each VN is connected to no more than 1 degree-1 CN, and at least 1 VN is not connected to any degree-1 CN. The VNs that have no neighboring degree-1 CN are called inner bits,**

Table 3.2: Error Profile of the (2209, 1978) Array LDPC Code and Effectiveness of Quenching

ETS	Error count	Resolved by quenching
(6,8)	6	3 (50%)
(7,9)	5	2 (40%)
(8,6)	124	105 (85%)
(8,8)	20	13 (65%)
(9,5)	37	33 (89%)
(10,4)	12	8 (67%)
(10,6)	9	5 (56%)
(10,8)	7	7 (100%)
other ETS	23	12 (52%)
non-ETS	31	11 (58%)
Total	274	199 (73%)

and the VNs that have only 1 neighboring degree-1 CN are called outer bits.

In the type II (8,6) ETS illustrated in Figure 3.6, the inner bits, v_7 and v_8, are connected to all satisfied checks, through which they reinforce the outer bits in the ETS. The inner bits are more "deeply" trapped than the outer bits since they are not connected to any unsatisfied checks. One iteration of heating helps correct the outer bits, but it does not propagate to the inner bits. The immediate cooling after only one iteration of heating hampers the full recovery. In annealing language, the temperature of the outer bits rise after the heating step, but the inner bits are still cold. Therefore, we propose a second post-processing method called extended heating by setting $P > 1$ in Algorithm 1.

Compared to quenching, extended heating prolongs heating to P iterations, where $P > 1$, before cooling. The idea is to heat all the bits in the ETS, including both outer and inner bits, to raise the temperature evenly. The neighborhood set is updated after

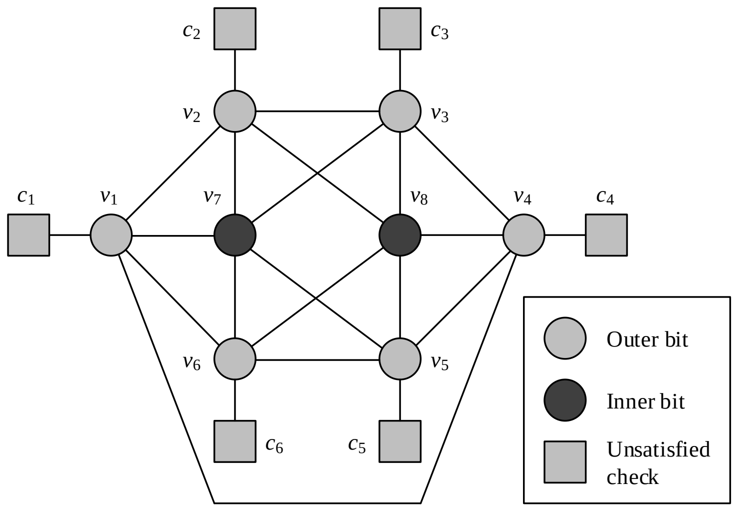


Figure 3.6: Illustration of a type II (8,6) ETS.

each iteration of heating, allowing the set to be enlarged to include inner bits so that heating can be propagated to them. Prolonged heating allows the bits in a ETS to accumulate enough energy to avoid falling back to the same local minimum.

Among the 236 ETS errors from FPGA emulations, there are only 1 type I ETS error and 184 type II ETS errors that are listed in Table 3.3. Quenching with $P = 1$ and $A_0 = 1$ resolves the type I ETS error but only 84% of the type II ETS errors. In comparison, extended heating with $P = 10$ and $A_0 = 1$ resolves 97% of the type II ETS errors, which demonstrates its effectiveness. When the number of inner bits is large, e.g., in (10,4), (10,6), and (11,5) ETS errors, the success rate of quenching is particularly low, but extended heating works well consistently.

3.4.3 Type III ETS and Focused Heating

Besides type I and type II ETS errors, there are 58 additional ETS errors collected for the $Q4.0$ (2209, 1978) array LDPC decoder. The (6,8) ETS shown in Figure 3.7 is an example of them. The (6,8) ETS is neither type I, nor type II, because two of the bits, v_1 and v_4 are each connected to two unsatisfied checks. It is an example of type

Table 3.3: Type II ETS Error Profile of the (2209, 1978) Array LDPC Code and Effectiveness of Quenching and Extended Heating

ETS	Inner bits	Error count	Resolved by quenching	Resolved by extended heating
(8,6)	2	124	105 (85%)	121 (98%)
(9,5)	4	37	33 (89%)	36 (97%)
(10,4)	6	12	8 (67%)	12 (100%)
(10,6)	4	5	3 (60%)	4 (80%)
Other	-	6	6 (100%)	6 (100%)
Total	-	184	155 (84%)	179 (97%)

III ETS. **A Type III ETS is one in which 1 or more VNs are each connected to more than 1 degree-1 CNs. The VNs that have only 1 neighboring degree-1 CN are called singular bits, and the VNs that have more than 1 neighboring degree-1 CNs are called plural bits.**

A type III ETS typically has more unsatisfied checks than the size of the ETS. Since the neighborhood set is traced from the unsatisfied checks, the neighborhood set is relatively larger. A large neighborhood set means more bits, mostly correct bits, are perturbed in heating. Table 3.4 above lists the dominant type III ETS errors that have been collected in the error floor region. Quenching with $P = 1$ and $A_0 = 1$ resolves only 55% of the type III ETS errors. Extended heating with $P = 10$ and $A_0 = 1$ resolves 86%. Overheating is the problem in both cases that cause the two methods to be not as effective.

In a type III ETS, a plural bit, e.g., v_1 or v_4 in Figure 3.7, is connected to more than one unsatisfied checks. Therefore, a plural bit candidate can be identified as one that is connected to more than one unsatisfied checks. After the plural bit candidates are identified, they can be corrected by bit flipping, allowing the unsatisfied checks, e.g., c_1, c_2, c_5 and c_6 in Figure 3.7, to be turned to satisfied checks. After bit flipping, heating can be applied to a smaller and focused neighborhood set to be more effective.

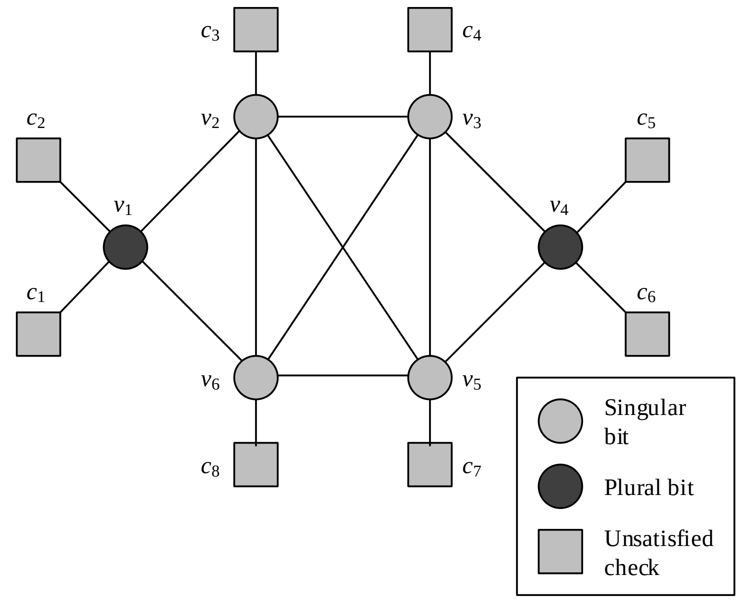


Figure 3.7: Illustration of a type III (6,8) ETS.

Table 3.4: Type III ETS Error Profile of the (2209, 1978) Array LDPC Code and Effectiveness of Extended Heating and Focused Heating

ETS	Plural bits	Error count	Resolved by		
			quenching	extended heating	extended and focused heating
(6,8)	2	5	3 (60%)	4 (80%)	5 (100%)
(8,8)	2	11	7 (64%)	10 (91%)	11 (100%)
(8,8)	1	4	4 (100%)	4 (100%)	4 (100%)
(8,8)	4	4	1 (25%)	4 (100%)	4 (100%)
(10,8)	1	4	4 (100%)	4 (100%)	4 (100%)
other	-	30	13 (43%)	24 (80%)	28 (93%)
Total	-	58	32 (55%)	50 (86%)	56 (97%)

In this case, bit flipping acts as another form of perturbation. To control the noise injection, we use soft bit flipping, i.e., reduce the reliability of the soft decision to a low value B_0 to weaken the plural bits without a significant impact on the correct bits outside the ETS. We call this method focused heating as described in Algorithm 2.

Algorithm 2. Focused Heating

1. BP decoding: run for M iterations. If there are unsatisfied checks, continue post-processing.

2. Post-processing:

   (a) Constraining: run L iterations of soft bit flipping.

   (b) Cooling: run N iterations of BP.

Focused heating uses L iterations of soft bit flipping to selectively weaken the plural bits and shrink the neighborhood set, so that extended heating can be applied to a focused neighborhood set. In practice, extended heating and focused heating need to be combined because a type III ETS error can include inner bits that need to be resolved by extended heating. Assume a gap of G iterations that separates extended and focused heating. Extended and focused heating with $P = 10$ and $A_0 = 1$ (for extended heating), $L = 5$ and $B_0 = 3$ (for focused heating), and $G = 10$ resolves 97% of the type III ETS errors, as shown in Table 3.4, more effective than quenching or extended heating.

In total, extended and focused heating can be applied to resolve 99% of the 236 ETS errors collected in the error floor region of the $Q4.0$ (2209, 1978) array LDPC decoder. Note that although the method is designed for ETS errors, extended and focused heating can resolve 82% of the 38 non-ETS errors. Table 3.5 lists the summary of the results. The BER in the error floor region is lowered by more than 2 orders of magnitude at $E_b/N_0 = 6.0$dB, as shown in Figure 3.8. This experiment is done with

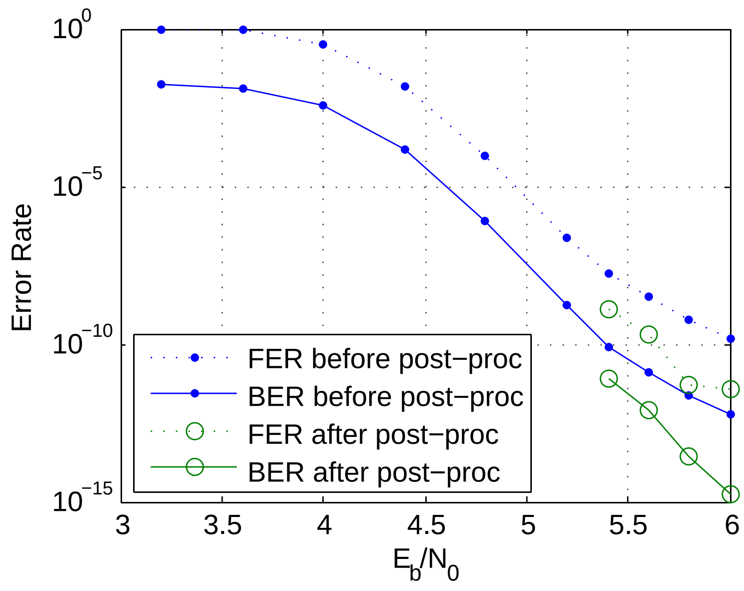


Figure 3.8: Error rate of the (2209, 1978) array LDPC code before and after post-processing using extended and focused heating.

$P = 10$ and $A_0 = 1$ (for extended heating), $L = 5$ and $B_0 = 3$ (for focused heating), and $G = 10$.

3.5 Application of Post-Processing Methods – Case Study on an IEEE 802.11n LDPC Code

The focused and extended heating methods are developed based on the (2209, 1978) array LDPC code, but the methods are generally applicable. We demonstrate these methods on an arbitrarily selected rate-0.83 (1944, 1620) LDPC code for the IEEE 802.11n standard [32]. The H matrix of the (1944, 1620) LDPC code is made up of a 4×24 array of 81×81 identity matrices, cyclic shifted identity matrices, or zero matrices. The H matrix is described in Figure 3.9 [32], where a "0" indicates an 81×81 identity matrix, a number x, $x > 0$, indicates an 81×81 matrix obtained by right cyclic shifting of the identity matrix by x, and a "-" indicates an 81×81 zero matrix.

Table 3.5: Summary of ETS and Non-ETS Errors of the (2209,1978) Array LDPC Decoder in the Error Floor Region and the Effectiveness of Post-Processing by Combined Extended and Focused Heating

E_b/N_0	Error type	Number of errors	Resolved by extended and focused heating
5.6 dB	Type I ETS	1	1 (100%)
	Type II ETS	74	73 (99%)
	Type III ETS	20	19 (95%)
	Non-ETS	27	21 (78%)
5.8 dB	Type I ETS	0	-
	Type II ETS	82	82 (100%)
	Type III ETS	21	20 (95%)
	Non-ETS	9	9 (100%)
6.0 dB	Type I ETS	0	-
	Type II ETS	33	33 (100%)
	Type III ETS	5	5 (100%)
	Non-ETS	2	1 (50%)
Total	ETS errors	236	233 (99%)
	Non-ETS errors	38	31 (82%)

13	48	80	66	4	74	7	30	76	52	37	60	-	49	73	31	74	73	23	-	1	0	-	-
69	63	74	56	64	77	57	65	6	16	51	-	64	-	68	9	48	62	54	27	-	0	0	-
51	15	0	80	24	25	42	54	44	71	71	9	67	35	-	58	-	29	-	53	0	-	0	0
16	29	36	41	44	56	59	37	50	24	-	65	4	65	52	-	4	-	73	52	1	-	-	0

Figure 3.9: Parity check matrix of the (1944, 1620) LDPC code for IEEE 802.11n standard.

The (1944, 1620) LDPC code is structured but not regular. Note that the identity matrices are laid out in a staircase on the right hand side to allow for an efficient encoder design. This design however leads to a low minimum column degree of 2. which dictates the majority of the error patterns.

We implemented a $Q5.0$ decoder for this LDPC code on FPGA and collected 1830 errors in the error floor region ($E_b/N_0 = 5.0$ dB, 5.4 dB, 5.6 dB, and 5.8 dB). More than 99% of the errors are ETS errors, and the remaining are non-ETS errors. Type I, type II and type III account for 7.3%, 70% and 22.5% of the ETS errors, respectively. The dominant type II ETS errors, (3,2), (4,1), (5,1), and (5,2), account for 76% of the type II ETS errors, and their structures are illustrated in Figure 3.10. Since they all contain inner bits, extended heating can be applied. The dominant type III ETS errors, (1,2), (1,3), (2,3) and (2,4), account for 82% of the type III ETS errors. Their structures are illustrated in Figure 3.11. Extended and focused heating are applicable to these errors.

Extended and focused heating is used to post-process the errors collected in the $Q5.0$ (1944, 1620) IEEE 802.11n LDPC decoder. Table 3.6 shows that the overall success rate is 95%. The error floor is reduced by one to two orders of magnitude after post-processing, as shown in Figure 3.12. The results are obtained with $P = 10$ and $A_0 = 1$ (for extended heating), $L = 5$ and $B_0 = 1$ (for focused heating), and $G = 10$. In comparison, quenching alone [51] resolves 23% of the errors and the bi-mode

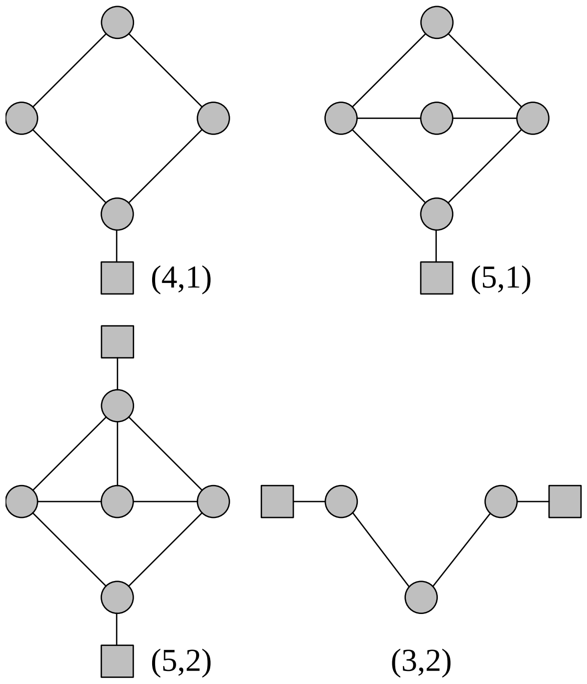


Figure 3.10: Dominant type II ETS structures in the (1944, 1620) LDPC code for the IEEE 802.11n standard.

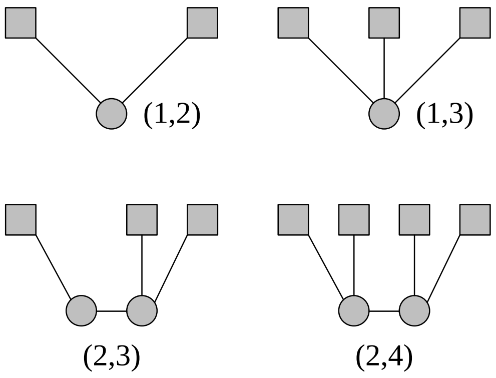


Figure 3.11: Dominant type III ETS structures in the (1944, 1620) LDPC code for IEEE 802.11n standard.

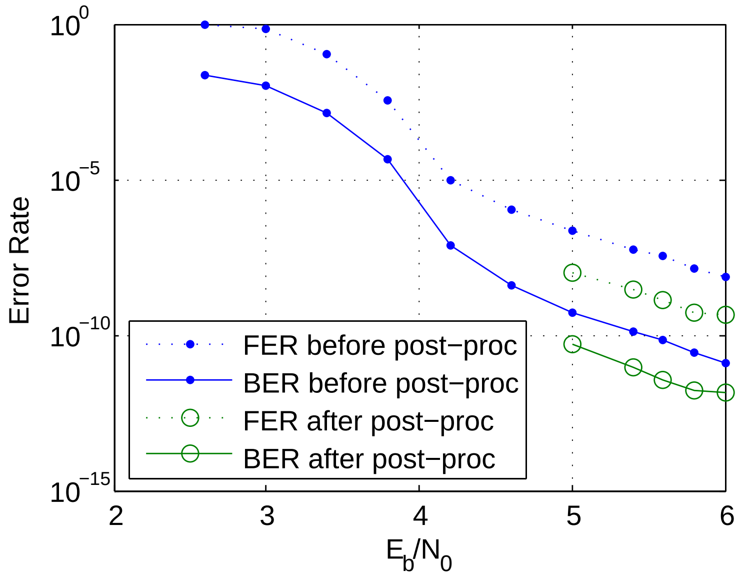


Figure 3.12: Error rate of the (1944, 1620) IEEE 802.11n LDPC code before and after post-processing using extended and focused heating.

syndrome erasure decoding [50] resolves 59% of the errors based on our simulations.

The device utilization of a row-parallel 802.11n LDPC decoder is listed in Table 3.7. The addition of extended and focused heating introduces less than 10% overhead. The results of this work are compared in Table 3.8 with two prior designs [48, 63] that included hardware design and evaluation. This work demonstrates a lower datapath overhead than [63] and requires significantly less memory than [48]. As a deterministic method, this work features a lower latency than [48] because it is integrated as part of BP decoding, while [48] requires trial and error.

3.6 Summary

Error floors of structured LDPC codes are caused by local minima due to non-codeword ETS and ETS-like errors. Inspired by simulated annealing, we design post-processing methods to perturb the local minimum state, followed by cooling to help decoding converge to the global minimum.

We use three well-known LDPC code examples, a (2048, 1723) RS-LDPC code,

Table 3.6: Summary of ETS and Non-ETS Errors of the (1944,1620) IEEE 802.11n LDPC Decoder in the Error Floor Region and the Effectiveness of Post-Processing by Combined Extended and Focused Heating

E_b/N_0	Error type	Number of errors	Resolved by extended and focused heating
5.0 dB	type I ETS	34	31 (91%)
	type II ETS	351	338 (96%)
	type III ETS	91	86 (95%)
	non-ETS	4	4 (100%)
5.4 dB	type I ETS	26	23 (88%)
	type II ETS	324	311 (96%)
	type III ETS	100	94 (94%)
	non-ETS	1	1 (100%)
5.6 dB	type I ETS	38	37 (97%)
	type II ETS	319	306 (96%)
	type III ETS	102	96 (94%)
	non-ETS	2	1 (50%)
5.8 dB	type I ETS	35	32 (91%)
	type II ETS	284	267 (94%)
	type III ETS	119	113 (95%)
	non-ETS	0	0 (N/A)
Total	type I ETS	133	123 (92%)
	type II ETS	1278	1222 (96%)
	type III ETS	412	389 (94%)
	non-ETS	7	6 (86%)

Table 3.7: Device Utilization of 4-Row-Parallel IEEE 802.11n (1944, 1620) decoders (based on Xilinx Virtex-5 XC5VLX155T FPGA)

Design	Baseline	Quenching	Extended + focused heating
Slice registers	4,432 (4.60%)	4,611 (4.79%)	4,633 (4.81%)
Slice LUTs	10,066 (10.4%)	10,732 (11.1%)	10,901 (11.2%)
Occupied slices	4,782 (19.7%)	4,834 (19.9%)	4,844 (19.9%)
BRAMs	35 (16.4%)	35 (16.4%)	35 (16.4%)

Table 3.8: Comparison of Low-Floor LDPC Decoder Implementations

	This work	[20]	[28]
Implementation	FPGA	Synthesis	Silicon
Method	Generalized post-processing	Backtracking	Quenching
Code	Any	Any	(2048,1723) RS-LDPC
Datapath overhead	8.3%	7%	13.7%
Memory overhead	4.5%	46%	N/A

Table 3.9: Summary of Post-Processing Parameters

Code	Method	P	A_0	L	B_0	G
(2048,1723) RS-LDPC	Quenching	1	1	-	-	-
(2209, 1978) array LDPC	Extended + focused heating	10	1	5	3	10
(1944, 1620) 802.11n LDPC	Extended + focused heating	10	1	5	1	10

a (2209, 1978) array LDPC code, and a (1944, 1620) 802.11n LDPC code for the IEEE 802.11n standard to demonstrate three types of ETS structures: type I with one-to-one correspondence between each unsatsified check and ETS bit, type II with inner bits, i.e., ETS bits that are not connected to any unsatisfied check, and type III with plural bits, i.e., ETS bits that are connected to more than one unsatisfied checks.

Three post-processing methods are proposed to resolve ETS errors. The quenching algorithm uses one heating step followed by immediate cooling to resolve type I ETS errors. The extended heating algorithm prolongs heating to multiple steps to allow the inner bits to accumulate enough energy to resolve type II ETS errors. The focused heating algorithm applies soft bit flipping to the plural bits in order to correct them and narrow down the neighborhood set for more effective heating. The post-processing parameters used in this work are summarized in Table 3.9.

The post-processing methods can be easily integrated as part of BP decoding, adding minimal overhead to the hardware implementation. As these methods are conditionally triggered when the decoder fails to converge at a very low BER level, the impact on decoding throughput and energy consumption is negligible.

The methods are demonstrated by post-processing the errors collected in the error floor region of the three LDPC code examples. The success rate is over 95% for ETS errors and over 80% for non-ETS errors for the IEEE 802.11n (1944,1620) LDPC code.

CHAPTER IV

High-Throughput Architecture and Implementation for NB-LDPC Codes

In this section, we present high-throughput decoder design through algorithm-architecture co-optimization for a class of regular-(2,d_c) NB-LDPC codes. A 1.22 Gb/s fully parallel decoder chip[1] is implemented for a GF(64) (160,80) regular-(2,4) NB-LDPC code in 65 nm CMOS with fine-grained dynamic clock gating.

4.1 Extended Min-Sum Decoding Algorithm

An NB-LDPC code is decoded by an iterative message passing on a factor graph. A number of efficient algorithms have been proposed with different error-correcting performance and implementation complexity [53, 37, 2, 70, 125, 64]. The EMS algorithm [70] offers a good tradeoff: it achieves a performance close to the original BP algorithm and its complexity is relatively low. The truncated EMS algorithm claims an even lower complexity and demonstrates great potential for practical adoption [71]. We briefly introduce the five steps of the truncated EMS algorithm [71] assuming $n_m < q$: (1) variable nodes are initialized with the prior log-likelihood ratios (LLR): one LLR is associated with one of q possible symbols, the largest n_m of which

[1]Special thanks to Dr. Youn-Sung Park who leaded the chip design project. I was responsible for the front-end RTL design.

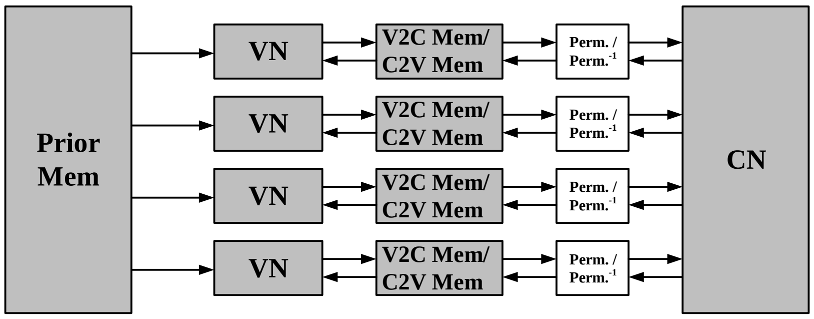


Figure 4.1: Layered EMS decoding of a (2,4)-regular NB-LDPC code.

along with their GF indices are stored in vector L and β_L respectively in descending order; (2) variable-to-check (v-c) messages are permuted based on the H matrix and sent to the check nodes. Note that in the first iteration, the priors are used as the v-c messages; (3) for each adjacent variable node v_j, check node c_i computes the check-to-variable (c-v) mes-sage $V_{ij}[k]$, $k \in \{0, \ldots, n_m–1\}$, that the parity-check equation is satisfied if $v_j = \beta_{V_{ij}}[k]$. The computation is done using a forward-backward recursion. Only the n_m highest probabilities are computed and stored; (4) the c-v messages are inverse permuted before being sent to the variable nodes; (5) each variable node v_j is updated with messages from the adjacent check nodes. A v-c message $U_{ji}[k]$, $k \in \{0, \ldots, n_m–1\}$, is computed for each adjacent check node c_i, based on the prior L and all adjacent c-v messages except from check node c_i. The procedure repeats itself from step (2).

A high-level block diagram of a row-layered NB-LDPC decoding is shown in Figure 4.1 for the selected (2, 4)-regular code. The architecture consists of 4 VNs and 1 CN. The processing schedule for this architecture is shown in Figure 4.2. CN reads v-c messages and performs forward-backward recursion on the 4-stage trellis in three steps: (1) 1 forward step, (2) 1 backward step, and (3) 4 merging steps. Forward and backward steps can be overlapped, and 4 merging steps can be parallelized. Each step in the recursion is done by an elementary CN (ECN). ECN carries out the max-log computation and sorts the results. The latency of each ECN is at least $2n_m$ clock

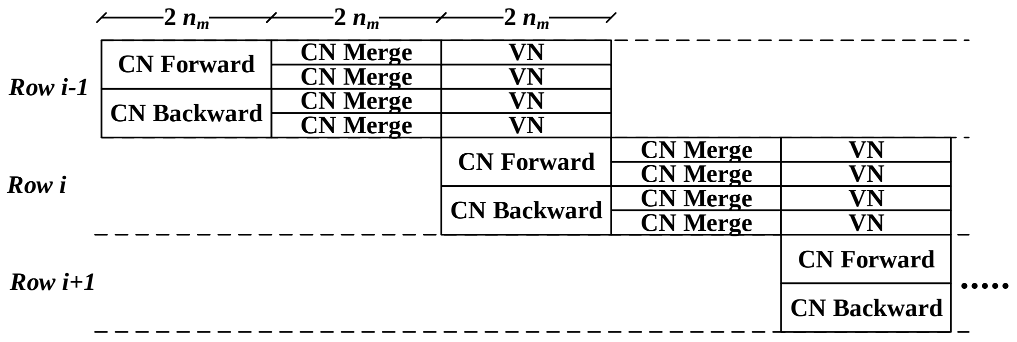


Figure 4.2: Scheduling of the (2,4)-regular layered EMS decoding.

cycles. The bubble check algorithm [128] reduces the sorter length from n_m to approximately $\sqrt{n_m}$. VN starts after CN is complete and c-v messages have been written to memory. A feature of this code that limits $d_v = 2$ simplifies the VN operation: a v-c message can be calculated by the vector addition of the c-v message and the prior. However, the addition requires matching of GF indices using a content-addressable memory. Sorting the results is also expensive, with a latency of at least $2n_m$ cycles. Altogether the throughput is one row per $4n_m$ cycles. With 80 rows in the selected code and $n_m = 16$, one decoding iteration takes on the order of 5,120 cycles! The decoding latency and throughput can be further degraded due to the structure of the (2, dc)-regular codes. The non-structured H matrix introduces data dependencies that require lengthy pipeline stalls to be inserted. The challenges with the decoder architecture call for new processing node designs and efficient memory access schemes.

4.2 Decoder Architecture Design For Low Latency and High Throughput

We introduce improvements in both CN and VN operations in order to enable a pipeline with lower latency and hence higher throughput.

4.2.1 Low-Latency ECN and Improved Pipeline Schedule

ECN is the elementary building block of CN, and it is used for forward, backward, or merging operation. ECN takes two LLR vectors U and I along with their GF indices β_U and β_I to produce a new V and its β_V using the max-log algorithm [4]: V[i] = $\max_{S(\beta_V[i])}$ (U[j] + I[p]), $i \in \{0, \ldots, n_m–1\}$ and $S(\beta_V[i])$ is the set of all combinations of β_V[i], β_U[j], and β_I[p] that satisfy β_V[i] + β_U[j] + β_I[p] = 0 over GF(q).

The proposed low-latency ECN is based on the bubble check algorithm [128]. We improve upon the original algorithm by prefetching and relaxing redundancy control, which together shorten the latency of an ECN operation from at least $2n_m$ to n_m + L$_{S-ECN}$ + 2 clock cycles, where L$_{S-ECN}$ is the sorter length used in ECN and L$_{S-ECN}$ ¡ n_m [128]. Simulation shows that relaxing redundancy control introduces functional performance loss at high error rate, but the loss becomes negligible at low error rate that is of more practical interest. The proposed ECN is described below, assuming both U and I are sorted in descending order:

(1) insert U[j] + I[0], where j = 0, ..., L$_{S-ECN}$ – 1, sequentially to the sorter in descending order. Note that along with the sum U[j] + I[p], the two indices j and p are also recorded.

(2) Set j_{curr} = 0 and p_{curr} = 1.

(3) Fetch U[j_{curr}] and $I[p_{curr}]$. Compute S_{in} = $U[j_{curr}]$ + $I[p_{curr}]$. Insert S_{in}. if the sorter is full, output the maximum value in the sorter S_{max} while inserting S_{in}.

(4) Find the next pair of indices j_{next} and p_{next}. Define a direction flag R, with R = 1 initially. The second largest value in the sorter is denoted $S_{max,2}$, which is the sum of $U[j_{max,2}]$ and $I[p_{max,2}]$.

(a) If ($S_{max,2} < S_{in}$), then j = j_{curr} and p = p_{curr}, else j = $j_{max,2}$ and p = $p_{max,2}$

(b) if (j = 0), then R = 1, else if (p = 0 and j > L$_{S-ECN}$ – 1), R = 0.

(c) Set j_{next} = j + !R and p_{next} = p + R (! denotes inversion).

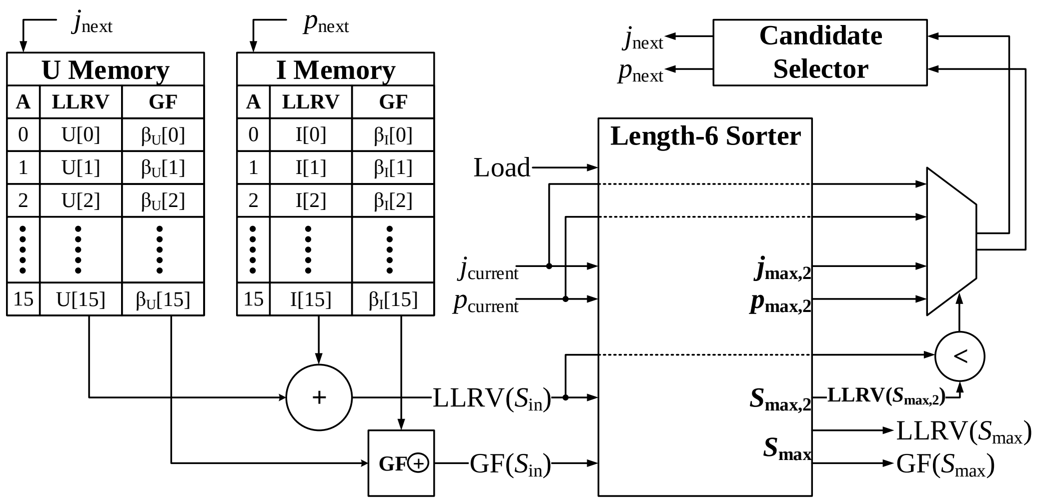


Figure 4.3: ECN architecture.

(5) Set $j_{curr} = j_{next}$ and $p_{curr} = p_{next}$.

(6) Go back to (3) until all n_m values have been output from the sorter.

The main difference between the proposed algorithm and the bubble check algorithm is the use of S_{max},2 and S_{in} to decide one cycle ahead the next inputs. Prefetching shortens the latency because both sorting and reading can execute con-currently without stalls. The algorithm permits redundancy in the output and therefore it stops after n_m outputs are done. The high-level architecture of the ECN is shown in Figure 4.3. The length of the sorter L$_{S-ECN}$ is determined by n_m. For the case of n_m = 16, the maximum number of pending candidates will be 6 and therefore L$_{S-ECN}$ = 6 is the best choice [128]. The latency of the first ECN output is L$_{S-ECN}$ + 2 = 8 cycles, which enables early starts on ECN or VN operations. In total, one complete ECN operation takes 24 cycles for 16 outputs.

The proposed ECN allows the pipeline latency of CN to be shortened for a higher throughput. Figure 4.4 shows the improved pipeline schedule, where the 4 parallel merge steps start when the first output from the forward and backward steps are ready. With a short 8-cycle latency, the merge steps are effectively overlapped with the forward and backward steps. Furthermore, using two sets of memories in CN

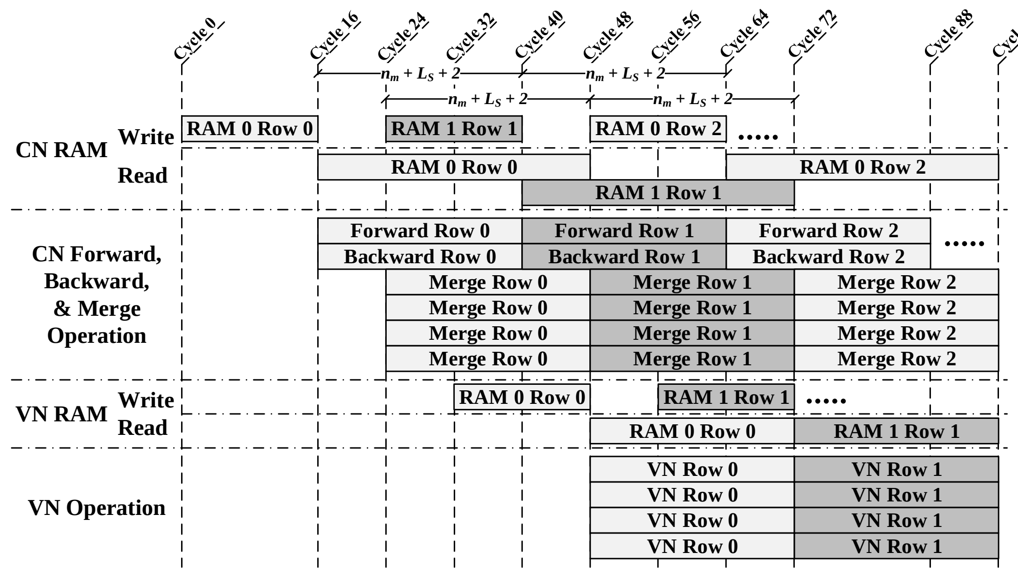


Figure 4.4: Overlapped pipeline schedule.

RAM for alternating rows, the forward and backward step of the next row can start right after the current row is complete, hence making full utilization of the hardware. The improved pipeline schedule enables a high CN throughput of 1 row processing in every 24 cycles.

4.2.2 Low-Latency VN

In order to fully take advantage of the latency reduction of the CN and the improved pipeline schedule, the VN operation also needs to be improved to avoid becoming the bottle-neck. The four outputs produced by the CN need to be processed in parallel by VNs within 24 cycles, or else it will stall the pipeline.

A VN takes two LLR vectors V (c-v message) and L (prior likelihoods) along with their GF indices β_V and β_L to produce a new U (v-c message) and its GF index β_U [72]. Since $n_m < q$, the VN operation needs to scan both the V vector and L vector for matching entries with a latency of at least $2n_m$.

We propose a simplified VN algorithm to achieve a low-latency by skimming the prior messages, i.e., allocate only L_{S-VN} cycles to scan the L vector, where L_{S-VN}

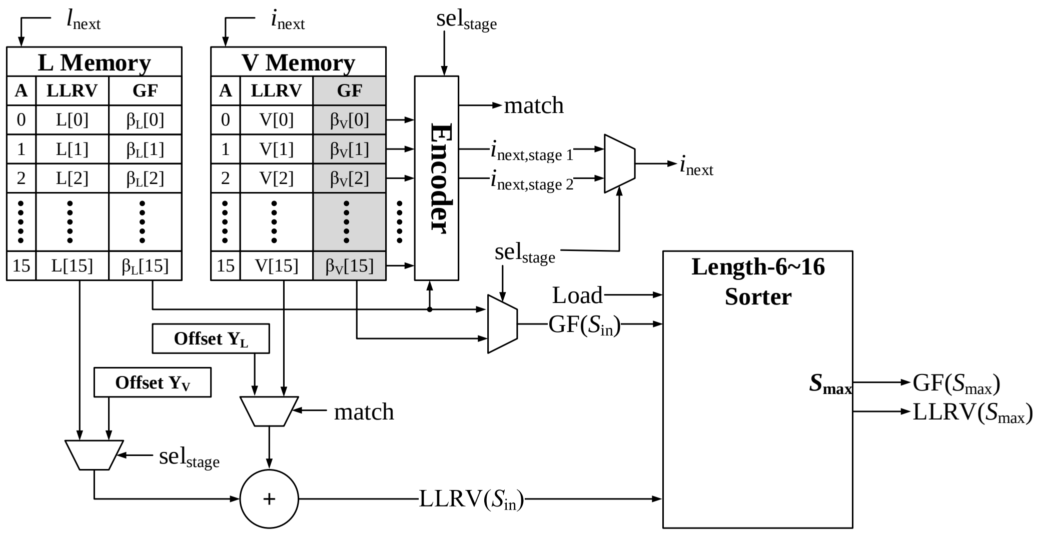


Figure 4.5: EVN architecture.

is the sorter length used in VN and L$_{S-VN}$ ¡ n_m. The algorithm is divided into two stag-es, assuming both L and V are sorted in descending order:

(1) Scan the top L$_{S-VN}$ entries in β_L sequentially: search $\beta_L[l]$,$l \in \{0, \ldots, L_{S-VN}$–1$\}$, in β_V for matching entries. Compute $S_{in} = L[l] + V[i]$, if β_L[l] $= \beta_V$[i]; or $S_{in} = L[l]$ + Y_V, if no matching entry is found, where Y_V is a compensation constant. Insert S_{in} to the sorter.

(2) Scan the n_m entries in V sequentially. If an entry $V[i]$, $i \in \{0, \ldots, n_m$–1$\}$, has not been matched in (1), compute $_{in} = V[i] + Y_L$, where Y_L is a compensation constant. The maxi-mum entry in the sorter is output every time a new $_{in}$ is inserted to the sorter.

The architecture of the proposed VN is shown in Figure 4.5. A content-addressable memory is used to search matching entries. If we choose L$_{S-VN} = 6$, the VN operation takes 24 cycles to produce 16 outputs for the perfect interleaving with CN as shown in Figure 4.4. Note that since L$_{S-VN} < n_m$, some low order entries in vector L will be missed, which degrades the functional performance by 0.65 dB at FER of 10^{-5} as shown in Figure 4.6. To compensate the performance loss, we can increase L$_{S-VN}$ and use two sets of VNs to accommodate a higher processing latency without stalling

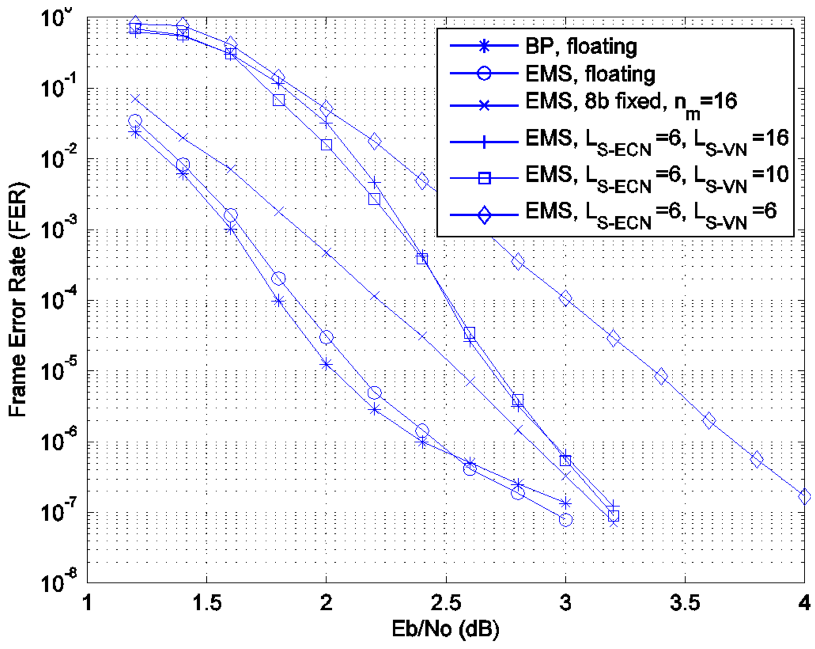


Figure 4.6: Performances of a (2,4)-regular, (960,480) NB-LDPC over GF(64).

the pipeline, i.e., while the first set is working on one row, the second set kicks in to process the second row immediately when the inputs are available. Alternatively, a short pipeline stall can be inserted. Simulation results in Figure 4.6 show that a small increase to $\mathrm{L}_{S-VN} = 10$ almost eliminates the performance loss.

4.2.3 Memory Conflict Resolution

Unlike quasi-cyclic codes, the (2, d_c)-regular NB-LDPC code lacks a systematic structure. When multiple VNs operate in parallel, memory access conflicts arise due to both intra- and inter-iteration data dependencies, causing pipeline stalls. In particular, CN accesses 4 v-c messages at the same time, requiring them to be stored in 4 sets of memory. CN then passes the c-v messages to 4 VNs that will output to a centralized memory to be read by later CN processing. How-ever, as illustrated in Figure 4.7, writing to the c-v memory can be problematic because conflicts can occur when two or more VNs write to the same memory set. The worst-case conflict is when all 4 VNs are attempting to write to one set. Our solution is to subdivide

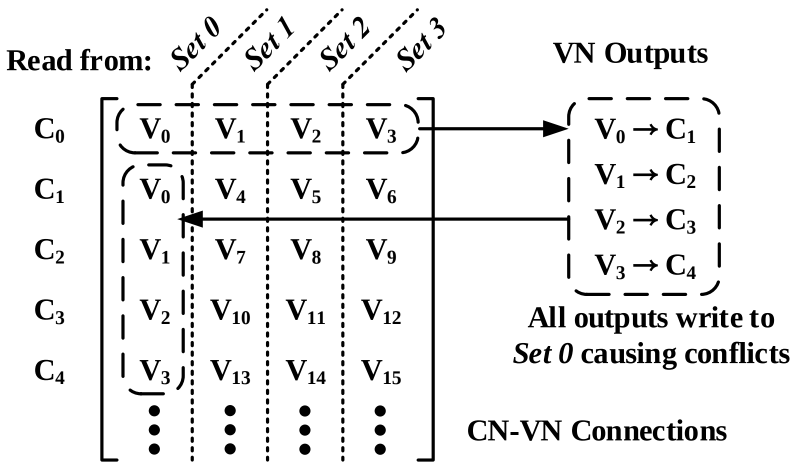


Figure 4.7: Example of worst case intra-iteration memory conflicts.

each set into smaller subsets determined by the code structure. The conflicting v-c messages would be written to different subsets within the same set. Note that this solution would require a separate look-up table in each set to select the correct subset for each read and write access, but the size of the v-c memory will remain the same.

Read-after-write (RAW) hazards can also occur due to data dependencies across consecutive iterations. For instance, if the VN that writes back to the first row shown in Figure 4.7 does not finish in time (e.g., $v_0 \rightarrow c_0$), the next iteration would read the old values. We resolve this inter-iteration data dependency by shuffling the rows in the H matrix such that the last row does not produce any output that is needed by the first row. The conflict-free scheme ensures a stall-free pipeline for a high throughput.

4.3 Low-Power Circuits Design by Fine-Grained Dynamic Clock Gating

To estimate the power consumption, a fully parallel nonbinary LDPC decoder has been synthesized and place and routed in a 65 nm CMOS process. Figure 4.8 shows the power breakdown of the decoder. The switching power of sequential circuits is the dominant portion, claiming 65% of the total power. The leakage power and the

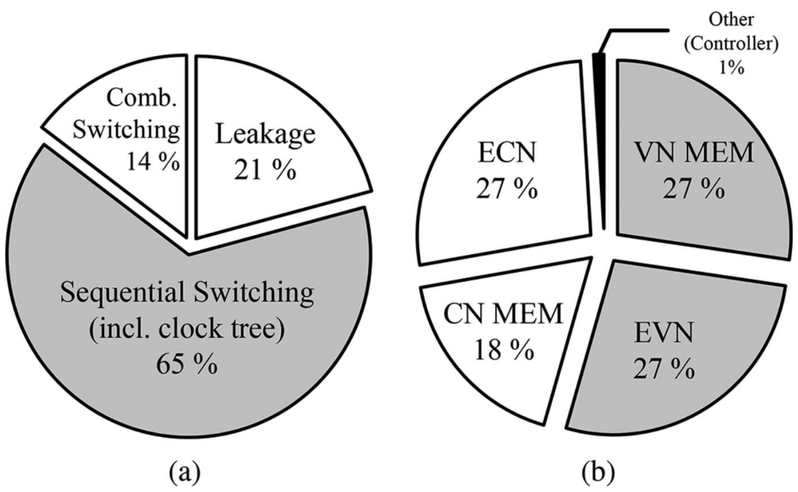


Figure 4.8: (a) Power breakdown of the 65 nm synthesized fully parallel nonbinary LDPC decoder, and (b) the distribution of sequential logic used in the decoder.

switching power of combinational circuits claim the remaining 21% and 14% of the total power,respectively. Further breakdown of the switching power of sequential circuits in Figure 4.8(b) shows that the switching power of the VN and CN memories and the sorters in EVNs and ECNs account for almost all of the sequential switching power.

The high dynamic power consumption prompts us to design a dynamic clock gating strategy to reduce the power consumption of the decoder. Clock gating disables the clock input to sequential circuits to save switching power, which in turn cuts the switching of combinational circuits. The use of clock gating is motivated by the observation that the majority of the VNs converge within a few decoding iterations before reaching the decoding iteration limit. Therefore, it is possible to clock gate the VNs and CNs that have reached convergence to save power. To achieve the most power savings, the clock gating is implemented at a fine-grained node level, i.e., at each VN and CN, and the clock gating is enabled dynamically during run time. The fine-grained dynamic clock gating requires convergence detection at the node level, i.e., each VN detects when it has reached convergence and can be clock gated.

The node-level convergence detection is different from the conventional convergence detection done at the global level by checking whether all parity checks have been met. Although clock gating can also be based on global convergence detection, the power savings would be greatly diminished.

4.3.1 Node-Level Convergence Detection

Node-level convergence detection is not equivalent to global convergence detection. Our proposed node-level convergence detection is designed to match the accuracy of the global convergence detection without causing BER degradation. The node-level convergence detection is based on two convergence criteria: (1) meet the minimum number of decoding iterations , and (2) VN's hard decisions remain unchanged for the last consecutive iterations. The two criteria are designed to prevent false convergence and ensure stability. Each VN checks the criteria upon completing each decoding iteration. If the criteria are met, the VN is clock gated. If a VN is clock gated, parts of the CN that are used for storing and processing messages from and to the VN are also clock gated. A CN is completely clock gated when all its connected VNs have been clock gated.

With node-level convergence detection, it is possible that a VN converges to the correct decision and is frozen, preventing it from changing to an incorrect decision. On the other hand, it is also possible that a VN converges to an incorrect decision and is frozen, and preventing other VNs from correcting this VN later. To best match the BER and FER performance of global convergence detection, and need to be set appropriately.

Fine-grained dynamic clock gating can be compared to early termination [129, 130] that is commonly used in decoder designs. Early termination relies on global convergence detection, whereas fine-grained dynamic clock gating is based on node-level convergence detection, and it allows a large number of VNs and CNs to be turned off before the global convergence is reached.

The idea of early termination can be combined with fine-grained dynamic clock gating to save power and improve throughput by terminating the decoder once all the VNs and CNs are clock gated. We term the approach decoder termination to differentiate it from early termination, because decoder termination relies on node-level convergence detection, whereas early termination commonly relies on global convergence detection.

4.3.2 Fine-Grained Dynamic Clock Gating

The clock gating architecture is illustrated in Fig. 11. The convergence detector inside each VN monitors the hard decisions in each iteration to check whether the hard decisions have changed between iterations. A counter keeps track of the number of consecutive iterations that the hard decisions have remained unchanged. When the convergence criteria are met, the convergence detector enables the clock gating latch (CG latch) to turn off the clock input to all sequential circuits with the exception of essential control circuits that are needed for recovering from the clock gating state. The majority of the VN's dynamic power is saved, leaving only leakage.

The convergence detector propagates the clock gating signal to the CNs to enable the CG latch of V2C message memories in the CNs, as noted in [?]. Clock gating V2C memories eliminates the unnecessary memory updates to save dynamic power. In this way, CN is partially clock gated. When all the connected VNs are clock gated, as indicated by their clock gating signals, a central CG latch is enabled to completely turn off the CN.

A decoder termination controller monitors the VN clock gating signals. When all the VNs are clock gated (and CNs are clock gated as a result), the decoder terminates the current frame and moves on to the next input frame. Decoder termination reduces the average number of decoding iterations per code frame and therefore improves the

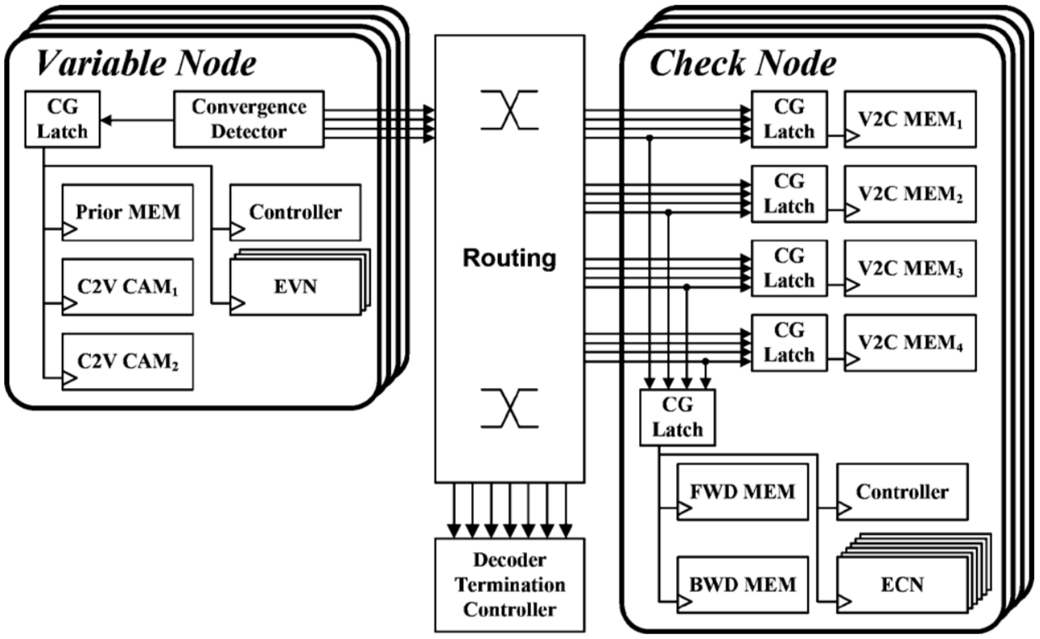


Figure 4.9: Implementation of fine-grained dynamic clock gating for the variable and check node.

decoding throughput for a net gain in energy efficiency.

In our implementation, each VN stores only the hard decision (6 bit) from the previous iteration. In each iteration, the VN compares the hard decision with the previous hard decision, and increments a 4-bit counter if they agree. If not, the counter is reset. After the comparison, the stored hard decision is replaced by the current hard decision for the next iteration. The node-level convergence detection requires only 6 bits of storage per VN (or 960 bits for the entire decoder), a small logic in each VN to compare a pair of 6-bit decisions, and a 4-bit counter. Compared to the size of the nonbinary VN and CN, the overhead for node-level convergence detection is negligible.

To check the effectiveness of fine-grained dynamic clock gating, we simulated the decoder's behavior with node-level convergence detection. Figure 4.10 shows the percentage of nodes that have been clock gated in each decoding iteration across various SNR levels. The decoding iteration limit is set to 30, and the convergence criteria are set to $M = 10$ and $T = 10$. Even at a low SNR of 2.8 dB, more than 85%

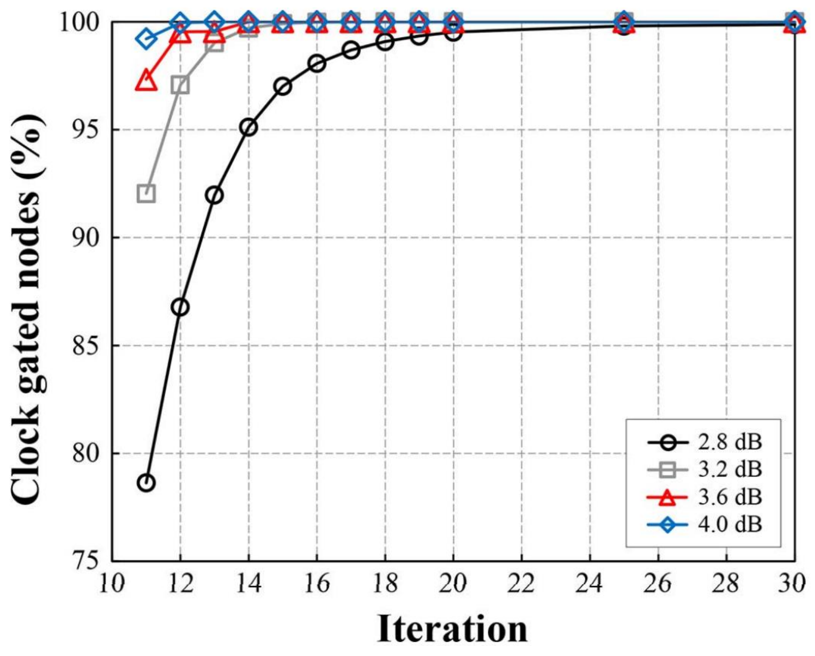


Figure 4.10: Cumulative distribution of clock gated nodes at each iteration for various SNR levels with a decoding iteration limit of 30. The parameters used for clock gating are $M = 10$ and $T = 10$.

of the VNs are clock gated after 12 iterations. After 14 iterations, 95% of the VNs are clock gated. At higher SNRs, the VNs are clock gated at an even faster pace.

The setting of M determines how much power can be saved. The lower the M, the earlier the clock gating can be applied, and the more power we can save. However, a lower M degrades the BER. There is a tradeoff between power consumption and BER. We set $M = 10$ to ensure no loss in BER across a wide range of SNR. We could use a lower at high SNR to achieve more power savings without affecting BER.

4.4 Decoder Chip Implementation and Measurements

A decoder test chip for the GF(64) (160, 80) regular-(2,4) NB-LDPC code was implemented in a ST-Microelectronics 65 nm 7-metal general-purpose CMOS technology. The chip microphotograph is shown in Figure 4.11. The test chip measures 4.40 mm $\times$ 2.94 mm, and the core measures 3.52 mm $\times$ 2.00 mm, or 7.04 mm^2. The

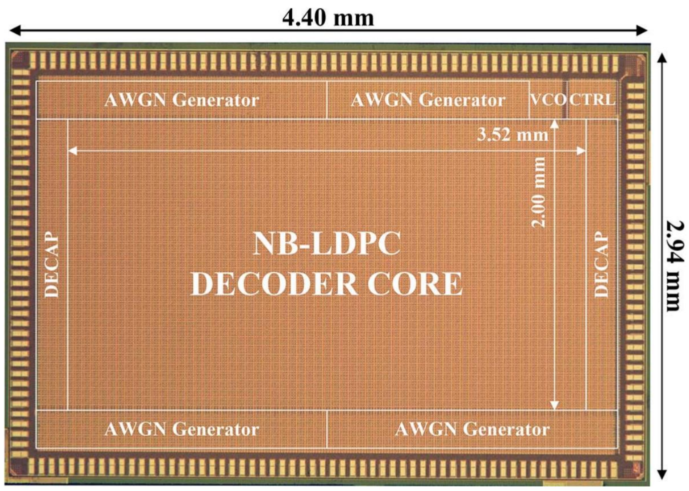


Figure 4.11: Chip microphotograph of the decoder test chip. Locations of the test peripherals and the decoder are labeled.

memory used in this decoder is implemented using registers.

Figure 4.12 shows the BER and FER curves for various configurations. The 5-bit decoder incurs a relatively large quantization loss (compared to floating point) at low SNR, because the 5 bit word length is not sufficient to separate the candidate elements for a VN at low SNR. At moderate to high SNR, the candidate elements can be more easily separated, which explains the much smaller quantization loss at high SNR. Figure 4.12 is based on two months of extensive testing. With a decoding iteration limit of 100, the decoder achieves a BER of at 4.2 dB, a significant improvement over binary LDPC codes of similar block length, e.g., the rate-1/2 672-bit binary LDPC code for the IEEE 802.11ad standard provides a BER of at 4.2 dB. Structured binary LDPC codes of similar block length also encounter severe error floors, which is not seen in the NB-LDPC code. With a more practical 30 iterations and our proposed node-level convergence criteria of and , the decoder still provides an excellent BER performance that is very close to the 100-iteration BER performance.

The NB-LDPC decoder test chip operates at a maximum clock frequency of 700 MHz at 1.0 V and room temperature for a coded throughput of 477 Mb/s with 30 decoding iterations. The test chip consumes 3.993 W, which translates to an en-

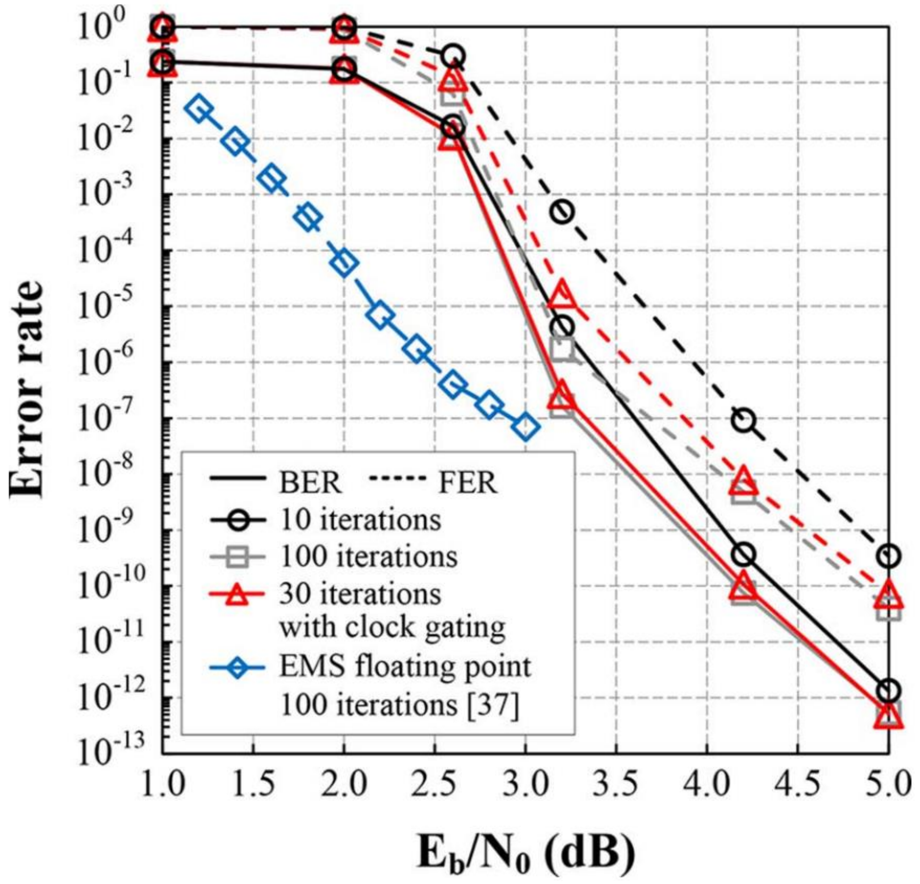


Figure 4.12: Bit error rate and frame error rate performance of the GF(64) (160,80) regular-(2,4) NB-LDPC code using 5-bit quantization and floating point.

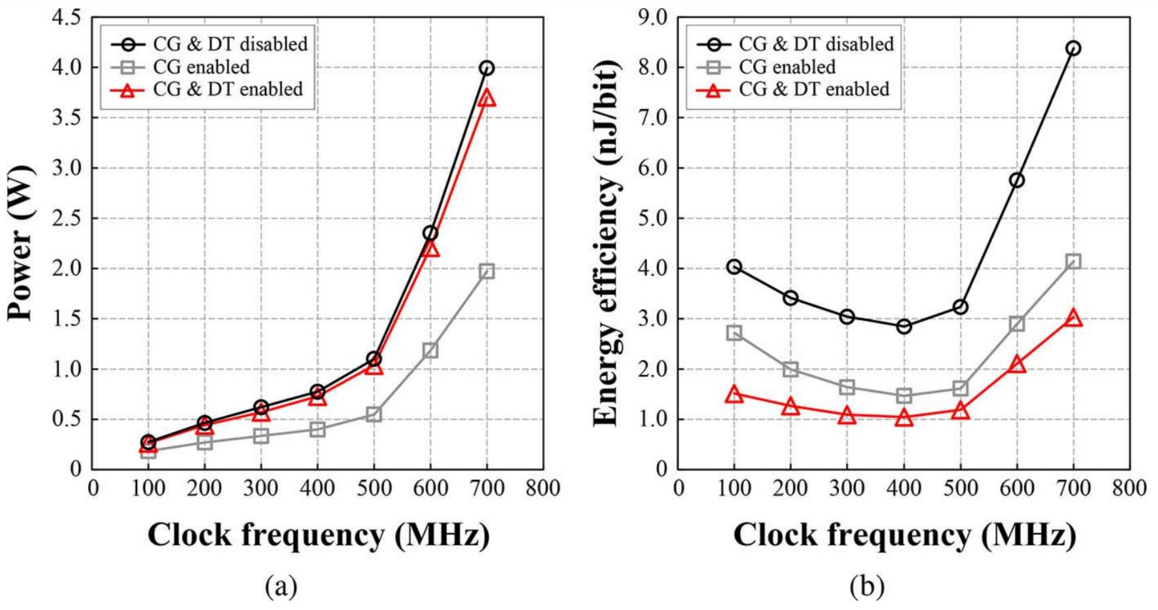


Figure 4.13: Measured NB-LDPC decoder (a) power and (b) energy efficiency at 5.0 dB SNR and 30 decoding iterations. CG denotes clock gating and DT denotes decoder termination. The parameters used for clock gating and decoder termination are $M = 10$ and $T = 10$. This minimum supply voltage is used at each clock frequency.

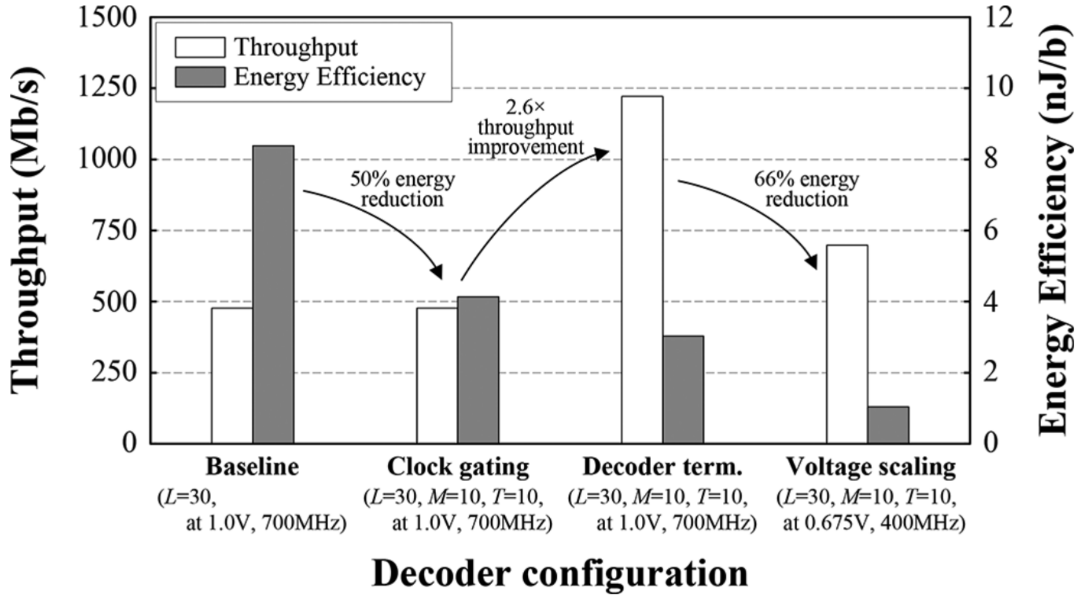


Figure 4.14: Illustration of throughput and energy efficiency of various decoder configurations at 5.0 dB SNR. L, M and T represents decoding iteration limit, minimum decoding iteration, and consecutive iteration threshold, respectively.

Table 4.1: Decoder Chip Measurement Summary

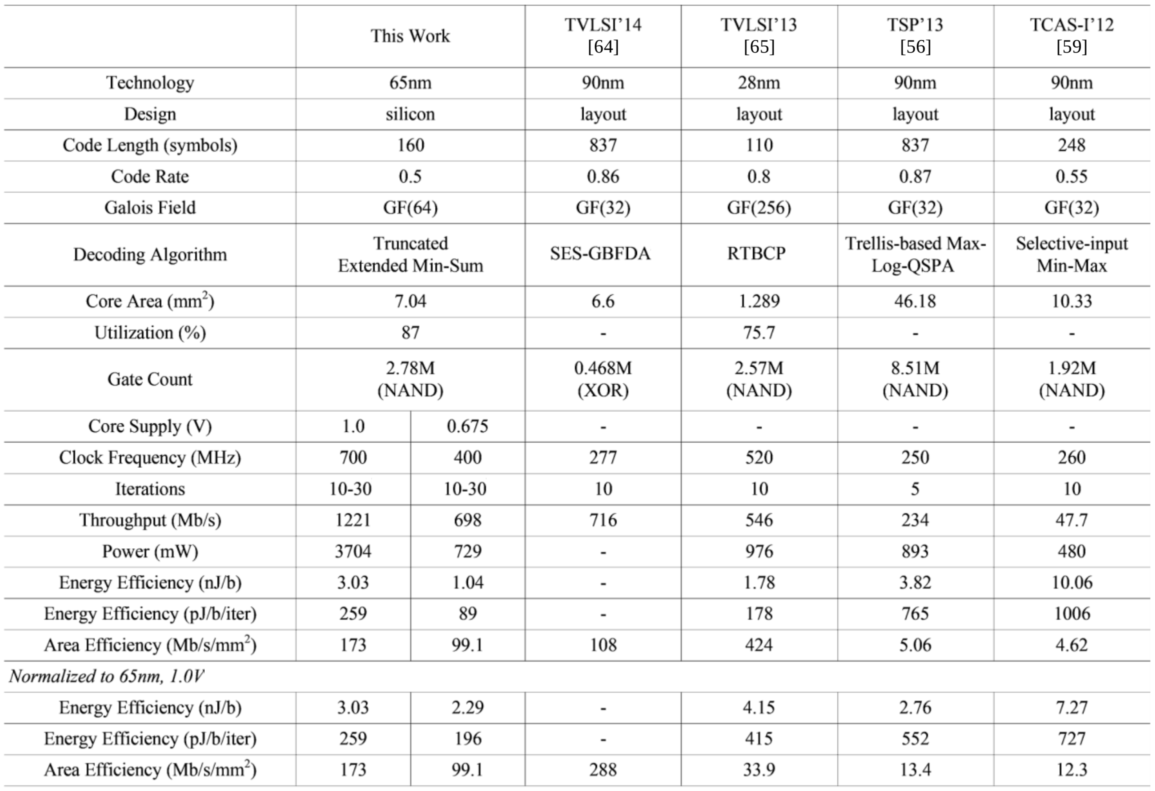
Decoder Configuration	Throughput[1] (Mb/s)	Power[1] (W)	Energy Efficiency[1] (nJ/b)
700MHz @ 1.0V 30 Iterations	477	3.993	8.38
700MHz @ 1.0V 30 Iterations \w CG[2]	477	1.974	4.14
700MHz @ 1.0V 30 Iterations \w CG & DT[2]	1221	3.704	3.03
400MHz @ 0.675V 30 Iterations \w CG & DT[2]	698	0.729	1.04

[1] Measured at 5.0 dB SNR.
[2] CG denotes clock gating, and DT denotes decoder termination. The parameters used for clock gating and decoder termination are $M = 10$ and $T = 10$.

ergy efficiency of 8.38 nJ/b. Figure 4.13, Figure 4.14 and Table 4.1 summarize the measured power consumption of the NB-LDPC decoder test chip. To improve the energy efficiency, fine-grained dynamic clock gating is enabled with node-level convergence criteria of and , reducing the power consumption by 50% and improving the energy efficiency to 4.14 nJ/b. To achieve a higher throughput, decoder termination is enabled to increase the throughput from 477 Mb/s to 1.22 Gb/s at 5.0 dB SNR . The power consumption increases due to a higher activity, but the energy efficiency improves to 3.03 nJ/b, or 259 pJ/b/iteration. Voltage and frequency scaling can be applied to further reduce the power consumption and improve the energy efficiency. Scaling the supply voltage from 1.0 V to 675 mV reduces the maximum clock frequency from 700 MHz to 400 MHz and improves the energy efficiency to 1.04 nJ/b, or 89 pJ/b/iteration, at a reduced throughput of 698 Mb/s.

Table 4.2 lists the nonbinary LDPC decoder test chip along with other state-of-the-art NB-LDPC decoder designs [64, 65, 56, 59] published prior to our design. It

Table 4.2: Comparison with State-of-the-art NB-LDPC Decoders Prior to this Work



	This Work		TVLSI'14 [64]	TVLSI'13 [65]	TSP'13 [56]	TCAS-I'12 [59]
Technology	65nm		90nm	28nm	90nm	90nm
Design	silicon		layout	layout	layout	layout
Code Length (symbols)	160		837	110	837	248
Code Rate	0.5		0.86	0.8	0.87	0.55
Galois Field	GF(64)		GF(32)	GF(256)	GF(32)	GF(32)
Decoding Algorithm	Truncated Extended Min-Sum		SES-GBFDA	RTBCP	Trellis-based Max-Log-QSPA	Selective-input Min-Max
Core Area (mm^2)	7.04		6.6	1.289	46.18	10.33
Utilization (%)	87		-	75.7	-	-
Gate Count	2.78M (NAND)		0.468M (XOR)	2.57M (NAND)	8.51M (NAND)	1.92M (NAND)
Core Supply (V)	1.0	0.675	-	-	-	-
Clock Frequency (MHz)	700	400	277	520	250	260
Iterations	10-30	10-30	10	10	5	10
Throughput (Mb/s)	1221	698	716	546	234	47.7
Power (mW)	3704	729	-	976	893	480
Energy Efficiency (nJ/b)	3.03	1.04	-	1.78	3.82	10.06
Energy Efficiency (pJ/b/iter)	259	89	-	178	765	1006
Area Efficiency (Mb/s/mm^2)	173	99.1	108	424	5.06	4.62
Normalized to 65nm, 1.0V						
Energy Efficiency (nJ/b)	3.03	2.29	-	4.15	2.76	7.27
Energy Efficiency (pJ/b/iter)	259	196	-	415	552	727
Area Efficiency (Mb/s/mm^2)	173	99.1	288	33.9	13.4	12.3

is important to note that none of the previous designs has been fabricated in silicon. This work is the first silicon that has been published to the best of our knowledge. The decoder claims higher throughput and energy efficiency (in pJ/b/iter), when normalized to 65 nm and 1.0 V, than the best previously reported post-layout results. The truncated EMS algorithm allows us to achieve excellent BER performance compared to other simplified algorithms.

4.5 Summary

We present a fully parallel NB-LDPC decoder to take advantage of the low wiring overhead that is intrinsic to NB-LDPC codes. To further enhance the throughput, we apply a one-step look-ahead to the elementary CN design to reduce the clock period, and interleave the CN and VN operations for a short iteration latency of 47 cycles. We implement a fine-grained clock gating at the node level to allow the majority of the processing nodes to be clock-gated long before reaching the iteration limit. A 7.04 mm^2 65 nm decoder test chip is designed for the GF(64) (160, 80) regular-(2, 4) NB-LDPC code. The decoder implements fine-grained dynamic clock gating and decoder termination to achieve a high throughput of 1.22 Gb/s at 700 MHz, consuming 3.03 nJ/b, or 259 pJ/b/iteration. The test chip demonstrates a superior error correcting performance compared to binary LDPC decoders. Voltage and frequency scaling of the test chip to 675 mV and 400 MHz further improve the energy efficiency to 89 pJ/b/iteration at a reduced throughput of 698 Mb/s.

CHAPTER V

HiMA:A Fast and Scalable History-based Memory Access Engine for Differentiable Neural Computer

5.1 Analysis of DNC Kernels

We first analyze DNC's memory access and computational profile, followed by a simulation study of DNC running bAbI dataset [131] on CPU and GPU.

5.1.1 Theoretical Kernel Analysis

Table 5.1 lists DNC's computational kernels with their corresponding primitives, associated memory access complexity and the NoC traffic condition when mapped to a tiled architecture. Recall that the external memory M is modeled as a $N \times W$ matrix ($N > W$) and the number of read heads is R. We categorize DNC kernels into two types: 1) state kernels for maintaining memory states and determining how the external memory is accessed, and 2) access kernels that do not maintain states and perform the actual access to the external memory. NTM only needs access kernels, while DNC requires a variety of new state kernels to support history-based mechanisms. These new state kernels impose critical challenges:

- *Computation*: Kernels such as usage sort and forward-backward require compute-intense large-scale data sorting of $O(N)$ or matrix-vector multiplication of $O(N^2)$

Table 5.1: Analysis of DNC Kernels

Type	Category	Kernel Name	Key Primitives	Ext. Mem Access	State Mem Access	Total NoC Traffic
Access Kernels	Content-based Weighting	Normalize	inner-prod	$O(NW)$	$O(W)$	$O(N_t N)$
		Similarity	inner-prod	$O(NW)$	$O(W)$	$O(N_t)$
	Memory Access	Memory Write	el-add/sub/mult, outer-prod	$O(NW)$	$O(N)$	$O(N_t N)$
		Memory Read	transpose, mat-vec mult	$O(NW)$	$O(N)$	$O(N_t NW)$
State Kernels (*New in DNC*)	History-based Write Weighting	Retention	el-mult, vec acc-prod	No	$O(RN)$	No
		Usage	el-add/sub/mult	No	$O(N)$	No
		Usage Sort	sort (Section 4.3)	No	$O(N)$	$O(N)$
		Allocation	vec acc-prod	No	$O(N)$	$O(N_t)$
		Wr. Weight Merge	el-add/sub	No	$O(N)$	No
	History-based Read Weighting	Linkage	mat expand, outer-prod, el-add/sub/mult	No	$O(N^2)$	$O(N_t N)$
		Precedence	el-add, vec acc-sum	No	$O(N)$	$O(N_t)$
		Forward-backward	transpose, mat-vec mult	No	$O(N^2)$	$O(N_t N^2)$
		Rd. Weight Merge	el-add	No	$O(RN)$	No

complexity, which can be major bottlenecks.

- *Memory Access*: Kernels such as linkage and forward-backward require $O(N^2)$ accesses to the new linkage memory, which easily surpass the memory access by the access kernels used by other MANNs like NTM.

- *NoC Traffic*: In a tiled architecture, some kernels such as linkage rely on inter-tile traffic. The amount of NoC traffic can be as high as $O(N_t N^2)$ depending on state and external memory partitions, and the traffic pattern is non-uniform over time and space due to the different primitives and state memories involved.

The challenges require designing an efficient NoC and memory organization, maximizing distributed processing, and providing efficient computational kernels such as sort.

5.1.2 Kernel Runtime Analysis

We simulated DNC inference on an Nvidia 3080Ti GPU and an Intel Core i7-9700K CPU using the bAbI dataset [131] in NLP. The bAbI dataset consists of 20 tasks. Each task is independent of the others, tests one aspect of an intended NLP behaviour and includes 10,000 QA examples. It is the only publicly available and practically meaningful dataset to date to demonstrate DNC's performance. In our experiments, we ran all tasks in the bAbI dataset and recorded the average runtime. Figure 5.1 captures runtime breakdown of DNC kernels in different categories. The average GPU inference time is 5.16 ms/test, $2.12\times$ faster than the 10.94 ms/test inference time on CPU. On both the GPU and the CPU, the NN (an LSTM) takes less than 5% of the total runtime, while the memory unit takes more than 95% of the total runtime. It highlights the need of a memory access engine for DNC. Additional insights can be derived from the kernel runtime shown in Figure 5.1.

- History-based write weighting, including retention, usage sort and allocation, accounts for 72% of the runtime on the GPU. GPU's highly parallelized hardware are not the most suitable for speeding up large-scale sorting.

- History-based read weighting, including linkage and precedence, relies on vector and matrix operations that can be extensively parallelized by the GPU. This part uses only 9% of the GPU's runtime.

- Content-based write/read weighting, including normalization and similarity, involves many multiply-accumulate (MAC) and softmax operations. It costs 12% of the runtime on the GPU and 22% on the CPU, but it is not the dominant part on either platform.

- Memory write and read[1] are much faster on the GPU (4% of the runtime) than on the CPU (53% of the runtime), because they are dominated by parallelizable weighting operations using MAC arrays.

[1]In the context of DNC, memory write and memory read are not simply write to or read from memory. Applying write weighting to data before write to memory and applying read weighting to data read from memory are the dominant operations in memory write and memory read, respectively.

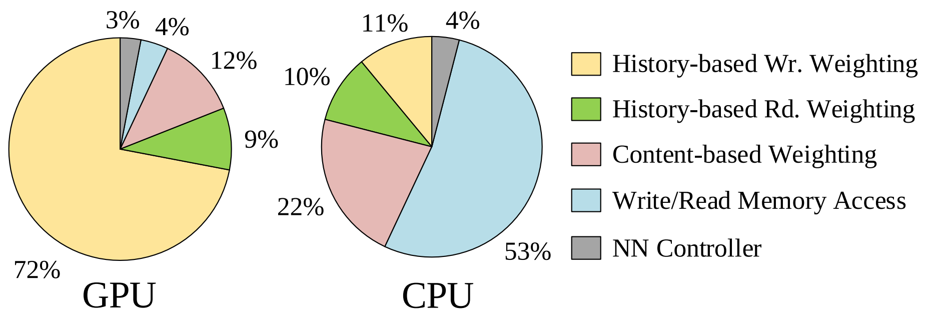


Figure 5.1: Kernel runtime breakdown on CPU/GPU for the bAbI dataset. The external memory size is $N \times W = 1024 \times 64$ and the LSTM is 1-layer of size 256.

The results show that improving the DNC performance requires both optimized computational kernels such as sort and highly parallel matrix operations. GPU and CPU follow a centralized-memory architecture and high premiums in power and area are paid in sustaining high-bandwidth interface and versatile memory hierarchy. In designing an accelerator targeting high performance, high energy efficiency and low cost, a distributed, tiled architecture is the preferred approach.

5.2 HiMA Architecture Design

HiMA is a memory access engine that follows a distributed, tiled architecture that consists of one CT and many PTs with an NoC linking the tiles. Based on kernel analysis and simulation results, we summarize HiMA's architectural design goals.

Scalable and versatile NoC: A fixed NoC is sub-optimal for the complex NoC traffic by DNC's unique state kernels. The H-tree NoC in [100] suffers from traffic saturation with more than 8 PTs as shown in Figure 5.2(d). A more scalable NoC that supports DNC's versatile kernels is desired.

Efficient memory partition: DNC's external memory is large in size, and its state memories can be even larger. For example, the linkage matrix requires a memory of $N \times N$ $(N > W)$. A strategy is needed to optimally partition large external and state memories and distribute them to tiles with the goal of minimizing the NoC traffic amount when executing DNC kernels.

Distributed compute kernels: DNC's kernels are ideally distributed to the tiles. The sort kernel is especially important. It is a performance bottleneck and it is not supported by existing NN/MANN accelerators [91, 92, 93, 94, 95, 96, 100, 99, 97, 98]. The goal is to distribute such kernels along with distributed memory to tiles, while minimizing the NoC traffic amount.

5.2.1 Scalable Multi-Mode NoC

Reconfigurable NoCs have been proposed for DNN accelerators for different tensor sizes in kernels like convolution. Specifically, MAERI [132] and HERALD [96] employ a multi-layer binary tree as show in Figure 5.2(a) with configurable interconnects between adjacent sub-trees at each level. These designs are suitable for DNN's reduction, collection and multi-cast dataflows. They are however sub-optimal for DNC's diverse traffic patterns, especially for transpose and matrix-vector multiplication that require communications between distant tiles. Data transfer between two distant tiles need to pass through their mutual root tile, which can become the traffic congestion point that drastically increases the inter-tile communication latency. The H-tree NoC demonstrated by MANNA [100] was designed for access kernels that require only two types of inter-tile communication: 1) broadcast of interface vectors from CT to PTs, and collection of read vectors from the PTs; and 2) transfer of submatrices of the external memory between PTs for transpose and matrix-vector multiplication. In implementing the access kernels, the H-tree NoC does not present a bottleneck up to 16 tiles [100].

To support history-based memory access, the new state kernels introduce far more inter-tile traffic and a diverse traffic profile. To see the suitability of the H-tree NoC and the multi-layer binary tree, we mapped DNC to a tiled architecture utilizing these two NoCs. To check scalability, we simulate the speedup by increasing the

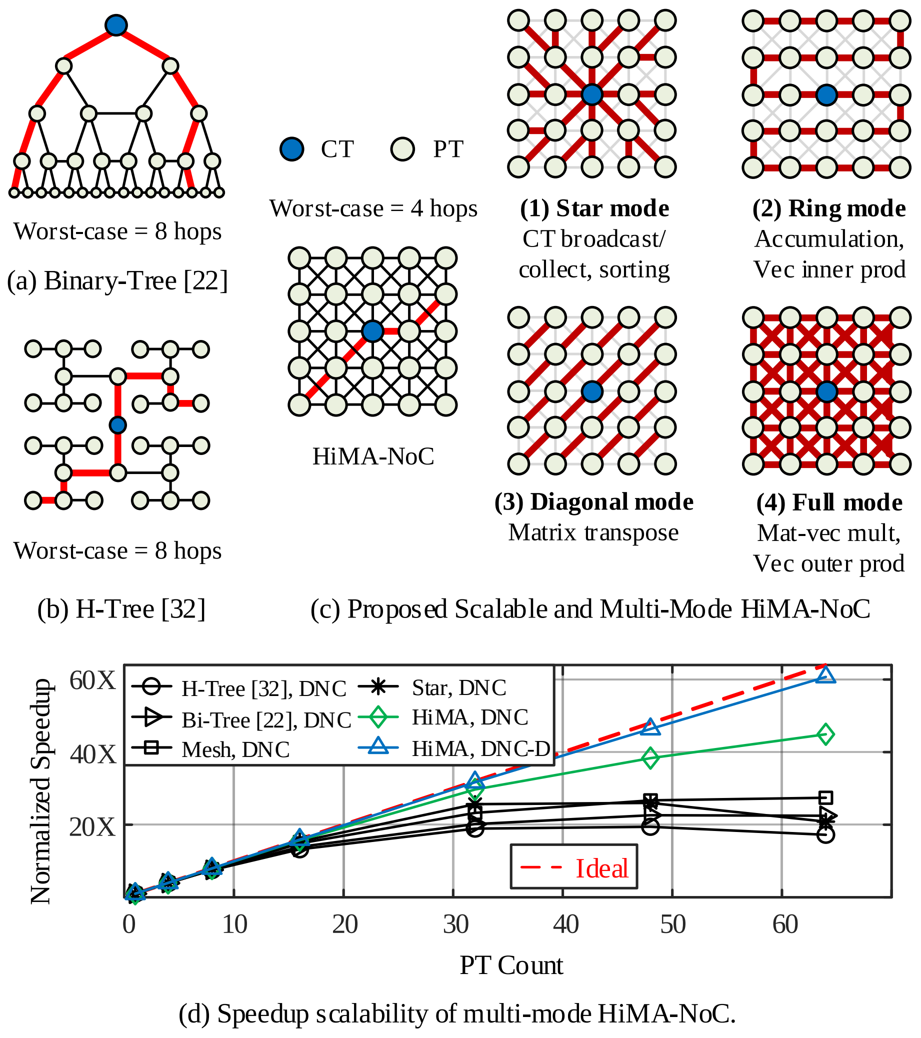


Figure 5.2: Scalable and Multi-Mode HiMA-NoC.

number of PTs. Here we assume ideal CT and PTs where the memory bandwidth or the computational parallelism do not present bottlenecks, and ideal routers that can handle any traffic congestion by stalling. Figure 5.2(d) shows that the speedup starts to saturate beyond the 8× point for both NoCs. The H-tree NoC requires traffic between two tiles to go through their mutual root tile, as show in Figure 5.2(b), resulting in traffic congestion at highest root node for the distant pairs of PTs. The binary-tree NoC [132] is an enhanced H-tree by additional interconnects between adjacent sub-trees at each level. It outperforms the H-tree slightly, but its scalability still saturates at a low level.

We analyze the traffic profile of DNC primitives and the suitable NoC topologies.

- Interface vector broadcast, read vector collection and sorting require only CT-PT traffic. A star NoC is the most suitable, where all PTs are connected directly to the CT with a distance of 1 hop. However, the CT needs a complex router and can become a traffic congestion point with increasing PT count, limiting scalability.

- Accumulation of products or sums and vector inner product require sending accumulated results from one PT to the next PT. A ring NoC is the most suitable.

- Matrix transpose requires transferring on-tile submatrices to other tiles along diagonals, as shown in Figure 5.2(c). A diagonally-connected NoC is suitable.

- Matrix vector multiplication and vector outer product require each tile to send its local submatrices to all other tiles for computation. A full-duplex mesh NoC is the most suitable. However, the scalability of a full-duplex mesh is even worse than the star NoC, because every tile, not only CT, can become a traffic congestion point.

The analysis shows that a fixed NoC topology does not meet DNC's diverse traffic profile. We propose a multi-mode NoC (namely, HiMA-NoC) to shorten the transfer distance, reduce the traffic congestion and enhance the scalability. Figure 5.2(c) shows an example HiMA-NoC for 5×5 tiles. It is made by adding diagonal connections in a mesh NoC. The worst-case inter-tile transfer distance is kept at only 4 in the 5×5 example. Compared to a fixed NoC that improves traffic conditions for some primitives but may worsen for other primitives, HiMA-NoC can be configured in run-time to efficiently support different traffic patterns through multi-mode routers (Section 5.4). Using the same simulation setup, HiMA-NoC provides a more scalable speedup than the fixed H-tree, mesh or star NoC, as shown in Figure 5.2(d). Note that HiMA-NoC does not reduce the amount of traffic, but enhances the tile-to-tile communication latency.

5.2.2 Submatrix-Based Memory Partition

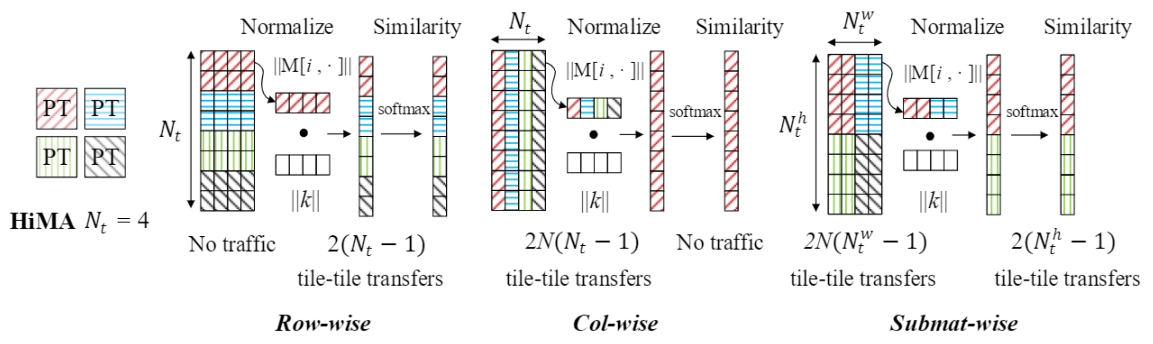
DNC's external memory and state memories need to be partitioned and distributed to the tiles during design time. The partition affects the available access bandwidth, the achievable compute parallelism, and the amount and the patterns of inter-tile data communication. SIMBA[95], a state-of-the-art distributed NN accelerator, distributes weights to tiles and efficiently supports convolution and FC workloads for DNNs. Manna [100] partitions external memory row-wise so that each PT receives N/N_t rows of the external memory, where N_t is the number of PTs. State memories were not discussed in [100].

HiMA's memory partitions are designed for new state memories (nonexistent in NNs or other variants of MANNs like NTM) and external memory. DNC requires accessing to various memories concurrently, and the traffic patterns are non-uniform depending on the primitives that are running. There need to be more considerations on memory partition to reduce the amount of traffic.

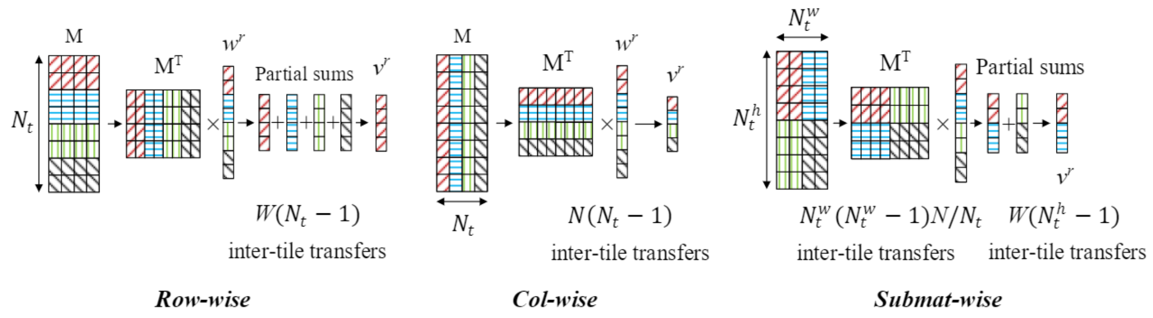
5.2.2.1 External Memory Partition

The external memory is accessed by the access kernels, first in computing content-based weighting including normalization and similarity, followed by memory write or read. Figure 5.3(a) illustrates three possible external memory partitions and their corresponding traffic in computing content-based weighting for a small example of HiMA with $N_t = 4$.

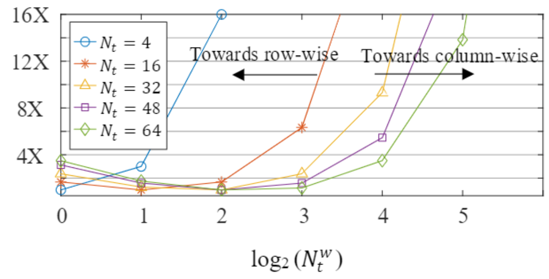
The row-wise partition of the external memory eliminates inter-tile transfer for normalization because normalization is computed on a row of memory, which are



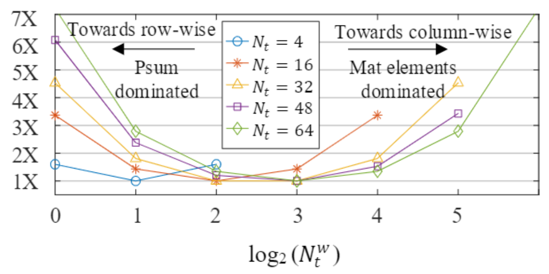
(a) Normalization and similarity in content-based weighting


M
M^T
w^r
Partial sums
v^r
N_t
$W(N_t-1)$ inter-tile transfers
$N(N_t-1)$ inter-tile transfers
N_t^w
N_t^h
$N_t^w(N_t^w-1)N/N_t$ inter-tile transfers
$W(N_t^h-1)$ inter-tile transfers
Row-wise
Col-wise
Submat-wise

(b) Transpose and matrix-vector multiplication in history-based weighting and soft read



(c) Normalized traffic amount for memory read kernel and external memory partition for various *Nt*



(d) Normalized traffic amount for forward-backward kernel and submat-wise linkage memory partition for various *Nt*

Figure 5.3: External memory partition in a 2×2 tile for (a) content-based weighting, and (b) transpose and matrix-vector multiplication in history-based weighting and soft read; (c) inter-tile traffic for memory read kernel with various external memory partition; (d) inter-tile traffic for forward-backward kernel with various linkage memory partition.

stored in the same PT. When calculating similarity, one PT produces only a partial sum (psum), and it collects the psums from the rest of $N_t - 1$ PTs to compute the global sum, followed by scaling and softmax. The softmax result is then distributed to the $N_t - 1$ PTs. Hence the number of inter-tile transfers is $2(N_t - 1)$. Alternatively, if we follow the column-wise partition of the external memory, normalization requires $2N(N_t - 1)$ inter-tile transfers, but similarity can be computed locally.

The row-wise and column-wise partitions can be viewed as special cases of a generalized submatrix-based partition where the external memory is divided into N_t^h block rows and N_t^w block columns, where $N_t = N_t^h \times N_t^w$ and $N_t^h, N_t^w \in Z^+$. As shown in Figure 5.3(a), using submatrix-based partition, normalization and similarity calculations cost $2N(N_t^w - 1)$ and $2(N_t^h - 1)$ inter-tile transfers, respectively. Based on Eq. (5.1), given N and N_t, to minimize the inter-tile traffic, $N_t^w = 1$ and $N_t^h = N_t$. In other words, *the row-wise partition of the external memory costs the minimum inter-tile traffic in computing content-based weighting.*

$$\underset{N_t^h, N_t^w}{\operatorname{argmin}} \left(2N(N_t^w - 1) + 2(N_t^h - 1) \right) \tag{5.1}$$

The memory write to the external memory requires element-wise operations that can be executed locally by PTs in parallel. The memory read from the external memory requires inter-tile traffic to support memory matrix transpose and matrix-vector multiplication. Figure 5.3(b) illustrates the inter-tile transfer patterns. Similarly, the optimal partition for matrix transpose and matrix-vector multiplication can be derived mathematically as Eq. (5.2).

$$\underset{N_t^h, N_t^w}{\operatorname{argmin}} \left(\frac{N_t^w (N_t^w - 1) N}{N_t} + W(N_t^h - 1) \right) \tag{5.2}$$

Due to the quadratic dependence on N_t^w, N_t^w should be kept at the minimum to minimize the inter-tile transfers. Therefore, we demonstrate that *the row-wise partition of the external memory costs the minimum inter-tile traffic.*

5.2.2.2 State Memory Partition

The state kernels require a set of state memories: usage, linkage, precedence, write weight and read weight. State memories of size N (usage, precedence, write weight) or $N \times R$ (read weight) can be straightforwardly partitioned to N/N_t or $N/N_t \times R$ parts and distributed to the N_t PTs.

The linkage memory, on the other hand, has a size of $N \times N$. The linkage memory is used by the forward-backward kernel in matrix transpose and matrix-vector multiplication primitives. The inter-tile transfer patterns look similar to Figure 5.3(b), except that the input matrix is $N \times N$ instead of $N \times W$. Similarly, we find the number of inter-tile transfers based on the generalized submatrix-based partition and formulate the optimization mathematically in Eq. (5.3).

$$\underset{N_t^h, N_t^w}{\operatorname{argmin}} \Bigg(\underbrace{\frac{N_t^h(N_t^h - 1)}{N_t} + N_t^w}_{\text{Forward}} + \underbrace{\frac{N_t^w(N_t^w - 1)}{N_t} + N_t^h}_{\text{Backward}} \Bigg) \tag{5.3}$$

The partition choices and the impact on inter-tile traffic are plotted in Figure 5.3(c) for a $N \times W = 1024 \times 64$ external memory used in running the bAbI dataset. The five sets of curves correspond to various number of tiles N_t, and they cover a range of N_t^w choices. The results show that both the low-end of N_t^w (corresponds to row-wise partition, or transfer psums) and the high-end of N_t^w (corresponds to column-wise partition, or transfer matrix elements) are suboptimal. The minimum inter-tile traffic is somewhere in between. As an example, for $N_t = 16$, the optimal submatrix partition for the linkage memory is $N_t^h \times N_t^w = 4 \times 4$.

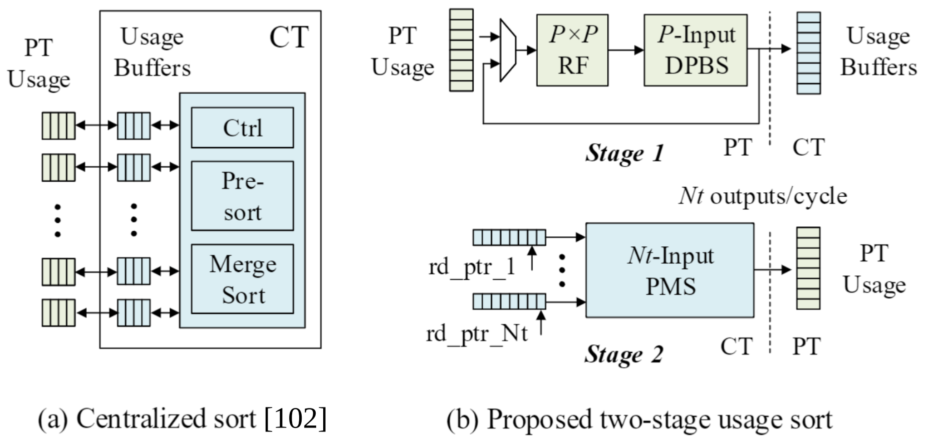


Figure 5.4: Two-stage usage sort.

5.2.3 Two-Stage Usage Sort

Usage vector sort is a bottleneck primitive. In HiMA, the usage vector is distributed and stored in parts on the PTs. A conventional solution using centralized merge sort [102], as shown in Figure 5.4(a), takes $N\log N$ cycles for a length-N usage vector. To achieve a lower latency, we propose a local-global two-stage sort for the distributed tile architecture: 1) a local usage vector of size $n = N/N_t$ is first sorted by each PT, 2) global merge sort by CT to combine N_t sorted local usage vectors.

We illustrate the two-stage sort for an example of $N_t = 4$ tiles and an external memory of $N = 1024$ rows. Each PT keeps a local usage vector of length $n = 256$. In stage 1 in each PT, a local usage vector is reshaped into a $P \times P$ matrix where $P = \sqrt{n} = 16$. We apply a fast multi-dimensional sorting algorithm (MDSA) [133] to complete the local sort in only 6 phases. The 2D MDSA sorter is illustrated in Figure 5.4(b), which is composed of a $P \times P$ register file (RF) and a P-input dual-mode pipelined bitonic sorter (DPBS) [133] supporting both ascending and descending order. The 16-input DPBS can be pipelined into $D_{DPBS} = 5$ stages. A length $n = 256$ local usage vector can be sorted in only $6 \times (P + D_{DPBS}) = 126$ cycles.

In stage 2, sorted local usage vectors are sent from PTs to CT for global merge sort. CT utilizes N_t memory banks to store the local usage vectors. We apply an N_t-input parallel merge sorter (PMS) [134] in CT to support N_t outputs per cycle, which are to be written back to the corresponding PTs as shown in Figure 5.4(b). The pointers are updated to keep track of the status of each memory bank. The 4-input PMS can be pipelined into $D_{PMS} = 7$ stages. The global merge sort for the example $N_t = 4$ takes only $n + D_{PMS} = 263$ cycles. With the proposed local-global two-stage sort, usage sort computation latency is reduced to only $6 \times (P + D_{DPBS}) + n + D_{PMS} = 389$ cycles compared to $N\log N$ cycles for the centralized merge sort.

5.3 Algorithmic Techniques

A key advantage of a distributed, tiled architecture is more opportunities for parallel processing. However, only part of DNC primitives like element-wise operations can take full advantage of distributed processing, while most of the primitives need to operate on the entire external memory or the entire state memories, resulting in excessive traffic and limited hardware scalability. We aim to distribute most of the processing to individual PTs by presenting a distributed version of the DNC model named DNC-D, while minimizing the accuracy loss over DNC.

5.3.1 Distributed Execution for DNC

In DNC, the LSTM provides an interface vector to the memory unit as the input and receives a read vector as the output. As shown in Figure 5.5, in DNC-D the LSTM provides a sub interface vector to each distributed PT instead of broadcasting one global interface vector to all the PTs. Soft read and soft write are executed locally on each PT's local portion of the external memory and state memories.

DNC-D could degrade the inference accuracy. To minimize the loss, we introduce a trainable weighted sum to merge the N_t output read vectors v_i^r from the PTs, where $i \in \{1, ..., N_t\}$, and compute the final read vector v^r as output to the LSTM as (5.4) below:

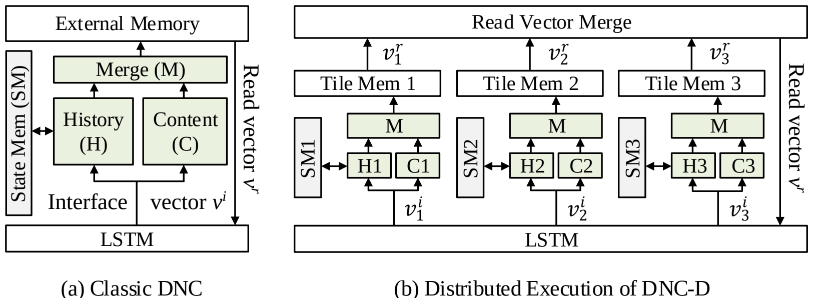


Figure 5.5: Illustration of memory operation in DNC and distributed execution in DNC-D.

$$v^r = \sum_{i=1}^{N_t} \alpha_i^r v_i^r, \tag{5.4}$$

where the trainable weights $\alpha_i^r \in [0, 1]$ are determined by the LSTM. The distributed execution offers several advantages: 1) it eliminates the inter-PT communication, 2) it reduces the computations on PTs related to inter-PT data, and 3) it removes the global sort. Without any inter-PT traffic, HiMA achieves nearly optimal speedup scaling as shown in Figure 5.2(d). In Section 5.5, we study the improved hardware efficiency and the accuracy loss of DNC-D.

5.3.2 Approximation Techniques

We introduce two optional approximation techniques to further reduce the compute complexity.

Usage Skimming: The usage vectors are collected in computing the write allocation. In practice, we observe that the least significant usage entries have little effect on computation of the write allocation. We propose usage skimming to discard the K smallest usage entries. Usage skimming reduces the complexity of usage sort and write allocation proportionally. In Section 5.5, we evaluate the inference accuracy impact of usage skimming with various K and show its hardware efficiency

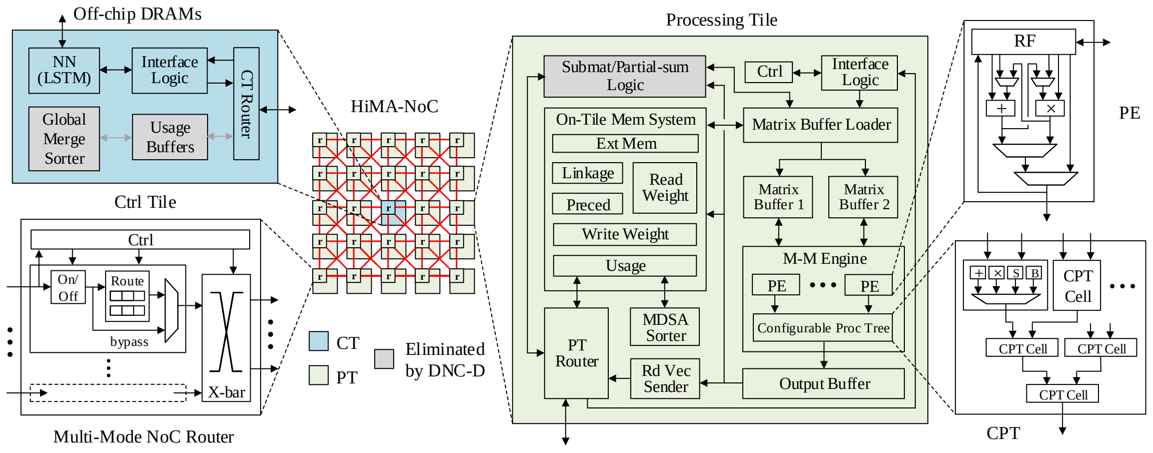


Figure 5.6: HiMA architecture and CT/PT designs.

enhancements.

Softmax Approximation: Softmax is a timing-critical compute block. State-of-the-art softmax approximations include look-up-table (LUT) based [135] or piece-wise linear approximation (PLA) based [136] exponential function. The drawback of the LUT-based is that the number of entries in the table increases exponentially with the input bit width. We combine PLA and LUT approaches: we apply the PLA-based approximation with a small number of line pieces, each of which is an affine function with a slope; we utilize a LUT of affine functions that stores the corresponding function parameters. The design costs only 1 multiply and 1 add.

5.4 HiMA Prototype

Putting everything together, the HiMA architecture is depicted in Figure 5.6. It is composed of a CT with surrounding PTs connected via the proposed HiMA-NoC. HiMA incorporates all the architectural design techniques and optionally the algorithmic optimizations and approximations.

Controller Tile: CT contains the LSTM and it also executes kernels that require global-level processing. An LSTM implementation employed by [100] is used in this work and it handles the LSTM inference and communication with off-chip memories. The CT design is illustrated in Figure 5.6. It sends the interface vectors to the PTs and collects the read vectors from the PTs through routers. The global usage buffers and merge sorter are employed for the 2nd-stage usage sort. Note that using DNC-D, the distributed DNC model, the 2nd-stage usage sort can be eliminated for smaller area.

Processing Tile: A PT's memory system consists of an external memory bank and state memory banks for linkage, precedence, usage, read weighting and writing weighting. The memory partitions are determined based on the optimized submatrix-based partitions. PT's compute modules support vector and matrix operations for the primitives outlined in Table 5.1. Two matrix buffers hold the data for processing from the on-PT memories, the PT router or the interface collector. A matrix buffer loader is used to format and store the data to the corresponding buffers. A matrix-matrix engine (M-M engine) is developed to perform matrix and vector operations. The M-M engine is made of a systolic array of processing elements (PEs) with a configurable processing tree (CPT) to support configurations of different sizes of vectors and matrices.

Each PE consists of a small RF to hold the intermediate values. The PE design supports bypass, add, multiply, multiply-then-add or add-then-multiply modes. The CPT consists of multiple stages of compute cells (CPT cell) including adders, multipliers, special function units (SFUs) and bypass routes. It follows a binary tree for reduction and enables faster executions for accumulation of products, inner product and psum accumulation. PT also includes a length-N/N_t MDSA sorter for on-tile usage sorting. The proposed architecture provides parallelism through the PE array and the multi-entry RF inside PE. The size of PE array, depth of RF inside PE, and the number of CPT stages can be scaled up to provide a higher degree of parallelism.

Multi-Mode NoC Router: Figure 5.6 illustrates the 8-way multi-mode router that supports different HiMA-NoC modes specified in Figure 5.2. In addition to the conventional router logic and buffers, input/output ports are controlled by on/off switches to enable traffic only in certain directions. For example, only the east/west ports are enabled for the inner tiles in the ring mode; and only the northeast/southwest ports are enabled in the diagonal mode. Feed-through single-cycle transfer is enabled when the input buffer in the forward direction is empty, bypassing router logic and reducing the latency for non-congested tiles. The multi-mode router is implemented by route LUTs dedicatedly designed to support proposed modes and a controller that monitors the buffer conditions and generates control signals for each mode.

5.5 Evaluations and Benchmarking

We developed a parameterized RTL simulator for HiMA to evaluate its silicon area, inference speed and power consumption. All designs utilize a 32-bit precision for a fair comparison with state-of-the-art MANN accelerators [102, 100]. We verified the designs against a functional model of DNC in Python at kernel level as well as system level. To estimate area, we synthesized designs at a 500 MHz clock frequency in a 40nm CMOS technology. We used Ansys PowerArtist to obtain power measurements of executing DNC kernels based on switching activities.

The HiMA-baseline architecture employs the H-tree NoC used in [100]. Proposed architectural features, including HiMA NoC, the optimized submatrix-wise memory partition and the two-stage usage sort, are incorporated in the optimized HiMA architectures. HiMA can be further enhanced by the DNC-D model, the usage skimming and the softmax approximation. Based on the architectural and algorithmic features, we create two HiMA architectural prototypes, HiMA-DNC that runs DNC and HiMA-DNC-D that runs DNC-D. Each prototype is equipped with $N_t = 16$ PTs and 1 CT, and supports an external memory of size up to $N \times W = 1024 \times 64$ for processing the bAbI dataset.

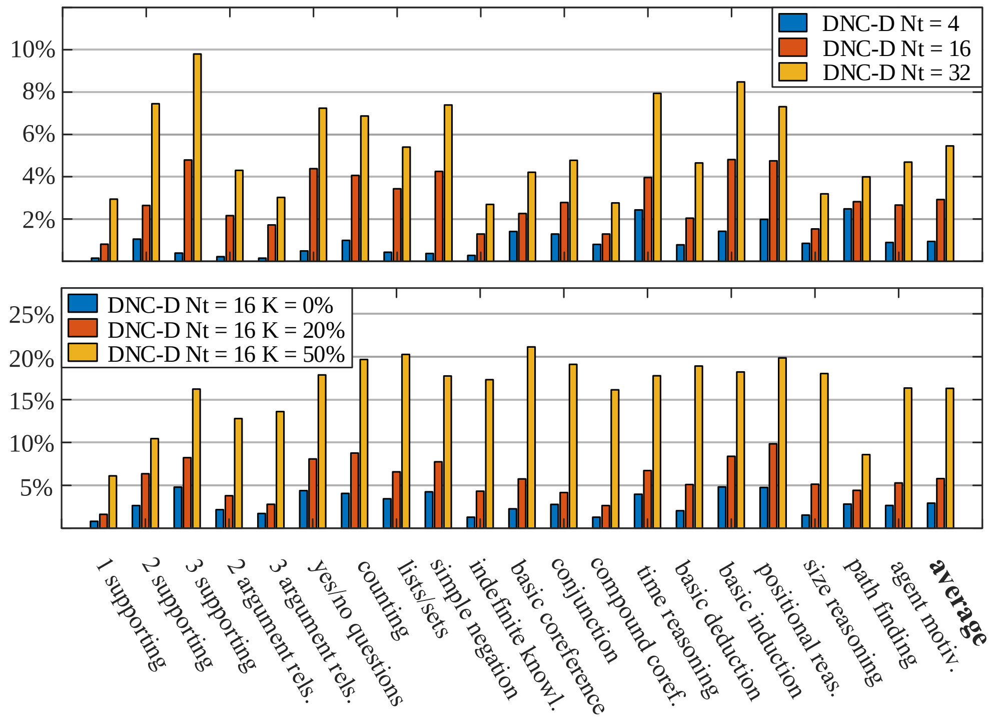
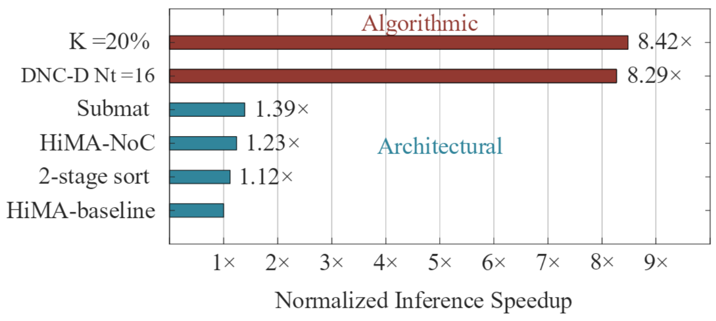


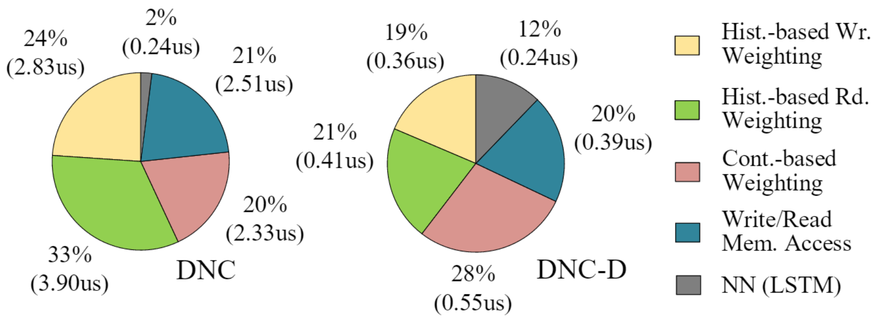
Figure 5.7: Inference error by DNC-D on 20 tasks of bAbI dataset for various N_t and usage skimming K.

5.5.1 Inference Accuracy

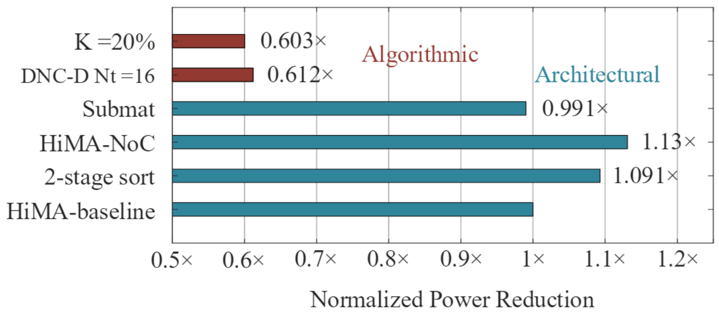
To evaluate the inference accuracy of the DNC-D model, we performed simulations using the bAbI dataset and report the error rates over DNC across 20 benchmark tasks in Figure 5.7. The error rate of the DNC-D model increases with the number of distributed tiles N_t. If N_t is capped at 32, the average error rate of DNC-D is kept below 6% over DNC. With a usage skimming rate of $K = 20\%$ and $N_t = 16$, DNC-D demonstrates an error rate of 5.8% higher than DNC. Further increasing the skimming rate to 50% increases the error rate above 15% over DNC. The proposed algorithmic features trade inference accuracy for a higher hardware efficiency. One can select N_t based on the accuracy tolerance. Parameters used in approximations can be selected based on simulations.



(a) Speedup breakdown with architectural and algorithmic optimizations



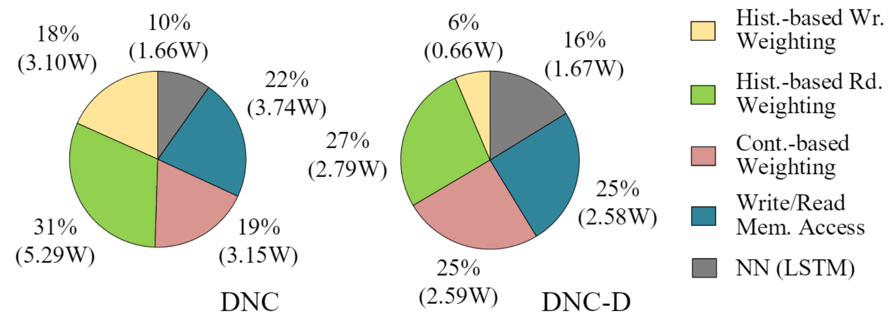
(b) Kernel runtime breakdown of HiMA running DNC and DNC-D



(c) Power reduction with architectural and algorithmic optimizations

5.5.2 Inference Speed

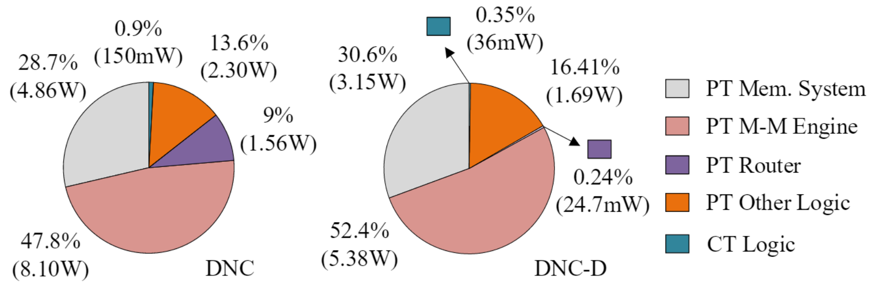
Figure 5.8(a) itemizes the inference speedup after steps of architectural optimizations over a HiMA-baseline ($N_t = 16$): 1) the two-stage sort provides a 1.12× speedup over the HiMA-baseline; 2) replacing the H-tree NoC in the HiMA-baseline by the multi-mode HiMA-NoC reduces the communication latency and improves the inference speed to 1.23× over the baseline. The improvement is mainly due to run time savings of traffic-intensive kernels involving matrix transpose and matrix-vector mul-



(d) Kernel power breakdown of HiMA running DNC and DNC-D

Area (mm^2)	HiMA baseline	HiMA DNC	HiMA DNC-D
PT	4.92	5.01	4.22
PT Mem	2.07	2.07	1.53
CT	0.43	0.52	0.18
Total	79.14	80.69	67.71
Power (W)	16.80	16.96	10.28

(e) HiMA silicon area and power with a 40nm CMOS technology



(f) Module power breakdown of HiMA running DNC and DNC-D

Figure 5.8: HiMA speed, silicon area and power with $N_t = 16$.

tiplications, such as linkage, forward-backward and memory read; 3) applying the submatrix-wise partition increases the inference speed to 1.39× over the baseline, where the speedup is mainly attributed to the reduced traffic amount. These improvements are based on architectural features only. The architecturally optimized HiMA-DNC achieves an inference time of 11.8 μs per test. Figure 5.8(b) shows the kernel run time breakdown in executing DNC. History-based write weighting and read weighting are the most significant, taking 24% and 33% of the run time, respectively.

To further improve the speed, we can apply DNC-D with distributed execution

($N_t = 16$). HiMA-DNC-D achieves a 8.3× inference speedup over the baseline as shown in Figure 5.8(a). The improvement is due to several factors: 1) the elimination of all inter-PT traffic, 2) the computation reduction on PTs, and 3) the elimination of global usage sort. Applying a $K = 20\%$ usage skimming and the softmax approximation increases the inference speedup to 8.4× over the baseline. The architecturally and algorithmically optimized HiMA-DNC-D with $K = 20\%$ usage skimming shortens the inference time to 1.95 μs per test. As shown in Figure 5.8(b), the run time for history-based write weighting and read weighting in DNC-D are reduced by 87% and 89% compared to the run time in DNC, respectively.

5.5.3 Silicon Area and Power

Both HiMA-DNC and HiMA-DNC-D prototypes contain $N_t = 16$ PTs, and they implement all the architectural features. Additionally, HiMA-DNC-D employs a simpler PT and CT due to the elimination of the inter-PT communication and the associated global processing. Figure 5.8(e) compares the silicon area and power consumption of HiMA-DNC and HiMA-DNC-D to HiMA-baseline. HiMA-DNC has a PT area of 5.01 mm^2. The architectural features cost an overhead of 1.8% for the PT over the baseline PT. PT's memory system occupies 2.07 mm^2, including an external memory of 16.4 KB, a linkage memory of 262 KB and multiple 256 B state memories. The linkage memory and the external memory account for 81.3% and 4.8% of the PT memory area, respectively.

Figure 5.8(c) itemizes the power impact of architectural features: 1) the two-stage sort adds 9% power over the baseline due to the introduction of local sorters in each PT; 2) adopting the multi-mode HiMA-NoC increases the power by another 4%; 3) applying the submatrix-wise partition reduces the total power to 0.9% below the baseline, where the power saving comes from the reduced data movement. In all, HiMA-DNC consumes 16.96 W for running a complete DNC inference.

HiMA-DNC-D has a smaller on-PT linkage memory and the centralized sorter is eliminated in CT. It results in a reduced PT area of 4.22 mm^2 and a reduced CT area of 0.18 mm^2. HiMA-DNC-D employs a simpler router that only supports CT-PT traffic as DNC-D eliminates all inter-PT traffic. HiMA-DNC-D uses 16.1% less silicon area and consumes 39.4% less power than HiMA-DNC.

Figure 5.8(d) and Figure 5.8(f) show the kernel and module power breakdown. Notably, DNC-D reduces the power of history-based write weighting by 79% due to the elimination of global usage sort in CT and the usage transfers between CT and PTs. DNC-D also cuts 98.4% of the router power because of the elimination of all inter-PT traffic. Since DNC-D allows PT to compute based only on local memories, the computation and traffic reduction result in power savings across all relevant kernels and modules.

HiMA can be scaled up with more tiles to support a larger external memory and a higher degree of parallelism. As shown in Figure 5.9(a), the power of HiMA-DNC grows super-linearly with N_t mainly because of the increased traffic and the related computations on each PT, while DNC-D improves the power scalability close to the ideal (linear) scaling.

5.5.4 Comparison with State-of-the-Art Accelerators

Figure 5.9(b) compares HiMA's performance to the state-of-the-art MANN accelerators as well as an Nvidia 3080-Ti GPU and an Intel Core i7-9700K CPU. The speedup is normalized to the GPU. Figure 5.9(c) and Figure 5.9(d) compare HiMA's area and power to the MANN accelerators. GPU and CPU are omitted in area and power comparisons since it would be unfair to compare area and power of an accelerator to general-purpose computing platforms. The area and power are normalized to Farm [102]. The area is also normalized based on each design's process technology.

Farm achieves a 68.5× faster speed over the GPU. Farm's faster speed is mainly

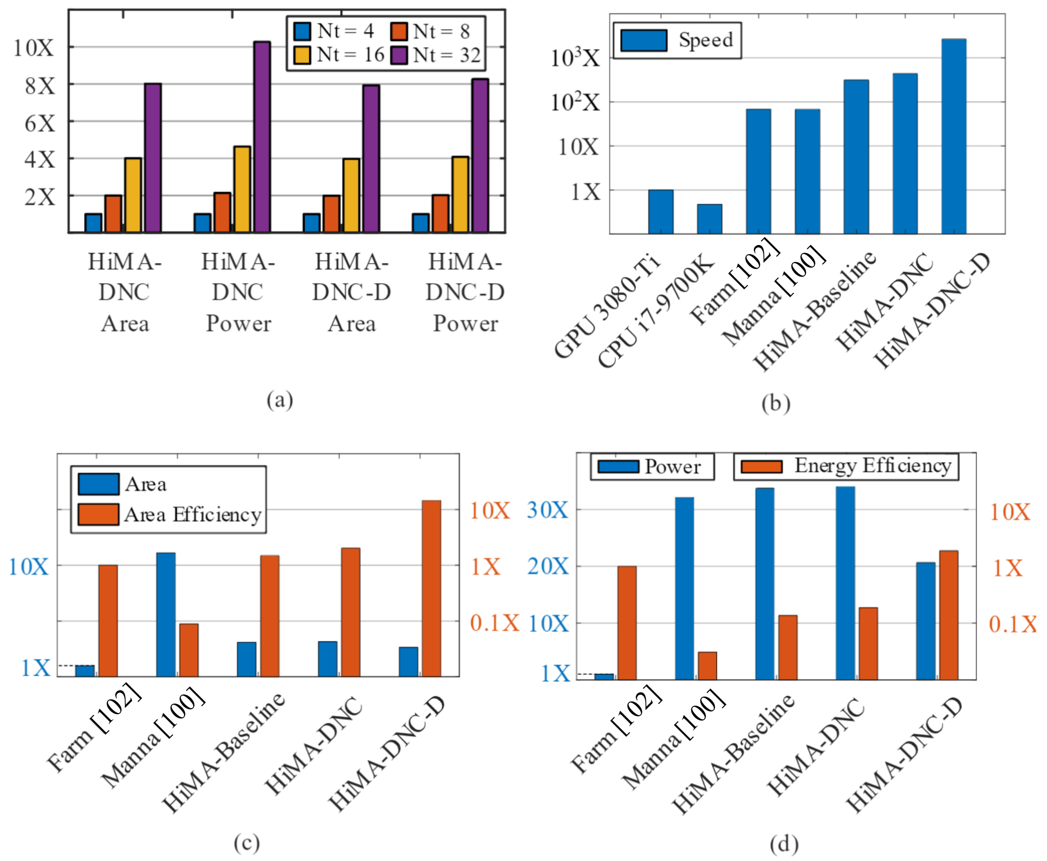


Figure 5.9: (a) Area and power scalability of HiMA-DNC and HiMA-DNC-D for various N_t to support larger external memory. (b)-(d) Performance, area and power comparison of HiMA ($N_t = 16$) with state-of-the-art MANN accelerators and GPU/CPU (area efficiency is measured by throughput/area, and energy efficiency is measured by throughput/power).

attributed to its small memory size (up to $N = 256$) and mixed-signal designs. However, Farm's centralized-memory architecture is not scalable to a larger size to support practical problems and the mixed-signal computation is not yet feasible at a large enough scale. The 16-tile NTM accelerator Manna [100] utilizes an H-tree NoC. It achieves a similar speedup as Farm, but it costs 11× area and 32× power to support 20× larger external memory than Farm. Manna still cannot run DNC due to the lack of support for history-based memory access.

HiMA-baseline uses the same H-tree NoC as Manna and it supports DNC's history-based memory access. It has a 4× larger external memory than Farm while using only 3.16× the area of Farm. HiMA-baseline consumes a higher power than Manna to support DNC's history-based mechanisms. HiMA-DNC achieves a 1.39× faster speed over HiMA-baseline thanks to the architectural features. The overhead of the architectural features is almost negligible, which explains why HiMA-DNC uses similar area and power as HiMA-baseline. HiMA-DNC-D takes advantage of the DNC-D model to increase the speed by 8.4× over HiMA-baseline and reduces the area by 14.4% and power by 38.8% over HiMA-baseline. Compared to Manna that was designed in a 15nm technology, the 40nm HiMA-DNC-D demonstrates 39.1× faster speed, 164.3× better area efficiency and 61.2× better energy efficiency.

5.6 Summary

We present HiMA, a distributed, tile-based accelerator, to efficiently speed up history-based memory access for advanced MANN models like DNC. A multi-mode NoC is designed to support different traffic patterns and improve latency and scalability. Submatrix-wise memory partition is developed to minimize the amount of data movements. To achieve better hardware efficiency, we leverage the tiled architecture to design a two-stage usage sort. To fundamentally improve the efficiency of HiMA's distributed architecture, we distribute not only memory, but also memory operations to the tiles in the form of a new DNC-D model. The HiMA compute kernels can be further optimized by skimming insignificant usage entries and applying an efficient approximation to the softmax function.

We create two HiMA architectural prototypes: HiMA-DNC that runs DNC and HiMA-DNC-D that runs DNC-D. The results show that HiMA-DNC and HiMA-DNC-D achieve 6.47× and 39.1× higher speed, 22.8× and 164.3× better area efficiency, and 6.1× and 61.2× better energy efficiency than Manna, the state-of-the-art tiled MANN accelerator for NTM. Compared to an Nvidia 3080Ti GPU, HiMA-DNC and HiMA-DNC-D outperform by up to 437× and 2,646× in speed, respectively.

CHAPTER VI

Hardware Acceleration for Neural Ordinary Differential Equations

6.1 Neural ODE Theory

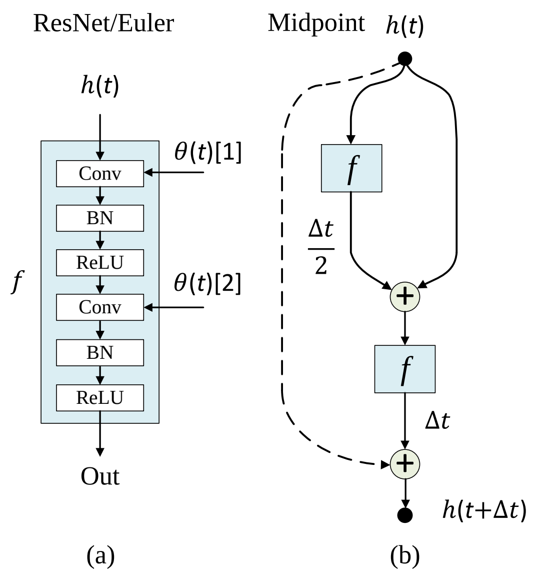
Recall that neural ODE is extended from classical ResNet through Euler discretization (6.1):

$$\frac{\mathrm{d}h(t)}{\mathrm{d}t} = f(h(t), t, \theta),\ h(0) = x,\ t \in [0, T] \tag{6.1}$$

where the inference becomes an integration from time 0 to the evaluation time point T as shown in (6.2):

$$h(T) = h(0) + \int_{0}^{T} f(h(t), t, \theta)dt \tag{6.2}$$

The f function in NODE is typically a shallow NN. An example f function of NODE is shown in Figure 6.1(a). It is essentially an NN that has 2 stages of {Conv, BN, ReLU}. $\theta(t)[1]$ and $\theta(t)[2]$ are the associated weights for the 2 Conv layers at time stamp t. A deeper NN can be used for f function to improve the modeling



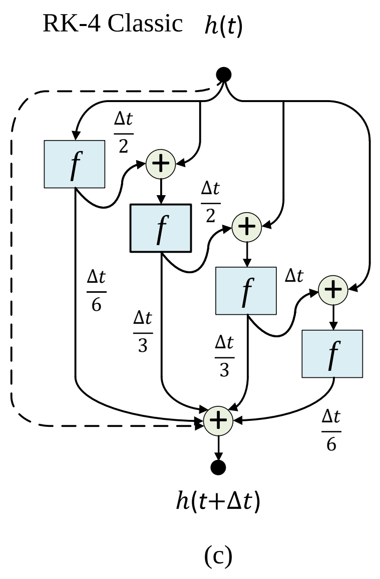


Figure 6.1: NODE with different ODE solvers: (a) Euler (classical ResNet), (b) Midpoint and (c) RK-4 Classic.

capabilities with higher network complexity. Classical ResNet [8] uses a first-order Euler discretization from $h(t)$ to $h(t + \triangle t)$ with a stepsize $\triangle t$.

6.1.1 Adaptive Stepsize Runge-Kutta Method

More complicated numerical integrators, such as midpoint in Figure 6.1(b) or 4-th order Runge-Kutta (RK-4 Classic) in Figure 6.1(c), can be used to minimize the numerical integration error. The stepsize $\triangle t$ can be determined by fixed integration grids. However, the truncation error (TE) may be unstable if the integration grids are not set properly. Hence, advanced ODE solvers often use adaptive stepsize instead of fixed stepsize to further improve the numerical integration accuracy as shown in Figure 6.2. However, adaptive stepsize requires a stepping algorithm to estimate the TE iteratively until reaching the optimal stepsize, incurring extra computations and longer latency. From hardware perspective, a reconfigurable architecture is needed to support the multitude of NN configurations and ODE solver choices.

The adaptive stepsize Runge-Kutta (RK) method is by far the most commonly used ODE solver in NODE. For completeness, we briefly introduce its numerical

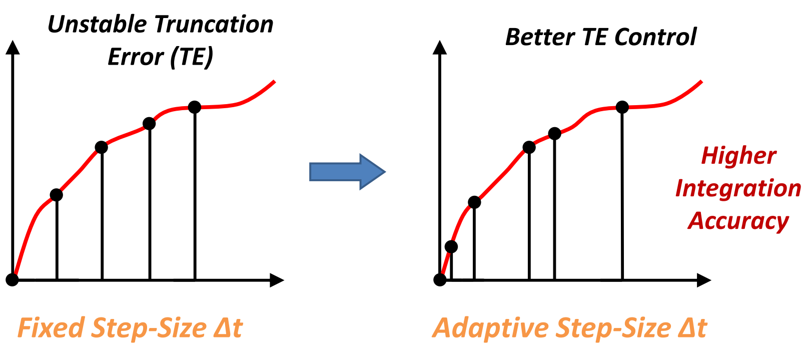
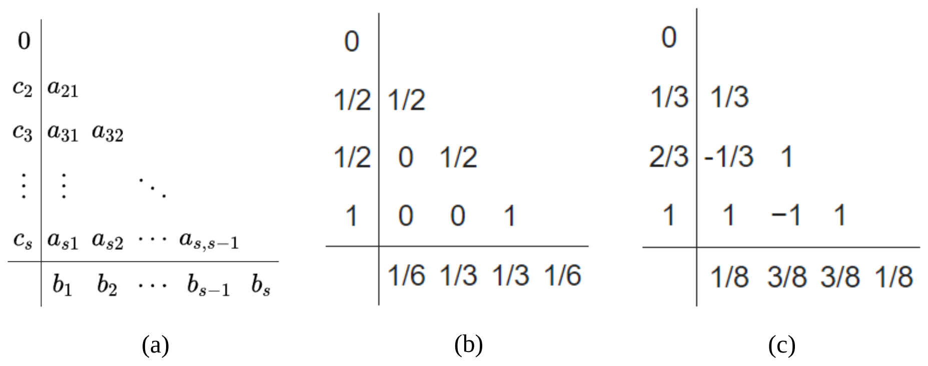


Figure 6.2: Fixed and adaptive stepsize integration.



(a)

$$\begin{array}{c|ccccc} 0 & & & & & \\ c_2 & a_{21} & & & & \\ c_3 & a_{31} & a_{32} & & & \\ \vdots & \vdots & & \ddots & & \\ c_s & a_{s1} & a_{s2} & \cdots & a_{s,s-1} & \\ \hline & b_1 & b_2 & \cdots & b_{s-1} & b_s \end{array}$$

(b)

$$\begin{array}{c|cccc} 0 & & & & \\ 1/2 & 1/2 & & & \\ 1/2 & 0 & 1/2 & & \\ 1 & 0 & 0 & 1 & \\ \hline & 1/6 & 1/3 & 1/3 & 1/6 \end{array}$$

(c)

$$\begin{array}{c|cccc} 0 & & & & \\ 1/3 & 1/3 & & & \\ 2/3 & -1/3 & 1 & & \\ 1 & 1 & -1 & 1 & \\ \hline & 1/8 & 3/8 & 3/8 & 1/8 \end{array}$$

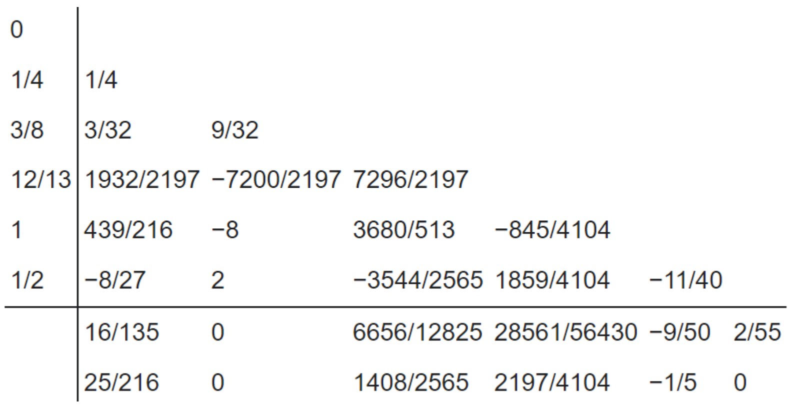
Figure 6.3: Butcher tableau for: (a) s-th order RK method, (b) 4-th order RK method, and (c) 4-th RK method with 3/8 rule.

integration procedure as (6.3):

$$\begin{aligned} \frac{\mathrm{d}h(t)}{\mathrm{d}t} &= f(t, h(t)) \\ h(t+\triangle t) &= h(t) + \triangle t \sum_{i=1}^{s} b_i k_i \\ k_i &= f(t + c_i\triangle t, h(t) + \triangle t \sum_{j=1}^{i-1} a_{ij} k_j) \end{aligned} \tag{6.3}$$

where s specifies a particular order of RK method and the coefficients are usually arranged in a mnemonic device known as a Butcher tableau as in Figure 6.3.

The stepsize $\triangle t$ in (6.3) can be determined by adaptive stepping algorithms. The most widely used adaptive stepping algorithm for RK method was proposed by



0						
1/4	1/4					
3/8	3/32	9/32				
12/13	1932/2197	−7200/2197	7296/2197			
1	439/216	−8	3680/513	−845/4104		
1/2	−8/27	2	−3544/2565	1859/4104	−11/40	
	16/135	0	6656/12825	28561/56430	−9/50	2/55
	25/216	0	1408/2565	2197/4104	−1/5	0

Figure 6.4: Butcher tableau for adaptive stepsize RK-4 method.

Fehlberg [137] in 1969. By performing one extra calculation for error estimation, the integration error can be controlled automatically. For the adaptive-stepsize RK-4 method, the Butcher tableau is given below in Figure 6.4 (detailed derivations can be found in [137]). The first row of coefficients at the bottom of the table gives the 5-th order accurate method and the second row gives the 4-th order accurate method.

The optimal stepsize is searched and updated as following:

1) k_i's are computed based on the adaptive stepsize Butcher tableau (e.g., Figure 6.4 for adaptive stepsize RK-4);

2) the weighted average is derived as (6.4):

$$h(t + \triangle t) = h(t) + \sum_{i=1}^{s} C(i) k_i \tag{6.4}$$

where $C(i)$ is the weights of k_i that are derived based on RK order s. Readers can refer to [137] for detailed derivations;

3) the estimation of the truncation error (TE) is then computed by (6.5):

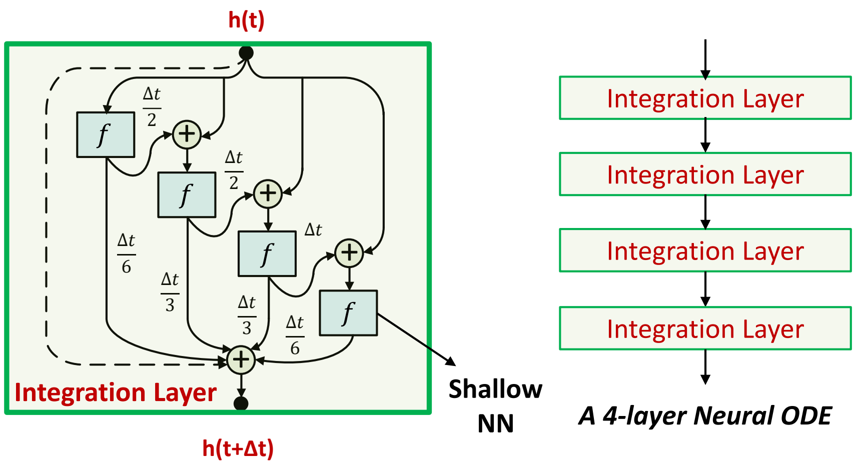


Figure 6.5: A 4-layer neural ODE with classical RK-4 integration layer.

.

$$TE = |\sum_{i=1}^{s} C(i)k_i| \tag{6.5}$$

4) at the completion of the step, a new stepsize can be calculated by (6.6):

$$\triangle t_{new} = 0.9 \cdot \triangle t \cdot (\frac{\epsilon}{TE})^{0.2} \tag{6.6}$$

if $TE > \epsilon$, the step is repeated with the new stepsize $\triangle t_{new}$; otherwise, the step is completed and $\triangle t_{new}$ is used for the next step.

6.1.2 Neural ODE Inference

Suppose a numerical integration step from $h(t)$ to $h(t+\triangle t)$ is called an integration layer, Figure 6.5 demonstrates a 4-layer neural ODE with RK-4 integration layer. The embedded f function is usually a shallow NN as shown in Figure 6.1(a). We can see that neural ODE inference can be viewed as multiple stages of NN inferences with more complex connections, which can be accelerated by state-of-the-art NN accelerators through proper scheduling. Hence, we do not focus on accelerating NODE inference in this work.

6.1.3 Neural ODE Training

Training a neural ODE involves reverse propagation through numerical integration layers. For completeness, we briefly introduce the training method [11].

6.1.3.1 Reverse Propagation through Numerical Integrator

The main technical difficulty for NODE is performing reverse propagation for gradient estimation through the ODE solver during training. The training process of NODE can be formulated as an optimization problem in (6.7):

$$\operatorname*{argmin}_{\theta} L(h(T), y) \tag{6.7}$$

where L is the loss function and y is the target output. To optimize L, we require gradients with respect to θ. The first step is to determine how the gradient of the loss depends on the hidden state $h(t)$ at each instant. This quantity is called the *adjoint* $a(t)$ as in (6.8). Its dynamics are given by another ODE, which can be thought of as the instantaneous analog of the chain rule as (6.8):

$$\frac{\mathrm{d}a(t)}{\mathrm{d}t} = -a(t)^T \frac{\partial f(h(t), t, \theta)}{\partial h(t)},\ a(T) = \frac{\partial L}{\partial h(T)} \tag{6.8}$$

The analytical form of the loss function gradient in the continuous case can be derived as (6.9):

$$\frac{\mathrm{d}L}{\mathrm{d}\theta} = -\int_{T}^{0} a(t)^T \frac{\partial f(h(t), t, \theta)}{\partial \theta} dt \tag{6.9}$$

Detailed proofs can be found in [11, 138]. NODE training operations can be summarized as four steps:

1. Solve $h(t)$ in time $0 \to T$.

2. Determine adjoint $a(T)$ with the boundary condition at time T in (6.8).

3. Solve $a(t)$ in time $T \to 0$ following (6.8) and boundary condition $a(T)$.

4. Calculate parameter gradients by (6.9) and update network parameters.

Note that in order to calculate (6.9), $a(t)$ and $h(t)$ are required for every t. Since $a(t)$ and $h(t)$ are solved in the opposite directions, we need to either memorize $h(t)$, or find a method to recover $h(t)$. (6.9) is the analytical form and needs to be numerically calculated in practice. Figure 6.6 demonstrates the training dataflow for a 2-layer NODE.

6.1.3.2 Numerical Implementations for Analytical Form

There are several numerical implementations [11, 138, 139, 140] in the literature for the analytical form $\frac{\mathrm{d}L}{\mathrm{d}\theta}$ in Section 6.1.3.1. Forward-pass is similar for different methods; however, the backward-pass are different. We categorize these methods as below.

Baseline Method The baseline method [11] directly back-propagate through the forward-pass trajectory using a numerical ODE solver, as shown in Figure 6.7. It saves all the computation graph (including search for optimal stepsize) in memory. The memory cost and depth are $O(N_f \times N_t \times m)$. The computation cost is doubled considering both forward and reverse passes.

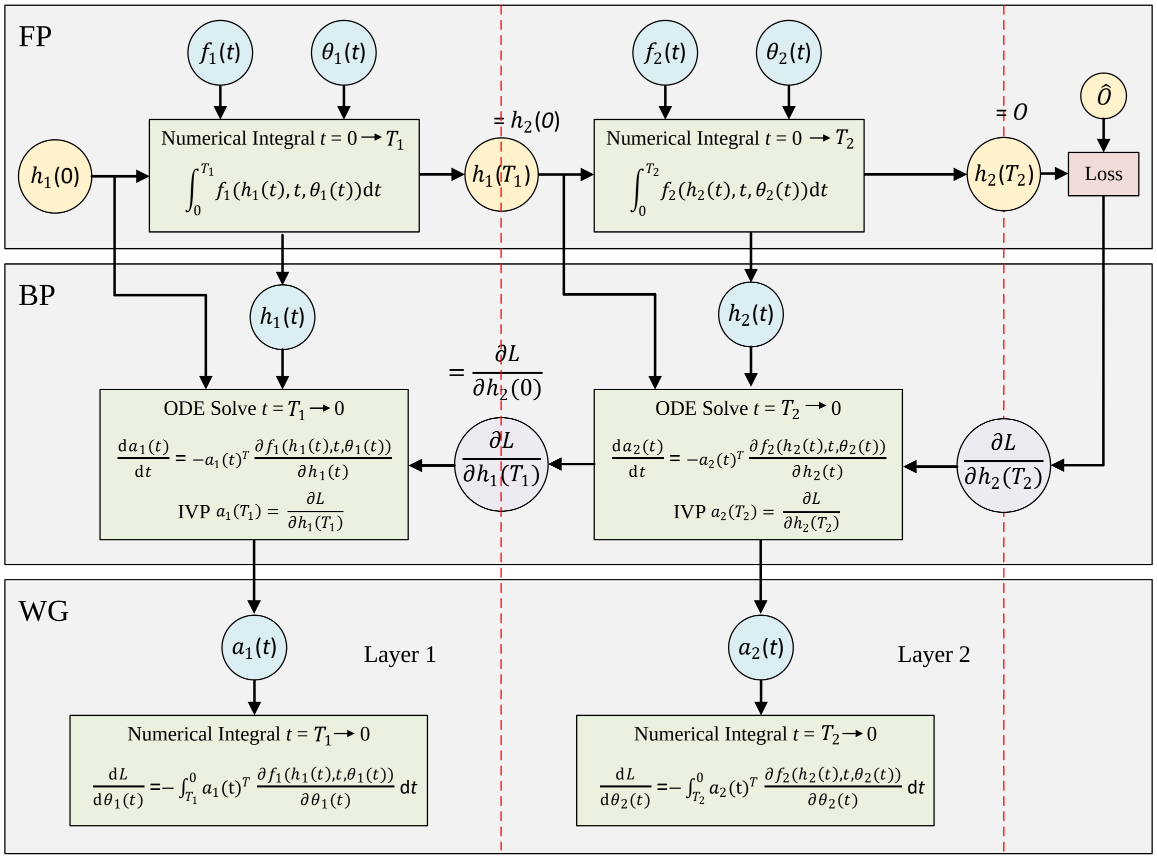

FP
$f_1(t)$
$\theta_1(t)$
$f_2(t)$
$\theta_2(t)$
$\hat{O}$
$= h_2(0)$
$= O$
$h_1(0)$
Numerical Integral $t = 0 \to T_1$
$\int_0^{T_1} f_1(h_1(t), t, \theta_1(t))\mathrm{d}t$
$h_1(T_1)$
Numerical Integral $t = 0 \to T_2$
$\int_0^{T_2} f_2(h_2(t), t, \theta_2(t))\mathrm{d}t$
$h_2(T_2)$
Loss
BP
$h_1(t)$
$h_2(t)$
$= \frac{\partial L}{\partial h_2(0)}$
ODE Solve $t = T_1 \to 0$
$\frac{\mathrm{d}a_1(t)}{\mathrm{d}t} = -a_1(t)^T \frac{\partial f_1(h_1(t),t,\theta_1(t))}{\partial h_1(t)}$
IVP $a_1(T_1) = \frac{\partial L}{\partial h_1(T_1)}$
$\frac{\partial L}{\partial h_1(T_1)}$
ODE Solve $t = T_2 \to 0$
$\frac{\mathrm{d}a_2(t)}{\mathrm{d}t} = -a_2(t)^T \frac{\partial f_2(h_2(t),t,\theta_2(t))}{\partial h_2(t)}$
IVP $a_2(T_2) = \frac{\partial L}{\partial h_2(T_2)}$
$\frac{\partial L}{\partial h_2(T_2)}$
WG
$a_1(t)$
Layer 1
$a_2(t)$
Layer 2
Numerical Integral $t = T_1 \to 0$
$\frac{\mathrm{d}L}{\mathrm{d}\theta_1(t)} = -\int_{T_1}^0 a_1(t)^T \frac{\partial f_1(h_1(t),t,\theta_1(t))}{\partial \theta_1(t)}\mathrm{d}t$
Numerical Integral $t = T_2 \to 0$
$\frac{\mathrm{d}L}{\mathrm{d}\theta_2(t)} = -\int_{T_2}^0 a_2(t)^T \frac{\partial f_2(h_2(t),t,\theta_2(t))}{\partial \theta_2(t)}\mathrm{d}t$

Figure 6.6: Training dataflow for a 2-layer NODE.

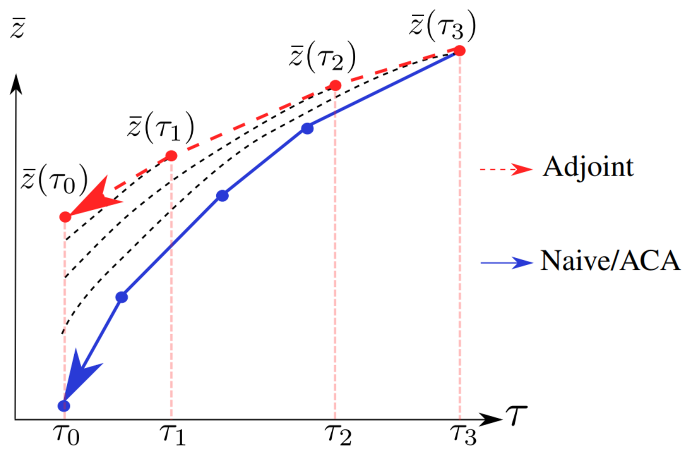


Figure 6.7: Backward-pass propagation. The blue curve is the same trajectory as in forward-pass. Both naive method and ACA method accurately recover the forward-pass trajectory, while adjoint method forgets the forward-pass trajectory.

Adjoint Method The adjoint method [11] forgets about forward-pass trajectory $h(t)$; instead, it remembers boundary condition $h(T)$ and $a(T)$, then solves $h(t)$ and $a(t)$ in reverse-time $T \to 0$ as a separate ODE. We use $\bar{h}$ to denote reverse-time solution. Because $a(t)$ and $\bar{h}(t)$ are solved in the same direction, the integration in Eq.(6.9) only records current values, achieving $O(N_f)$ memory cost. Since the adjoint method needs to solve $\bar{h}(t)$ in reverse-time, it requires extra $O(N_f \times N_r \times m)$ computation, so the total computation cost is $O(N_f \times (N_t + N_r) \times m)$. Note that $\bar{h}(t)$ is not the same as $h(t)$ due to numerical errors, the adjoint method will cause errors in gradient estimation.

Adaptive Checkpoint Adjoint (ACA) Method To solve the inaccuracy of adjoint method, the ACA method [138] stores forward-pass $h(t)$ in memory for backward-pass to avoid numerical errors, while also controlling the memory cost. ACA deletes the search process in baseline method and only back-propagates through the accepted steps, hence has a shallower computation graph $(N_f \times N_t)$. ACA only stores $h(t_i)_{i=1}^{N_t}$, and deletes the computation graph for $f(h(t_i), t_i)_{i=1}^{N_t}$, hence the memory cost is only $N_h(N_f + N_t)$. In this work, we adopt ACA method for NODE hardware acceleration.

6.2 Algorithm-Architecture Co-Design for NODE

We tackle hardware acceleration of NODE training through algorithm-hardware co-optimizations: 1) we explore the sparsity and early termination opportunities in NODE training algorithms, aiming to reduce the computational complexity while still maintaining excellent training accuracy; 2) we study state-of-the-art ODE solvers and propose a network-on-chip (NoC) architecture for NODE compute kernels with reconfigurable interconnects to handle a variety of ODE solvers; 3) we carry out software/hardware co-optimizations for further enhanced hardware efficiency with hardware reuse and hierarchical memory design.

6.2.1 Neural Activation Sparsity

Sparsification can lead to more efficient models that reduces the representational complexity using only a subset of the weight or activation. We study the sparsification where neural activations are set to zero during the forward integration and reverse propagation of training. We further observe that the iterative stepsize search accounts for majority of the latency in forward integration; hence, an early termination mechanism is proposed that can skip the unnecessary computations in error normalization.

The output activations of any RELU-based NN layer are naturally sparse due to the the property of RELU activation function. Intuitively, with random inputs, half of the output values of an NN layer would be 0's. The sparsity can be significantly higher than 50% with sparsification. State-of-the-art sparsification methods include 1) α-thresholding, where activations with magnitudes smaller than α are set to 0's, and 2) top-K sparsification, where neural activations with K largest magnitudes are kept and others are set to 0's. We select α-thresholding for NODE sparsification due

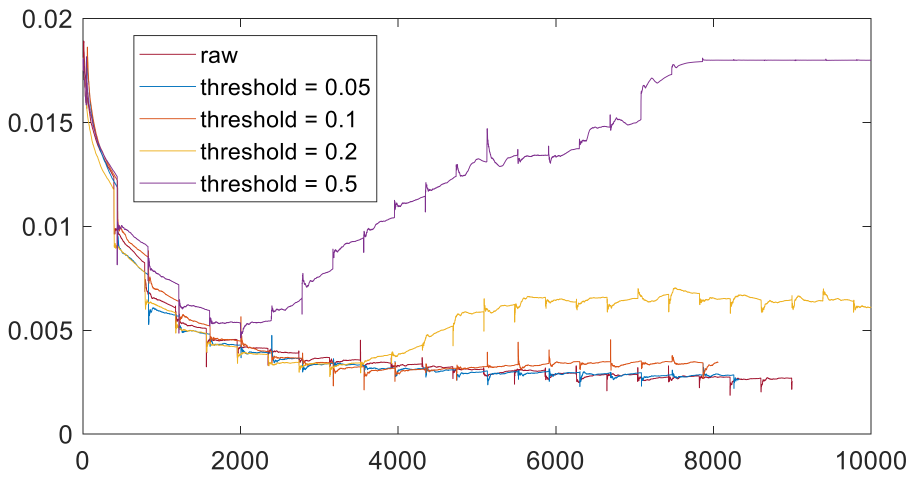


Figure 6.8: Training loss with a variety of fixed thresholding on activation when using a 4-layer NODE (with adaptive stepsize RK-4 and f function in Figure 6.1) on CIFAR-10 dataset.

to its simple hardware implementations without comparators or sorters.

We simulate a 4-layer NODE using adaptive stepsize RK-4 and f function in Figure 6.1 on a CIFAR-10 dataset. The training loss with a variety of fixed thresholding on output activations are demonstrated in Figure 6.8. We observe that fixed thresholding of $\alpha = 0.1$ have negligible degradation in training loss and the sparsity of neural activations can be as high as 80%, as shown in Figure 6.9. This presents an opportunity to greatly reduce the computational complexity and enables new hardware architecture for sparse activation training. The sparsity processor design is ongoing.

6.2.2 Top-Level Architecture[1] with Adaptive Stepsize ODE Solvers

To efficiently support adaptive stepsize RK ODE solvers, we propose a top-level architecture as shown in Figure 6.10. It utilizes a network-on-chip architecture with an array of NN cores to accelerate NODE's embedded NNs. The NoC routers can be configured during run-time to support a variety of ODE solvers. A stepsize engine is instantiated to compute the initial stepsize based on input activations, derive the output truncation error based on output activations, and compare the truncation error

[1]Acknowledgement to Junkang (Jerry) Zhu for his contributions

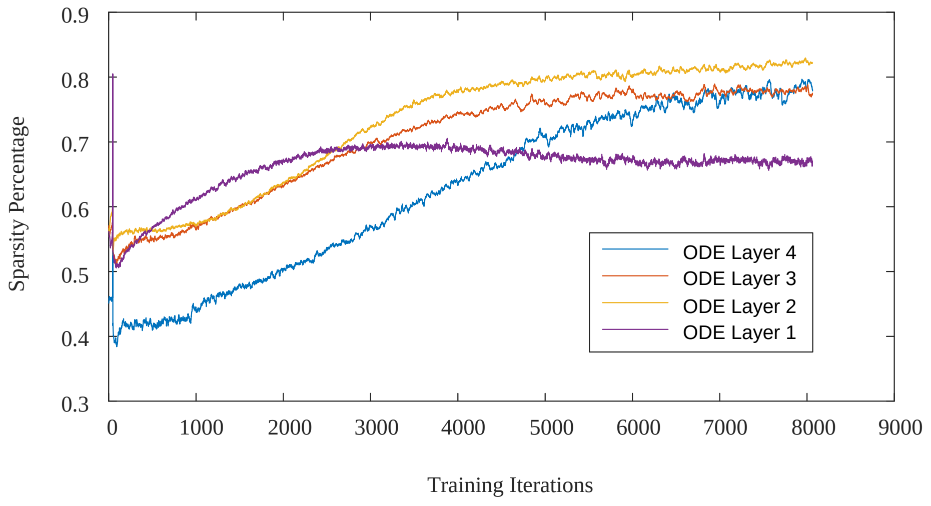


Figure 6.9: Sparsity percentage of each layer with a fixed thresholding of $\alpha = 0.1$ on neural activation when using a 4-layer NODE (with adaptive stepsize RK-4 and f function in Figure 6.1) on CIFAR-10 dataset.

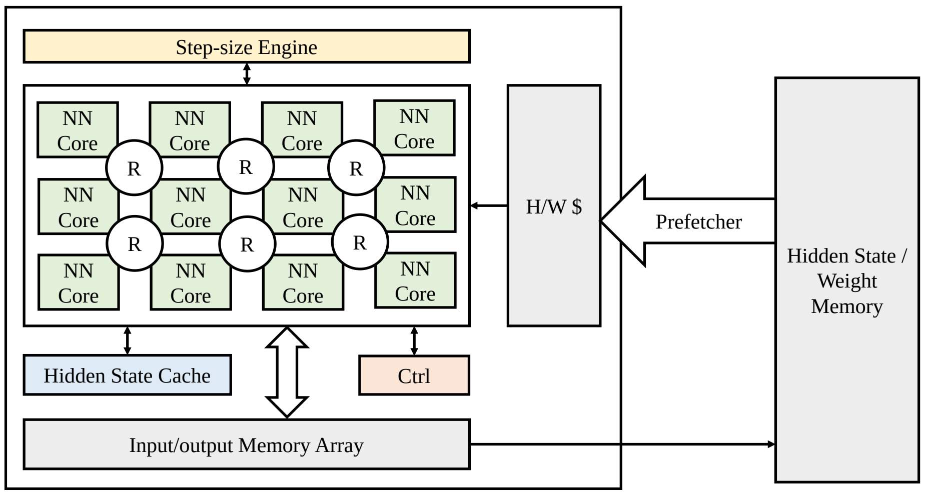


Figure 6.10: Top-level architecture for accelerating NODE with adaptive stepsize RK methods.

to a given ϵ. It consists of an error norm calculator with early termination: once the accumulated truncation error $TE > \epsilon$, the error norm computations terminate and an updated stepsize is calculated based on (6.6).

The hidden states and trained weights of forward integration steps are stored in a main memory and are pre-fetched to an hidden state and weight cache (denoted by H/W \$ in Figure 6.10). At the beginning of a step, the hidden states and weights are loaded to corresponding NN cores for inference. Upon completion of each step, the H/W cache are updated based on the output activations and newly trained weights. Moreover, the adaptive stepsize ODE solvers introduce stepsize searching iterations within a numerical integration step: the intermediate activations from completed k's are stored in a hidden state cache that can be frequently written to and read from for computations of downstream k's.

The architecture of NN core is illustrated in Figure 6.11. It consists of a conv core, a norm core, and a ReLU core with corresponding programmable LUTs and local buffers (denoted by P and B in Figure 6.11, respectively). The programmable LUTs are used to configure NN layers (for embedded f in Figure 6.5). The local buffers are used to store intermediate activations of NN layers. A scratchpad memory is instantiated to store input/output activations as well as weights in each NN core. The NN core also includes a pre-fetcher/sender block to transfer data from/to caches or other NN cores. Note that a sparsity processor is implemented to exploit the sparsity of activations and weights for reduced hardware complexity.

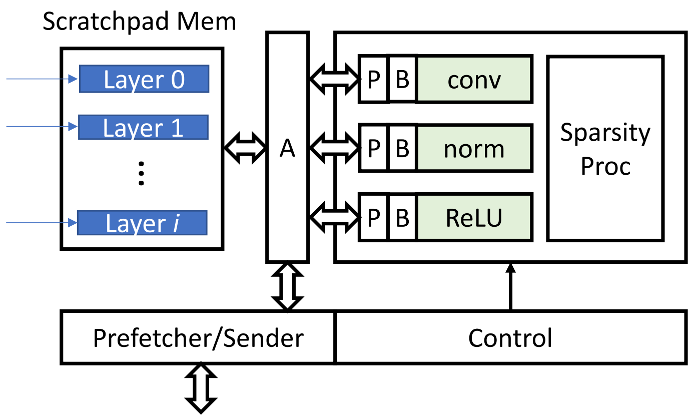


Figure 6.11: NN core architecture for accelerating f function in NODE.

.

CHAPTER VII

DNC-Aided SCL Flip Decoding of Polar Codes

Capacity-achieving polar codes [116] have been adopted in modern communication systems such as 5th generation (5G) wireless standard. They can be decoded sequentially on a trellis using successive cancellation list (SCL) [117] decoder. Upon receiving log-likelihood ratios (LLRs), SCL calculates path metrics (PMs) following a bit after bit order. A list of L most likely paths are kept during decoding and decoded bits are determined by the most likely path that passes cyclic redundancy check (CRC). However, the decoding performance is not very satisfactory with moderate code length N. Once wrong bit decisions occur on the trellis, they have no chance to be corrected due to the sequential decoding order.

To solve this problem, flip algorithms are used when standard decoding fails with CRC. Error bit positions are searched and flipped in subsequent decoding attempts. Clearly, the key for successful flip decoding is to accurately identify error bit positions. As shown in Figure 7.1, heuristic methods [141, 142, 143, 144, 145, 146, 147, 148, 149, 150, 151, 152, 153, 154] use explicit mathematical metric to estimate the likelihood of each bit being an error bit. The likelihoods are sorted to obtain the flip position set.

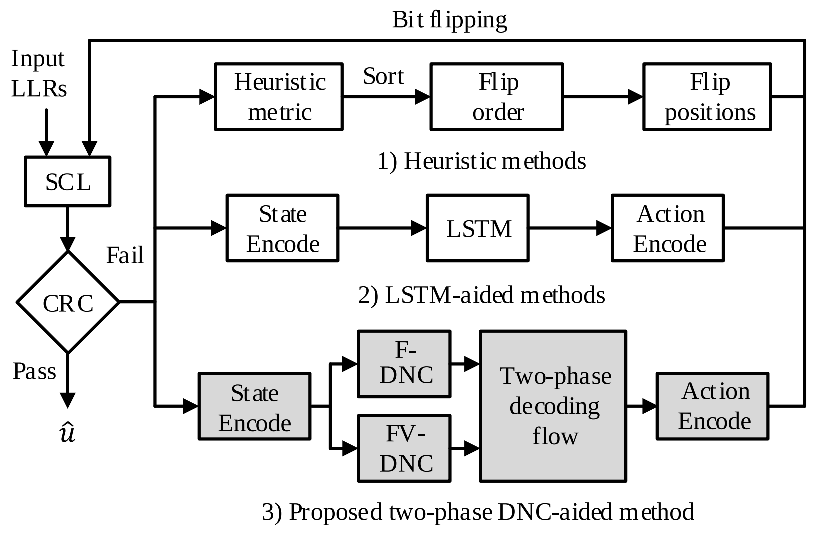


Figure 7.1: Overview of 1) Heuristic bit flipping, 2) LSTM-aided bit flipping and 3) proposed DNC-aided two-phase bit flipping.

7.1 State-of-the-art Flip Algorithms

Heuristic methods like [141, 142, 143, 144, 147] use received LLRs or their absolute values as the metric to derive flip positions. Specifically, [144] introduces a critical set to reduce the search space of flip positions for lower complexity. [147] subdivides the codeword into partitions, on which SC-Flip (SCF) is run for shorter latency. However, these methods can only flip one bit at a time. [148, 149, 152, 151] propose a dynamic SC-Flip (DSCF) that allows flipping of multiple bits at a time and improves the latency of SCF. Multi-bit flipping requires identifying multiple error bit positions concurrently. DSCF introduces a new metric considering not only received LLRs but also the trajectories in the sequential SCL decoding. [151, 152] introduce variations of DSCF to improve the accuracy of identifying error bit positions. [146, 154] extends the bit-flipping from SC to SCL for a SCL-Flip decoding (SCLF). Similarly, SCF is a special case of SCLF when $L = 1$.

However, the optimal flipping strategy is still an open problem to date. Recent works on flip algorithms involve deep learning (DL). Recently developed DL-aided SCF/SCLF [155, 156, 145, 157] exploit a trained LSTM to locate error bit posi-

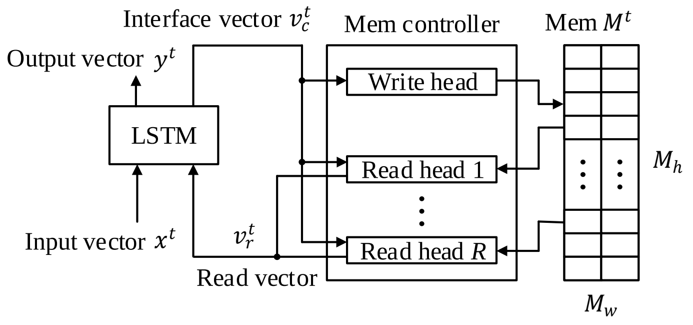


Figure 7.2: Top-level architecture of DNC.

tions instead of heuristic metric. They have shown slightly better performance than heuristic methods for short polar codes of length 64 or 128. However, the accuracy of identifying error bit positions is limited by the scalability of LSTMs when code length increases. On the other hand, state-of-the-art LSTM methods use simple state and action encoding that do not support multi-bit flipping efficiently, resulting in more decoding attempts compared to heuristic methods.

7.2 Differentiable Neural Computer (DNC)

DNC addresses LSTM's scalability problem with help of an external memory. Since its invention, DNC has found many applications like question answering [158, 159]. DNC can be considered as an LSTM augmented with an external memory through soft read and write heads, as shown in Figure 7.2. In this work, we use DNCs to enhance the accuracy of identifying error bit positions.

A top level architecture of DNC is demonstrated in Figure 7.2. DNC periodically receives x^t as input vector and produces y^t as output vector at time t. The output vector y^t is usually made into a probability distribution using softmax. At time t, the DNC 1) reads an input x^t, 2) writes the new information into the external memory using interface vector v_c^t through memory controller, 3) reads the updated memory M^t and 4) produces an output y^t. Assume the external memory is a matrix of M_h

slots, each slot is a length-M_w vector. To interface with this external memory, DNC computes read and write keys to locate slots. The memory slot is found using similarity between key and slot content. This mechanism is known as the content-based addressing. In addition, DNC also uses dynamic memory allocation and temporal memory linkage mechanisms for computing write and read weights. We omit the mathematical descriptions of DNC here and readers can refer to [160] for more details.

7.3 DNC-Aided Flip Decoding

Bit-flipping can be modeled as a game and the DNC is the player to identify flip positions towards successful decoding. Upon CRC failure, the DNC player needs to take an action based on current state, either reverting falsely flipped positions or adding more flip positions. The proposed DNC-aided method includes: 1) new state and action encoding; and 2) a DNC-aided two-phase decoding flow.

7.3.1 State and Action Encoding

One of the keys for efficient DNC inference is to design good input (state) and output (action) vector for training and inference. We discuss the encoding of existing LSTM-based approaches [155, 145, 156, 157] and present a new encoding scheme.

7.3.1.1 State Encoding

a straightforward way to encode states is to directly use the received LLR sequence r_0^{N-1}. [155, 145] use the amplitudes of received LLRs as the LSTM input. [156] uses the amplitudes of received LLRs combining the syndromes generated by CRC for state encoding. However, path metric information in sequential decoding are discarded in these methods, resulting in a loss in representing error path selection probability. [157] proposed a state encoding by taking the PM ratio of discarded paths and survival paths. However, this representation requires extra computations for PM summations at each bit position and does not include received LLR information.

In this work, we introduce a new state encoding scheme using the gradients of L survival paths concatenated with received LLRs. It takes both PMs and received LLRs into consideration. The PM gradients $\nabla\mathcal{P}(\ell)_i$ for i-th bit can be described as (7.1):

$$\nabla\mathcal{P}(\ell)_i = \ln(1 + e^{-(1-2\hat{u}_i(\ell))L^{\hat{u}_i}(\ell)}) \tag{7.1}$$

Note that $\nabla\mathcal{P}(\ell)_i$ can be directly taken from existed PM calculations in standard SCL without extra computations. The state encoding S is therefore a vector as (7.2) and is used as DNC input in this work.

$$S = \{\nabla\mathcal{P}(\ell)_0^{N-1}, r_0^{N-1}\} \tag{7.2}$$

7.3.1.2 Action Encoding

the one-hot scheme used in state-of-the-art LSTM-based flip algorithms are efficient in identifying the first error bit, but lacks the capability to flip multiple bits at a time. This results in more decoding attempts. To improve bit flipping efficiency, we propose a soft multi-hot (i.e. ω-hot) flip vector v_f to encode both first error bit and subsequent error bits, aiming to correctly flip multiple bits in one attempt. v_f is a length-N vector that has ω non-zero entries. An action is therefore encoded by v_f. Each possible flip position in v_f is a non-zero soft value indicating the flip likelihood of the bit.

For training purpose, we introduce a scaled logarithmic series distribution (LSD) to assign flip likelihoods to the ω flip positions, where $p \in (0, 1)$ is a shape parameter of LSD. The intention is to create a distribution with descending probabilities for first error bit position and subsequent error bit positions and to provide enough likelihood differences between them. Suppose the k-th bit in polar code has an index $\mathcal{I}_{\mathcal{F}}(k)$ in the flip position set $\mathcal{F}$. Non-zero entries of v_f can be derived as (7.3):

$$\begin{gathered} v_f(k) = \mathcal{K} \frac{-1}{\ln(1-p)} \frac{p^{\mathcal{I}_{\mathcal{F}}(k)}}{\mathcal{I}_{\mathcal{F}}(k)} \text{ for } k \in \mathcal{F} \\ \text{where scaling factor } \mathcal{K} = 1 / \int_{\mathcal{F}} v_f \end{gathered} \tag{7.3}$$

Reference v_f generation for training are discussed in Section 7.4. The impacts of parameters ω and p on the accuracy of identifying error bit positions are discussed in Section 7.5.1..

7.3.2 DNC-Aided Two-Phase Decoding Flow

We design a two-phase flip decoding flow aiming to reduce the number of SCL attempts while achieving good error correction performance. The two phases in this flow are: i) multi-bit flipping and ii) successive flip decoding trials. In the first phase, the received symbols are first decoded with a standard decoder. If it fails CRC, a flip DNC (F-DNC) exploits the state encoding S to score the actions, i.e., estimate the probability of each bit being error bits and output a flip vector v_f. Figure 7.3 shows an example of $\omega = 3$ where $\mathcal{F} = \{7, 9, 2\}$ is the flip position set with descending likelihoods {0.4, 0.3, 0.1}. To avoid wrong flips of subsequent positions with insignificant flip likelihoods, an α-thresholding is applied to keep only positions with $v_f(i) > \alpha, i = \{0, ..., N-1\}$, for multi-bit flipping. A subsequent decode attempt is then carried out with multi-bit flipping of bit positions $\{7, 9\}$ in the example.

If CRC still fails after multi-bit flipping, we enter Phase-II that successively re-

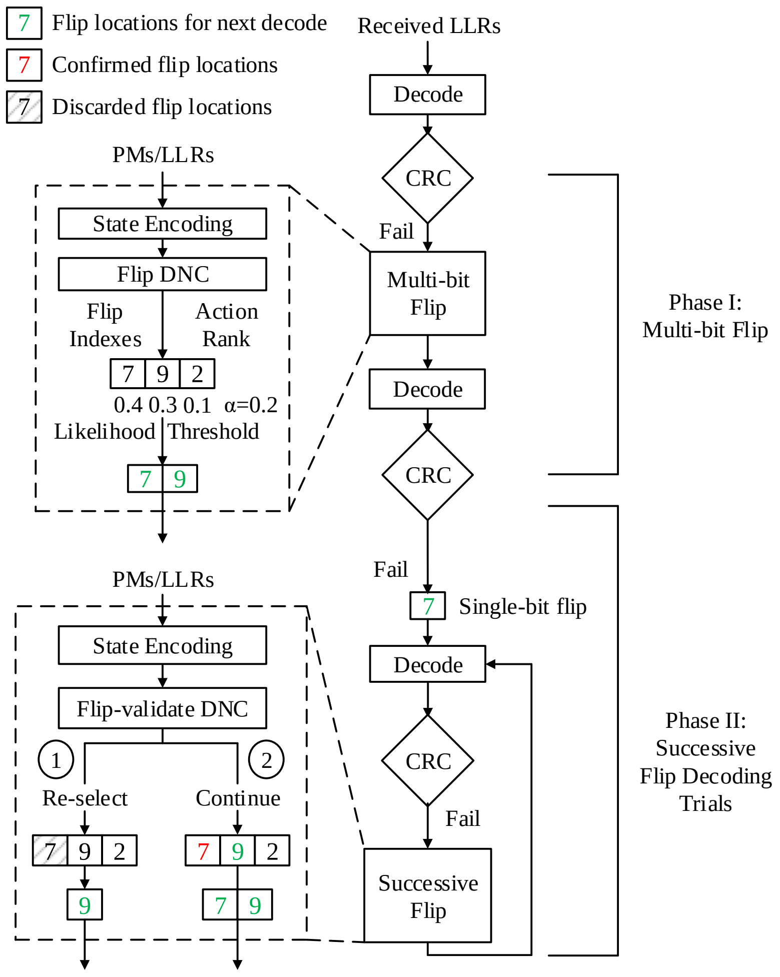


Figure 7.3: DNC-aided two-phase flip decoding ($\omega = 3$ case).

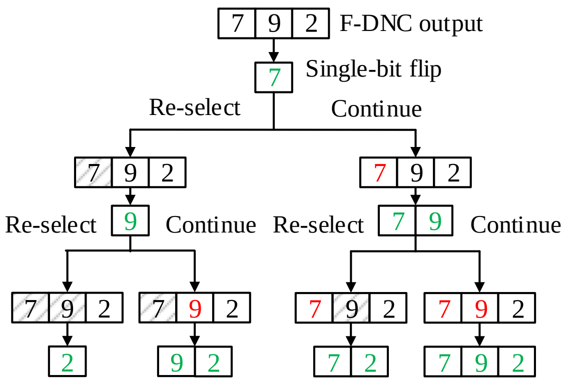


Figure 7.4: Flip attempts in Phase-II for different FV-DNC output combinations ($\omega = 3$ case).

select or confirm a single error bit position. The reasons of failed decoding in Phase-I are either: 1) first error bit position is wrong; or 2) first error bit position is right but some subsequent flip positions are wrong. Our proposed solution is to flip each possible error bit position one at a time and use a flip-validate DNC (FV-DNC) to confirm if this is a correct flip before moving to the next possible error bit position. The first attempt in Phase-II flips the highest ranked error bit position in $\mathcal{F}$, i.e., bit 7 in the example shown in Figure 7.3.

If FV-DNC invalidates the single-bit flip (bit 7 in this case), we discard bit 7 and re-select the flip position to next bit 9 in $\mathcal{F}$. Alternatively, if FV-DNC confirms the flip of bit 7, we continue by adding bit 9 into the flip queue $\mathcal{Q}_f$ and flip $\mathcal{Q}_f = \{7, 9\}$ in next attempt. The process runs successively until CRC passes or reaching the end of $\mathcal{F}$. Figure 7.4 shows all possible flip combinations given different FV-DNC output combinations in the $\omega = 3$ case. The number of decoding attempts of Phase-II is bounded by ω. The two-phase DNC-SCLF can be described as Algorithm 7.5.

7.4 Training Methodology

In this section, we discuss training for the DNCs used in proposed DNC-SCLF. The training is conducted off-line and does not increase the run-time decoding complexity.

```
**Algorithm 1:** DNC-Aided SCL-Flip Decoding
$\hat{u}_0^{N-1}, S \leftarrow$ SCL($r_0^{N-1}$)
**if** CRC($\hat{u}_0^{N-1}$) = pass **return** $\hat{u}_0^{N-1}$
**Phase-I**: Multi-bit Flipping
$\mathcal{F}, \omega, v_f \leftarrow$ F-DNC($S$)
$\hat{u}_0^{N-1} \leftarrow$ SCL($r_0^{N-1}, \mathcal{F}_{v_f \geq \alpha}$)
**if** CRC($\hat{u}_0^{N-1}$) = pass **return** $\hat{u}_0^{N-1}$
**Phase-II**: Successive Flip Decoding Trials
$\mathcal{Q}_f = \{\mathcal{F}[0]\}$
**for** $i = 0, 1, ..., \omega - 1$ **do**
    $\hat{u}_0^{N-1}, S \leftarrow$ SCL($r_0^{N-1}, \mathcal{Q}_f$)
    **if** CRC($\hat{u}_0^{N-1}$) = pass or $i = \omega - 1$ **return** $\hat{u}_0^{N-1}$
    $\mathcal{R} \leftarrow$ FV-DNC($S$)
    **if** $\mathcal{R}$ = continue **then**
        $\mathcal{Q}_f = \{\mathcal{Q}_f, \mathcal{F}[i+1]\}$
    **else**
        $\mathcal{Q}_f$[end] $= \mathcal{F}[i+1]$
    **end**
**end**
```

Figure 7.5: Algorithm 7.3.2 DNC-Aided SCL-Flip Decoding

We adopt the cross-entropy function which has been widely used in classification tasks [161].

7.4.1 F-DNC Training

In the first training stage, we run extensive SCL decoder simulations and collect error frames upon CRC failure. The F-DNC training database consists of pairs of S from (7.2) as DNC input and a corresponding v_f from (7.3) as reference output. S can be straightforwardly derived based on received LLRs and PMs of collected error frames. However, v_f is determined by parameter ω and p, whose values will affect the training and inference efficiency. We first label the error bit positions w.r.t the transmitted sequence for each sample as candidate flip positions. Intuitively, small ω and p strengthen the likelihood of identifying first error bit position, but attenuate the likelihoods of subsequent error bit positions. Hence there is a trade-off between the accuracy of identifying first error bit position and the accuracy of identifying

Table 7.1: F-DNC/FV-DNC Hyper-parameters Set

Parameter	Description
LSTM controller	1 layer of size 128
Size of access heads	1 write head, 4 read heads
Size of external memory	$M_h = 256, M_w = 128$
Size of training set	10^6 for F-DNC, 3×10^7 for FV-DNC
Size of validation set	5×10^4
Mini-batch size	100
Dropout probability	0.05
Optimizer	Adam
Environment	Tensorflow 1.14.0 on Nvidia GTX 1080Ti

subsequent error bit positions. In this work, we carried out reference v_f generations with $\omega = \{2, 5, 10\}$ and $p = \{0.2, 0.8\}$. The experimental results with these parameter choices are discussed in Section 7.5.

7.4.2 FV-DNC Training

The error frames that can not be decoded correctly in Phase-I enter Phase-II, where single bit positions are flipped and tested successively as shown in Figure 7.4. This is to prevent wrong flips that will lead the DNC player into a trapping state and can never recover. The FV-DNC is a classifier taking either "re-select" or "continue" action given the knowledge of received LLRs and PMs from most recent attempt. The key for FV-DNC training is to create a well-categorized database that can detect trapping state effectively. We carry out supervised flip decoding attempts based on reference v_f in F-DNC database. For each collected error:1) the first 5 error bit positions in reference v_f are flipped bit after bit and their corresponding state encoding S are recorded. These samples result in a “continue” action. 2) After flipping each of the first 5 error bit positions, we flip 5 random positions and record their state encoding S. These samples indicate trapping state and result in a “re-

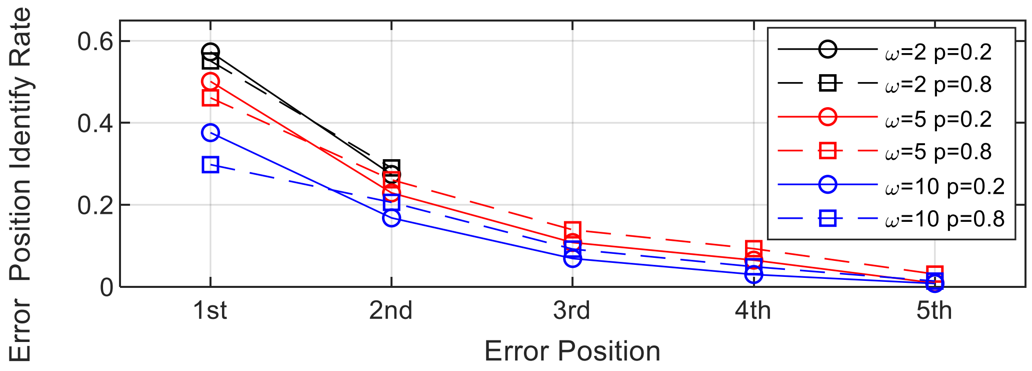


Figure 7.6: Rate of identifying error bit positions for $\omega = \{2, 5, 10\}$ and $p = \{0.2, 0.8\}$ for SC decoding of (256,128) polar code.

select" action. For each collected frame, we have 5 samples for "continue" action and 25 samples for "re-select" action.

7.5 Experiments and Analysis

To show the competitiveness of DNC in tackling long-distance dependencies in polar decoding trellis, we evaluate the performances for polar codes of length $N = 256$, 1024 with SC and SCL ($L = 4$). The code rate is set to 1/2 with an 16b CRC. Error frames are collected at SNR 2dB. In this paper we do not focus on the hyper-parameter optimizations for DNC and just demonstrate a set of configurations that work through our experiments for F-DNC and FV-DNC in Table 7.1.

7.5.1 Accuracy of Identifying Error Bits

Firstly, we study the impacts of parameters ω and p introduced in action encoding. For a fair comparison, we pick the same code length $N = 256$ and SC decoding used in heuristic method [149] and LSTM-based method [155]. Figure 7.6 presents the accuracy of identifying the first 5 error bit positions. For a given ω, a smaller p ($p = 0.2$) enhances the probability of identifying the first error bit position, but attenuates the probability of identifying subsequent error bit positions. We achieve a

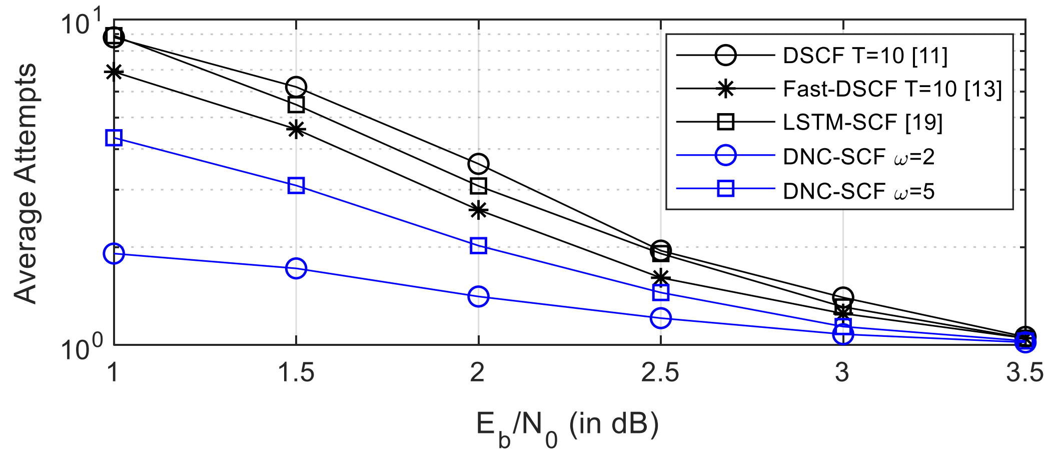


Figure 7.7: Number of extra decoding attempts of DNC-SCF and state-of-the-art flipping algorithms for (1024, 512) polar code.

0.573 success rate of identifying the first error bit position with $\omega = 2$, outperforming the 0.425 and 0.51 success rate with heuristic DSCF [149] and LSTM-aided SCF [155], respectively. Comparing $\omega = 2$ and $\omega = 5$ with same $p = 0.8$, a bigger ω helps to identify more error bit positions, but the success rates of identifying each position are degraded.

We pick $p = 0.8$ in our two-phase DNC-SCLF experiments to strengthen the success rates of identifying subsequent error bit positions and slightly sacrifice the success rate of identifying first error bit position. This is because with help of FV-DNC, even though F-DNC may not identify the first error bit position accurately in Phase-I, the two-phase decoding can re-select it in Phase-II. We use an $\alpha = 0.03$ for thresholding through our experiments.

7.5.2 Complexity and Latency

Metric calculation and sorting in heuristic methods can be implemented inside standard SC/SCL decoders. However, DL-aided algorithms introduce higher complexity and require an inference accelerator to interact with the decoder. We use GPU that achieves a speed of 1.7 ms/inference. For practical adoptions, a dedicated accelerator can be implemented for faster inference. Bit flipping is conditionally triggered when the standard decoder fails and the triggering rate is lower than the FER. DL-aided algorithms are more suitable for the low FER regime where the inference latency can be hidden behind successful decoding runs with help of LLR buffers. In this work we do not focus on the inference acceleration and LLR buffering strategy, but focus on the average number of flip decoding attempts that determines the overall latency.

Assume β_1 is the rate of successful decoding with multi-bit flipping in Phase-I, the average number of decoding attempts T_{avg} for a DNC-aided flip decoding can be calculated as (7.4):

$$T_{avg} = \beta_1 + \omega_{2,\text{avg}}(1 - \beta_1) \tag{7.4}$$

where $\omega_{2,\text{avg}}$ is the average number of attempts in Phase-II and $\omega_{2,\text{avg}} \leq \omega$. Figure 7.7 demonstrates the T_{avg} for proposed DNC-SCF and the state-of-the-art techniques. At a 2dB SNR, DNC-SCF with $\omega = 2$ improves the average decoding attempts by 45.7% and 54.2% compared to state-of-the-art heuristic [151] and LSTM-aided methods [156], respectively.

7.5.3 Error-Correction Performance

We compare coding gain of DNC-SCF at FER 10^{-4} with state-of-the-art heuristic methods [149, 151] and LSTM-based methods [156] for a (1024, 512) polar code and 16b CRC. DNC-SCF $\omega = 2$ achieves 0.5dB coding gain w.r.t SC decoder. Increasing ω to 5 provides another 0.31dB coding gain. DNC-SCF $\omega = 5$ also outperforms DSCF [149] or Fast-DSCF [151] with $T = 10$ by 0.03dB and 0.05dB, respectively, while reducing the number of decoding attempts by 45.7%. Further increasing ω to

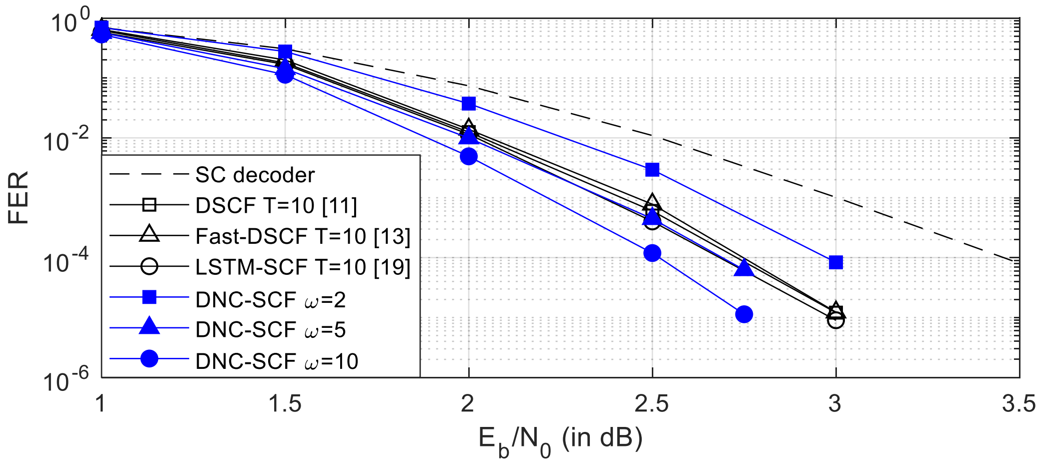


Figure 7.8: FER performance comparison of DNC-SCF and state-of-the-art flipping algorithms for (1024,512) polar code and 16b CRC.

DNC-SCF $\omega = 10$ provides 0.21dB coding gain compared to DSCF $T = 10$ while reducing the number of decoding attempts by 18.9%.

The LSTM-based approach in [155] does not report FER, but has shown up to 10% improvement in the accuracy of identifying first error bit position over DSCF with $T = 1$ at 1dB SNR for (64, 32) polar code. Another LSTM-based SCF [156] provides FER for (64, 32) polar code with $T = 6$ and claims 0.2dB improvement over DSCF $T = 6$. The FER of [156] with 1024b and $T = 10$ is shown in Figure 7.8, worse than DNC-SCF $\omega = 5$. LSTM's capability of identifying error bit positions gets weakened when code length increases.

We further compare the FER of DNC-SCLF ($L = 4$) on (256, 128) polar code and 16b CRC with state-of-the-art heuristic methods [146, 154] and LSTM-based approaches [145, 157] as shown in Figure 7.9. At FER 10^{-4}, DNC-SCLF $\omega = 2$ achieves a 0.27dB coding gain w.r.t standard SCL. Increasing ω to 5 results in 0.59dB coding gain from the standard SCL. DNC-SCLF $\omega = 5$ achieves 0.21dB and 0.01dB better performance than heuristic SCLF [154] and LSTM-SCLF [157] with $T = 10$, respectively. Further increasing ω to DNC-SCLF $\omega = 10$ improves the coding gain to 0.34dB and 0.16dB compared with [154] and [157], respectively.

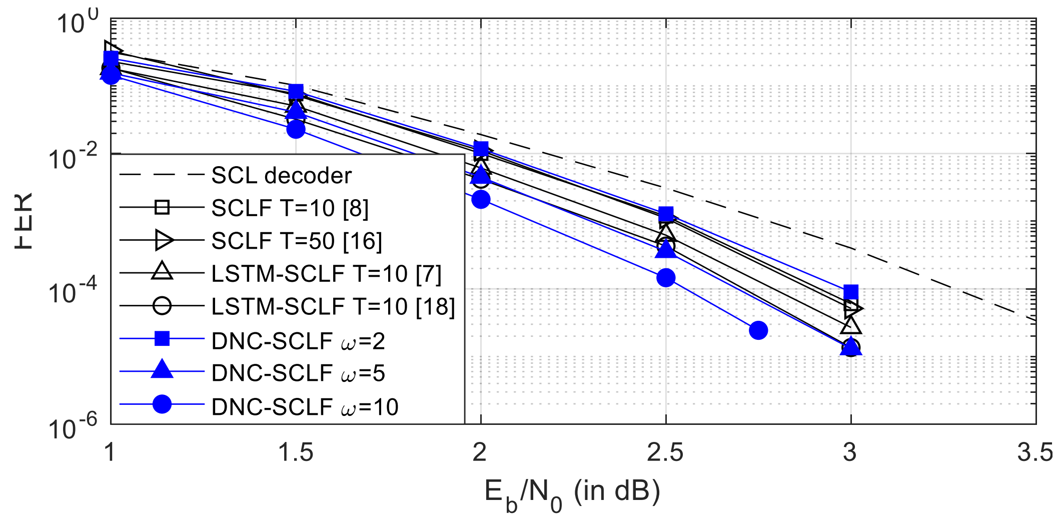


Figure 7.9: FER performance comparison of DNC-SCLF ($L = 4$) and state-of-the-art flipping algorithms for (256,128) polar code and 16b CRC.

7.6 Summary

In this paper, we present a new DNC-aided SCLF decoding. We propose a two-phase decoding assisted by two DNCs, F-DNC and FV-DNC, to identify error bit positions for multi-bit flipping and to re-select error bit positions for successive flip decoding trials, respectively. The multi-bit flipping reduces number of flip decoding attempts while successive flip decoding trials lowers the probability of going into trapping state. Training methods are proposed accordingly to efficiently train F-DNC and FV-DNC. Simulation results show that the proposed DNC-SCLF helps to identify error bits more accurately, achieving better error correction performance and reducing the number of flip decoding attempts than the the state-of-the-art flip algorithms. We plan to investigate the parameter optimizations for proposed DNC-SCLF in follow-up research.

CHAPTER VIII

Conclusion

The explosive growth of data and the needs for high-speed data communications and processing continuously drive the development of new hardware for transmitting more data reliably and processing more data to obtain a higher level of intelligence. Domain-specific communication and machine learning accelerators require algorithm-architecture co-design to derive efficient solutions for future applications. We have explored into the designs of: 1) channel decoders for polar codes, LDPC codes, and nonbinary LDPC codes; and 2) NN accelerators such as DNC and NODE. We also looked into interdisciplinary area of AI-aided communication, such as DNC-aided polar flip decoding.

Recently invented polar codes are the first provably capacity achieving code. We present a chip in a 40nm CMOS technology that implements a ST-SCL decoder for polar codes. In this design, a given polar code is split into 4 sub-codes and decoded separately with smaller sub-decoders followed by a reconciliation step in every decoding stage. Taking advantage of the under-utilized PEs in the sub-decoders, 8 frames are interleaved and decoded in parallel to achieve a high throughput and area efficiency. The decoder supports variable code lengths up to 1024b and variable code rates by programming the control LUTs. Per-block clock gating is implemented to further reduce the power consumption and improve the energy efficiency. The 0.64mm^2

test chip is measured to achieve a decoding throughput of 3.25Gb/s at 430MHz and the nominal supply voltage of 0.9V, consuming 13.17pJ/b, and it demonstrates a competitive error-correction performance. Voltage and frequency scaling of the chip to 0.6V and 100MHz further improves the energy efficiency to 7.4pJ/b at a reduced throughput of 760Mb/s.

Error floors of structured binary LDPC codes are caused by local minima due to non-codeword ETS and ETS-like errors. Inspired by simulated annealing, we design post-processing methods to perturb the local minimum state, followed by cooling to help decoding converge to the global minimum. The post-processing methods can be easily integrated as part of BP decoding, adding minimal overhead to the hardware implementation. As these methods are conditionally triggered when the decoder fails to converge at a very low BER level, the impact on decoding throughput and energy consumption is negligible. The success rate of resolving errors in the error-floor region is over 95% for ETS errors and over 80% for non-ETS errors for the IEEE 802.11n (1944,1620) LDPC code.

The NB-LDPC codes defined over GF(q) are extended from binary LDPC codes to improve error-correction performance. However, its decoding complexity is significantly higher. We present a fully parallel NB-LDPC decoder to take advantage of the low wiring overhead that is intrinsic to NB-LDPC codes. To further enhance the throughput, we apply a one-step look-ahead to the elementary CN design to reduce the clock period, and interleave the CN and VN operations. We implement a fine-grained clock gating at the node level to allow the majority of the processing nodes to be clock-gated long before reaching the iteration limit. A 7.04 mm^2 65 nm decoder test chip is designed for the GF(64) (160, 80) regular-(2, 4) NB-LDPC code. The decoder implements fine-grained dynamic clock gating and decoder termination to achieve a high throughput of 1.22 Gb/s at 700 MHz, consuming 3.03 nJ/b, or 259 pJ/b/iteration.

The recently developed differentiable neural computer (DNC) is a memory-augmented neural network (MANN) that has been shown to outperform in representing complicated data structures and learning long-term dependencies. DNC's higher performance is derived from new history-based attention mechanisms in addition to the previously used content-based attention mechanisms. History-based mechanisms require a variety of new compute primitives and state memories, which are not supported by existing neural network (NN) or MANN accelerators. We present HiMA, a tiled, history-based memory access engine with distributed memories in tiles. HiMA incorporates a multi-mode network-on-chip (NoC) to reduce the communication latency and improve scalability. An optimal submatrix-based memory partition strategy is applied to reduce the amount of NoC traffic; and a two-stage usage sort method leverages distributed tiles to improve computation speed. To make HiMA fundamentally scalable, we create a distributed version of DNC called DNC-D to allow almost all memory operations to be applied to local memories with trainable weighted summation to produce the global memory output. Two approximation techniques, usage skimming and softmax approximation, are proposed to further enhance hardware efficiency. HiMA prototypes are created in RTL and synthesized in a 40nm technology. By simulations, HiMA running DNC and DNC-D demonstrates 6.47× and 39.1× higher speed, 22.8× and 164.3× better area efficiency, and 6.1× and 61.2× better power efficiency over the state-of-the-art MANN accelerator. Compared to an Nvidia 3080Ti GPU, HiMA demonstrates speedup by up to 437× and 2,646× when running DNC and DNC-D, respectively.

The recently invented neural ordinary differential equations (NODE) demonstrate superior performance in modeling continuous-time events and normalizing flows. Their higher performance are derived by using numerical ODEs between NN's discrete hidden layers. To train NODE, state-of-the-art NN accelerators need to be attached to general-purpose CPUs or GPUs for elaborate ODE operations, which is unlikely to deliver a high efficiency. To address this, we present a programmable accelerator architecture for training NODE. We analyze the NODE training process, the neural activation sparsity, the data dependency and reuse patterns, and their impacts on hardware architecture design. By simulations, the sparsity operations based on adaptive thresholding during NODE training results in up to 80% neural activation sparsity, while still maintaining excellent training accuracy. We develop an accelerator architecture that employs an NoC architecture with distributed processing elements (PEs) and hierarchical memories, focusing on maximizing performance for elaborate NODE operations. The architecture supports reconfigurability for NODE's embedded NNs as well as a variety of adaptive step-size ODE solvers. The hardware efficiency can be enhanced with algorithm-architecture co-optimizations.

Applying DNC on polar decoding, we propose a new DNC-aided flip algorithm to improve the error-correction performance. Successive-cancellation list (SCL) decoding of polar codes has been adopted for 5G. However, the performance is not very satisfactory with moderate code length. Heuristic or deep-learning-aided (DL-aided) flip algorithms have been developed to tackle this problem. The key for successful flip decoding is to accurately identify error bit positions. New state and action encoding are developed for better DNC training and inference efficiency. The proposed method consists of two phases: i) a flip DNC (F-DNC) is exploited to rank the most likely flip positions for multi-bit flipping; ii) if decoding still fails, a flip-validate DNC (FV-DNC) is used to re-select error bit positions for successive flip decoding trials. Supervised training methods are designed accordingly for the two DNCs. Simulation results show that the proposed DNC-aided SCL-Flip (DNC-SCLF) decoding demonstrates up to 0.34dB coding gain improvement or 54.2% reduction in average number of decoding attempts compared to prior works.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] R. Gallager. Low-density parity-check codes. *IRE Transactions on Information Theory*, 8(1):21–28, 1962. doi: 10.1109/TIT.1962.1057683.

[2] M.C. Davey and D. MacKay. Low-density parity check codes over gf(q). *IEEE Communications Letters*, 2(6):165–167, 1998. doi: 10.1109/4234.681360.

[3] Erdal Arikan. Channel polarization: A method for constructing capacity-achieving codes for symmetric binary-input memoryless channels. *IEEE Transactions on information Theory*, 55(7):3051–3073, 2009.

[4] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25:1097–1105, 2012.

[5] Ronan Collobert, Jason Weston, Léon Bottou, Michael Karlen, Koray Kavukcuoglu, and Pavel Kuksa. Natural language processing (almost) from scratch. *Journal of machine learning research*, 12(ARTICLE):2493–2537, 2011.

[6] David E Rumelhart, Bernard Widrow, and Michael A Lehr. The basic ideas in neural networks. *Communications of the ACM*, 37(3):87–93, 1994.

[7] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[8] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[9] Alex Graves, Greg Wayne, and Ivo Danihelka. Neural turing machines. *arXiv preprint arXiv:1410.5401*, 2014.

[10] Alex Graves, Greg Wayne, Malcolm Reynolds, Tim Harley, Ivo Danihelka, Agnieszka Grabska-Barwińska, Sergio Gómez Colmenarejo, Edward Grefenstette, Tiago Ramalho, John Agapiou, et al. Hybrid computing using a neural network with dynamic external memory. *Nature*, 538(7626):471–476, 2016.

[11] Ricky TQ Chen, Yulia Rubanova, Jesse Bettencourt, and David Duvenaud. Neural ordinary differential equations. *arXiv preprint arXiv:1806.07366*, 2018.

[12] Erdal Arikan. A performance comparison of polar codes and reed-muller codes. *IEEE Communications Letters*, 12(6):447–449, 2008. doi: 10.1109/LCOMM.2008.080017.

[13] Ido Tal and Alexander Vardy. List decoding of polar codes. *IEEE Transactions on Information Theory*, 61(5):2213–2226, 2015.

[14] A. Balatsoukas-Stimming, M. B. Parizi, and A. Burg. Llr-based successive cancellation list decoding of polar codes. *IEEE Transactions on Signal Processing*, 63(19):5165–5179, Oct 2015. ISSN 1053-587X. doi: 10.1109/TSP.2015.2439211.

[15] B. Yuan and K. K. Parhi. Low-latency successive-cancellation list decoders for polar codes with multibit decision. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 23(10):2268–2280, Oct 2015. ISSN 1063-8210. doi: 10.1109/TVLSI.2014.2359793.

[16] J. Lin, C. Xiong, and Z. Yan. A high throughput list decoder architecture for polar codes. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 24(6):2378–2391, June 2016. ISSN 1063-8210. doi: 10.1109/TVLSI.2015.2499777.

[17] C. Xiong, J. Lin, and Z. Yan. A multimode area-efficient scl polar decoder. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 24(12):3499–3512, Dec 2016. ISSN 1063-8210. doi: 10.1109/TVLSI.2016.2557806.

[18] X. Liu, Q. Zhang, P. Qiu, J. Tong, H. Zhang, C. Zhao, and J. Wang. A 5.16gbps decoder asic for polar code in 16nm finfet. In *arXiv:1807.01451 [cs.IT]*, Aug 2018. doi: 10.1109/ISWCS.2018.8491225.

[19] R. Shrestha and A. Sahoo. High-speed and hardware-efficient successive cancellation polar-decoder. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 66(7):1144–1148, July 2019. ISSN 1558-3791. doi: 10.1109/TCSII.2018.2877140.

[20] H. Yoon, S. Hwang, and T. Kim. A 655mbps successive-cancellation decoder for a 1024-bit polar code in 180nm cmos. In *2018 IEEE Asian Solid-State Circuits Conference (A-SSCC)*, pages 281–284, Nov 2018. doi: 10.1109/ASSCC.2018.8579335.

[21] Y. Chen, W. Sun, C. Cheng, T. Tsai, Y. Ueng, and C. Yang. An integrated message-passing detector and decoder for polar-coded massive mu-mimo systems. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 66(3):1205–1218, March 2019. ISSN 1558-0806. doi: 10.1109/TCSI.2018.2879860.

[22] S. A. Hashemi, M. Mondelli, S. H. Hassani, C. Condo, R. L. Urbanke, and W. J. Gross. Decoder partitioning: Towards practical list decoding of polar codes. *IEEE Transactions on Communications*, 66(9):3749–3759, Sep. 2018. ISSN 0090-6778. doi: 10.1109/TCOMM.2018.2832207.

[23] M. Mousavi, Y. Fan, C. Tsui, J. Jin, B. Li, and H. Shen. Efficient partial-sum network architectures for list successive-cancellation decoding of polar codes. *IEEE Transactions on Signal Processing*, 66(14):3848–3858, July 2018. ISSN 1053-587X. doi: 10.1109/TSP.2018.2839586.

[24] C. Xiong, J. Lin, and Z. Yan. Symbol-decision successive cancellation list decoder for polar codes. *IEEE Transactions on Signal Processing*, 64(3):675–687, Feb 2016.

[25] S. A. Hashemi, C. Condo, and W. J. Gross. A fast polar code list decoder architecture based on sphere decoding. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 63(12):2368–2380, Dec 2016. ISSN 1549-8328. doi: 10.1109/TCSI.2016.2619324.

[26] S. A. Hashemi, C. Condo, and W. J. Gross. Fast and flexible successive-cancellation list decoders for polar codes. *IEEE Transactions on Signal Processing*, 65(21):5756–5769, Nov 2017. ISSN 1941-0476. doi: 10.1109/TSP.2017.2740204.

[27] S. A. Hashemi, C. Condo, M. Mondelli, and W. J. Gross. Rate-flexible fast polar decoders. *IEEE Transactions on Signal Processing*, 67(22):5689–5701, Nov 2019. ISSN 1941-0476. doi: 10.1109/TSP.2019.2944738.

[28] D. Kim and I. Park. A fast successive cancellation list decoder for polar codes with an early stopping criterion. *IEEE Transactions on Signal Processing*, 66 (18):4971–4979, Sep. 2018. ISSN 1941-0476. doi: 10.1109/TSP.2018.2864580.

[29] P. Giard, A. Balatsoukas-Stimming, T. C. Müller, A. Bonetti, C. Thibeault, W. J. Gross, P. Flatresse, and A. Burg. Polarbear: A 28-nm fd-soi asic for decoding of polar codes. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 7(4):616–629, Dec 2017. ISSN 2156-3357. doi: 10.1109/JETCAS.2017.2745704.

[30] Marc PC Fossorier, Miodrag Mihaljevic, and Hideki Imai. Reduced complexity iterative decoding of low-density parity check codes based on belief propagation. *IEEE Transactions on communications*, 47(5):673–680, 1999.

[31] IEEE standard for local and metropolitan area networks part 16: Air interface for fixed and mobile broadband wireless access systems amendment 2: Physical and medium access control layers for combined fixed and mobile operation in licensed bands and corrigendum 1. *IEEE Std 802.16e-2005 and IEEE Std 802.16-2004/Cor 1-2005*, pages 1–822, 2006.

[32] IEEE standard for information technology– local and metropolitan area networks– specific requirements– part 11: Wireless LAN medium access control (MAC)and physical layer (PHY) specifications amendment 5: Enhancements for higher throughput. *IEEE Std 802.11n-2009*, pages 1–565, Oct 2009.

[33] IEEE standard for information technology–telecommunications and information exchange between systems–local and metropolitan area networks–specific requirements-part 11: Wireless LAN medium access control (MAC) and physical layer (PHY) specifications amendment 3: Enhancements for very high throughput in the 60 ghz band. *IEEE Std 802.11ad-2012*, pages 1–628, Dec 2012.

[34] IEEE standard for information technology-telecommunications and information exchange between systems-local and metropolitan area networks-specific requirements part 3: Carrier sense multiple access with collision detection (CSMA/CD) access method and physical layer specifications. *IEEE Std 802.3an-2006*, pages 1–167, 2006.

[35] Tom Richardson. Error floors of LDPC codes. In *Proc. Annu. Allerton Conf. Commun. Control and Computing*, volume 41, pages 1426–1435, 2003.

[36] Zhengya Zhang, Lara Dolecek, Borivoje Nikolic, Venkat Anantharam, and M Wainwright. Design of LDPC decoders for improved low error rate performance: quantization and algorithm choices. *IEEE Trans. Commun.*, 57(11): 3258–3268, 2009.

[37] David JC MacKay and Michael S Postol. Weaknesses of Margulis and Ramanujan-Margulis low-density parity-check codes. *Electron. Notes in Theoretical Comput. Sci.*, 74:97–104, 2003.

[38] Stefan Landner and Olgica Milenkovic. Algorithmic and combinatorial analysis of trapping sets in structured LDPC codes. In *Int. Conf. Wireless Networks, Commun. and Mobile Computing*, volume 1, pages 630–635. IEEE, 2005.

[39] Tao Tian, Christopher R Jones, John D Villasenor, and Richard D Wesel. Selective avoidance of cycles in irregular LDPC code construction. *IEEE Trans. Commun.*, 52(8):1242–1247, 2004.

[40] Hua Xiao and Amir H Banihashemi. Improved progressive-edge-growth(PEG) construction of irregular LDPC codes. *IEEE Commun. Lett.*, 8(12):715–717, 2004.

[41] Gianluigi Liva, William E Ryan, and Marco Chiani. Quasi-cyclic generalized LDPC codes with low error floors. *IEEE Trans. Commun.*, 56(1):49–57, 2008.

[42] Reza Asvadi, Amir H Banihashemi, and Mahmoud Ahmadian-Attari. Lowering the error floor of LDPC codes using cyclic liftings. *IEEE. Trans. Inf. Theory*, 57(4):2213–2224, 2011.

[43] G. Spourlis, I. Tsatsaragkos, N. Kanistras, and V. Paliouras. Error floor compensation for ldpc codes using concatenated schemes. In *Signal Processing Systems (SiPS), 2012 IEEE Workshop on*, pages 155–160, Oct 2012.

[44] S. Shieh. Concatenated bch and ldpc coding scheme with iterative decoding algorithm for flash memory. *Communications Letters, IEEE*, PP(99):1–1, 2015.

[45] K.T. Sarika and P.P. Deepthi. A novel high speed communication system based on the concatenation of rs and qc-ldpc codes. In *Emerging Research Areas and 2013 International Conference on Microelectronics, Communications and Renewable Energy (AICERA/ICMiCR), 2013 Annual International Conference on*, pages 1–5, June 2013.

[46] Chad A Cole, SG Wilson, EK Hall, and Thomas R Giallorenzi. Analysis and design of moderate length regular LDPC codes with low error floors. In *Annu. Conf. Inf. Sci. and Syst.*, pages 823–828. IEEE, 2006.

[47] Andres I Vila Casado, Miguel Griot, and Richard D Wesel. LDPC decoders with informed dynamic scheduling. *IEEE Trans. Commun.*, 58(12):3470–3479, 2010.

[48] Xiaoheng Chen, Jingyu Kang, Shu Lin, and Venkatesh Akella. Hardware implementation of a backtracking-based reconfigurable decoder for lowering the error floor of quasi-cyclic ldpc codes. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 58(12):2931–2943, 2011.

[49] Huang-Chang Lee and Yeong-Luh Ueng. Ldpc decoding scheduling for faster convergence and lower error floor. *IEEE Transactions on Communications*, 62 (9):3104–3113, 2014.

[50] Yang Han and William E Ryan. Low-floor decoders for LDPC codes. *IEEE Trans. Commun.*, 57(6):1663–1673, 2009.

[51] Zhengya Zhang, Lara Dolecek, Borivoje Nikolic, Venkat Anantharam, and Martin J Wainwright. Lowering LDPC error floors by postprocessing. In *IEEE Global Telecommun. Conf.*, pages 1–6. IEEE, 2008.

[52] Huang-Chang Lee, Po-Chiao Chou, and Yeong-Luh Ueng. An effective low-complexity error-floor lowering technique for high-rate qc-ldpc codes. *IEEE Communications Letters*, 22(10):1988–1991, 2018.

[53] David JC MacKay. Good error-correcting codes based on very sparse matrices. *IEEE Trans. Inf. Theory*, 45(2):399–431, 1999.

[54] Jun Lin and Zhiyuan Yan. Efficient shuffled decoder architecture for nonbinary quasi-cyclic ldpc codes. *IEEE transactions on very large scale integration (VLSI) systems*, 21(9):1756–1761, 2012.

[55] Fang Cai and Xinmiao Zhang. Relaxed min-max decoder architectures for nonbinary low-density parity-check codes. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 21(11):2010–2023, 2013. doi: 10.1109/TVLSI.2012.2226920.

[56] Yeong-Luh Ueng, Kuo-Hsuan Liao, Hsueh-Chih Chou, and Chung-Jay Yang. A high-throughput trellis-based layered decoding architecture for non-binary ldpc codes using max-log-qspa. *IEEE Transactions on Signal Processing*, 61(11): 2940–2951, 2013. doi: 10.1109/TSP.2013.2256905.

[57] Xiaoheng Chen and Chung-Li Wang. High-throughput efficient non-binary ldpc decoder based on the simplified min-sum algorithm. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 59(11):2784–2794, 2012.

[58] Xinmiao Zhang, Fang Cai, and Shu Lin. Low-complexity reliability-based message-passing decoder architectures for non-binary ldpc codes. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 20(11):1938–1950, 2011.

[59] Yeong-Luh Ueng, Chen-Yap Leong, Chung-Jay Yang, Chung-Chao Cheng, Kuo-Hsuan Liao, and Shu-Wei Chen. An efficient layered decoding architecture for nonbinary qc-ldpc codes. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 59(2):385–398, 2011.

[60] Xiaoheng Chen, Shu Lin, and Venkatesh Akella. Efficient configurable decoder architecture for nonbinary quasi-cyclic ldpc codes. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 59(1):188–197, 2011.

[61] Yaoyu Tao, Youn Sung Park, and Zhengya Zhang. High-throughput architecture and implementation of regular (2, dc) nonbinary ldpc decoders. In *2012 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 2625–2628. IEEE, 2012.

[62] Xinmiao Zhang and Fang Cai. Reduced-complexity decoder architecture for non-binary ldpc codes. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 19(7):1229–1238, 2010.

[63] Zhengya Zhang, Venkat Anantharam, Martin J Wainwright, and Borivoje Nikolic. An efficient 10GBASE-T ethernet LDPC decoder design with low error floors. *IEEE J. Solid-State Circuits*, 45(4):843–855, 2010.

[64] Francisco Garcia-Herrero, María José Canet, and Javier Valls. Nonbinary ldpc decoder based on simplified enhanced generalized bit-flipping algorithm. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 22(6):1455–1459, 2013.

[65] Jun Lin and Zhiyuan Yan. An efficient fully parallel decoder architecture for nonbinary ldpc codes. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 22(12):2649–2660, 2013.

[66] Lingqi Zeng, Lan Lan, Ying Y Tai, Shumei Song, Shu Lin, and Khaled Abdel-Ghaffar. Transactions papers-constructions of nonbinary quasi-cyclic ldpc codes: A finite field approach. *IEEE Transactions on Communications*, 56 (4):545–554, 2008.

[67] S. Song, B. Zhou, S. Lin, and K. Abdel-Ghaffar. A unified approach to the construction of binary and nonbinary quasi-cyclic ldpc codes based on finite fields. *IEEE Transactions on Communications*, 57(1):84–93, 2009. doi: 10.1109/TCOMM.2009.0901.060129.

[68] B. Zhou, J. Kang, S.W. Song, S. Lin, K. Abdel-Ghaffar, and M. Xu. Construction of non-binary quasi-cyclic ldpc codes by arrays and array dispersions - [transactions papers]. *IEEE Transactions on Communications*, 57(6):1652–1662, 2009. doi: 10.1109/TCOMM.2009.06.070313.

[69] Charly Poulliat, Marc Fossorier, and David Declercq. Design of regular (2,d/sub c/)-ldpc codes over gf(q) using their binary images. *IEEE Transactions on Communications*, 56(10):1626–1635, 2008. doi: 10.1109/TCOMM.2008.060527.

[70] David Declercq and Marc Fossorier. Decoding algorithms for nonbinary ldpc codes over gf (q). *IEEE transactions on communications*, 55(4):633–643, 2007.

[71] Adrian Voicila, David Declercq, Francois Verdier, Marc Fossorier, and Pascal Urard. Low-complexity decoding for non-binary ldpc codes in high order fields. *IEEE Transactions on Communications*, 58(5):1365–1375, 2010. doi: 10.1109/TCOMM.2010.05.070096.

[72] Valentin Savin. Min-max decoding for non binary ldpc codes. In *2008 IEEE International Symposium on Information Theory*, pages 960–964, 2008. doi: 10.1109/ISIT.2008.4595129.

[73] Alex Graves, Greg Wayne, Malcolm Reynolds, Tim Harley, Ivo Danihelka, Agnieszka Grabska-Barwińska, Sergio Gómez Colmenarejo, Edward Grefenstette, Tiago Ramalho, John Agapiou, et al. Hybrid computing using a neural network with dynamic external memory. *Nature*, 538(7626):471–476, 2016.

[74] Ronan Collobert, Jason Weston, Léon Bottou, Michael Karlen, Koray Kavukcuoglu, and Pavel Kuksa. Natural language processing (almost) from scratch. *J. Mach. Learn. Res.*, 12:2493–2537, November 2011. ISSN 1532-4435.

[75] A. Graves, A. Mohamed, and G. Hinton. Speech recognition with deep recurrent neural networks. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 6645–6649, 2013.

[76] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016.

[77] Alex Graves. Generating sequences with recurrent neural networks. *arXiv preprint arXiv:1308.0850*, 2013.

[78] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112, 2014.

[79] Junyoung Chung, Caglar Gulcehre, Kyunghyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. In *NIPS 2014 Workshop on Deep Learning, December 2014*, 2014.

[80] Sainbayar Sukhbaatar, Jason Weston, Rob Fergus, et al. End-to-end memory networks. In *Advances in neural information processing systems*, pages 2440–2448, 2015.

[81] Jason Weston, Sumit Chopra, and Antoine Bordes. Memory networks. *arXiv preprint arXiv:1410.3916*, 2014.

[82] Ankit Kumar, Ozan Irsoy, Peter Ondruska, Mohit Iyyer, James Bradbury, Ishaan Gulrajani, Victor Zhong, Romain Paulus, and Richard Socher. Ask me anything: Dynamic memory networks for natural language processing. In *International conference on machine learning*, pages 1378–1387, 2016.

[83] Alex Graves, Greg Wayne, and Ivo Danihelka. Neural turing machines. *arXiv preprint arXiv:1410.5401*, 2014.

[84] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *arXiv preprint arXiv:1706.03762*, 2017.

[85] Junhyuk Oh, Valliappa Chockalingam, Honglak Lee, et al. Control of memory, active perception, and action in minecraft. In *International Conference on Machine Learning*, pages 2790–2799. PMLR, 2016.

[86] Antreas Antoniou, Amos Storkey, and Harrison Edwards. Data augmentation generative adversarial networks. *arXiv preprint arXiv:1711.04340*, 2017.

[87] Abhik Singla, Sindhu Padakandla, and Shalabh Bhatnagar. Memory-based deep reinforcement learning for obstacle avoidance in uav with limited environment knowledge. *IEEE Transactions on Intelligent Transportation Systems*, 2019.

[88] Diana Borsa, Bilal Piot, Rémi Munos, and Olivier Pietquin. Observational learning by reinforcement learning. *arXiv preprint arXiv:1706.06617*, 2017.

[89] Jakob Foerster, Nantas Nardelli, Gregory Farquhar, Triantafyllos Afouras, Philip HS Torr, Pushmeet Kohli, and Shimon Whiteson. Stabilising experience replay for deep multi-agent reinforcement learning. In *International conference on machine learning*, pages 1146–1155. PMLR, 2017.

[90] Ignasi Clavera, Jonas Rothfuss, John Schulman, Yasuhiro Fujita, Tamim Asfour, and Pieter Abbeel. Model-based reinforcement learning via meta-policy optimization. In *Conference on Robot Learning*, pages 617–629. PMLR, 2018.

[91] Manoj Alwani, Han Chen, Michael Ferdman, and Peter Milder. Fused-layer cnn accelerators. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12. IEEE, 2016.

[92] Hyoukjun Kwon, Prasanth Chatarasi, Michael Pellauer, Angshuman Parashar, Vivek Sarkar, and Tushar Krishna. Understanding reuse, performance, and hardware cost of dnn dataflow: A data-centric approach. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 754–768, 2019.

[93] Eric Chung, Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Adrian Caulfield, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, et al. Serving dnns in real time at datacenter scale with project brainwave. *iEEE Micro*, 38(2):8–20, 2018.

[94] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W Keckler, and William J Dally. Scnn: An accelerator for compressed-sparse convolutional neural networks. *ACM SIGARCH Computer Architecture News*, 45(2):27–40, 2017.

[95] Yakun Sophia Shao, Jason Clemons, Rangharajan Venkatesan, Brian Zimmer, Matthew Fojtik, Nan Jiang, Ben Keller, Alicia Klinefelter, Nathaniel Pinckney, Priyanka Raina, Stephen G. Tell, Yanqing Zhang, William J. Dally, Joel Emer, C. Thomas Gray, Brucek Khailany, and Stephen W. Keckler. Simba: Scaling deep-learning inference with multi-chip-module-based architecture. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, page 14–27, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450369381. doi: 10.1145/3352460.3358302. URL `https://doi.org/10.1145/3352460.3358302`.

[96] Hyoukjun Kwon, Liangzhen Lai, Tushar Krishna, and Vikas Chandra. Herald: Optimizing heterogeneous dnn accelerators for edge devices. *arXiv preprint arXiv:1909.07437*, 2019.

[97] Hanhwi Jang, Joonsung Kim, Jae-Eon Jo, Jaewon Lee, and Jangwoo Kim. Mnnfast: A fast and scalable system architecture for memory-augmented neural networks. In *Proceedings of the 46th International Symposium on Computer Architecture*, pages 250–263, 2019.

[98] Seongsik Park, Jaehee Jang, Seijoon Kim, and Sungroh Yoon. Energy-efficient inference accelerator for memory-augmented neural networks on an fpga. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1587–1590. IEEE, 2019.

[99] Ashish Ranjan, Shubham Jain, Jacob R Stevens, Dipankar Das, Bharat Kaul, and Anand Raghunathan. X-mann: A crossbar based architecture for memory augmented neural networks. In *Proceedings of the 56th Annual Design Automation Conference 2019*, pages 1–6, 2019.

[100] Jacob R Stevens, Ashish Ranjan, Dipankar Das, Bharat Kaul, and Anand Raghunathan. Manna: An accelerator for memory-augmented neural networks.

In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 794–806, 2019.

[101] Akane Saito, Yuki Umezaki, and Makoto Iwata. Hardware accelerator for differentiable neural computer and its fpga implementation. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, pages 232–238, 2017.

[102] Nagadastagiri Challapalle, Sahithi Rampalli, Nicholas Jao, Akshaykrishna Ramanathan, John Sampson, and Vijaykrishnan Narayanan. Farm: A flexible accelerator for recurrent and memory augmented neural networks. *Journal of Signal Processing Systems*, pages 1–15, 2020.

[103] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.

[104] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[105] Yaoyu Tao, Sung-Gun Cho, and Zhengya Zhang. A configurable successive-cancellation list polar decoder using split-tree architecture. *IEEE Journal of Solid-State Circuits*, 56(2):612–623, 2020.

[106] Yaoyu Tao, Sung-Gun Cho, and Zhengya Zhang. A 3.25 gb/s, 13.2 pj/b, 0.64 mm 2 configurable successive-cancellation list polar decoder using split-tree architecture in 40nm cmos. In *2019 Symposium on VLSI Circuits*, pages C240–C241. IEEE, 2019.

[107] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simmulated annealing. *Science*, 220(4598):671–680, 1983.

[108] VLADIMÍR Černỳ. Thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm. *J. Optimization Theory and Applicat.*, 45(1):41–51, 1985.

[109] Colin R Reeves. *Modern heuristic techniques for combinatorial problems.* John Wiley & Sons, Inc., 1993.

[110] Yaoyu Tao, Shuanghong Sun, and Zhengya Zhang. Efficient post-processors for improving error-correcting performance of ldpc codes. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 66(10):4032–4043, 2019.

[111] John L Fan. Array codes as low-density parity-check codes. In *Proc. Int. Symp. Turbo Codes and Related Topics*, volume 546, 2000.

[112] Youn Sung Park, Yaoyu Tao, and Zhengya Zhang. A fully parallel nonbinary ldpc decoder with fine-grained dynamic clock gating. *IEEE Journal of Solid-State Circuits*, 50(2):464–475, 2014.

[113] Youn Sung Park, Yaoyu Tao, and Zhengya Zhang. A 1.15 gb/s fully parallel nonbinary ldpc decoder with fine-grained dynamic clock gating. In *2013 IEEE International Solid-State Circuits Conference Digest of Technical Papers*, pages 422–423. IEEE, 2013.

[114] Yaoyu Tao and Zhengya Zhang. Hima: A fast and scalable history-based memory access engine for differentiable neural computer. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 845–856, 2021.

[115] Yaoyu Tao and Zhengya Zhang. Dnc-aided scl-flip decoding of polar codes. *arXiv preprint arXiv:2101.10498*, 2021.

[116] E. Arikan. Channel polarization: A method for constructing capacity-achieving codes for symmetric binary-input memoryless channels. *IEEE Transactions on Information Theory*, 55(7):3051–3073, July 2009. ISSN 0018-9448. doi: 10.1109/TIT.2009.2021379.

[117] I. Tal and A. Vardy. List decoding of polar codes. In *2011 IEEE International Symposium on Information Theory Proceedings*, pages 1–5, July 2011. doi: 10.1109/ISIT.2011.6033904.

[118] K. Niu and K. Chen. Crc-aided decoding of polar codes. *IEEE Communications Letters*, 16(10):1668–1671, October 2012. ISSN 1089-7798. doi: 10.1109/LCOMM.2012.090312.121501.

[119] B. Li, H. Shen, and D. Tse. An adaptive successive cancellation list decoder for polar codes with cyclic redundancy check. *IEEE Communications Letters*, 16 (12):2044–2047, December 2012. ISSN 1089-7798. doi: 10.1109/LCOMM.2012.111612.121898.

[120] B. Li, H. Shen, and D. Tse. Parallel decoders of polar codes. *arXiv:1309.1026v1 [cs.IT]*, Sep 2013.

[121] F. Ercan, T. Tonnellier, and W. J. Gross. Energy-efficient hardware architectures for fast polar decoders. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 67(1):322–335, Jan 2020. ISSN 1558-0806. doi: 10.1109/TCSI.2019.2942833.

[122] W. Song, H. Zhou, K. Niu, Z. Zhang, L. Li, X. You, and C. Zhang. Efficient successive cancellation stack decoder for polar codes. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 27(11):2608–2619, Nov 2019. ISSN 1557-9999. doi: 10.1109/TVLSI.2019.2925029.

[123] Jinghu Chen, Ajay Dholakia, Evangelos Eleftheriou, Marc PC Fossorier, and Xiao-Yu Hu. Reduced-complexity decoding of LDPC codes. *IEEE Trans. Commun.*, 53(8):1288–1299, 2005.

[124] C. Cheng, J. Yang, H. Lee, C. Yang, and Y. Ueng. A fully parallel ldpc decoder architecture using probabilistic min-sum algorithm for high-throughput applications. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 61(9): 2738–2746, Sep. 2014. ISSN 1549-8328. doi: 10.1109/TCSI.2014.2312479.

[125] A. J. Blanksby and C. J. Howland. A 690-mw 1-gb/s 1024-b, rate-1/2 low-density parity-check code decoder. *IEEE Journal of Solid-State Circuits*, 37(3): 404–412, March 2002. ISSN 0018-9200. doi: 10.1109/4.987093.

[126] Y.S. Park H. Li and Z. Zhang. Reconfigurable architecture and automated design flow for rapid fpga-based ldpc code emulation. In *ACM Int. Symp. Field-Programmable Gate Arrays (FPGA)*, pages 167–170, Feb 2012.

[127] I. Djurdjevic, Jun Xu, K. Abdel-Ghaffar, and Shu Lin. A class of low-density parity-check codes constructed based on Reed-Solomon codes with two information symbols. *IEEE Commun. Lett.*, 7(7):317–319, July 2003.

[128] Emmanuel Boutillon and Laura Conde-Canencia. Bubble check: a simplified algorithm for elementary check node processing in extended min-sum non-binary ldpc decoders. *Electronics Letters*, 46(9):633–634, 2010.

[129] Ahmad Darabiha, Anthony Chan Carusone, and Frank R Kschischang. Power reduction techniques for ldpc decoders. *IEEE Journal of Solid-State Circuits*, 43(8):1835–1845, 2008.

[130] Xin-Yu Shih, Cheng-Zhou Zhan, Cheng-Hung Lin, and An-Yeu Wu. An 8.29 mm¡formula formulatype="inline"¿¡tex¿^{2}¡/tex¿ ¡/formula¿ 52 mw multi-mode ldpc decoder design for mobile wimax system in 0.13 ¡formula¿¡tex¿μ¡/tex¿¡/formula¿m cmos process. *IEEE Journal of Solid-State Circuits*, 43(3):672–683, 2008. doi: 10.1109/JSSC.2008.916606.

[131] Jason Weston, Antoine Bordes, Sumit Chopra, Alexander M Rush, Bart van Merriënboer, Armand Joulin, and Tomas Mikolov. Towards ai-complete question answering: A set of prerequisite toy tasks. *arXiv preprint arXiv:1502.05698*, 2015.

[132] Hyoukjun Kwon, Ananda Samajdar, and Tushar Krishna. Maeri: Enabling flexible dataflow mapping over dnn accelerators via reconfigurable interconnects. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, page 461–475, New York, NY, USA, 2018. ISBN 9781450349116.

[133] A. Norollah, D. Derafshi, H. Beitollahi, and M. Fazeli. Rths: A low-cost high-performance real-time hardware sorter, using a multidimensional sorting algorithm. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 27 (7):1601–1613, 2019. doi: 10.1109/TVLSI.2019.2912554.

[134] Susumu Mashimo, Thiem Van Chu, and Kenji Kise. High-performance hardware merge sorter. In *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 1–8, 2017. doi: 10.1109/FCCM.2017.19.

[135] Ioannis Kouretas and Vassilis Paliouras. Simplified hardware implementation of the softmax activation function. In *2019 8th International Conference on Modern Circuits and Systems Technologies (MOCAST)*, pages 1–4, 2019.

[136] Xiao Dong, Xiaolei Zhu, and De Ma. Hardware implementation of softmax function based on piecewise lut. In *2019 IEEE International Workshop on Future Computing (IWOFC)*, pages 1–3, 2019.

[137] Erwin Fehlberg. *Low-order classical Runge-Kutta formulas with stepsize control and their application to some heat transfer problems*, volume 315. National aeronautics and space administration, 1969.

[138] Juntang Zhuang, Nicha Dvornek, Xiaoxiao Li, Sekhar Tatikonda, Xenophon Papademetris, and James Duncan. Adaptive checkpoint adjoint method for gradient estimation in neural ode. In *International Conference on Machine Learning*, pages 11639–11649. PMLR, 2020.

[139] Juntang Zhuang, Nicha C Dvornek, Sekhar Tatikonda, and James S Duncan. Mali: A memory efficient and reverse accurate integrator for neural odes. *arXiv preprint arXiv:2102.04668*, 2021.

[140] Alejandro F Queiruga, N Benjamin Erichson, Dane Taylor, and Michael W Mahoney. Continuous-in-depth neural networks. *arXiv preprint arXiv:2008.02389*, 2020.

[141] O. Afisiadis, A. Balatsoukas-Stimming, and A. Burg. A low-complexity improved successive cancellation decoder for polar codes. In *2014 48th Asilomar Conference on Signals, Systems and Computers*, pages 2116–2120, Nov 2014. doi: 10.1109/ACSSC.2014.7094848.

[142] C. Condo, F. Ercan, and W.J. Gross. Improved successive cancellation flip decoding of polar codes based on error distribution. In *Proc. IEEE Wireless Commun. Netw. Conf. Workshops (WCNCW)*, pages 19–24, Apr 2018. doi: 10.1109/ACSSC.2014.7094848.

[143] F. Ercan, C. Condo, and W. J. Gross. Improved bit-flipping algorithm for successive cancellation decoding of polar codes. *IEEE Transactions on Communications*, 67(1):61–72, Jan 2019. ISSN 1558-0857. doi: 10.1109/TCOMM.2018.2873322.

[144] Z. Zhang, K. Qin, L. Zhang, H. Zhang, and G. T. Chen. Progressive bit-flipping decoding of polar codes over layered critical sets. In *GLOBECOM 2017 - 2017 IEEE Global Communications Conference*, pages 1–6, Dec 2017. doi: 10.1109/GLOCOM.2017.8254149.

[145] X. Liu, S. Wu, Y. Wang, N. Zhang, J. Jiao, and Q. Zhang. Exploiting error-correction-crc for polar scl decoding: A deep learning based approach. *IEEE Transactions on Cognitive Communications and Networking*, pages 1–1, 2019. ISSN 2372-2045. doi: 10.1109/TCCN.2019.2946358.

[146] F. Cheng, A. Liu, Y. Zhang, and J. Ren. Bit-flip algorithm for successive cancellation list decoder of polar codes. *IEEE Access*, 7:58346–58352, 2019.

[147] F. Ercan, C. Condo, S. A. Hashemi, and W. J. Gross. Partitioned successive-cancellation flip decoding of polar codes. In *2018 IEEE International Conference on Communications (ICC)*, pages 1–6, May 2018. doi: 10.1109/ICC.2018.8422464.

[148] L. Chandesris, V. Savin, and D. Declercq. An improved scflip decoder for polar codes. In *2016 IEEE Global Communications Conference (GLOBECOM)*, pages 1–6, Dec 2016. doi: 10.1109/GLOCOM.2016.7841594.

[149] L. Chandesris, V. Savin, and D. Declercq. Dynamic-scflip decoding of polar codes. *IEEE Transactions on Communications*, 66(6):2333–2345, June 2018. ISSN 1558-0857. doi: 10.1109/TCOMM.2018.2793887.

[150] Y. Tao, S. G. Cho, and Z. Zhang. A configurable successive-cancellation list polar decoder using split-tree architecture. *IEEE Journal of Solid-State Circuits*, 56(2):612–623, 2021. doi: 10.1109/JSSC.2020.3005763.

[151] F. Ercan, T. Tonnellier, N. Doan, and W. J. Gross. Practical dynamic sc-flip polar decoders: Algorithm and implementation. *IEEE Transactions on Signal Processing*, 68:5441–5456, 2020. doi: 10.1109/TSP.2020.3023582.

[152] F. Ercan and W. J. Gross. Fast thresholded sc-flip decoding of polar codes. In *ICC 2020 - 2020 IEEE International Conference on Communications (ICC)*, pages 1–7, 2020. doi: 10.1109/ICC40277.2020.9149099.

[153] C. Condo, V. Bioglio, and I. Land. Sc-flip decoding of polar codes with high order error correction based on error dependency. In *2019 IEEE Information Theory Workshop (ITW)*, pages 1–5, 2019. doi: 10.1109/ITW44776.2019.8989249.

[154] Y. H. Pan, C. H. Wang, and Y. L. Ueng. Generalized scl-flip decoding of polar codes. In *GLOBECOM 2020 - 2020 IEEE Global Communications Conference*, pages 1–6, 2020. doi: 10.1109/GLOBECOM42002.2020.9321982.

[155] X. Wang, H. Zhang, R. Li, L. Huang, S. Dai, Y. Yourui, and J. Wang. Learning to flip successive cancellation decoding of polar codes with lstm networks. arXiv:1902.08394, Feb 2019. doi: 10.1109/ACSSC.2014.7094848.

[156] B. He, S. Wu, Y. Deng, H. Yin, J. Jiao, and Q. Zhang. A machine learning based multi-flips successive cancellation decoding scheme of polar codes. In *2020 IEEE 91st Vehicular Technology Conference (VTC2020-Spring)*, pages 1–5, 2020.

[157] C-H. Chen, C-F. Teng, and A-Y. Wu. Low-complexity lstm-assisted bit-flipping algorithm for successive cancellation list polar decoder. In *45th IEEE International Conference on Acoustics, Speech, and Signal Processing*, May 2020. doi: 10.1109/ICC.2018.8422464.

[158] Sainbayar Sukhbaatar, arthur szlam, Jason Weston, and Rob Fergus. End-to-end memory networks. In *Advances in Neural Information Processing Systems 28*, pages 2440–2448. 2015.

[159] Ankit Kumar, Ozan Irsoy, Peter Ondruska, Mohit Iyyer, James Bradbury, Ishaan Gulrajani, Victor Zhong, Romain Paulus, and Richard Socher. Ask me anything: Dynamic memory networks for natural language processing. In *Proceedings of The 33rd International Conference on Machine Learning*, volume 48, pages 1378–1387, 20–22 Jun 2016.

[160] A. Graves, G. Wayne, and M. Reynolds et al. Hybrid computing using a neural network with dynamic external memory. *Nature*, 538:471–476, Oct 2016. ISSN 1558-0857. doi: https://doi.org/10.1038/nature20101.

[161] Y. Lecun, Y. Bengio, and G. Hinton et al. Deep learning. *Nature*, 521(7553): 436–444, Oct 2015. ISSN 1558-0857. doi: https://doi.org/10.1038/nature20101.