Algorithm-Hardware Co-Design for Artificial Intelligence: From Energy-Efficient Edge Processing to High-Performance GPU Acceleration

by

Pierre Abillama

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
2025

Doctoral Committee:

Associate Professor Hun-Seok Kim, Chair
Professor David Blaauw
Associate Professor Ronald Dreslinski
Professor Dennis Sylvester

Pierre Abillama
pabillam@umich.edu
ORCID iD: 0000-0001-9523-6152

© Pierre Abillama 2025

DEDICATION

To my parents, Pierrette and Walid, for their ferocious and unconditional love, endless support, unwavering faith, and for being the truest examples of strength, sacrifice, and selflessness. To my mother, for teaching me what it means to be a warrior, and to my father, for showing me that you're never too old to rock and roll. To my brother, Jerry, for his kindness, his big heart, and for always lifting me up when I needed it most. And to my grandparents, Georgette and Pierre, for making me feel like the most special kid in the world, even when I'm not. This one's for all of you.

ACKNOWLEDGEMENTS

I don't even know where to start, thank God it's over!

First, I owe my deepest gratitude to my advisor, Professor Hun-Seok Kim, for giving me the opportunity to pursue a Ph.D. and for his invaluable guidance, mentorship, and support throughout my doctoral studies. I am also very thankful to Professors David Blaauw and Dennis Sylvester, whose insights and expertise shaped the direction of this research and greatly influenced my growth as a researcher. I am grateful as well to Professor Ronald Dreslinski for serving on my committee and for his thoughtful feedback. I also wish to thank my undergraduate advisor, Professor Kia Bazargan, and Dr. Rasoul Faraji, for taking me in when I had no experience in digital hardware design. Without the opportunity they gave me to learn and demonstrate my research abilities, I would not be where I am today.

I'm especially thankful to all my colleagues and friends in the MICL group for the discussions, the late nights in the office, for witnessing my reactions to that week's Champions League matches, and for making the lab a genuinely fun and supportive place to be. Changwoo, Zichen, Qirui, Hyochan, Hyunwon, Rohit, Demba, Yufan, Heejin, Jungho, Ji Yoon, Xiangdong, Sara, Mohammad, and Yu, thank you for making this experience unforgettable.

Beyond the lab, I was lucky to find several communities that made Ann Arbor feel like home. Demba and the Fuller Park soccer crew reignited my love for the game; Wednesday evenings and Saturday mornings were the highlight of my early Ph.D. years. Trevor, Aman, Sarah, Kevin, Eli, Fahad, Sophie, Anna, and Noah filled my weekends with poker nights, bonfires, and football watch parties, while Jack, Gracie, Micah, Sasi, Lauren, Rebecca, Vangelis, Geoff, Courtney, John, and Shashank brightened things with countless lunches, dinners, and nights out that made everything so much more bearable.

I was also fortunate to have the unwavering support of longtime friends I met before coming to Ann Arbor, who, despite the distance, played a huge part in my well-being. Mourad, Kim, and Elie reminded me what it feels like to be a kid again and to have fun even when I was overworking myself. I'm deeply grateful for the laughter, the great conversations, and for making my trip to Montreal so memorable after my first tapeout. Sangbone, thank you for keeping me sane and for picking up every call when I needed to vent or when life felt overwhelming. Having someone to confide in made this journey infinitely easier. I'm

also thankful to Bruce and Mootz for visiting and for the fun trips along the east coast that helped me unwind and reset. And to all my friends from Minnesota and back home, now scattered around the world, thank you for cheering me on and believing in me from afar.

I couldn't have made it through my first year at Michigan, at the height of COVID, without my extended family. Tacha, Chelsie, and Imad, thank you for welcoming me into your community of Lebanese and Arab Michiganders, for helping me move and settle into my first apartment, and for inviting me to countless Thanksgivings, Labor Days, and Fourth of July barbecues. Your kindness, hospitality, and home-cooked meals gave me a sense of belonging when I needed it most. To the rest of my family, thank you for your constant encouragement and love from afar. Elie, Valerie, Eliane, and Souheil, your messages, humor, and belief in me always lifted my spirits when things got tough. Knowing you were rooting for me made the distance feel a little smaller and the challenges a little lighter.

To my parents, thank you for inspiring me to pursue a Ph.D. and for being my unwavering foundation. Mom, Dad, Jerry, and Titi, thank you for all the love and care you've given me, for crossing oceans just to be by my side during the most turbulent times, and for giving so much of yourselves to help me push through difficult circumstances. Thank you for your endless patience when I was overwhelmed, anxious, or just plain insufferable. Your support and attention never faltered, and I can't begin to express how much that meant to me.

To Andie, my rock and soulmate, my source of endless joy and light. Without you, I couldn't have persevered or completed this degree. I am forever indebted to you for the laughter, love, comfort, and inspiration you brought into my life during one of its most challenging moments. Moving a piano with you up a flight of stairs proved that nothing is impossible, not even this Ph.D. Being with you has taught me how to be more patient, caring, and thoughtful, and has given my life a deeper sense of purpose.

And finally, to my late Jeddo, who always saw the best in me and made me believe I could reach it when I first started my Ph.D. I wish you could see where I am today. I hope I've made you proud.

TABLE OF CONTENTS

DEDICATION ii

ACKNOWLEDGEMENTS iii

LIST OF FIGURES viii

LIST OF TABLES x

ABSTRACT xi

CHAPTER

1 Introduction 1

1.1 Deep Learning: Challenges and Applications 1
1.2 Overview of Algorithm Techniques 3
1.2.1 Transform-Domain Computations 3
1.2.2 Model Pruning and Compression 4
1.2.3 Quantization 7
1.3 Overview of Hardware Techniques 9
1.3.1 Architecture-Level Techniques 9
1.3.2 Circuit-Level Techniques 11
1.4 Dissertation Overview 12

2 SONA: A Hardware Accelerator Architecture for Transform-Domain Neural Networks with Sparse-Orthogonal Weights 14

2.1 Introduction 14
2.2 Background 16
2.2.1 WHT-Domain Convolution 16
2.2.2 Sparse-Orthogonal Kernels in HTNNs 18
2.2.3 Bit-Sparse Canonical-Signed-Digit (BS-CSD) Computation 19
2.2.4 Fully-Connected Layer (FCL) with Structured Sparsity 20
2.3 SONA Architecture and Implementation 20
2.3.1 Overall Architecture 21
2.3.2 Reconfigurable Transform Datapaths 23
2.3.3 WHT-Domain Sparse-Orthogonal Convolution Dataflow 24
2.3.4 Memory Organization 28
2.3.5 Column-Based-Block Sparse Fully-Connected Layer 30

2.3.6 Bit-Sparse Canonical-Sign-Digit (BS-CSD) MAC Unit 33
2.4 Evaluation 35
2.4.1 Experimental Methodology 35
2.4.2 Simulator-Based Results 37
2.4.3 Comparisons with Sparsity-Aware Accelerators 42
2.4.4 Post-Synthesis Design Comparison 43
2.5 Related Works 44
2.6 Conclusion 47

3 A 22 nm 9.51 TOPS/W Neural Engine with 2 MB MRAM Leveraging Sparse-Orthogonal Walsh-Hadamard Transform Computations and Dynamic Power Gating 48

3.1 Introduction 48
3.2 Neural Engine Architecture 51
3.3 Neural Engine Dataflow 52
3.3.1 WHT-Convolution Layer 52
3.3.2 Fully-Connected Layer 54
3.4 MRAM-Weight Cache Subsystem 55
3.5 Measurements 58
3.6 Conclusion 59

4 One-Hot Multi-Level Leaky Integrate-and-Fire Spiking Neural Networks for Enhanced Accuracy-Latency Tradeoff 62

4.1 Introduction 62
4.2 Background & Related Works 65
4.2.1 Leaky Integrate-and-Fire Model 65
4.2.2 ANN-SNN Conversion 66
4.2.3 Direct Training 66
4.2.4 Quantized-Activation ANNs 68
4.2.5 Quantized-Activation SNNs 69
4.3 Proposed One-Hot Multi-level LIF-based SNNs 70
4.3.1 Multi-level LIF Model 70
4.3.2 One-hot Multi-level LIF Model 71
4.4 Energy Consumption Estimation 75
4.5 Experiments & Results 78
4.5.1 Static Image Classification 79
4.5.2 Dynamic Image Classification 84
4.5.3 Ablation Studies 88
4.5.4 Memory Access Energy Impact 89
4.6 Future Directions 93
4.7 Conclusion 94

5 Memory-Efficient Acceleration of Block Low-Rank Foundation Models on Resource Constrained GPUs 95

5.1 Introduction 95

5.2 Background . . . 98
5.2.1 Weight Structures . . . 98
5.2.2 GPU Performance . . . 101
5.3 Profiling and Characterizing Bottlenecks . . . 102
5.3.1 Single-Token Inference . . . 103
5.3.2 Multi-Token Inference . . . 103
5.4 Memory-Efficient Kernels . . . 105
5.5 Experiments and Results . . . 109
5.6 Conclusion . . . 115
6 Conclusion . . . 116
6.1 Summary of Contributions . . . 116
6.2 Future Work . . . 118
BIBLIOGRAPHY . . . 121

LIST OF FIGURES

FIGURE

1.1 Vision model ImageNet accuracy versus computational and memory complexity. 2
1.2 Deep neural network weight and activation pruning. 5
1.3 Comparison between 3-bit uniform quantization (left) and non-uniform power-of-two quantization (right). 7

2.1 Comparison between HTNN WHT-domain convolution layer and conventional CNN convolutional layer. 18
2.2 Overall architecture with highlighted WHT convolution layer path. 22
2.3 Permuted reconfigurable transform datapaths 23
2.4 HTNN pseudcode for three candidate dataflows. 25
2.5 Memory access energy comparison for ConvPool-CNN-C and ResNet-20 against HTNN dataflows: output/weight/input stationary (OS/WS/IS) 27
2.6 MASR transform memory organization for mapping transform activations onto memory banks and words ($I = 32$, $N_P = 4$, $n_t = 3$). 29
2.7 Overhead of index-based encoding methods for FCL weight matrices with unstructured sparsity. 30
2.8 Fully-connected layer path. 31
2.9 BS-CSD-MAC unit and 8-bit BS-CSD multiplier. 33
2.10 ResNet-20 per-layer speedup on CIFAR-10. 38
2.11 VGG-Nagadomi per-layer speedup on CIFAR-10. 38
2.12 ConvPool-CNN-C per-layer speedup on CIFAR-100. 38
2.13 ResNet-18 per-layer speedup on ImageNet. 39
2.14 VGG-Nagadomi FCL speedup on CIFAR-10. 39
2.15 ResNet-20 per-layer energy efficiency on CIFAR-10. 40
2.16 VGG-Nagadomi per-layer energy efficiency on CIFAR-10. 40
2.17 ConvPool-CNN-C per-layer energy efficiency on CIFAR-100. 40
2.18 ResNet-18 per-layer energy efficiency on ImageNet. 42
2.19 SONA energy breakdown on ResNet-20. 42
2.20 VGG-Nagadomi FCL energy efficiency on CIFAR-10. 43

3.1 Motivation for the proposed accelerated HTNN processor. 49
3.2 MRAM active and leakage energy challenges for accelerated HTNN processor. . 50
3.3 MRAM performance and access time challenges for accelerated HTNN processor. 50
3.4 Overall architecture of the proposed HTNN processor. 52
3.5 Summary of HTNN processor instruction set architecture. 53
3.6 Heterogeneous WHT-domain sparse-orthogonal convolution dataflow. 53

3.7 Column-based block structured sparsity fully-connected layer dataflow. 55
3.8 MRAM-weight cache multi-clock dynamic power gating architecture. 56
3.9 Chip summary and die photo. 57
3.10 Measured peak efficiency and voltage-frequency efficiency scaling of WHT-convolution using 2 and 3 WHTs. 58
3.11 Measured efficiency layer breakdown at mid performance points for ResNet-20 HTNN. 61

4.1 LIF neuron model (left) vs. one-hot M-LIF neuron model with $S_o = 4$ lanes (right). 64
4.2 Output activation and surrogate gradient functions for conventional LIF neuron model (left) and one-hot multi-level LIF neuron model for $0 \leq s < S_o - 1$ (right). 67
4.3 Example of triangular surrogate gradient for $S_o = 4$ one-hot M-LIF neuron output spike lanes. 74
4.4 32-bit floating-point format and power of two multiplication details where b_i is the i-th bit, *sign* is the most significant bit, and E is the 8-bit exponent. 76
4.5 Workflow of multi-level input layer encoding for dynamic vision tasks. 86
4.6 Example image in DVS-CIFAR-10 using baseline input layer encoding with $T_F = 10$ binary frames and $S_i = 1$ spike lane (left) and the proposed multi-level input layer encoding with $Q_F = 5$ one-hot frames and $S_i = 3$ spike lanes (right). . . . 87
4.7 Comparison of per-layer spike rates for VGG16 on ImageNet. 92

5.1 Roofline model of BF16 operations for NVIDIA A40 GPU. 97
5.2 Monarch (left) and BLAST (right) weight parametrization and linear layer execution for $b_1 = b_2 = 3$ blocks and rank $r = 4$. 99
5.3 Basic GPU architecture (left) and execution model (right). 101
5.4 Performance of Llama-7B layers with low-rank methods versus dense, for single-token (left) and multi-token (right) inference on an NVIDIA A40 GPU. 103
5.5 Roofline of $\texttt{Q}/\texttt{K}/\texttt{V}/\texttt{O}_{\texttt{proj}}$ during multi-token inference on A40. 104
5.6 Speedup of Triton fused low-rank matrix multiplication over the PyTorch low-rank baseline on NVIDIA A40 GPU. Results are shown across different input/output feature dimensions for two fixed ranks: $r = 128$ (left) and $r = 256$ (right). 106
5.7 Pseudo-code for fused permutation and `bmm` Monarch kernel (❷). 106
5.8 Pseudo-code for BLAST partial fusion (❹). 108
5.9 Compute diagram for BLAST permutation-only fusion (❺). 108
5.10 Layer-wise performance comparison across BLR methods and GPUs. 109
5.11 End-to-end inference performance across models and platforms. 113
5.12 Tradeoff between perplexity and speedup over dense compared across methods. 114

LIST OF TABLES

TABLE

2.1 SONA configuration and area summary. 35
2.2 Accelerator benchmark comparison summary. 43
2.3 Post-synthesis design comparison summary. 44

3.1 Comparison table versus state-of-the-art. 60
3.2 Measured impact of MRAM optimizations on performance and efficiency at high performance point for ResNet-20 HTNN. 61

4.1 Iso-architecture comparison with SNNs for static image classification on CIFAR and ImageNet. * denotes self-implementation results. 82
4.2 Iso-architecture comparison with LQ-ANNs for static image classification on CIFAR and ImageNet. * denotes self implementation results. 84
4.3 Comparison with quantized-activation SNNs for static image classification on CIFAR and ImageNet. 85
4.4 Comparison with prior works for dynamic image classification on DVS-CIFAR-10. † denotes data augmentation. $*$ denotes self-implementation results. 88
4.5 Ablation study varying the number of spike lanes for one-hot M-LIF VGGSNN on DVS-CIFAR-10 with fixed timesteps $T = 3$ and $T = 5$. 89
4.6 Impact of memory access energy for static image classification. $*$ denotes self-implementation results. 91

5.1 Accuracy of foundation models using different (B)LR model compression factors (CF). 101
5.2 FLOP and memory traffic for linear layers under different BF16 weight structures. 104
5.3 Layer configurations for dense layers replaced by low-rank, Monarch, and BLAST counterparts across evaluated models. 112

ABSTRACT

Deep learning has become a cornerstone of modern artificial intelligence (AI), driving transformative advances in fields such as computer vision, natural language processing, and generative models for text and images. However, as the complexity of neural networks continues to rise, the computational and memory demands necessary to maintain state-of-the-art performance have also increased significantly. This dissertation addresses the growing complexity of deep neural networks and the challenges associated with scaling and deploying them, particularly in resource-constrained environments such as edge devices. It explores the co-design of specialized hardware and neural networks, emphasizing the need for efficient integration between software and hardware to achieve optimal performance. This dissertation presents four works that aim to maximize the benefits of software optimizations, minimize accuracy loss during model simplification, and deliver real-world speedups on existing hardware platforms.

The first work introduces a hardware accelerator for heterogeneous transform-domain neural networks, which reduces computational overhead through a specialized dataflow and optimized cache organization. Evaluations demonstrate that the proposed accelerator enhances energy efficiency and performance by up to 5.1× and 5.2×, respectively, for convolution layers, compared to previous dense accelerators. The second work presents an all-on-chip-weight implementation of a heterogeneous transform-domain neural engine that uses dense magnetoresistive non-volatile memory for weight storage. This processor achieves a peak efficiency of 9.51 TOPS/W for convolutional layers and 22.86 TOPS/W for fully connected layers, positioning it as a promising solution for energy-constrained edge applications. The third work proposes a one-hot multi-level leaky integrate-and-fire neuron model,

which improves the accuracy-energy tradeoff in spiking neural networks by improving performance without compromising energy consumption. Evaluations reveal that networks using the proposed neurons enhance the accuracy-latency tradeoff for advanced network architectures, such as spike-driven transformers, achieving over 3% higher accuracy and requiring 4× fewer timesteps on ImageNet. Finally, the fourth work investigates techniques for optimizing structured-compressed foundation model inference on graphics processing units (GPUs), leveraging fused kernels and memory-efficient strategies to achieve substantial speedups. Together, these contributions not only advance the efficient deployment of deep learning models in resource-limited environments but also pave the way for future innovations in AI hardware and algorithm co-design.

CHAPTER 1

Introduction

1.1 Deep Learning: Challenges and Applications

In recent years, deep learning powered by neural networks has become a key driver of significant advancements in artificial intelligence (AI), enabling breakthroughs in fields such as computer vision, natural language processing (NLP), and generative models for text and images. This progress is exemplified by the success of convolutional neural networks (CNNs) in image classification [1] and object segmentation tasks [2], as well as by the ability of recurrent and transformer-based models [3], such as GPT [4], to process and generate coherent text.

However, as neural networks evolve, they have grown substantially in both depth and complexity, requiring an increasing number of parameters and computations to achieve state-of-the-art performance. The evolution of deep learning models for computer vision [5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18], illustrated in Figure 1.1, highlights this trend. Early networks were relatively compact but limited in their capabilities. As performance improved over time, networks became deeper and larger, which in turn corresponded to an increase in computational and memory demands. Most recently, the rise of transformer architectures as the dominant choice for NLP and generative tasks has further exacerbated these demands. For instance, the smallest GPT2 model, GPT2-S [19], contains approximately 117M parameters, slightly more than the 93M parameters of the largest ResNet

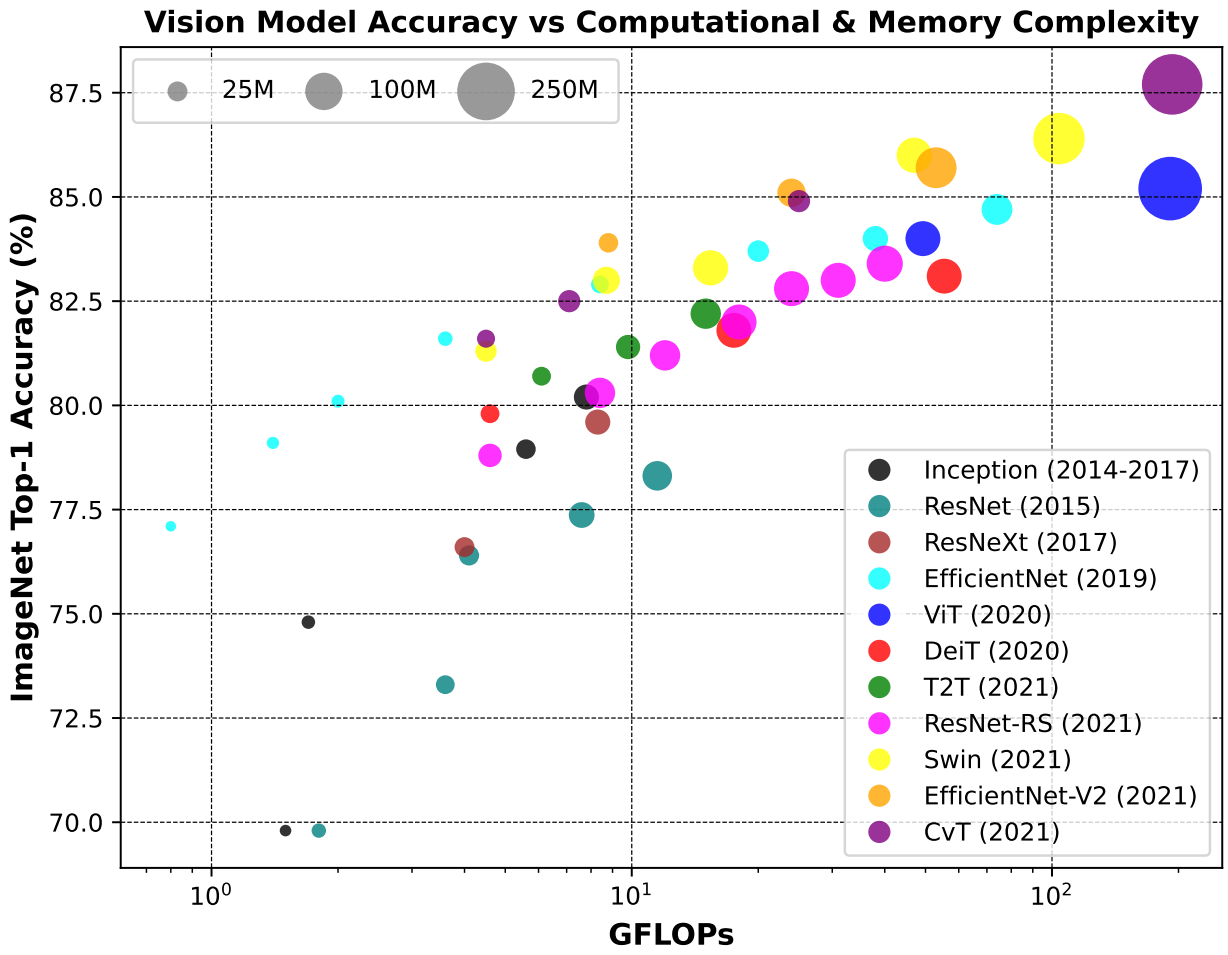


Figure 1.1: Vision model ImageNet accuracy versus computational and memory complexity.

model [11] for computer vision. While the GPT2-S model performs well for text generation, its performance degrades significantly with longer contexts and more complex tasks, such as question answering, summarization, and reasoning [20]. A larger GPT2 model, with approximately 1.5B parameters, offers much better performance on these complex tasks but demands substantially more computational resources.

This growing complexity has been supported by advancements in cloud-based hardware, particularly server-grade graphics processing units (GPUs), which offer higher memory capacity, bandwidth, and computational power to accommodate deep learning workloads [21]. Additionally, innovations like distributed training across multiple GPU nodes have further enabled model scaling [22]. Nevertheless, even with these advancements, server-grade GPUs

can become bottlenecks as the demand for more accurate and larger models continues to rise. In the case of language models, where increasing model size and extending context length often lead to higher accuracy, memory requirements during inference scale quadratically with sequence length and often exceed the individual memory capacity of even high-end GPUs.

Moreover, while much of the progress has centered around scaling models within cloud environments, there is also a growing shift toward deploying AI at the edge, where resource constraints introduce a new set of challenges. Running these models on smaller platforms such as laptops, smartphones, and intelligent edge systems presents significant challenges. Server-grade GPUs, while powerful, are often impractical for such applications due to their size, energy consumption, and cost. In many cases, deploying server-grade GPUs is unnecessary, as some AI models can maintain robust performance even when compressed [23, 24, 25], making smaller, more efficient hardware a viable alternative.

To bridge this gap, extensive research has focused on compressing models while preserving accuracy to enable deployment on resource-constrained platforms. However, these optimizations do not always translate effectively to existing hardware architectures, and many platforms struggle to convert algorithmic improvements into meaningful energy and speedup gains. Achieving better efficiency requires a co-design approach that integrates both hardware and algorithmic innovations. Custom hardware accelerators can be designed to maximize the benefits of model optimizations, while algorithm techniques can be tailored to map efficiently onto specific hardware architectures.

1.2 Overview of Algorithm Techniques

1.2.1 Transform-Domain Computations

A deep neural network consists of a sequence of linear and nonlinear computational layers. At its core, the fundamental operation is multiply-and-accumulate (MAC), which involves multiplying an input activation by a weight and accumulating the result. In CNNs, the

primary computational workload comes from convolution layers, where a $k \times k$ weight kernel is convolved with a $k \times k$ window of an $n \times n$ input, using a stride s. A common CNN configuration, a 3×3 stride-1 kernel, requires nine MACs per output.

Prior work [26, 27, 28] has shown that convolution can be computed more efficiently using transforms such as the discrete cosine transform (DCT), discrete Fourier transform (DFT), and Winograd minimal filtering. This efficiency gain stems from the equivalence of convolution in the spatial domain to element-wise multiplication in the transform domain [29], reducing arithmetic complexity by up to 4× depending on the tile size [26]. Consequently, Winograd-based convolution algorithms are integrated into state-of-the-art deep learning libraries, such as NVIDIA cuDNN [30], to enhance computational efficiency.

However, these methods also introduce challenges, as computing the transform on large feature maps can create significant overhead. In many cases, recomputing the transform is necessary to work around hardware cache size limitations, potentially negating the benefits of reduced arithmetic complexity [26]. Furthermore, combining this approach with other efficiency-oriented techniques can be nontrivial, as it may conflict with strategies such as model pruning and compression, discussed in the following subsection.

1.2.2 Model Pruning and Compression

Deep neural networks inherently exhibit a property known as sparsity, which refers to the proportion of zero-valued activations and weights propagating through the layers of the network. Prior research has demonstrated that these networks can be pruned to enforce even higher degrees of sparsity without significantly deteriorating accuracy [24, 23, 31]. By removing redundant connections and neurons as illustrated in Figure 1.2, sparsification reduces both the number of parameters and the computational workload, making it a widely used technique for model compression and computational acceleration. However, to fully exploit these benefits, hardware must be designed to skip computations involving zero-valued weights and activations. Additionally, hardware must also be designed to avoid unneces-

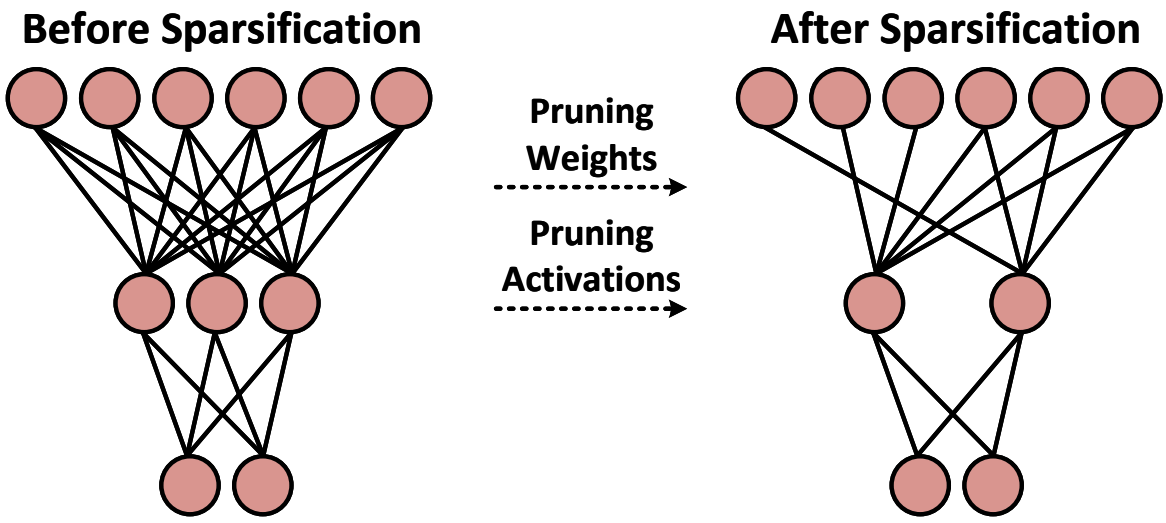


Figure 1.2: Deep neural network weight and activation pruning.

sary data transfers by skipping zero-valued data. Reducing memory traffic in this way can lead to substantial energy and throughput gains, particularly in memory-bounded inference scenarios where memory latency dominates compute latency, such as non-batched inference workloads.

Despite these advantages, harnessing sparsity for efficiency gains is challenging due to existing hardware constraints. The irregular nature of unstructured sparsity creates load imbalances and inconsistent access patterns that reduce overall hardware utilization since most modern general-purpose processors are designed for dense matrix computations. On the other hand, designing custom hardware to address these challenges introduces additional complexities, since mitigating load imbalance and efficiently handling sparse data structures often require costly techniques that can overshadow the benefits of sparsity [32, 33, 34]. As a result, even when networks achieve extreme sparsity levels, sometimes exceeding 90% [35], the observed speedup and energy-efficiency are often limited.

An alternative to unstructured sparsity is structured sparsity, where the locations of non-zero values in the neural network are predefined. Making sparsity predictable facilitates more efficient computation and memory access, reducing the overhead associated with irregular

memory access patterns and enabling scalable acceleration provided the underlying hardware is designed with the knowledge that structured sparsity exists in the workload. Although enforcing structured sparsity typically requires additional retraining to compensate for lost degrees of freedom, it can achieve comparable accuracy to unstructured sparsity and, in some cases, even match the accuracy of the original dense model [25, 36, 37].

Beyond sparsity, low-rank decomposition of weight matrices is another widely adopted compression technique, most recently in transformer architectures [38, 39, 40, 41]. Since linear layers are one of the primary computational bottlenecks in these models, learning a structured low-rank approximation of weight matrices can significantly reduce both computational and memory complexity. The effectiveness of this method relies on the decomposition rank being much smaller than the original feature dimensions, thus lowering the number of required operations. However, despite the theoretical reduction in operations, practical speedups can fall short of expectations, as adding intermediate operations can disrupt efficient execution on existing hardware. These intermediate operations cannot always be fused, leading to increased data movement and latency overheads which can offset the theoretical computational gains.

To address this issue, compression techniques must be designed with memory-aware implementations to ensure that computational reductions also translate into meaningful speedup. A notable example of the importance of memory-aware algorithm design is found in the FlashAttention implementation [42] of the self-attention operation in transformer-based models. While self-attention is fundamental to the model's representational power, its naive implementation on GPUs suffers from severe memory bottlenecks. In particular, moving the attention matrix to and from dynamic random access memory (DRAM) incurs significant latency, especially as the inference context length increases. Optimizing this operation requires a memory-aware implementation that reduces memory traffic, even if that means strategically introducing auxiliary computations.

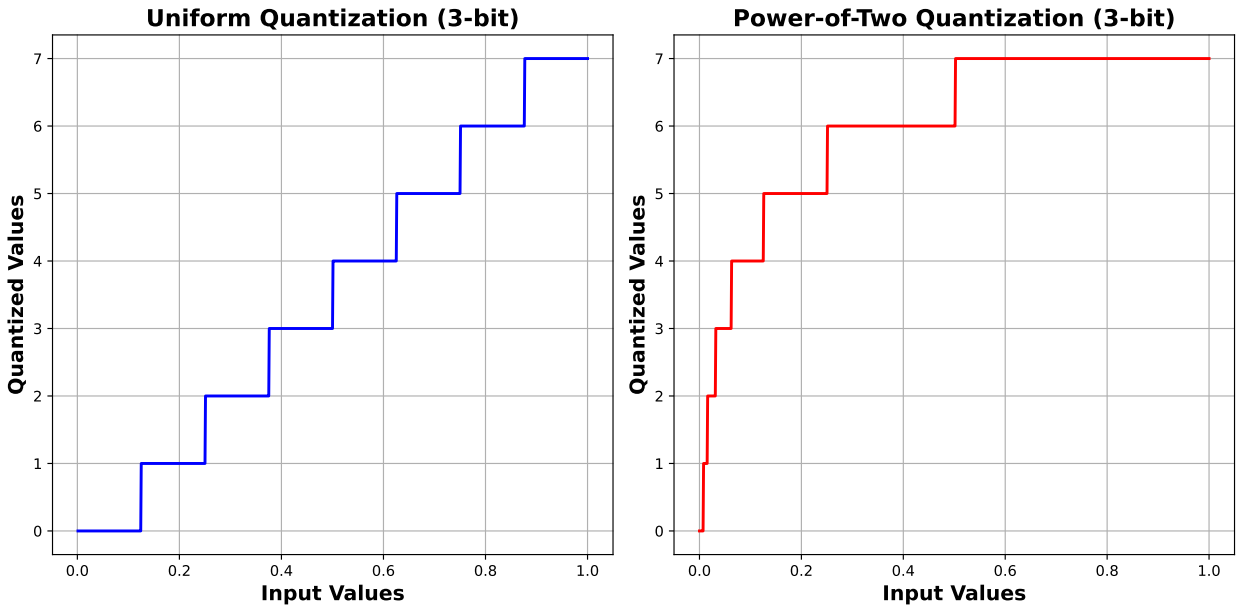


Figure 1.3: Comparison between 3-bit uniform quantization (left) and non-uniform power-of-two quantization (right).

1.2.3 Quantization

Reducing the numerical precision of neural network computations is another effective strategy for improving complexity. Lower bit-width representations significantly decrease the energy consumption of multiply-accumulate (MAC) operations in hardware [43], while also reducing memory traffic and the width of datapaths, thereby improving overall throughput and energy-efficiency. For example, in a 45 nm CMOS technology, an 8-bit integer multiplication consumes 0.2 pJ while a 32-bit integer multiplication consumes 3 pJ [43]. Quantization techniques enable these benefits by mapping high-precision floating-point weights and activations to lower-precision fixed-point or integer representations, thereby reducing both memory footprint and computational complexity.

A widely adopted approach is uniform quantization, where values are evenly spaced over a fixed dynamic range. This approach is illustrated in Figure 1.3 (left) using 3 bits. While applying quantization directly after training can result in significant accuracy degradation, this issue can often be mitigated through quantization-aware training (QAT). Prior research

has demonstrated that QAT allows models to recover much of the accuracy lost due to quantization during training, making it a practical approach for deploying low-precision models across various hardware platforms [44, 45, 46]. However, the extent of improvement depends on the hardware's ability to efficiently support the chosen bit-widths. Without a hardware architecture explicitly designed to exploit low-bit computations, algorithm mappings may underutilize compute resources and memory bandwidth, limiting the efficiency gains.

Beyond uniform quantization, researchers have explored non-uniform quantization strategies, where quantization levels are tailored to exploit hardware-friendly arithmetic operations. One such method is power-of-two quantization, which constrains weight and activation values to powers of two, allowing multiplication operations to be replaced by bit-shifts and additions [47, 48, 49, 50]. This approach is illustrated in Figure 1.3 (right) using 3 bits. Another strategy involves bit-sparsity, where quantization schemes restrict the number of non-zero digits in weight and activation representations. This technique reduces the number of partial products computed during multiplications, further lowering computational cost. Realizing the full benefits of non-uniform quantization often also requires specialized hardware capable of handling variable-bit-width arithmetic and flexible data encoding schemes [51, 52, 53].

Beyond conventional quantization, spiking neural networks (SNNs) [54, 55] have been proposed as an alternative paradigm to traditional artificial neural networks (ANNs) [56, 57]. Unlike ANNs, which perform inference in a single timestep using continuous activations, SNNs process information over multiple timesteps using sparse binary spike trains which is equivalent to a 1-bit activation. This event-driven computation model replaces energy-intensive multiplications with simple accumulations [58].

1.3 Overview of Hardware Techniques

1.3.1 Architecture-Level Techniques

As with any workload, the choice of hardware platform architecture plays a significant role in maximizing throughput and efficiency. Traditionally, central processing units (CPUs) have served as the primary computing platform for a wide range of applications due to their flexibility and general-purpose nature. However, CPUs are not well-suited for deep learning workloads, as they fail to fully exploit key computational properties such as massive data parallelism in high-dimensional matrix operations. Even with vector processing extensions such as Intel's advanced vector extensions (AVX), CPUs lack the scalability required to meet the growing complexity of modern neural networks. GPUs have emerged as a more effective alternative, leveraging their highly parallel architectures to accelerate dense tensor operations. However, they are still fundamentally general-purpose in nature, leading to large chip area and power consumption due to unnecessary control logic and features unrelated to deep learning workloads.

Domain-Specific Acceleration. This has driven the development of domain-specific accelerators (DSAs), which are specialized hardware architectures optimized for the computational demands of specific deep learning applications. By tailoring processing elements (PEs) and memory hierarchies to the structure of deep learning applications, DSAs maximize performance and energy efficiency while minimizing data movement, a critical bottleneck in these computations. Notable examples include Google's Tensor Processing Unit (TPU) [59], NVIDIA's Deep Learning Accelerator (DLA) [60], and custom application-specific integrated circuits (ASICs) such as the edge TPU [61] for low-power inference.

Parallel Pipelined Architectures. Parallel pipelined architectures enhance computational performance by dividing tasks across multiple processors (parallelism) and allowing each processor to handle different stages of a task at the same time (pipelining), leading to increased throughput and reduced processing time. A prime example of this approach is the

systolic array, which has been widely adopted for efficiently executing matrix-matrix multiplications, a core operation in deep learning. The TPU, for instance, employs a 256×256 systolic array where parallel MAC units are arranged in a grid, passing partial sums in a pipelined manner to reduce memory accesses [59]. The efficiency of such architectures is also heavily influenced by the underlying dataflow strategy, which dictates how data is transferred and reused within the processing elements. Selecting an optimal dataflow is crucial, as different strategies maximize data reuse along different dimensions, directly impacting performance and energy efficiency. In weight-stationary dataflows, weights remain fixed in the processing elements while input activations are streamed through the system, reducing redundant memory accesses for weight retrieval. This approach is particularly effective for CNNs performing inference on large images, where filter weights are reused across multiple input patches [59]. Eyeriss [62], an energy-efficient deep learning accelerator, adopts a row-stationary dataflow that balances the reuse of weights, inputs, and outputs, achieving reductions in data movement and power consumption.

Sparsity-Aware Datapaths. Beyond dataflow optimization, modern accelerators incorporate mechanisms to exploit sparsity in neural networks, further improving computational efficiency. Specialized datapaths, such as those featuring zero-gated MAC units and zero-skipping buffers, eliminate unnecessary computations involving zero operands. The SCNN accelerator [32], for instance, employs a compressed sparse representation to store only nonzero elements, along with their positions, reducing memory footprint. SCNN also employs crossbars and read-modify-write accumulator arrays to gather non-zero activation-weight pairs and scatter their product into their corresponding accumulation slots to ensure correct sparse partial sum computation. A notable example of structured sparsity being integrated into modern hardware is the introduction of sparsity-aware tensor cores in NVIDIA GPUs, which leverage a 2:4 sparsity pattern to dynamically skip computations involving zero values [21, 37].

1.3.2 Circuit-Level Techniques

Circuit-level techniques are essential for designing energy-efficient edge ASIC accelerators. Given the tight constraints on area and power in edge devices, it is critical to explore strategies that support on-chip storage of model weights and minimize power consumption during both active computation and idle periods. These optimizations directly impact key system-level goals such as improving energy efficiency, extending battery life, and reducing heat dissipation.

Embedded Non-Volatile Memory. On-chip non-volatile memory (NVM) technologies such as resistive RAM (RRAM) and magnetoresistive RAM (MRAM) have emerged as promising solutions for persistent data storage in edge deep neural network accelerators [63, 64, 65]. Their significantly higher area density compared to static RAM (SRAM) makes them attractive for implementing all-on-chip weight storage, reducing the energy cost of frequent off-chip memory accesses. RRAM offers benefits such as multi-bit storage per cell and lower fabrication cost, while MRAM is advantageous in terms of write endurance and read energy [66]. However, one key challenge in integrating NVMs on-chip is the relatively slow readout speed of large memory macros compared to the fast compute speed of PEs. To address this, designs often associate smaller SRAM-based caches with on-chip NVMs, operating at processor clock speeds and enabling fast access to frequently used weights. This architectural choice introduces the need for communication between components running at different clock domains, which is typically managed using asynchronous FIFO (First-In First-Out) buffers to ensure safe data transfer [67].

Power and Clock Gating. Reducing power consumption in edge accelerators requires effective strategies for both idle and active modes. Power gating is a widely used technique to reduce static power by disconnecting unused circuit blocks from the power supply when they are idle [68]. This is usually implemented by placing a p-channel metal–oxide–semiconductor (PMOS) power switch between the power rail and the block being gated. The impact of power gating is particularly significant in low-voltage operation as well as in the context of on-chip

NVMs [69], where static power can become more prevalent and various power gating modes offer different tradeoffs between leakage current and wake-up latency. For reducing active power, clock gating is another common approach, readily supported by modern computer-aided design (CAD) tools [70]. In its simplest form, clock gating involves using an enable signal to control the clock signal using an AND gate and a latch. When the enable is high, the clock propagates to the registers, allowing normal operation. However, when the block is not in use, setting the enable low prevents unnecessary clock toggling, thereby avoiding dynamic power dissipation in unused logic.

1.4 Dissertation Overview

Motivated by the recent success of domain-specific accelerators in improving energy efficiency and performance for edge AI, and in response to the growing complexity of deep neural networks, this dissertation presents four projects that co-design specialized hardware and neural networks. Together, these projects aim to: (1) maximize performance and energy efficiency gains from software-level optimizations, (2) mitigate accuracy loss from model simplification through hardware-aware design, and (3) ensure that algorithmic efficiency translates to real-world speedups, even on existing platforms such as GPUs. The organization of the rest of the dissertation is described below.

Chapter 2 introduces SONA, an accelerator architecture for heterogeneous transform-domain neural networks (HTNNs) with structured sparse-orthogonal weights. SONA leverages an HTNN-specific output-stationary dataflow and an optimized transform cache organization to reduce transform-domain computation overhead. It also incorporates a column-based block (CBB) structured sparsity pattern for fully connected layers (FCLs), enabling unified support for sparse convolutions and FCLs. To further improve efficiency, it features a bit-sparse canonical-signed-digit MAC unit that replaces multiplications with shifts and additions.

Chapter 3 details the first all-on-chip-weight silicon implementation of a structured-sparse HTNN accelerator. Using dense, retentive MRAM for weight storage, the design eliminates costly off-chip memory access, improving energy efficiency. It introduces layer-wise power gating, a double-buffered weight cache with pipelined prefetching, and decoupled MRAM/-compute clocks to reduce both active and leakage energy. Fabricated in 22 nm CMOS, the processor achieves a 40% energy reduction and peak efficiencies of 9.51 TOPS/W (convolution) and 22.86 TOPS/W (FCL).

Chapter 4 proposes a novel multi-level leaky integrate-and-fire (M-LIF) neuron model to improve the accuracy-energy tradeoff in SNNs. Unlike traditional LIF models, which rely solely on timestep increases, M-LIF SNNs also introduce a tunable dimension to encode power-of-two spike values. This preserves SNNs' multiplication-free operation while achieving higher accuracy with fewer timesteps and lower energy consumption.

Chapter 5 explores structured matrix decomposition techniques [38, 39] to accelerate foundation model inference for long sequences. This work identifies key inefficiencies in data movement after compilation via PyTorch that push multi-token inference into the memory-bound regime compared to traditional low-rank and dense methods. Using OpenAI's Triton [71], it proposes fused kernels, operation reordering, and tailored memory layouts to boost arithmetic intensity and memory access efficiency, achieving considerable speedups on resource-constrained GPUs.

CHAPTER 2

SONA: A Hardware Accelerator Architecture for Transform-Domain Neural Networks with Sparse-Orthogonal Weights

2.1 Introduction

Convolutional neural networks (CNNs) have become a fundamental technique in machine learning tasks, but their extension to low-cost and energy-constrained applications is limited by CNN model sizes and complexity. Recently, pruning methods aimed at reducing the number of non-zero parameters have been proposed to decrease model size and lower complexity [24, 23]. In turn, sparsity-aware accelerators that directly leverage sparsity have also been proposed [34, 72, 32, 73, 33] to improve the energy-efficiency of inference tasks.

Another way to reduce complexity is through transform-domain computations that replace convolutions in CNNs with element-wise multiplications. However, it is difficult to combine both sparsity and transform-domain computations since transform-domain models often do not allow aggressive weight pruning as shown in [74, 75]. Moreover, the sparsity that these works exploited was unstructured (random). Applying sparsity-aware architectures to these transform-domain models with less aggressive pruning can yield limited gains due to the unstructured nature of sparsity which imposes significant overheads on accelerator flexibility or results in reduced hardware utilization.

To overcome these challenges, a heterogeneous transform-domain neural network (HTNN) [36] was proposed as a framework to learn structured sparse-orthogonal weights where convolutions are replaced by element-wise multiplications. In an HTNN, two or more kernels in different transform domains share a multiplier without conflict as the non-zero weight positions are strictly orthogonal to each other. The authors in [36] demonstrate various CNN workloads that can be trained, pruned, and quantized in heterogeneous transform domains while maintaining inference accuracy. However, the expectation that HTNNs [36] can reduce computational complexity compared to equivalent sparse CNNs has not been demonstrated in hardware. Efficiently mapping HTNN models to a hardware architecture in a way that maximizes observable gain remains a significant challenge.

This work proposes SONA, a novel energy-efficient accelerator architecture for HTNNs. The proposed architecture employs an HTNN-specific output stationary dataflow coupled with an energy-efficient transform memory organization to reduce the overhead of overlapped transform-domain convolution. The proposed architecture demonstrates reconfigurable datapaths to compute the permuted variants of the Walsh-Hadamard transform (WHT) [76] for concurrently executed transform-domain kernels. Moreover, this work extends the sparse-orthogonal weight concept of HTNNs to fully connected layers (FCLs) by proposing a column-based-block (CBB) structured sparsity pattern. Structured sparsity in FCLs allows SONA to share a unified datapath between sparse convolution and sparse FCLs without compromising multiply-accumulate (MAC) array utilization. Furthermore, HTNN employs non-uniformly quantized bit-sparse canonical-signed-digit (BS-CSD) weights. At the circuit level, SONA proposes a novel BS-CSD-MAC unit (CMU) to replace multiplications with bit-shifts and additions.

This work evaluates SONA on CIFAR-10/100 [77] and ImageNet [78] and compares performance and energy-efficiency against representative dense and sparse deep neural network accelerator designs. Compared to prior dense and sparse CNN accelerators, SONA can reduce inference energy by 5.1× and 2.4×, and increase performance by 5.2× and 2.1×,

respectively, for convolution layers. For sparse FCLs, SONA can reduce inference energy by 2.4× and increase performance by 2.1× compared to prior work. The main contributions of this work are:

- SONA, the first demonstration of an energy-efficient hardware accelerator architecture for HTNNs with an HTNN-specific output stationary dataflow coupled with an energy-efficient memory organization.
- A CBB structured sparsity pattern for FCLs which enables SONA to share the datapath between sparse transform-domain convolution and sparse FCLs without compromising MAC array utilization.
- A BS-CSD-MAC unit to replace multiplications with bit-shifts/additions and concurrently process output channels.
- An evaluation of SONA against prior dense and sparse DNN accelerators using simulator as well as RTL-based implementations on CIFAR-10/100 and ImageNet.

2.2 Background

2.2.1 WHT-Domain Convolution

First, essential terminology used throughout the discussion is defined. An overview of the two-dimensional WHT is provided, explaining how convolutional layers are replaced by WHT-domain element-wise multiplications in HTNNs [36]. A CNN consists of a set of cascaded layers that includes linear layers such as convolution and non-linear layers such as ReLU. The term activations is used to refer to elements of a given layer's input feature map. The input feature map is of size $N_x \times N_y \times I_C$, where N_x and N_y denote the input feature map's spatial dimensions and I_C denotes the layer's number of input channels. The set of learnable parameters in a given linear layer is referred to as weight kernels. The cardinality of this set corresponds to the layer's number of output channels, denoted by O_C.

Each weight kernel is a three-dimensional tensor $K_x \times K_y \times I_C$ where K_x and K_y denote the spatial dimensions of the kernel.

WHT [76] is a generalized class of Fourier transforms given by a symmetric transform matrix $\mathbf{H}$ that contains only binary values ± 1. Transform-domain convolution consists of a WHT, element-wise multiplication, and inverse WHT. Although transform-domain convolution in HTNNs plays the same role as convolution in CNNs, they are not mathematically equivalent as explained in [36]. Furthermore, although WHT-domain convolution is generalizable to kernels of any size, this work focuses on the case proposed in [36] where stride-1 3×3 convolutions are replaced by stride-2 4×4 WHT-domain kernels given that it is a very popular configuration in CNNs. Note that the 1×1 point-wise convolution commonly used in MobileNetV1/2 for example [79, 80] is identical in both HTNNs and CNNs.

As shown in Figure 2.1 (red), WHT-domain convolution is based on $4 \times 4 \times I_C$ activation patches $\mathbf{X_p}$ extracted from the $N_x \times N_y \times I_C$ input feature map with a stride of 2×2. To calculate the 2×2 output channel $\mathbf{Y_p}$ of a layer for patch position $\mathbf{p}$, a two-dimensional 4-point WHT is first applied to each channel in $\mathbf{X_p}$ to yield $\mathbf{T_p} = \mathbf{H_{WHT}}^{\mathrm{T}} \mathbf{X_p} \mathbf{H_{WHT}}$ where $\mathbf{H_{WHT}}$ is the WHT matrix Equation (2.1).

This two-dimensional WHT can be seen as applying a 4-point WHT to the rows and then columns of each channel in $\mathbf{X_p}$. Similarly, $\mathbf{H_{WHT}}$ would be applied to the corresponding weight kernel. However, a kernel can be trained offline in the WHT-domain [36], meaning that its transform domain representation $\mathbf{K}$ can be directly used during inference without explicit transform computations. Subsequently, each channel in the transform domain kernel $\mathbf{K}$ and the WHT representation of $\mathbf{X_p}$ are multiplied element-wise $\mathbf{T_p} \odot \mathbf{K}$, and their $4 \times 4 \times I_C$ intermediate product is reduced to a 4×4 patch $\mathbf{Z_p}$ by accumulating along the input channels as shown in Figure 2.1. After applying the inverse WHT ($\mathbf{H_{WHT}}^{-1}$) to $\mathbf{Z_p}$, the output patch $\mathbf{Y_p}$ is obtained by taking the central 2×2 block of the inverse transform result. Thus the inverse transform can be simplified to applying a binary-valued 4×2 matrix $\mathbf{A_{IWHT}}$ in Equation (2.1) ($\mathbf{A_{IWHT}}^{\mathrm{T}} \mathbf{Z_p} \mathbf{A_{IWHT}}$), which corresponds to the middle two

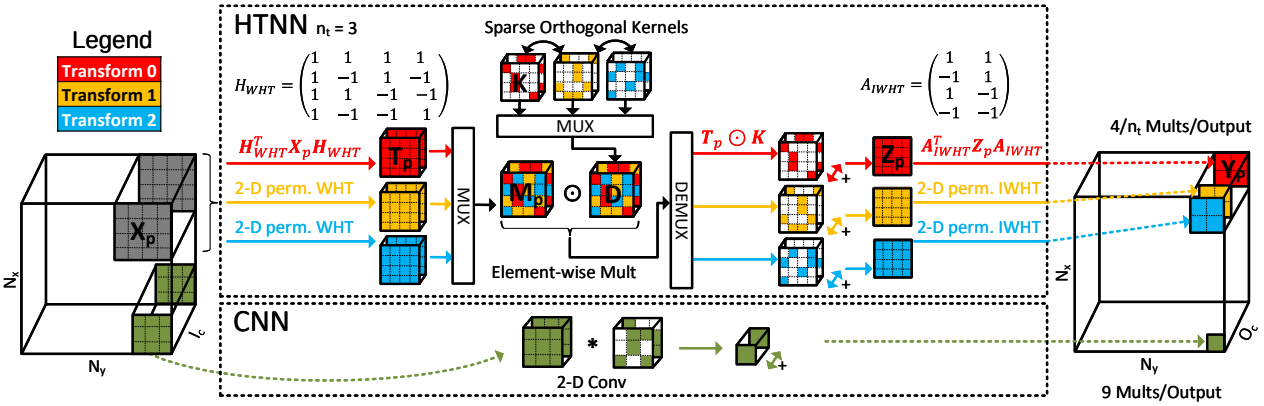


Figure 2.1: Comparison between HTNN WHT-domain convolution layer and conventional CNN convolutional layer.

columns of $\mathbf{H_{WHT}}^{-1}$. The entire operation is summarized in Equation (2.2). Since the (inverse) WHT matrix only contains binary values, it can be computed using additions/subtractions. Therefore, compared to a 3×3 convolution that requires 9 multiplications per output activation, the computational complexity is reduced by a factor of $2.25\times$ down to 4 multiplications per output activation.

$$\mathbf{H_{WHT}} = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{pmatrix}, \ \mathbf{A_{IWHT}} = \begin{pmatrix} 1 & 1 \\ -1 & 1 \\ 1 & -1 \\ -1 & -1 \end{pmatrix} \tag{2.1}$$

$$\mathbf{Y_p} = \mathbf{A_{IWHT}}^{\mathrm{T}} \Big[\big[\mathbf{H_{WHT}}^{\mathrm{T}} \mathbf{X_p} \mathbf{H_{WHT}} \big] \odot \mathbf{K} \Big] \mathbf{A_{IWHT}} \tag{2.2}$$

2.2.2 Sparse-Orthogonal Kernels in HTNNs

The HTNN framework [36] imposes a sparse-orthogonality property to transform domain convolutions. The framework deliberately searches multiple heterogeneous transforms, thus

the position of important weights in one transform domain is less likely to overlap with the ones in other transform domains. In HTNN, kernels belonging to different transforms are pruned in such a way that their non-zero weight positions do not overlap (*i.e.*, sparse-orthogonal). This potentially allows for efficient parallelized computation of multiple sparse-orthogonal kernels in hardware accelerators. The number of heterogeneous transform domains/variants in an HTNN is typically $n_t = 2$ or 3, and each transform domain is defined by multiplying a unique permutation matrix $\mathbf{P}$ to the left of $\mathbf{H_{WHT}}$. Kernels that are sparse-orthogonal to each other can be gathered to form a dense overlapped kernel $\mathbf{D}$ to process n_t output channels concurrently as shown in Figure 2.1 for $n_t = 3$. To concurrently compute the n_t output channels for a given patch position $\mathbf{p}$, activation patches $\mathbf{X_p}$ are transformed in n_t different WHT variants and overlapped using multiplexers to match the overlapping pattern of $\mathbf{D}$. The resulting overlapped activation patches $\mathbf{M_p}$ are multiplied element-wise with $\mathbf{D}$. The intermediate product's overlapping is then undone with de-multiplexers to accumulate across the input channels for each of the n_t concurrently computed output channels. Finally, the n_t accumulated patches are inverse transformed in their respective WHT domains to recover the n_t output channels.

2.2.3 Bit-Sparse Canonical-Signed-Digit (BS-CSD) Computation

HTNN [36] extends the structured sparsity concept to the number representation system to further reduce the hardware complexity by training weights in a BS-CSD format. CSD [81] is a numbering system that uses ternary digits $\{-1, 1, 0\}$ to represent an N-bit number with reduced non-zero digits. For example, the number 23 requires 4 non-zero bit in the conventional binary representation 010111_2 but only requires 3 non-zero digits in the CSD representation $(1, 0, -1, 0, 0, -1)$ since $23 = 2^5 - 2^3 - 2^0$ holds. To further take advantage of the CSD representation, HTNNs impose an additional bit-sparsity constraint to limit the number of non-zero digits in the CSD representation of weights to at most 2. This technique allows replacing multiplication with a single addition/subtraction without impacting the

inference accuracy [36]. SONA implements 8-bit input MAC units for BS-CSD weights that are non-uniformly quantized with at most 2 non-zero digits, whereas activations are uniformly quantized with 8-bit two's complement (non-CSD) representation.

2.2.4 Fully-Connected Layer (FCL) with Structured Sparsity

FCLs constitute a large portion of the weights that can be pruned to sparse matrices. Accelerators optimized for sparse FCLs such as [72] have shown that the location of non-zero weights significantly impacts compute and memory resource utilization. The choice between inner or outer-product matrix-vector multiplication [82, 83] can have an impact on the compressed weight representation and the hardware's ability to leverage dynamic activation sparsity. Despite employing compressed formats for data storage, unstructured sparsity in FCLs leads to extraneous zero padding and load imbalance where processing elements (PEs) are assigned a varying number of non-zero weights to multiply the same activation value. Although the original HTNN [36] addresses sparse convolution overhead, it does not provide a framework for learning structured sparsity for FCLs. To overcome this, SONA introduces a new column-based-block (CBB) structured sparsity pattern extending HTNNs to sparse FCLs under a unified architecture. HTNNs are trained with CBB sparse weights and employ an index-based weight-encoding representation to minimize the overhead of zero padding while maximizing the potential memory and MAC utilization and sharing the same hardware with transform-domain convolutions.

2.3 SONA Architecture and Implementation

The proposed SONA architecture is motivated by the drawbacks of existing unstructured sparse deep neural network accelerator architectures such as [32] and [72] as well as by the promise of HTNNs. Although the authors in [36] claim that HTNNs can achieve $4.9 - 6.8\times$ complexity reduction compared to sparse CNN models without compromising the inference

accuracy, this result has not been demonstrated in hardware. While the software implementation of HTNNs performs fewer operations, the expected gain is much smaller as it incurs overhead in handling WHT and sparse-orthogonality. The theoretical analysis for HTNN's gain in [36] does not address challenges at the hardware level for devising an accelerator architecture that can efficiently exploit the properties of HTNNs to maximize the observable gain.

2.3.1 Overall Architecture

Figure 2.2 shows the overall SONA architecture that unifies WHT-domain sparse-orthogonal convolution and CBB-sparse FCLs. All of the weights and activations are assumed to be stored initially in off-chip DRAM before being loaded into weight and activation memories, respectively. Weights are converted into their BS-CSD representation (Section 2.3.6) before being loaded into weight memory. SONA makes use of a $4 \times 4 \times n_t$ BS-CSD MAC unit (CMU) array and a $N_P \times n_t$ transform datapath array to process N_P patches in parallel using n_t WHT variants.

The execution of a WHT-domain convolution layer is illustrated in Figure 2.2. Paths that are not active during transform convolution are greyed out. Different colors in Figure 2.2 indicate $n_t = 3$ WHT variants with sparse-orthogonal weights that are processed concurrently. The nested-loop control to sweep the patch position, output channels, and input channels follows an output stationary (OS) ordering (Section 2.3.3). Once weights and activations are loaded into their respective memories, an input channel tile of size I for N_P 4×4 patches are sent to the transform array (Section 2.3.2). The resulting $N_P \times n_t$ transformed $4 \times 4 \times I$ patches are stored in transform memory consisting of $4 \times 4 \times n_t$ banks of depth I and data width $N_P \times 8$ bits. Transform memory organization details are discussed in Section 2.3.4. Subsequently, SONA performs element-wise BS-CSD MAC operations in the CMU array. All N_P activation $4 \times 4 \times I$ patches are multiplied by the same $4 \times 4 \times I$ kernel that merges n_t sparse-orthogonal weights to concurrently process n_t output channels, which are

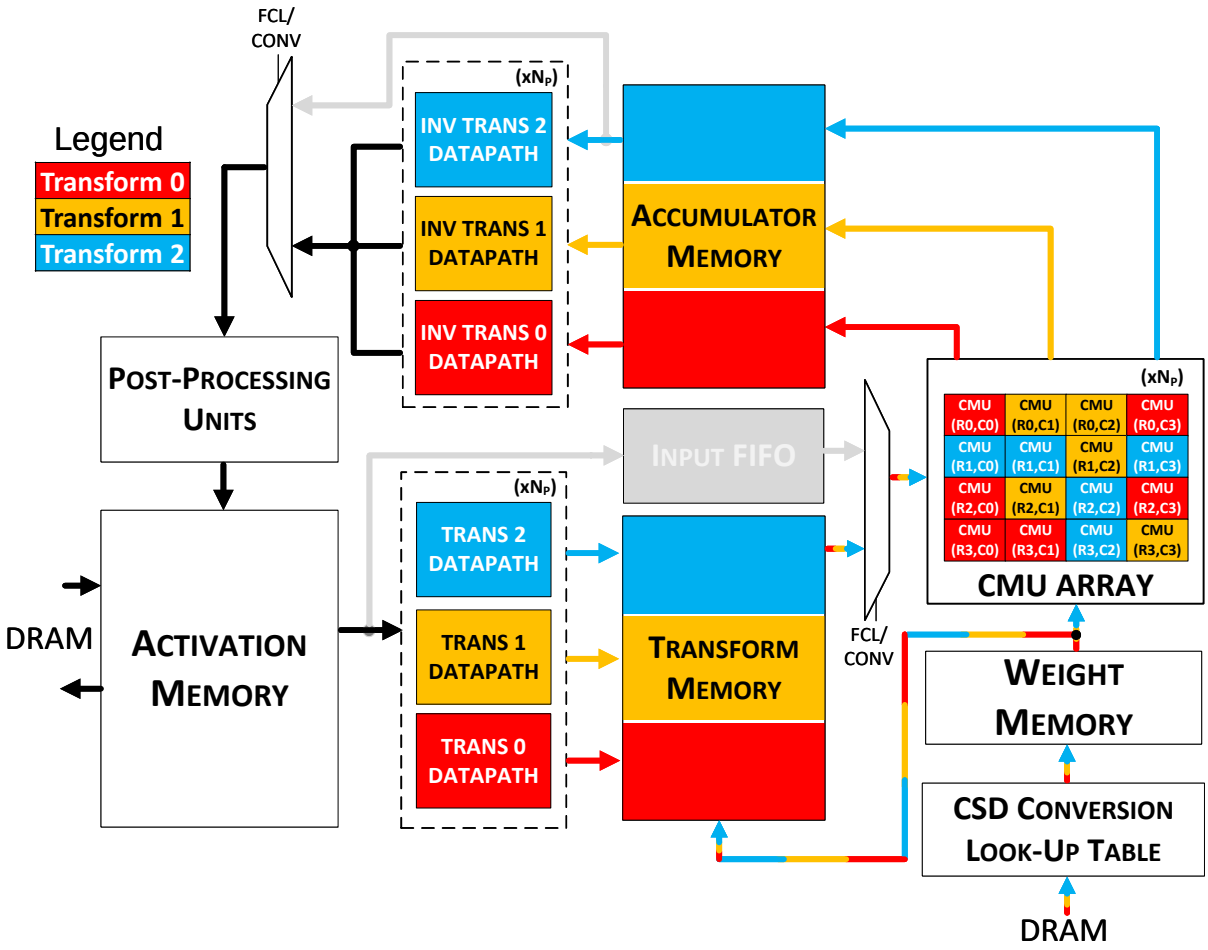


Figure 2.2: Overall architecture with highlighted WHT convolution layer path.

color-coded in Figure 2.2. Accumulation across the I channels is done locally at the CMU level. Upon completing the accumulation for a tile, the following group of orthogonal output channels is processed to maximize the reuse of the transformed activations. If the layer's number of input channels is larger than I, the intermediate result from the previous set of output channels is stored in accumulator memory that consists of $4 \times 4 \times n_t$ banks of depth $\frac{O_C}{n_t}$ and data width $N_P \times 24$ bits.

After all the layer's weight kernels and input channels have been processed, N_P accumulated $4 \times 4 \times O_C$ patches are sent to the inverse transform array and post-processing units (*e.g.*, ReLU) while the next set of N_P patches are transformed. The final $2 \times 2 \times O_C$ patches are written to the activation memory.

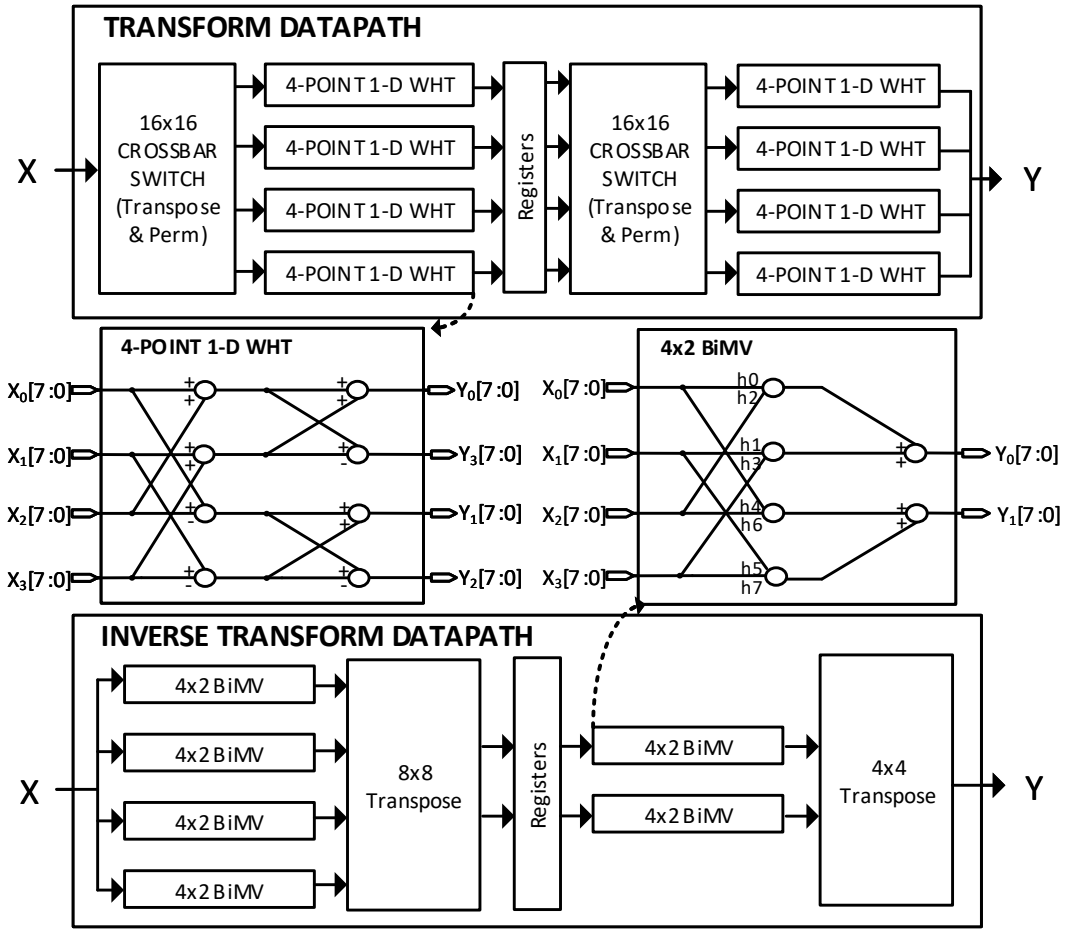


Figure 2.3: Permuted reconfigurable transform datapaths

2.3.2 Reconfigurable Transform Datapaths

The reconfigurable transform datapaths need to handle different permuted variants of the two-dimensional WHT, which are defined as $\mathbf{H_P} = \mathbf{PH_{WHT}}$ where $\mathbf{P}$ is the corresponding permutation matrix. A 4×4 two-dimensional non-permuted fast WHT requires $8 \times 8 = 64$ adders/subtractors. The transform operation can be reordered and split into two back-to-back identical operations as in Equation (2.3). First, the 4×4 input patch $\mathbf{X}$ is transposed and permuted. A transform is then applied to each row of the intermediate result. The operation is repeated a second time to produce the final transformed patch $\mathbf{Y}$. A diagram of the

proposed transform datapath is shown in Figure 2.3 (top). The transpose and permutation operations are combined into a single 16×16 reconfigurable crossbar switch. To implement the permuted variants of the inverse WHT operation as defined in $\mathbf{Y} = \mathbf{A_P}^T\mathbf{X}\mathbf{A_P}$, one might note that $\mathbf{H_{WHT}}$ is an orthogonal matrix and that the WHT transform datapath can be reused in Figure 2.3 (top) to compute the inverse WHT. However, this would be wasteful since only the central 2×2 block from the resulting operation is needed. The permuted inverse WHT variants ($\mathbf{A_P}$) correspond to the binary-valued (± 1) middle two columns of the $\mathbf{H_P}^{-1}$ matrices. Therefore, the inverse transform is implemented using reconfigurable 4×2 binary matrix-vector multiplication (BiMV) blocks and transpose interconnect networks as shown in Figure 2.3 (bottom). Each datapath requires $6 \times 6 = 36$ adders/subtractors but no multiplications. SONA employs $n_t \times N_P$ datapaths for each of the (inverse) transform to process N_P individual 4×4 patches using up to n_t variants in parallel. These datapaths are pipelined to not limit the performance of the overall architecture.

$$\mathbf{Y} = \mathbf{H_P}^T\mathbf{X}\mathbf{H_P} = (((\mathbf{X}^T\mathbf{P})\mathbf{H_{WHT}})^T\mathbf{P})\mathbf{H_{WHT}} \tag{2.3}$$

2.3.3 WHT-Domain Sparse-Orthogonal Convolution Dataflow

The choice of dataflow dictates memory access patterns and therefore plays a significant role in maximizing the efficiency of accelerators. Although CNN accelerator dataflows have been extensively studied [62, 84, 85, 86], those are not directly transferable to HTNN because its internal dataflow is dissimilar to that of ordinary convolution. Therefore, the choice of heterogeneous WHT-domain convolution dataflow was studied prior to conceiving SONA.

Let the input feature map, output feature map, and weight kernels be of size $N \times N \times I_C$, $N \times N \times O_C$, and $O_C \times I_C \times 4 \times 4$, respectively. When the HTNN layer uses n_t WHT domains, its computation loops over three parameters: patch position $\mathbf{p}$, orthogonal output channels $\frac{O_C}{n_t}$ (n_t orthogonal channels are computed together), and input channels I_C. For a candidate architecture, the memory sizes as well as the number of memory accesses along

Order 1: Output Stationary (OS)

```
for p in range(0, N^2/4)
  for o in range(0, OC/NT)
    for i in range(0, IC)
      result += DEMUX(
      MUX(trans_buff[i])
      .* wt_buf[o][i])
```

Order 2: Weight Stationary (WS)

```
for o in range(0, OC/NT)
  for i in range(0, IC)
    for p in range(0, N^2/4)
      acc_buff[p][o] += DEMUX(D
      .* MUX(trans_buff[p][i]))
```

Order 3: Input Stationary (IS)

```
for p in range(0, N^2/4)
  for i in range(0, IC)
    for o in range(0, OC/NT)
      acc_buff[o] += DEMUX(Mp
      .* wt_buff[o][i])
```

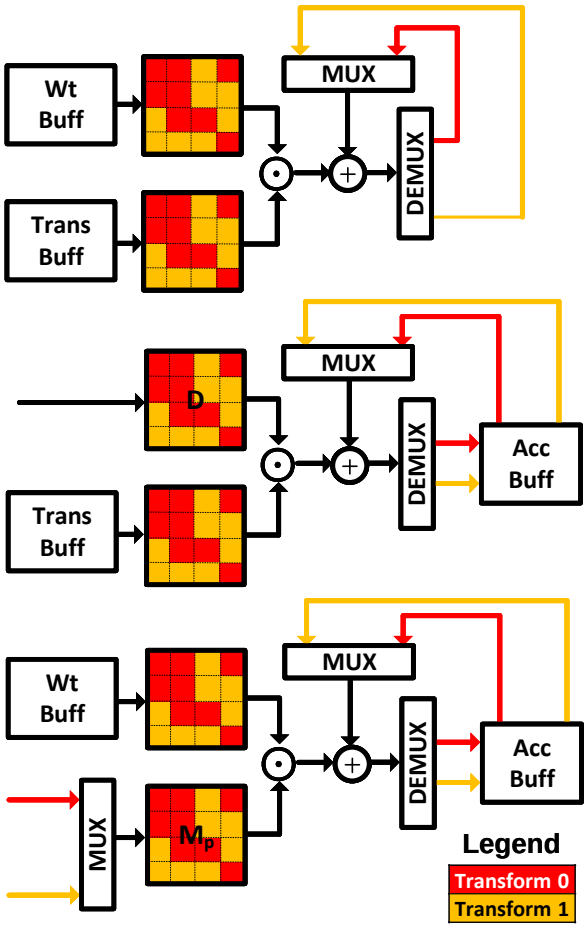


Figure 2.4: HTNN pseudcode for three candidate dataflows.

the datapath will depend on the order in which input and output channels are processed.

In Figure 2.4, three WHT-domain convolution dataflows corresponding to different re-orderings of the loops are outlined. The first is *output stationary* (OS), where the input is processed by applying n_t WHTs to each input channel patch at a given position. Transformed patches are then reused by multiple weight kernels and are stored in a buffer. Weight kernels are also reused across different patch positions and are stored in a buffer. However, output product patches are accumulated along the input channels which can be done locally at the processing element (PE) level. The second dataflow is *weight stationary* (WS), where the kernel weight is fixed at the PE level. The input is processed by applying n_t WHTs to all

patch positions in a single input channel. Transformed patches are reused by multiple kernels and are therefore stored in a buffer. The third dataflow is *input stationary* (IS). The input is processed by applying n_t WHTs to a single patch position at a single input channel, meaning there is no buffer for transformed activations. The weight kernels are reused across different patch positions and are stored in a buffer. Finally, in IS and WS dataflows, accumulation cannot be done locally at the PE level. A buffer is needed to store the output product patches, but the buffer size in IS is smaller than that in WS.

To identify an energy-efficient dataflow, a case study was performed on WHT-domain convolution layers of ResNet-20 and ConvPool-CNN-C. The buffer SRAM sizes and the number of read and write accesses were quantified in terms of generic layer parameters for the candidate dataflows. A memory architecture was then outlined where each layer of the studied network has on-chip buffer SRAM macros that are sized to fit its layer parameters without needing to tile within the layer, therefore off-chip memory accesses will be identical for all dataflows and are excluded. TSMC 28 nm memory compilers were employed to obtain unit access energies for SRAM macros. PE-internal register access was excluded as its contribution is negligible relative to the SRAM macro access energy contribution.

The tradeoffs shown in Figure 2.5 in terms of memory access energy cost for each dataflow indicate that the OS dataflow is the most energy efficient for WHT-domain convolution. Although similar observations were made for the conventional CNN dataflow [62], it is not straightforward to arrive at the same conclusion because the internal dataflow of heterogeneous WHT-domain convolution is dissimilar to that of ordinary convolution. WS and IS dataflows are primarily less energy efficient for WHT-domain convolution because they suffer from a large amount of read-modify-writes, which heavily contribute to the energy cost, whereas accumulations can be done at the PE level in the OS dataflow. OS benefits from the fact that the bandwidth of the transform buffer is less than that of the accumulator buffer because only $1/n_t$ of the transformed patches are used in a cycle to concurrently process n_t output channels. Despite the write bandwidth of the transform buffer being larger

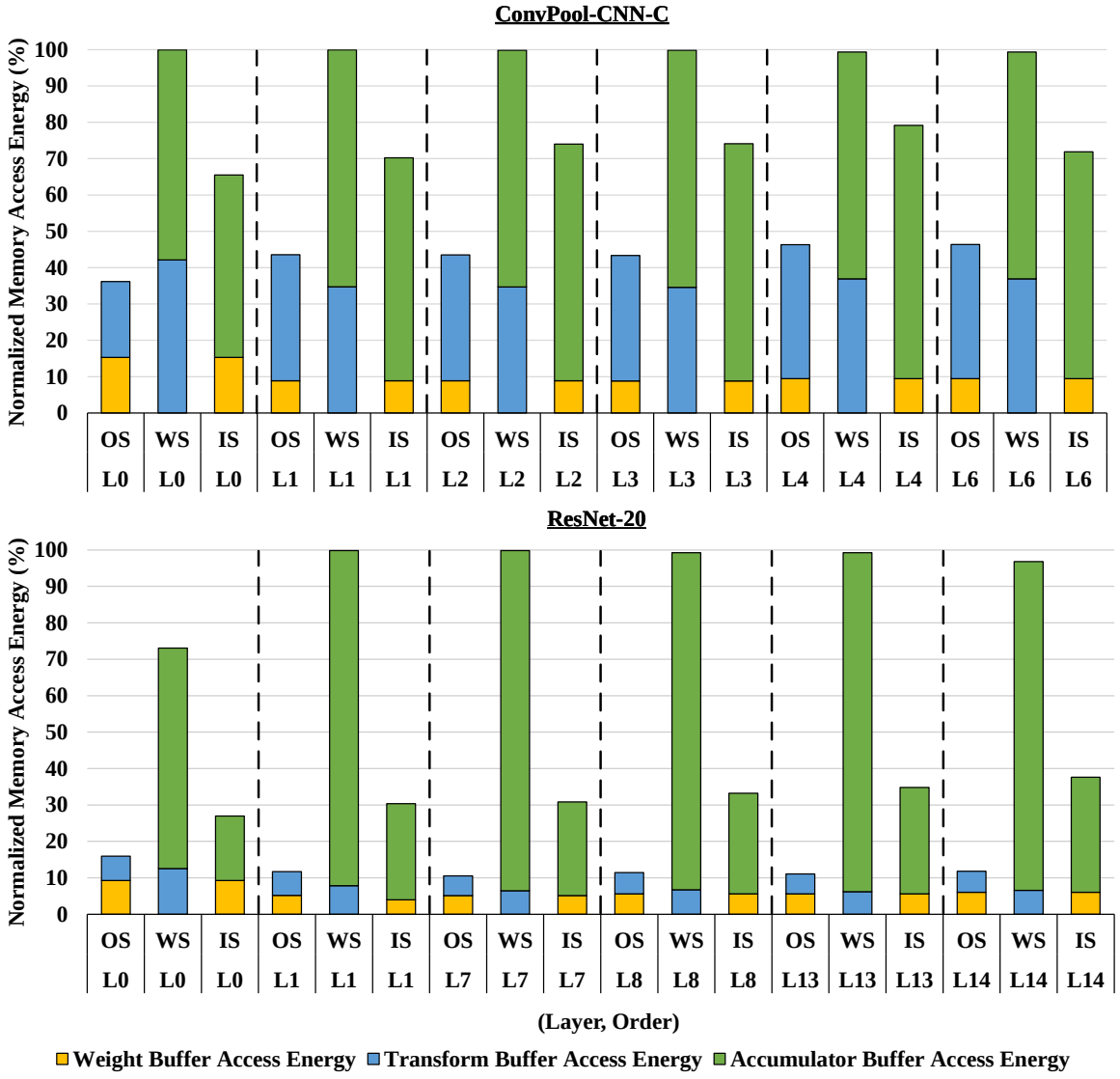


Figure 2.5: Memory access energy comparison for ConvPool-CNN-C and ResNet-20 against HTNN dataflows: output/weight/input stationary (OS/WS/IS)

than that of the accumulator buffer, the transform buffer is written to less frequently since the transformed patches are re-used for multiple kernels. To take advantage of this fact, the HTNN-specific OS dataflow requires an efficient memory organization for accessing the transform buffer, which will be discussed in the next section.

2.3.4 Memory Organization

Prior to being stored in weight memory, $n_t \leq 3$ orthogonal weight kernels are overlapped together. Each weight is associated with a 2-bit mask to indicate its corresponding WHT variant. The input activation patches are transformed in all n_t variant domains and re-used across the output channel dimension, but only $1/n_t$ are used during element-wise multiplications. It must be noted that the overlapping pattern is different from channel to channel within a single layer, which makes reusing transformed patch non-trivial. Therefore, it is critical to devise an energy-efficient memory organization that limits the access to only the required transformed patches in each cycle.

Assuming that N_P patches are processed in parallel, the transform memory is expected to hold $I \times N_P$ 8-bit transformed activation patches where I is the tile size for the number of input channels. The read and write bandwidths of this memory are $4 \times 4 \times N_P$ and $4 \times 4 \times N_P \times n_t$, respectively. One approach referred to as single patch single row (SPSR) consists of having $N_P \times n_t$ banks of depth I and word width $4 \times 4 \times 8$ bits. Another approach referred to as single activation single row (SASR) consists of having $N_P \times n_t \times 4 \times 4$ banks of depth I and word width 8 bits. SASR provides more flexibility than SPSR in controlling which activations are read during a cycle. With SPSR, $n_t \times N_P$ patches are read when only N_P patches are needed. In other words, SASR helps limit the number of unnecessary memory accesses. However, SASR incurs a larger overhead for peripheral memory circuity from employing many more smaller banks and therefore has the potential to be less area and energy efficient than SPSR.

As a compromise, this work proposes the multiple activation single row (MASR) scheme illustrated in Figure 2.6. In MASR, there are $n_t \times 4 \times 4$ banks of depth I and word width $N_P \times 8$ bits. This approach was chosen since the overlap pattern is solely dependent upon the kernel being processed and all N_P patch positions are multiplied by the same kernel in a cycle. When loading the transformed patches from transform memory, the weight mask can be used to disable $\frac{n_t - 1}{n_t}$ of the banks and load only the N_P overlapped transformed patches

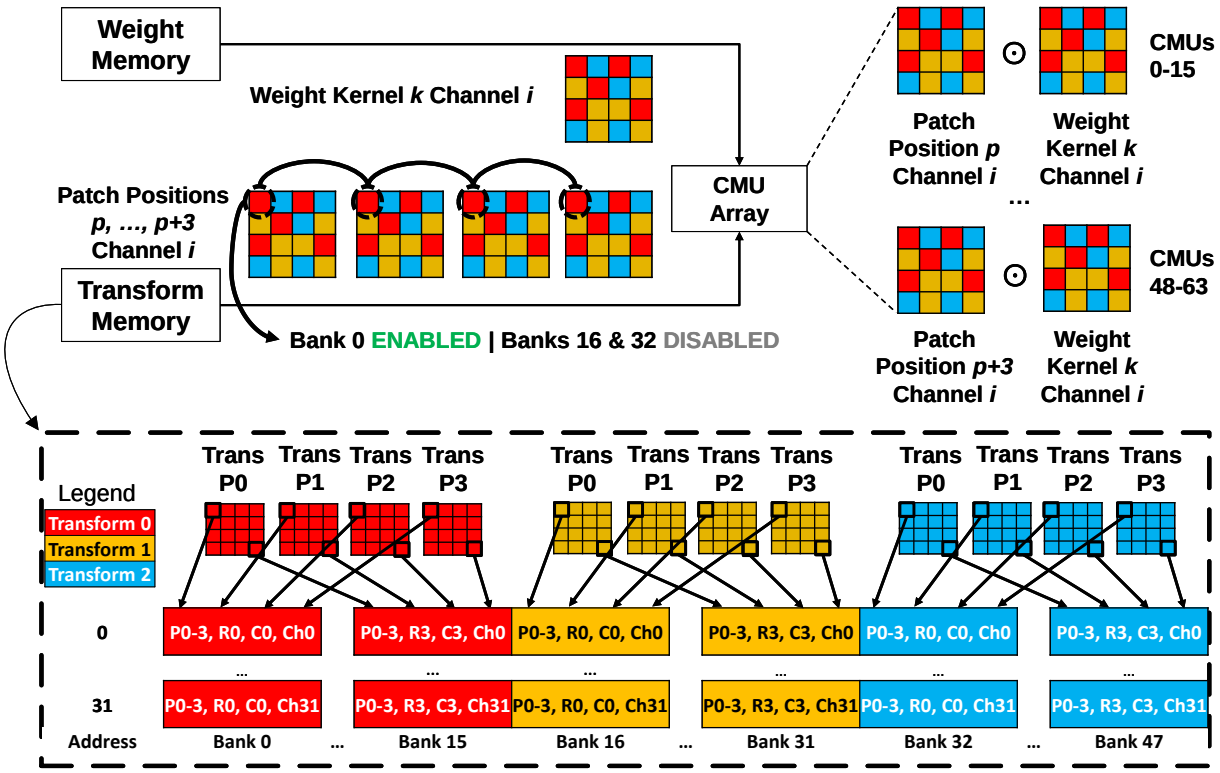


Figure 2.6: MASR transform memory organization for mapping transform activations onto memory banks and words ($I = 32$, $N_P = 4$, $n_t = 3$).

that are needed. Figure 2.6 illustrates MASR for $I = 32$, $N_P = 4$, $n_t = 3$. The top left weight of the loaded kernel is associated with variant 0, and all the activations of each transformed patch using variant 0 are in bank 0. For element-wise multiplication, that bank is enabled and the other two that contain the activations transformed using variants 1 and 2 at that location are disabled.

Experimental results using Arm register file compilers in TSMC 22nm technology indicate that for $I = 32$, MASR has $\{1.2\times, 1.7\times\}$ and $\{1.6\times, 2.4\times\}$ less access energy than SPSR and SASR, respectively, for $N_P = \{2, 4\}$ at the cost of being $\{1.8\times, 1.3\times\}$ less area efficient than SPSR. Note that SASR is overall the most flexible but least area-efficient approach and it is not necessarily energy efficient as the increased number of small memory banks incurs energy overhead to peripheral circuitry for memory banking. Thus to exploit patch

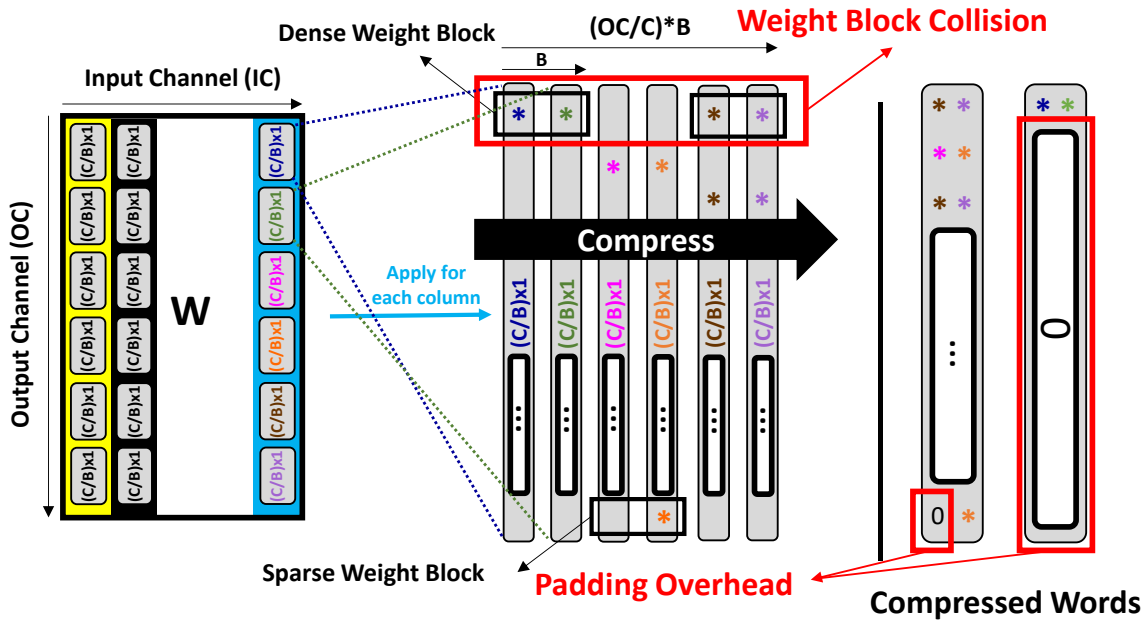


Figure 2.7: Overhead of index-based encoding methods for FCL weight matrices with unstructured sparsity.

parallelism, MASR becomes necessary to maximize the energy efficiency of the design.

2.3.5 Column-Based-Block Sparse Fully-Connected Layer

To fully exploit both weight and activation sparsity, an outer-product sparse FCL implementation based on an input stationary dataflow was explored. To best explain the proposed scheme, first consider a case where an index-based compression method is employed to represent the weight matrix **W** where the location of non-zero weight values are random (unstructured) as illustrated in Figure 2.7. The original column is first divided into equal-sized chunks of size C/B and reshaped into a C/B-row matrix, where C is the number of weights in a compressed word and B is the size of a weight block. Then to compress **W** (remove zeros) along its columns, the weights are packed densely in the direction shown in Figure 2.7. Each compressed word contains C/B blocks of weights (*i.e.*, C weights per word). Each weight is associated with the column number that it originates from in the C/B-row

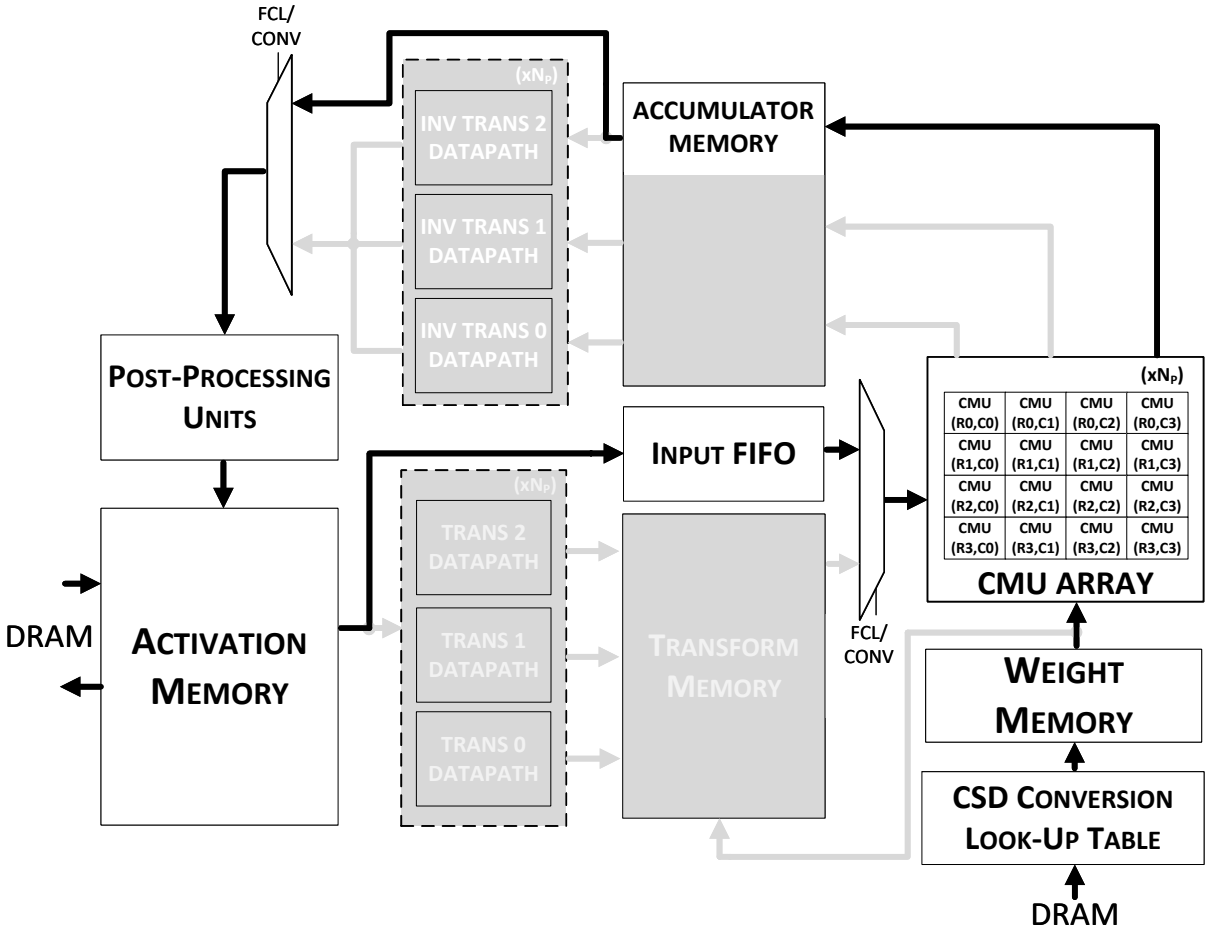


Figure 2.8: Fully-connected layer path.

matrix. Each weight's position in the compressed word corresponds to its position in the chunk. Despite being able to aggressively prune $\mathbf{W}$, the lack of structure on the location of the zero weights could lead to collisions after reshaping the column. In Figure 2.7, the existence of weight block collisions implies the need to have an additional weight word just for one non-zero weight block. Also note the possible existence of non-dense (underutilized) weight blocks. In general, these collisions can be spread across the matrix and translate to inefficient memory and MAC utilization.

To combat this inefficiency, a novel CBB structured pruning method for sparse FCLs that can be learned to minimize the overhead of zero padding while sharing the same hardware with transform-domain convolutions in HTNNs was proposed. During FCL training, the

following sparsity constraint on $\mathbf{W}$ is imposed. Given a target density d, the matrix is pruned such that the number of weight block collisions in each row of the reshaped column is the same. As a result, the overall impact of zero padding is minimized and the potential memory and MAC utilizations are maximized. To verify whether CBB structured sparsity can achieve high sparsity ratios while maintaining inference accuracy, the feasibility of this approach was tested on the FCLs of VGG-Nagadomi HTNN [36]. With this scheme, the FCL weights were trained, pruned, and quantized using 8-bit BS-CSD in PyTorch with $C = 64$, $B = 4$, and $d = 6.25\%$ for the CIFAR-10 dataset. The experimental results show that the top-1 accuracy post-training, post-pruning, and post-(BS-CSD) quantization are 92.29%, 92.74%, and 92.22%, respectively. This validates that CBB structured sparsity can be supplemented to HTNNs without compromising the accuracy. CBB structured sparsity can operate at different layer-dependent target densities d (in the range of 6.25 – 50%) that do not degrade the accuracy by controlling the number of collisions in each row of the reshaped columns of $\mathbf{W}$.

The execution of an FCL is illustrated in Figure 2.8, where inactive paths related to transform-domain convolution are greyed out. It employs an input stationary dataflow where only non-zero activation values in the input FIFO are detected and broadcast to the CMU array. In addition, outer-product multiplication is implemented for full CMU array utilization using only $1/n_t$ of the accumulator memory banks. CBB-sparse weights are represented using the proposed index-based encoding method. CBB structured sparsity parameters C and B are selected in conjunction with the SONA architecture configuration parameter N_P and the 4×4 patch-based dataflow. To share the datapath between transform-domain convolution and sparse FCL without degrading MAC array utilization, C and B are selected such that $C = 4 \times 4 \times N_P$ and $B = N_P$. Each one of the 4×4 utilized accumulator memory bank maps to one of the C/B rows in the reshaped columns of $\mathbf{W}$ shown in Figure 2.7. Each weight block maps to an address in an accumulator memory bank. The maximum number of output channels supported for an FCL is bounded by $4 \times 4 \times N_P \times O_C/n_t$ where O_C/n_t

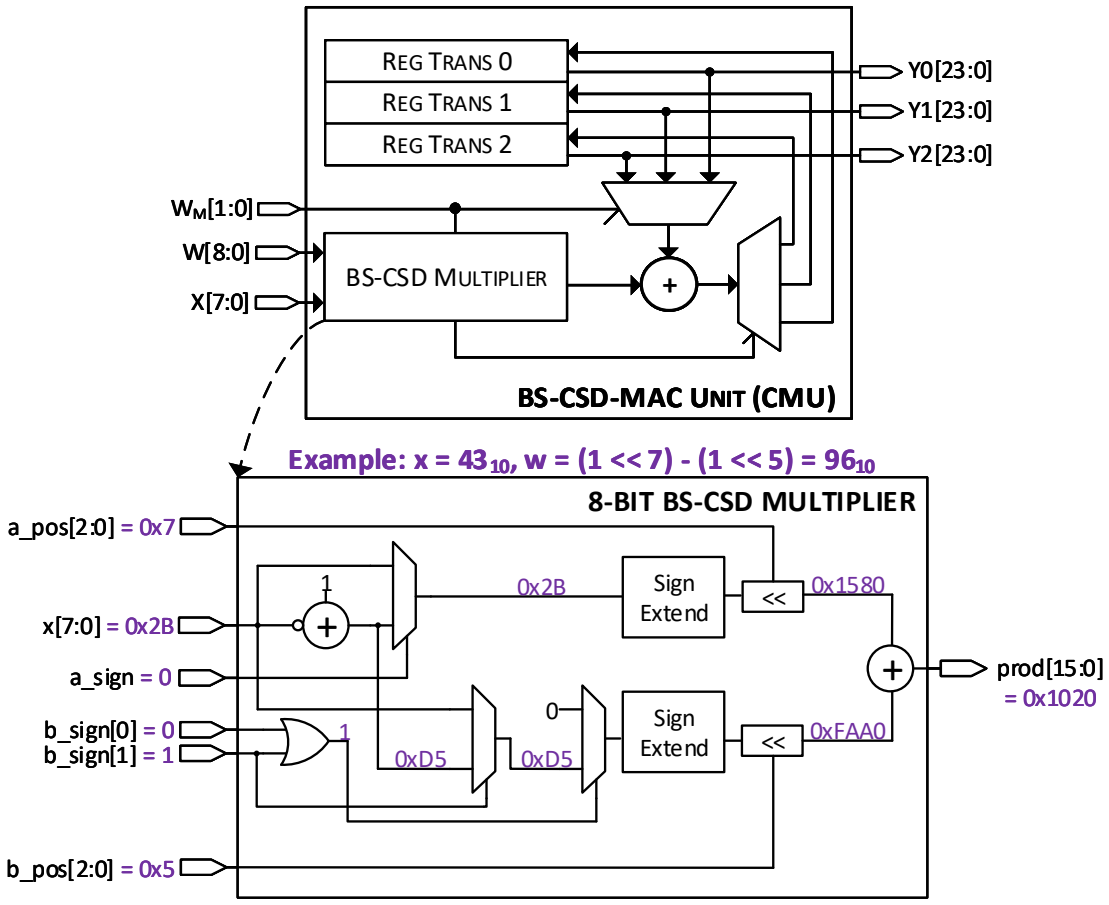


Figure 2.9: BS-CSD-MAC unit and 8-bit BS-CSD multiplier.

is the depth of an accumulator memory bank.

2.3.6 Bit-Sparse Canonical-Sign-Digit (BS-CSD) MAC Unit

Imposing structured sparsity to the number representation system, SONA employs 8-bit input MAC units for BS-CSD weights that are non-uniformly quantized with at most 2 non-zero digits. Activations are uniformly quantized with 8-bit two's complement (non-CSD) representation. To take advantage of the bit sparsity and non-uniform quantization when performing MAC operations, a hardware-friendly representation is required to represent non-zero weights. Given at most 2 non-zero CSD digits, which will be referred to as bits a and

b, the respective signs, a_{sign} and b_{sign}, and positions, a_{pos} and b_{pos}, of these bits are encoded to determine the operands of the final addition/subtraction. Without loss of generality, and assuming that $a_{pos} > b_{pos}$ ($a_{pos} \in \{0, 1, \ldots, 7\}$ and $b_{pos} \in \{0, 1, \ldots, 5\}$), the traditionally 8-bit weight values w are encoded using 9 bits in the form of $w = (a_{sign} \ll a_{pos}) + (b_{sign} \ll b_{pos})$ where $\ll$ denotes the left bit-shift.

Our proposed weight representation stems from the observation that the CSD representation of a number does not contain two adjacent non-zero digits. Thus, the relationship between a_{pos} and b_{pos} actually becomes $a_{pos} > b_{pos} + 1$. Taking advantage of the fact that there are 87 quantization levels (byproduct of having ≤ 2 non-zero digits), the footprint in off-chip memory can be reduced by storing each weight as a 7-bit code, which can be converted to a 9-bit representation using a look-up-table before storing it in SONA's weight memory. Figure 2.9 shows the circuit implementation for BS-CSD weight multiplication and illustrates an example where $x = 43$ and $w = (1, 0, -1, 0, 0, 0, 0, 0) = 96$. Using the proposed representation, multiplication between an 8-bit activation and BS-CSD weight is replaced by two bit-shifts and one 16-bit addition: $x \cdot w = ((a_{sign} \cdot x) \ll a_{pos}) + ((b_{sign} \cdot x) \ll b_{pos})$.

SONA employs $4 \times 4 \times N_P$ BS-CSD-MAC units (CMUs) shown in Figure 2.9 to perform dense (after merging n_t sparse-orthogonal weights) element-wise multiplications where N_P is the number of patch positions processed in parallel. In each cycle, an 8-bit BS-CSD multiplier and a 24-bit accumulation adder perform at most one non-zero and $n_t - 1$ implicit zero MAC operations. The inputs are the 8-bit two's complement activation x, the 9-bit BS-CSD weight w and a 2-bit weight mask w_M indicating the transform variant associated with w. Each weight is associated with a unique weight mask used to perform muxing/demuxing as the orthogonal overlapping pattern varies from channel to channel. As discussed in Section 2.3.3 the partial results are accumulated locally in registers that map to the concurrently computed n_t output channels.

Table 2.1: SONA configuration and area summary.

Component	**Size**	**Area (mm^2)**	**Power (mW)**
Activation Memory	420KB	1.262	2.3
Transform Memory	6KB	0.083	9.7
Accumulator Memory	18KB	0.192	4.9
Weight Memory	352KB	1.073	6.5
Input FIFO	1 KB	0.003	0.3
Inverse Trans Array	12 datapaths	0.022	1.6
Trans Array	12 datapaths	0.039	2.3
CMU Array	64 CMUs	0.023	8.6
Other	N/A	0.152	10.4
Total	N/A	2.85	46.2

2.4 Evaluation

2.4.1 Experimental Methodology

SONA was evaluated using $N_P = 4$ and a tile size of $I = 32$. Table 2.1 summarizes the SONA accelerator configuration. To estimate energy efficiency, performance, and area, the RTL design of SONA in Verilog (including control logic and compiled memories) was fully implemented and synthesized using Synopsys Design Compiler with TSMC 22 nm library under the typical corner, 0.8 V, and 25° C. The design was synthesized at a 500 MHz clock frequency. The area and power estimated from the synthesized result are summarized in Table 2.1. PrimePower (PTPX) is used to estimate the energy consumption of the synthesized design using the gate-level netlist and fast signal data base (FSDB) file containing switching activity.

To compare against prior dense and sparse accelerator designs whose RTL implementations are not publicly available, a simulator was utilized instead of PTPX to evaluate the relative speedup and energy efficiency of SONA for transform-domain convolution and sparse FCLs. The simulator was verified against the RTL implementation. For a dense CNN accelerator benchmark, Eyeriss [62] was selected and TimeLoop [87] was used to simulate its performance and memory access traces. For a sparse CNN accelerator benchmark, SCNN

[32] was selected and DNNSim [88] was used to simulate its performance and memory access traces. These benchmarks were chosen because they are representative dense and sparse CNN accelerator architectures whose open-source simulation models are available for accurate comparison. For a sparse FCL benchmark, EIE [72] was selected and a cycle-accurate simulator was built to simulate its performance and memory access traces. Its performance and memory access results were also verified against those reported by [72]. Arm memory compilers in TSMC 22 nm were used to estimate the energy consumption of on-chip buffers/memories. The energy cost of DRAM accesses was approximated to 100 pJ per 8 bits as in [89]. The configuration of all baseline designs was adjusted to have the same of number of multiplier instances as in SONA. The energy costs of an 8-bit conventional multiplier, an 8-bit BS-CSD multiplier, and 24/8-bit adders were obtained under the same 22 nm technology node to estimate the computation energy expenditure of SONA and baseline designs. The overhead from computing the (inverse) transforms was also included in comparisons. Finally, this work compares against [90], an optimized dense CNN accelerator whose RTL implementation was made available, using PTPX and post-synthesis FSDB dumps to get a more accurate relative energy-efficiency of SONA compared to a dense baseline.

These evaluations make use of neural network workloads provided in [36, 75, 23], ResNet-20 ($n_t = 3$) and VGG-Nagadomi ($n_t = 2$) using CIFAR-10, and C-CNN-C ($n_t = 3$) using CIFAR-100, as well as their respective equivalent CNN workloads. Furthermore, an additional benchmark was evaluated using ImageNet, ResNet-18 and its equivalent HTNN ($n_t = 3$). As the original HTNN [36] targets smaller networks for edge applications where training time is manageable, the training time is exacerbated for larger networks and datasets, rendering tasks such as ImageNet ResNet-18 unfeasible. To overcome the limitations of the training code released with [36], custom CUDA extensions were implemented. The training framework was also supplemented with CBB-structured sparsity learning for VGG-Nagadomi FCLs (Section 2.3.5) unavailable in [36]. These models were trained in PyTorch using the proposed functionally equivalent custom CUDA layers. This amounts to

rewriting parts of the code in CUDA and fusing implementations of many functions into a single function, which benefits from fewer kernel launches and reduced I/O data movement. Overall, the optimizations improve the training speed by $2-13\times$ on an NVIDIA A40 GPU compared to the code in [36]. For ResNet-18 HTNN, training is done using 4 NVIDIA A40 GPUs at a rate of ≈ 40 minutes/epoch. For reference, ResNet-18 CNN can be trained using an NVIDIA A40 GPU at the same rate. With our optimized training framework, the top-1 accuracies of {ResNet-20, VGG-Nagadomi, C-CNN-C, ResNet-18} HTNNs and baseline sparse CNNs are {91.56%, 92.80%, 70.51%, 69.8%} and {91.44%, 93.08%, 70.55%, 69.2%}, respectively. For HTNNs, weights are 8-bit BS-CSD quantized. For sparse CNNs, the deep compression strategy described in [23] was applied and weights are 8-bit uniformly quantized.

2.4.2 Simulator-Based Results

To highlight the benefits of different optimizations on speedup and energy efficiency, SONA's simulator-based results are presented incrementally in Figures 2.10-2.20 as follows: SONA* only includes the benefits of WHT-domain convolutions (Sections 2.3.2 & 2.3.3), SONA** is SONA* plus the benefits of using sparse-orthogonal kernels (Section 2.3.4), and SONA*** is SONA** with additional benefits of BS-CSD (Section 2.3.6). Based on these results, sparse-orthogonal computations contribute the most to speedup gains and energy savings, followed by WHT-domain element-wise multiplications and BS-CSD. For sparse FCLs, SONA's benefits stem entirely from optimizations for CBB structured sparsity (Section 2.3.5).

2.4.2.1 Speedup Comparison

Figures 2.10, 2.11, 2.12, and 2.13 show the speedup of SONA over Eyeriss and SCNN on the convolutional layers of ResNet-20 (CIFAR-10), VGG-Nagadomi, ConvPool-CNN-C, and ResNet-18 (ImageNet). These layers have different sizes as specified in [36, 91, 92, 93]. Note that SCNN leverages weight and activation sparsity, whereas SONA only leverages weight sparsity for transform-domain convolution. Nevertheless, SONA can achieve {4.4×, 2.1×},

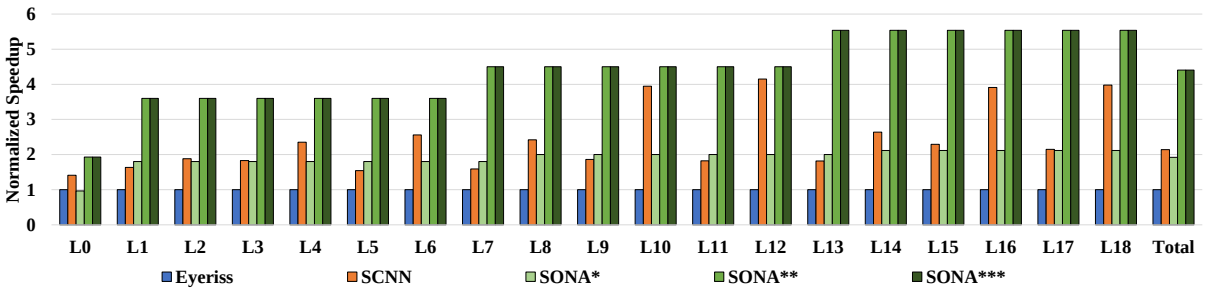


Figure 2.10: ResNet-20 per-layer speedup on CIFAR-10.

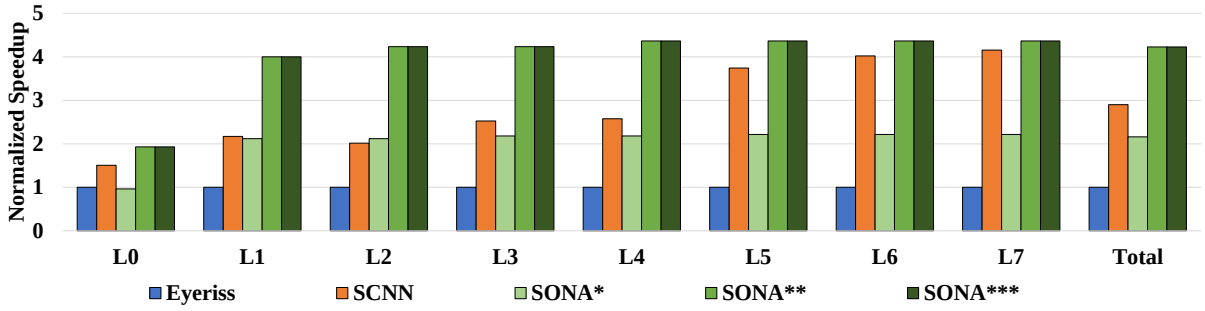


Figure 2.11: VGG-Nagadomi per-layer speedup on CIFAR-10.

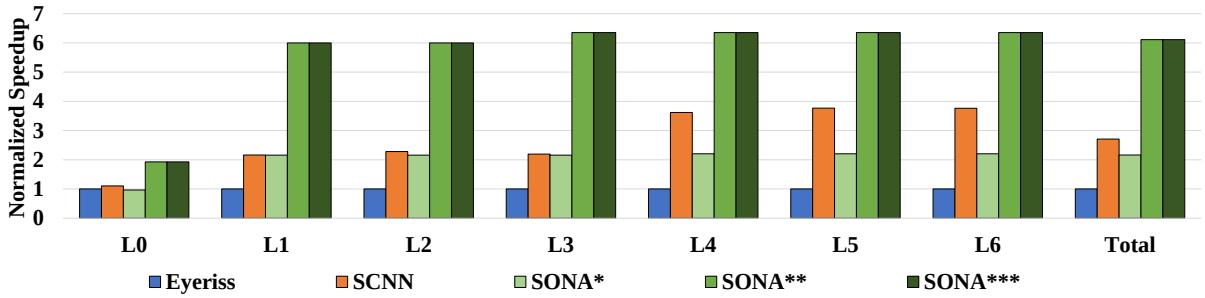


Figure 2.12: ConvPool-CNN-C per-layer speedup on CIFAR-100.

{4.2×, 1.5×}, {6.1×, 2.7×}, {6.1×, 2.1×} overall speedup compared to {Eyeriss, SCNN} across all layers of ResNet-20 (CIFAR-10), VGG-N, C-CNN-C, ResNet-18 (ImageNet), respectively. BS-CSD does not contribute to speedup gains as both conventional and BS-CSD multiplication can be done in a single cycle. Also, note that the relative speedup of SONA over SCNN diminishes when the activation density of the sparse CNN model is unusually low ($\approx 20\%$). However, SONA maintains higher speedup because it does not incur frequent

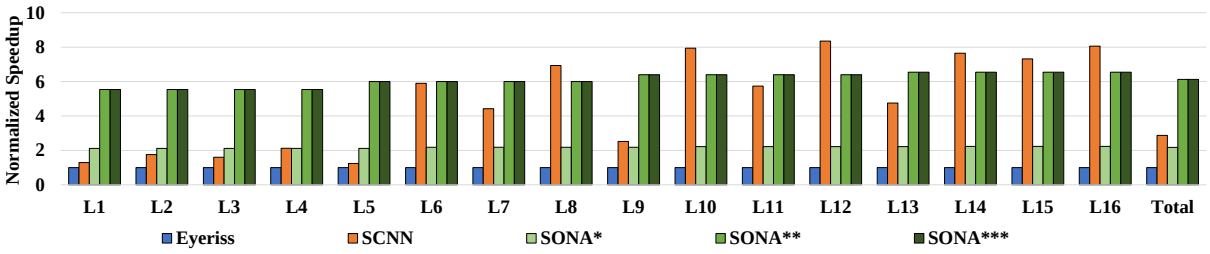


Figure 2.13: ResNet-18 per-layer speedup on ImageNet.

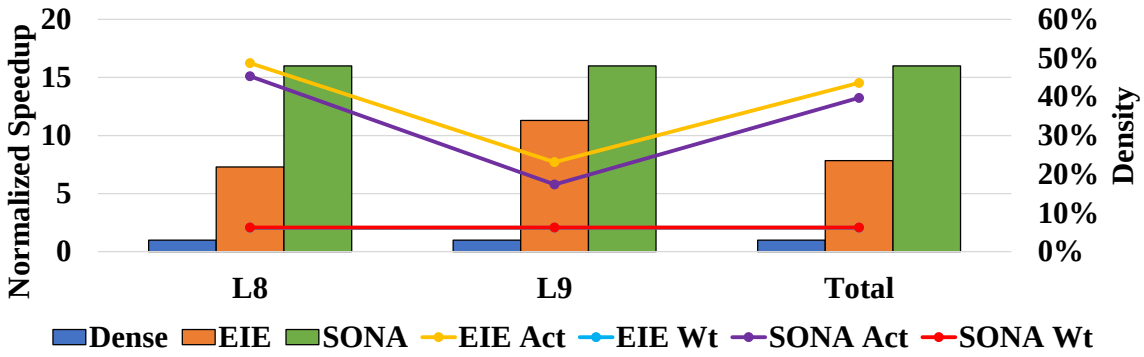


Figure 2.14: VGG-Nagadomi FCL speedup on CIFAR-10.

execution stalls from intra-PE fragmentation and from the way SCNN's specialized dataflow (PT-IS-CP-sparse) partitions the work across the PE arrays. SONA can concurrently compute n_t output channels with minimal overhead while also benefiting from the multiplicative gains of replacing convolution by element-wise operations in the heterogeneous WHT domains.

Figure 2.14 shows the speedup of SONA over EIE on the FCLs of VGG-Nagadomi. The speedup over a naive dense architecture that leverages neither weight nor activation sparsity is also included. Layer sizes can be found in [36, 92]. The speedup dependency on the sparsity of different layers is highlighted. In the case of FCLs, activation density observed from the HTNN model is illustrated as SONA leverages both activation sparsity and weight sparsity for FCLs using CBB-structured pruning. SONA can achieve approximately 16× and 2× overall speedup compared to the naive dense architecture and EIE, respectively.

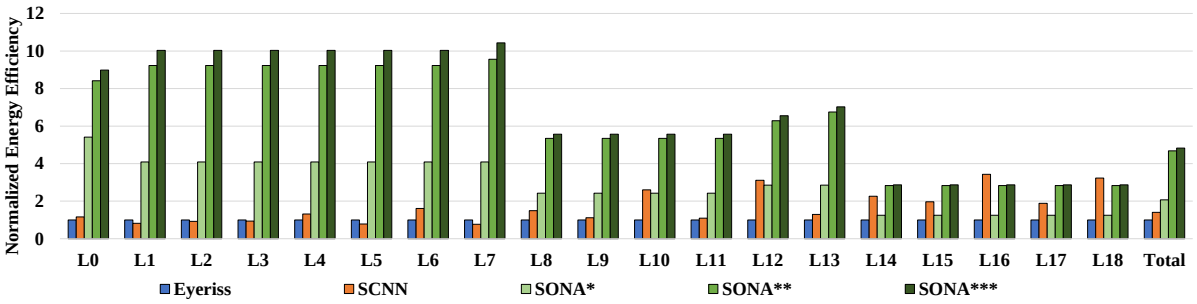


Figure 2.15: ResNet-20 per-layer energy efficiency on CIFAR-10.

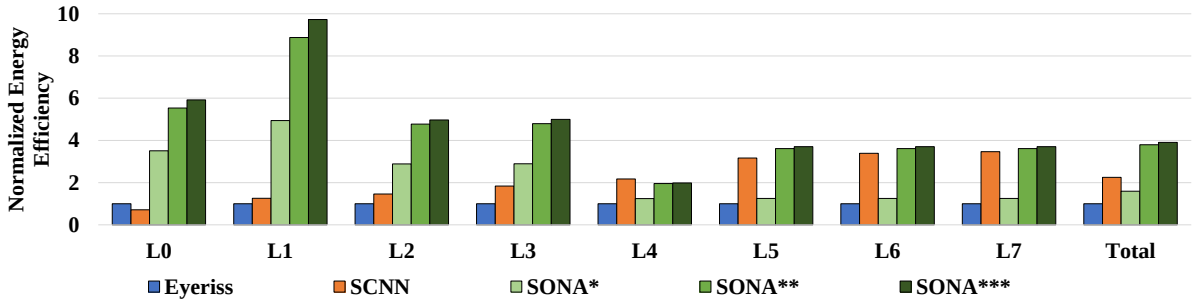


Figure 2.16: VGG-Nagadomi per-layer energy efficiency on CIFAR-10.

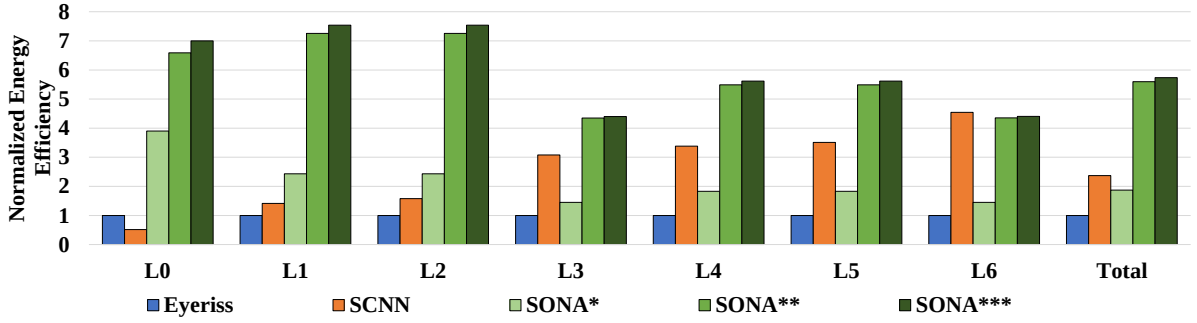


Figure 2.17: ConvPool-CNN-C per-layer energy efficiency on CIFAR-100.

Despite the existence of an activation queue, EIE suffers from load balancing issues because it uses compressed sparse column weight representation. As discussed in Section 2.3.5, CBB structured sparsity maps to SONA while maintaining full MAC array utilization.

2.4.2.2 Energy-Efficiency Comparison:

Figures 2.15, 2.16, 2.17, and 2.18 show the relative energy-efficiency of SONA over Eyeriss and SCNN on the convolutional layers of ResNet-20 (CIFAR-10), VGG-N, and C-CNN-C, ResNet-18 (ImageNet), respectively. The figures show that SONA can achieve {4.8×, 3.4×}, {3.9×, 1.7×}, {5.7×, 2.4×}, {5.9×, 2.2×} overall energy-efficiency gain compared to {Eyeriss, SCNN} across layers of ResNet-20 (CIFAR-10), VGG-N, and C-CNN-C, ResNet-18 (ImageNet), respectively. Note that SONA maintains higher energy efficiency despite lower weight density (in transform-domain) compared to SCNN for VGG-Nagadomi. Also note that SONA's energy efficiency compared to SCNN diminishes when the activation density of the sparse CNN model is unusually low ($\approx 20\%$) because SONA does not exploit activation sparsity. SCNN's choice of outer product for convolution leads to an input stationary dataflow that incurs an overhead cost of accumulation buffer structures with expensive read-modify-write accesses. This overhead in turn hurts SCNN's energy efficiency and detracts from the multiplicative gains one would expect by leveraging weight and activation sparsity. This disadvantage is not as noticeable for later layers in the examined networks where the activation density is very low ($\approx 20\%$) and the input feature map dimensions are small. However, Figures 2.15 and 2.16 show that for layers with larger activation densities (50-70%) and larger input feature map dimensions, SONA stands out because it is able to overlap the weight kernels and process sparse-orthogonal WHT-domain convolution using the OS dataflow.

Figure 2.19 illustrates the energy breakdown of SONA for ResNet-20 excluding DRAM access energy. Note that the overhead of the (inverse) transform arrays is only 3% of SONA's energy expenditure. The input transform memory access accounts for 15% of the energy, which, in the absence of MASR transform memory organization (Section 2.3.4), would increase by at least 1.7×. The BS-CSD MAC array accounts for 24% of the total energy which, in the absence of BS-CSD, would increase by 1.6×.

Figure 2.20 shows the relative energy efficiency of SONA over EIE and the naive dense

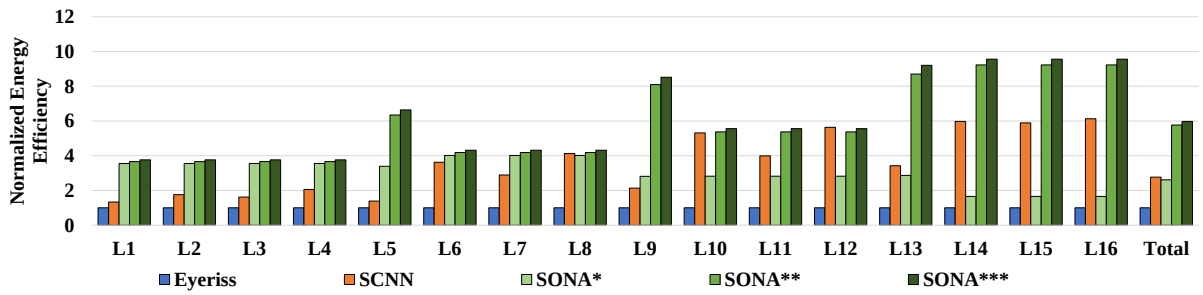


Figure 2.18: ResNet-18 per-layer energy efficiency on ImageNet.

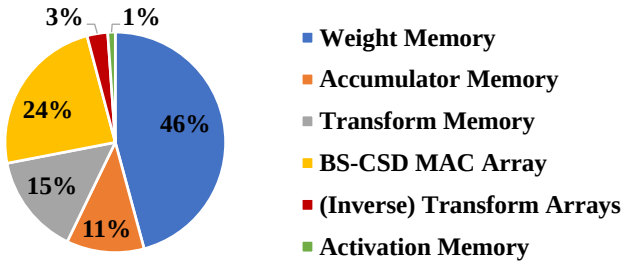


Figure 2.19: SONA energy breakdown on ResNet-20.

architecture on the FCLs of VGG-Nagadomi. SONA can improve the energy efficiency of the naive architecture and EIE by 14.9× and 2.4×, respectively, across FCLs of VGG-Nagadomi. Although both approaches significantly improve the energy efficiency relative to a naive architecture, SONA does not incur unnecessary memory storage and access as it capitalizes on CBB sparsity to maximize memory utilization and limit the zero padding overhead.

2.4.3 Comparisons with Sparsity-Aware Accelerators

Table 2.2 compares SONA with recent sparsity-aware accelerators. As open-source simulators are not available for those designs (except [32, 62]) to perform direct comparisons, indirect comparisons are provided via the energy and speedup gains claimed in [37, 94, 33, 73, 34] for common/similar benchmarks. For example, 8-bit Laconic [94] reports 1.4× speedup and

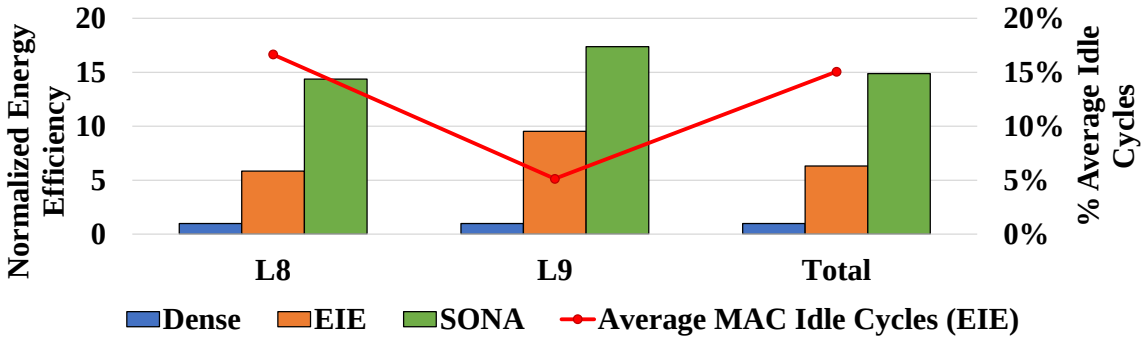


Figure 2.20: VGG-Nagadomi FCL energy efficiency on CIFAR-10.

0.9× energy efficiency compared to 8-bit SCNN, overall. It can be extrapolated that SONA is 1.5× more energy efficient and achieves a 2.7× speedup over [94], overall. The methodology described in this example is identical for [34, 73, 33, 37]. Table 2.2 shows that SONA exhibits higher energy efficiency and speedup in general. Section 2.5 discusses the inefficiencies in those prior designs and relative advantages of SONA.

Table 2.2: Accelerator benchmark comparison summary.

Accelerator A	**Overall Energy Efficiency of SONA over A**	**Overall Speedup of SONA over A**
Eyeriss [62]	5.1	5.2
SCNN [32]	2.4	2.1
SparTen [34]	1.7	1.3
Laconic [94]	2.7	1.5
Cambricon-X [73]	1.7	2.1
Cambricon-S [33]	1.0	1.2
S2TA [37]	0.8	2.5

2.4.4 Post-Synthesis Design Comparison

As mentioned, the RTL design of SONA was fully implemented in Verilog including all the necessary control logic and compiled memories. To better capture SONA's minimal overhead and to better situate its relative energy efficiency if it were to be implemented on a chip, the

SONA implementation using the same configuration in Table 2.1 was compared against the synthesized CNN accelerator design in [90] using the same technology node (TSMC 22 nm) and operating conditions (0.8 V). The authors in [90] have made the RTL implementation of their dense CNN accelerator available. Both designs were evaluated at their minimum slack design point which is 250 MHz for [90] and 500 MHz for SONA. Since the design in [90] has an 512 MAC array that is 8× more than the number of MACs in SONA (Table 2.1), [90] was also evaluated at 62.5 MHz so that both designs have the same number of MAC operations per unit time. Off-chip DRAM access energy was excluded because both designs store all weights and activations in on-chip memory for ResNet-20. If DRAM was included for bigger networks, SONA would benefit from fewer accesses by employing fewer sparse-orthogonal overlapped kernels.

Systolic array processing of convolution in [90] enables aggressive data reuse among PEs and significantly reduces the memory access energy. As a result, the CNN accelerator in [90] reports an outstanding energy efficiency (1.5 TOPS/W) and ultra-low power consumption (170 μW) at 0.58 V and 153 kHz in TSMC 40nm. Compared to this highly optimized CNN accelerator [90], SONA is $2.9 - 3.9\times$ more energy-efficient for ResNet-20 when both designs are synthesized and evaluated in the same TSMC 22 nm technology.

Table 2.3: Post-synthesis design comparison summary.

Design synthesized (22 nm, 0.8 V)	**SONA**	**[90]**	
# MACs	64	512	
Clock Frequency (MHz)	500	250	62.5
Energy Efficiency(Inference/J, ResNet-20)	60247	21031	15610

2.5 Related Works

Dense CNN accelerators have been explored [95, 96, 97, 59, 98, 99] to optimize the performance and energy-efficiency of inference tasks. Sparse-Winograd CNNs [74, 75] have been studied prior to HTNNs to reduce the computational complexity of convolution. Both ap-

proaches replace convolutions with element-wise operations in the transform-domain but differ in their choice of transform. HTNNs require multiple heterogeneous transforms and select WHT and its permutations for sparse orthogonal computations. Sparse-Winograd CNNs use a single Winograd transform and rely on unstructured sparsity. This work's training of ResNet-18 using ImageNet shows that HTNN can outperform [74] in terms of top-1 accuracy (69.8% for HTNN vs. 66.8% in [74]). It is also shown in [36] that the complexity of HTNN is 1.93 – 5.12× lower than that of sparse-Winograd models.

HTNN can replace any 3×3 convolution in general networks. For depthwise separable convolutions, however, sparse-orthogonality between kernels is foregone (i.e., $n_t = 1$) and HTNN's improvement will solely stem from element-wise multiplications in the WHT domain. For example, if applied to 3×3 convolutions in MobileNetV1/2 [79, 80], HTNN can theoretically save approximately 48%/56% MAC operations for those layers, respectively. However, MobileNetV1/V2 are not able to fully take advantage of the optimizations proposed by HTNN as they rely on 1×1 point-wise convolution as the primary workload. Although SONA supports 1×1 point-wise convolution, it is identical to that of MobileNetV1/2.

Pruning methods have also been developed [23, 25, 24, 31] to mitigate the growing model sizes and computational complexity of CNNs. To take advantage of both weight and/or activation sparsity, several sparsity-aware accelerators have been proposed in the literature [100, 34, 72, 96, 32, 73, 33, 37, 51, 52, 101, 94, 102]. SCNN [32] and EIE [72] were discussed extensively throughout this work. Both approaches explore unstructured sparsity where the location of the non-zero values are completely random and as a result suffer from load imbalance.

SparTen [34] and ExTensor [96] propose accelerator architectures that only perform the necessary multiplications at the cost of a high-overhead intersection operation or content-addressable memories. These high-overhead approaches to performing intersection dominate the computational energy expenditure of their respective accelerators while demonstrating a potential to cause frequent stalls, which hurt utilization. Cambricon-X [73] tackles load

imbalance using an indexing module requiring wide and costly multiplexers that quickly become a source of inefficiency. Cambricon-S [33] tries to reduce the multiplexer size in [73] by grouping weights together in a predefined block size therefore reducing the energy overhead and making it comparable to SONA as shown in Table 2.2. However, weight block locations are irregular and that continues to be a source of slight performance overhead. Although SONA only targets weight sparsity, it mitigates load balancing via structured sparse-orthogonal weights that enable parallel execution and hardware sharing with almost no overhead.

Other works exploit the inherent bit-level sparsity in weights and activations to enhance the performance and energy efficiency of CNN accelerators. Most recently, Laconic [94] focuses on eliminating the ineffectual multiplications stemming from intermediate zero bits. The optimizations proposed are applied to varying bit-widths and exhibit significant gains for higher bit precisions (e.g., 16 bits). Meanwhile, SONA targets HTNNs where weights are explicitly trained to have 8-bit BS-CSD weights (at most 2 non-zero ternary digits), therefore trading off configurability for increased opportunity of more specific optimizations in addition to the use of heterogeneous WHT-domain sparse orthogonal kernels.

Other approaches target structured model compression. PermDNN [103] reduces the indexing overhead of unstructured sparse methods by transforming sparse filters into a permuted diagonal matrix. However, it only targets FCL in DNNs, whereas SONA shares convolution and FCL datapaths without compromising MAC array utilization. CirCNN [104] leverages block-circulant matrices and Fast Fourier Transform (FFT) based convolution. Although [104] supports convolution and FCL, it does not leverage sparsity and requires complex FFT hardware, whereas SONA employs patch-based WHT hardware requiring additions/subtractions to enable sparse-orthogonal convolution in heterogeneous domains. S2TA [37] leverages dynamically bounded weight and activation structured sparsity using a time-unrolled systolic array to serialize MAC processing. Leveraging structured activation sparsity provides significant gains in energy at the cost of reduced speedup due to the time-unrolled

nature as shown in Table 2.2.

2.6 Conclusion

In this work, a novel accelerator architecture was introduced for transform-domain neural networks with structured sparse-orthogonal weights where convolution operations are replaced by element-wise multiplications. SONA employs an OS dataflow coupled with an energy-efficient memory organization to reduce the overhead of sparse-orthogonal WHT-domain kernels that are concurrently processed. Weights in SONA are non-uniformly quantized with BS-CSD representations to reduce multiplications to simpler additions. Moreover, SONA leverages CBB structured pruning patterns to efficiently map sparse FCLs onto the proposed accelerator while maintaining full MAC array utilization. Compared to prior dense and sparse neural networks accelerators, SONA can reduce inference energy by 5.1× and 2.4× and increase performance by 5.2× and 2.1×, respectively, for convolution layers. For sparse FCLs, SONA can reduce inference energy by 2.4× and increase performance by 2.1× compared to a prior sparse FCL accelerator.

CHAPTER 3

A 22 nm 9.51 TOPS/W Neural Engine with 2 MB MRAM Leveraging Sparse-Orthogonal Walsh-Hadamard Transform Computations and Dynamic Power Gating

3.1 Introduction

Neural network processing has become increasingly integrated into Internet-of-Things (IoT) devices, where stringent memory constraints require all model weights to be stored on-chip. This challenge has driven the exploration of alternative neural network computational techniques to improve efficiency without sacrificing accuracy. One promising approach discussed in Chapter 2 is HTNNs [36], which leverage the WHT to replace traditional convolutions with element-wise multiplications. As shown in Figure 3.1, HTNNs significantly reduce computational complexity, achieving up to a 4.9 – 6.8× improvement [36], all while maintaining model accuracy.

This method also employs structured sparsity during training [36, 105], where weights are partitioned across three permuted WHT variants. These variants are trained such that their non-zero entries remain non-overlapping, allowing them to be merged into a single dense kernel. This structured approach ensures that, even with only 33% weight density,

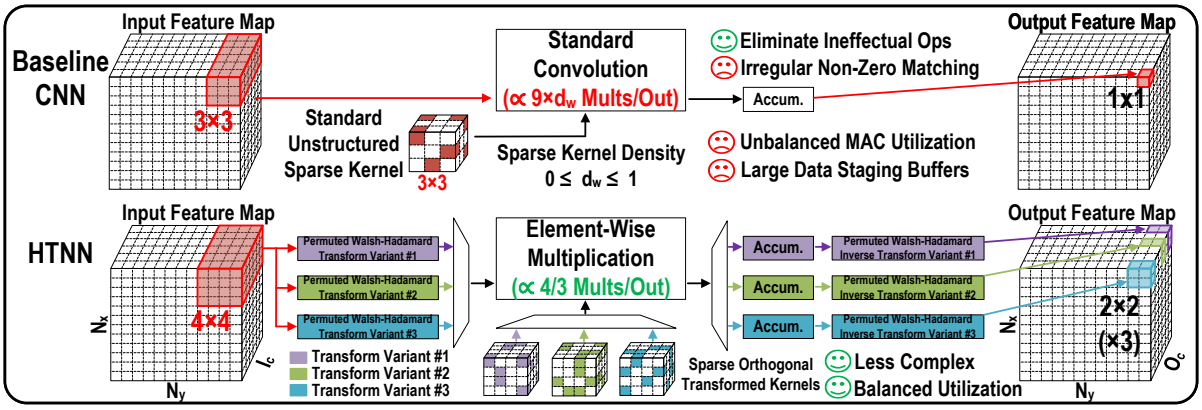


Figure 3.1: Motivation for the proposed accelerated HTNN processor.

the hardware can achieve full utilization, leading to a potential 3× efficiency gain. Although the work in Chapter 2 proposes an accelerator architecture based on these principles [105], its feasibility in silicon has not yet been demonstrated. The transition from modeling to a physical implementation presents several challenges, particularly when considering the need for all-on-chip weight storage. On-chip memory constraints, increased memory access latency, and maintaining computational efficiency while leveraging structured sparsity pose significant design hurdles that have yet to be fully addressed.

This work presents the first silicon implementation of a structured-sparse HTNN processor capable of accelerating both convolutional and fully connected layers (FCLs). To enable efficient on-chip execution, the design leverages a dense, retentive MRAM for weight storage, eliminating the need for external memory access and reducing data transfer overhead. However, this choice introduces several critical implementation challenges, as described in Figures 3.2 and 3.3, that must be addressed to achieve practical energy efficiency and performance.

First, while MRAM offers high density and long-term data retention, it suffers from substantial active and leakage energy consumption as shown in Figure 3.2, particularly during low-voltage operation. This energy overhead can severely limit efficiency if not properly managed. To mitigate this issue, the proposed processor incorporates a layer-wise dynamic power gating mechanism coupled with a weight caching strategy that minimizes MRAM access time

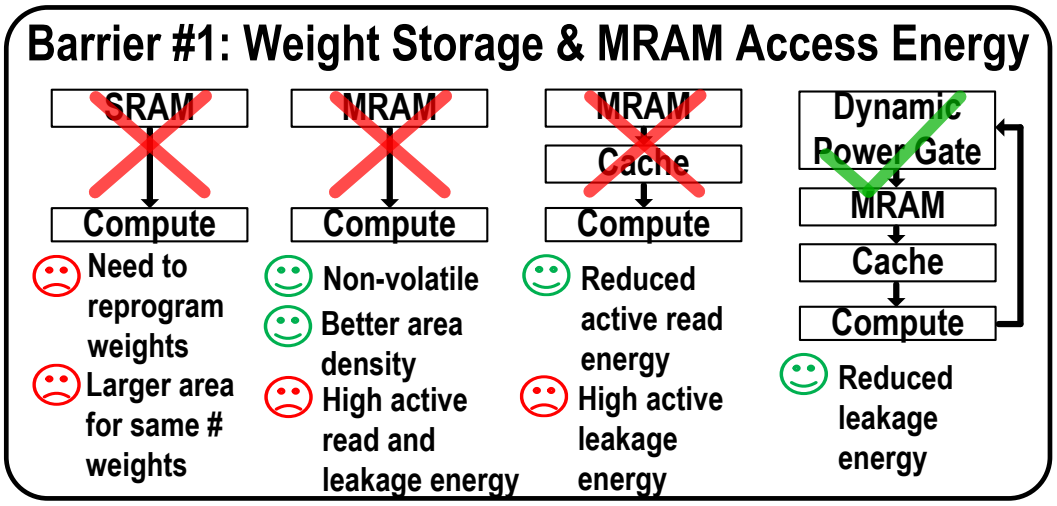


Figure 3.2: MRAM active and leakage energy challenges for accelerated HTNN processor.

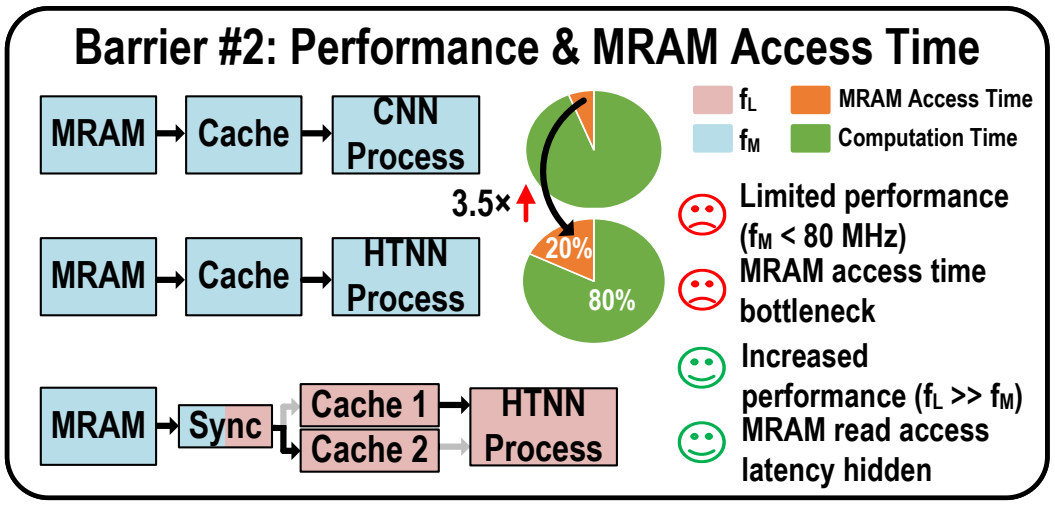


Figure 3.3: MRAM performance and access time challenges for accelerated HTNN processor.

and reduces leakage power. Second, despite the reduced computational complexity of HTNN operations, the overall performance is constrained by memory access latency. Due to the slow read speed of MRAM, the time spent on memory access increases by 3.5× compared to traditional neural network processing as shown in Figure 3.3, accounting for up to 43% of total execution time. To overcome this bottleneck, the processor employs a double-buffered

(ping-pong) weight cache with pipelined prefetching and decoupled MRAM/compute clocks to fully hide the MRAM read time.

The proposed HTNN processor was fabricated in a 22 nm CMOS process, demonstrating substantial improvements in both energy efficiency and performance. The combination of power-saving techniques leads to a 40% reduction in energy consumption, while achieving a best-in-class peak efficiency of 9.51 TOPS/W for convolutional layers and 22.86 TOPS/W for fully connected layers, including the overhead of activation and weight memory access energy. In addition to energy savings, the processor's architectural optimizations yield a 4.9× increase in throughput, reaching a peak performance of 195 GOPS.

3.2 Neural Engine Architecture

The proposed reconfigurable engine shown in Figure 3.4 supports a complete set of custom instructions loaded into a 16 kB SRAM such as WHT-domain convolution, max/average pooling, and sparse FCLs. A complete listing of the supported instruction set architecture is provided in Figure 3.5.

The compute core consists of two 4×3 arrays of reconfigurable 2D 4-point (inverse) WHT pipelined datapaths and a $4 \times 4 \times 4$ array of 8-bit bit-sparse canonical-sign-digit (BS-CSD) [36] MAC units (CMUs). Each (inverse) transform datapath is implemented with butterfly and transpose/permute crossbar structures. The CMUs are constructed with proposed BS-CSD multipliers as shown at the top left of Figure 3.6, and compute multiple kernels concurrently. Double-buffered local memories support overlap of data movement and computation for increased utilization. A transform cache is included to increase transformed patch reuse and limit WHT compute overhead. The architecture integrates a 2 MB MRAM to store all weights on-chip and a 375 kB multi-bank SRAM memory to store activations. The MRAM supports dynamic power gating and feeds a multi-clock domain double-buffered weight cache.

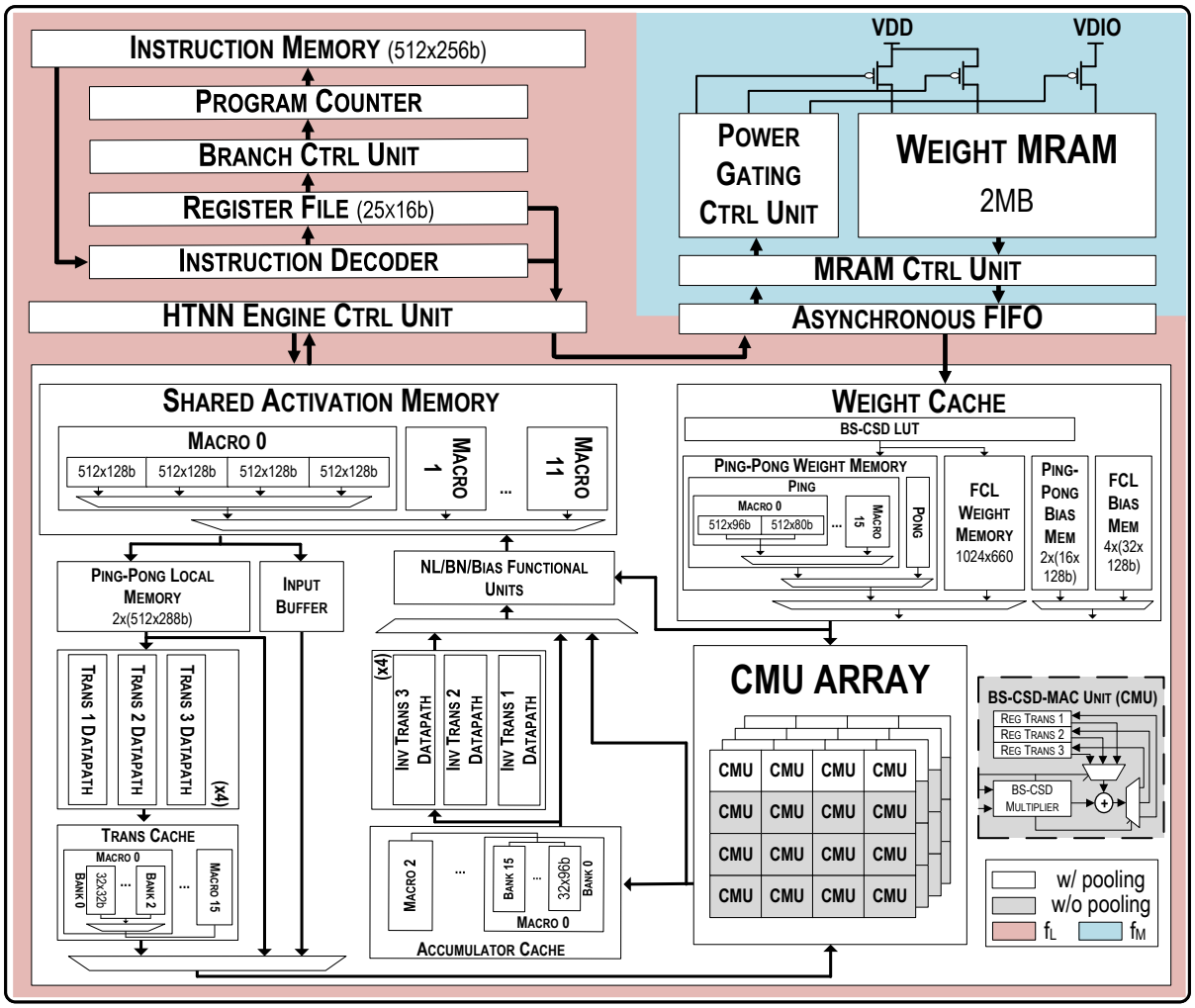


Figure 3.4: Overall architecture of the proposed HTNN processor.

3.3 Neural Engine Dataflow

3.3.1 WHT-Convolution Layer

Figure 3.6 shows the dataflow for WHT-domain convolution. The architecture processes up to four 4×4 patch positions in parallel using ≤ 3 WHT variants. BS-CSD weights are 8-bit quantized but stored in on-chip MRAM using 7 bits per weight since there are only 87 (non-uniformly spaced) possible values for the 8-bit BS-CSD representation. When written to a 437 kB weight cache, they are on-the-fly converted to a computation-friendly 9-bit format

OPERATION	OPCODE	Description
PE_WEIGHT	1	Load weights from MRAM to weight memory
PE_DOWNSAMPLE	2	Spatial downsampling
PE_MOV	3	Move data from shared to local activation memory
PE_TRANS_CONV	4	Transform domain convolution
PE_1x1_CONV	5	1x1 convolution
PE_FCL	6	Fully-connected layer
PE_POOL	7	Max or average pooling
PE_ADD	8	Element-wise addition
PE_RELU	9	Non-linear activation
NCX_{}	31	Sup-ops in NE controller processor

ARITHMETIC			
INSTR	OPCODE	INSTR	OPCODE
ADD	00001	ADDI	00010
SUB	00011	SUBI	00100
MULT	10001	MULTI	10010
MULTS	10011	LSR	11000
AND	10100	LSL	11001
OR	10101	LDS	00111
XOR	10110		
NAND	10111		
BRANCH		**SPECIAL**	
INSTR	OPCODE	INSTR	OPCODE
BEQ	01001	HALT	11111
BNE	01010	NOOP	00000
BLT	01011		
BGT	01100		
BLE	01101		
BGE	01110		

Figure 3.5: Summary of HTNN processor instruction set architecture.

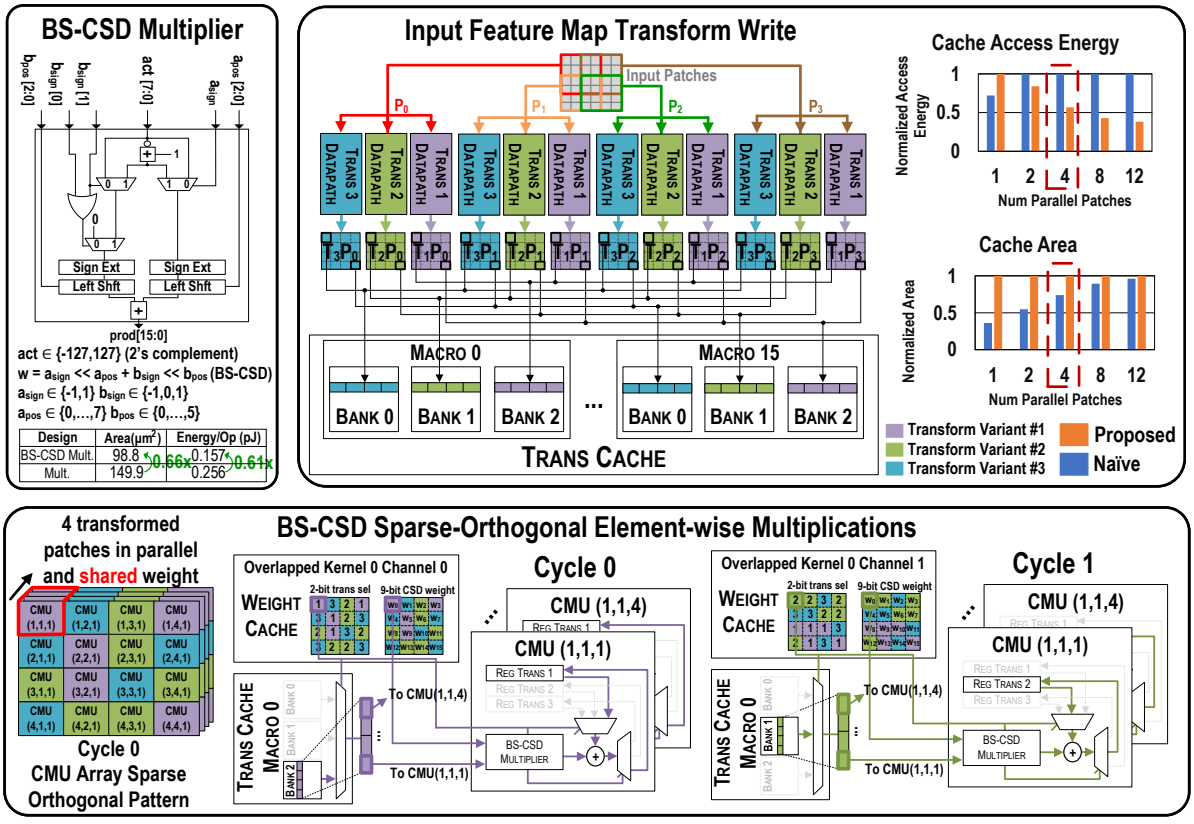


Figure 3.6: Heterogeneous WHT-domain sparse-orthogonal convolution dataflow.

(Figure3.6 top left) in which each weight is encoded with the sign and position of up to 2 non-zero CSD digits. The BS-CSD multiplier uses this unique format to efficiently generate partial products, achieving 34% energy reduction over a conventional multiplier. Structured sparsity ensures that non-zero weight positions of ≤ 3 WHT kernels do not overlap. Hence, each non-zero weight is associated with a 2-bit mask to allow concurrent processing of ≤ 3 overlapped kernels without any resource conflicts. During convolution, input patches in local memory are transformed and stored in transform cache for reuse across overlapped kernels. As overlap patterns vary across kernels and channels, a naive cache storing entire patches in a single word would have to read all activations prior to multiplexing them with the structured sparsity masking pattern. To enable energy-efficient patch reuse while exploiting patch parallelism, the proposed processor uses a cache to distribute 4 patches across multiple banks such that the needed 4×4 row/column location and WHT variant pairs can always be read by accessing 1/3 of the total 48 banks. Compared to a traditional cache structure, this 6 kB cache reduces access energy by 1.8× for a 30% cache area overhead and negligible total chip size increase as shown at the top of Figure 3.6. Transformed patches are then element-wise multiplied by their kernels (instead of performing convolution because transformed patches are in WHT domain) and accumulated locally at the CMU as shown at the bottom of Figure 3.6. Accumulated patches then undergo inverse WHT and post-processing before being written to shared memory.

3.3.2 Fully-Connected Layer

The dataflow for sparse FCL in the proposed HTNN processor uses CBB structured sparsity, employing an outer-product approach as shown at the top of Figure 3.7. Structured sparsity-aware NN training reshapes each column in the FCL weight matrix and prunes the reshaped column to have the same number of weight blocks in each of its rows, minimizing zero-pad memory overhead. To share the datapath with WHT convolution and to fully utilize the $4 \times 4 \times 4$ CMU array, the proposed processor uses a weight block size of 4 and reshapes

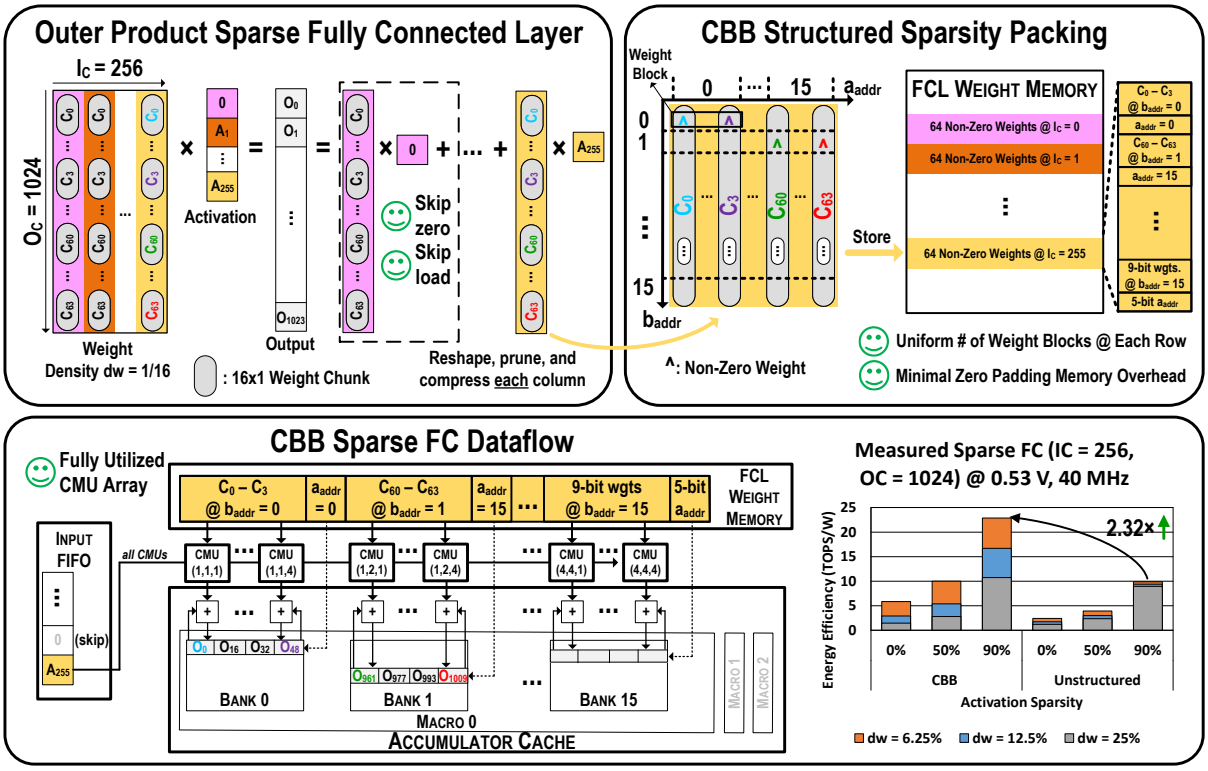


Figure 3.7: Column-based block structured sparsity fully-connected layer dataflow.

each column to have 16 rows. Only non-zero BS-CSD weight blocks are stored in MRAM and loaded into FCL weight memory. Each weight block is mapped to an accumulator cache bank. Given that non-contiguous outputs are being concurrently computed, a 5-bit bank address is stored with each weight block. The CBB structured sparse FCL outperforms unstructured sparse FCL by up to 2.32× with 90% activation and 93.75% weight sparsity as shown at the bottom of Figure 3.7.

3.4 MRAM-Weight Cache Subsystem

Although MRAM offers high density and non-volatility, its readout energy and speed present significant challenges, with an operating frequency limited to $f_M \leq 80$MHz. The proposed processor mitigates these limitations through a multi-clock domain, double-buffered MRAM-weight cache architecture as shown in Figure 3.8.

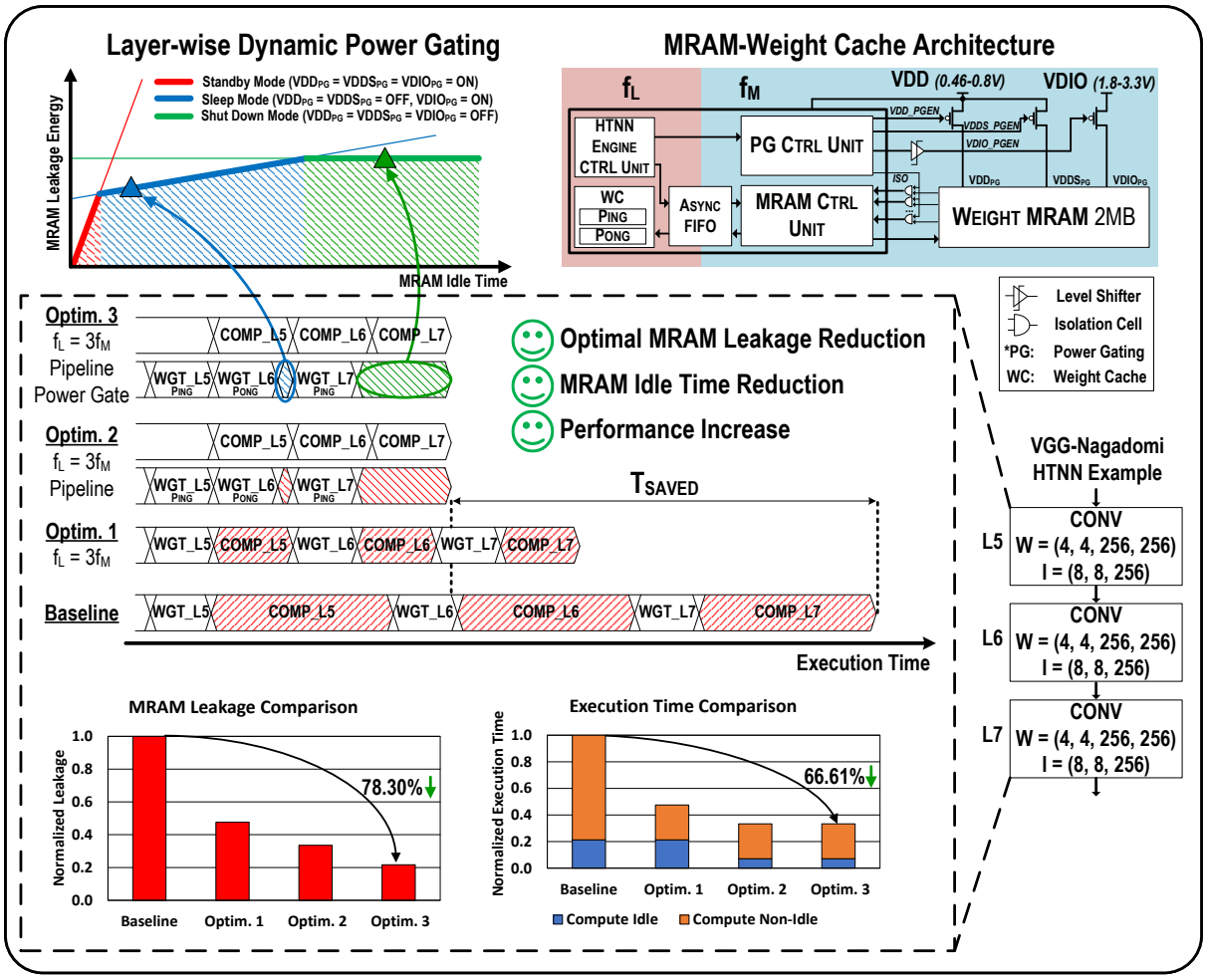
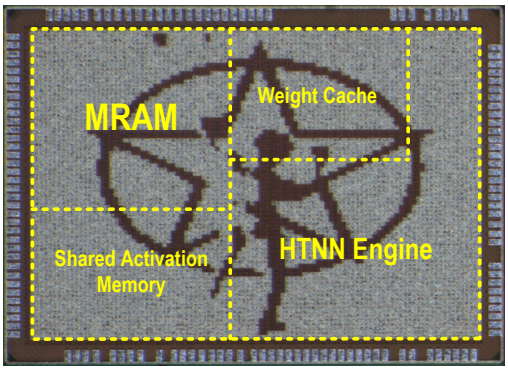


Figure 3.8: MRAM-weight cache multi-clock dynamic power gating architecture.

Specifically, the MRAM macro and controller operate at $f_M \leq 80$MHz, while the rest of the chip runs at a higher frequency of $f_L < 250$MHz. An asynchronous first-in first-out (FIFO) buffer manages clock-domain crossing between the MRAM controller and the SRAM-based weight cache. While multi-clocking reduces compute time, the latency of MRAM weight loading into the cache emerges as a major bottleneck, accounting for approximately 43% of execution time. To alleviate this issue, the proposed processor employs a double-buffering strategy, allowing MRAM weight loading and computation to be pipelined, thereby enhancing both utilization and overall performance. Additionally, to minimize the impact of MRAM active leakage, the subsystem incorporates a layer-wise dynamic power-gating

	Specification
Process	22 nm
Die Area	3.42mm × 2.56mm (8.76 mm^2)
Voltage	0.46 V - 0.8 V (VDD) 1.8 V - 3.3 V (VDIO)
Max Frequency	243 MHz
Power	2.03 mW - 45.67 mW
On-Chip Memory	877.5 kB SRAM + 2 MB MRAM
Performance	19.2 GOPS - 195 GOPS
Energy Efficiency	4.25 TOPS/W - 9.51 TOPS/W



Metal fill artwork based on 2112 album cover by Rush

Figure 3.9: Chip summary and die photo.

mechanism. The MRAM macro is divided into three power domains (VDD, VDDS, and VDIO), all of which are power-gated using PMOS headers controlled dynamically by the neural engine control unit. When the MRAM macro is idle, it is transitioned into either sleep mode (peripheral on, array off) or full shut-down mode. The optimal power mode is selected on a per-layer basis, depending on the expected MRAM idle time.

To evaluate the impact of these optimizations on MRAM leakage energy and execution time, an analysis based on VGG-Nagadomi HTNN layers five through seven is presented in Figure 3.8. Without optimizations, the compute core remains idle while waiting for the layer's weights to be loaded into the cache. Only after this loading process is completed can the computation begin. By first decoupling the MRAM and compute clock domains, the compute core's frequency is increased, effectively reducing both execution time and MRAM on-time (thereby minimizing active leakage). However, this improvement introduces a new challenge of compute core underutilization, addressed using the double-buffered weight cache architecture. During execution, after layer five's weights are loaded, computation begins immediately by reading from the weight cache's ping buffer. Simultaneously, layer six's weights are cached into the pong buffer, enabling continuous operation without stalls. This pipelining mechanism not only improves performance and compute core utilization but also significantly reduces MRAM on-time.

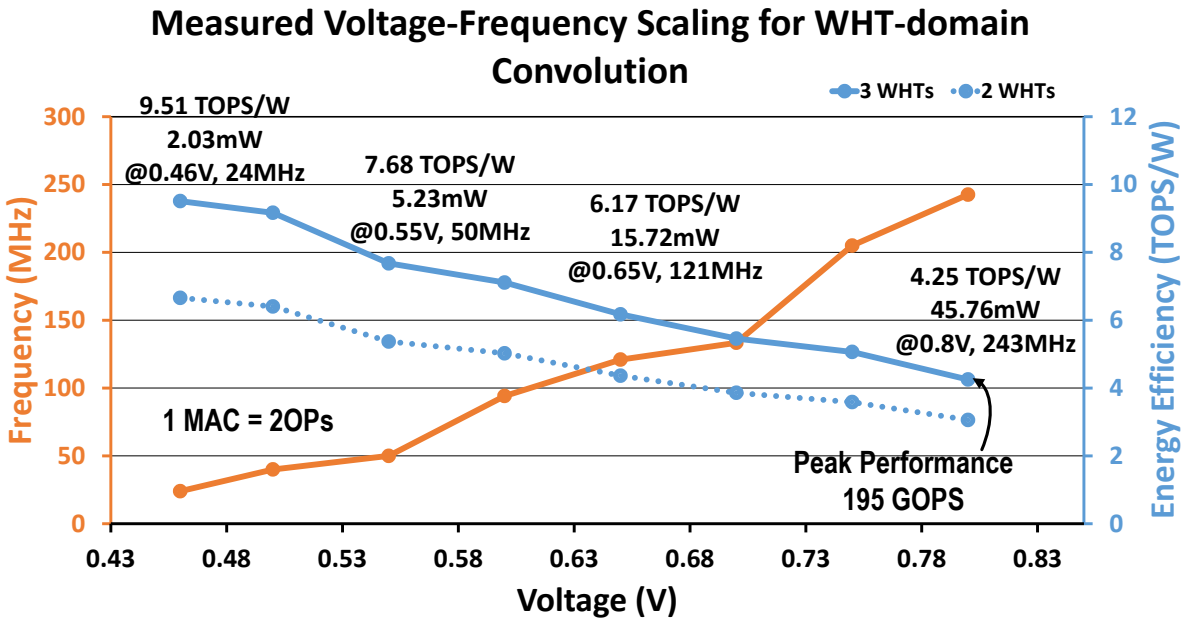


Figure 3.10: Measured peak efficiency and voltage-frequency efficiency scaling of WHT-convolution using 2 and 3 WHTs.

At low-voltage operation, MRAM leakage in standby mode can negatively impact energy efficiency. To mitigate this, the MRAM macro transitions from standby mode to either sleep or shut-down modes during idle periods. The optimal transition mode is determined based on the MRAM idle time. For instance, between weight loads for layers six and seven in Figure 3.8, the idle period is too short to justify full shut-down, so the system instead enters sleep mode to minimize leakage while maintaining responsiveness. By leveraging the proposed MRAM-weight cache subsystem, MRAM macro leakage is reduced by 78.3%, while execution time for layers five through seven of VGG-Nagadomi is cut by 66.61%.

3.5 Measurements

The proposed HTNN processor was fabricated using a 22nm ultra-low-leakage (ULL) technology node, as shown in Figure 3.9. To ensure a fair comparison with conventional architectures, the performance and efficiency of WHT convolutions are reported by translating

their operations into equivalent regular convolutions, allowing for a direct performance assessment. Additionally, the measured energy efficiency accounts for all overhead associated with activation and weight memory accesses, reflecting the actual energy consumption of a complete convolutional layer rather than just raw computational efficiency.

The processor demonstrates a peak measured energy efficiency of 9.51 TOPS/W when operating at a supply voltage of 0.46 V and a frequency of 24 MHz. At higher performance settings, it achieves a peak computational throughput of 195 GOPS at 0.8 V and 243 MHz for WHT-domain convolutions utilizing 3 WHTs, as illustrated in Figure 3.10. Beyond convolutional layers, the chip is also optimized for fully connected layers (FCLs) with structured sparsity. When processing sparse FCLs, it reaches an efficiency of up to 22.86 TOPS/W at 0.53 V and 40 MHz, leveraging 90% activation sparsity and 93.75% weight sparsity to maximize computational efficiency, as detailed in Figure 3.7.

Table 3.1 shows our processor outperforms convolutional energy efficiency of state-of-the-art dense and sparse designs targeting similar applications with end-to-end processing by 3.1× and 2.2×, respectively. The runtime performance and energy efficiency on ResNet-20 using 3 WHTs shown in Figure 3.11 and Table 3.2 demonstrate up to 159 GOPS at 0.8 V (243 MHz) and 4.46 TOPS/W at 0.6 V (40 MHz) using batched processing (multiple input per single MRAM load) as well as MRAM pipelining and dynamic power gating optimizations. Table 3.2 also shows that MRAM pipelining, clocking, and power gating optimizations improve ResNet-20 performance and energy efficiency by 4.9× and 1.4×, respectively, at 0.8 V.

3.6 Conclusion

This work presents the first silicon implementation of an HTNN processor, featuring a 2 MB MRAM for all-on-chip weight storage and targeting edge devices. The design overcomes the high leakage and access time of MRAM with several schemes such as dynamic power gating and multi-clock domain double-buffered caching. The proposed processor demonstrates best-

Table 3.1: Comparison table versus state-of-the-art.

	This work	[106]	[107]	[108]	[109]	[110]	[111]	[112]
Technology (nm)	22	22	22	40	22	28	16	65
Applications	AI-IoT	AI-IoT	AI-IoT	AI-IoT	AI-IoT	CNN Acceleration	DNN Acceleration	DNN Acceleration
Die Area (mm^2)	8.76	18.7	12	29.2	6.25	2.69	2.3	7.5
Layer Support	CNN (HTNN), FCL	CNN	CNN, FCL GAN	CNN, FCL	CNN, FCL GAN, AE	2-D/3-D Sparse CNN	CNN, FCL	CNN, FCL
Transform Support	**2-D WHT**	No	No	No	No	No	No	2-D FFT
Sparsity Support	**Orthogonal Weights (WHT CONV) CBB Structured Weights (FCL)**	No	No	No	Block-wise Sparse Inputs	Block-wise Sparse Inputs	Unstructured Sparse Inputs & Weights	Block-circulant Weights
INT precision (bit)	8	2–8	8	8	2–8	8	16	1–12
On-Chip NVM	**2 MB MRAM**	No	2 MB MRAM	2 MB RRAM	512 kB MRAM	No	No	No
Weight Storage	**On-Chip NVM**	SRAM	On-Chip NVM	On-Chip NVM	On-Chip NVM	Off-Chip	Off-Chip	Off-Chip
On-Chip SRAM (kB)	877.5	1152	1157	512	644	176	280.6	100
Supply Voltage (V)	0.46–0.8	0.5–0.8	0.44–1.0	1.1	0.4–0.9	0.48–0.92	0.55–0.8	0.54–1.15
Frequency (MHz)	24–243	100–240	0.8–70	200	0.033–150	40–450	33–480	25–200
Power (mW)	2.03–45.76	12.8–123	0.25–3.84	126	0.0017–20	6.6–179.4	16.3–364	13.3–339.2
Peak Performance (GOPS)	195[1,2]	122[1]	62.4[1]	920[1]	17.6[1]	760[1]	241.92[7,8]	398[9]
Peak Energy Efficiency (TOPS/W)	**9.51**[1,2] (WHT CONV) **22.86**[1,3,4] (FCL) **including act. & wgt. mem. accesses**	1.8[1] (CONV) N/A (FCL) **including act. & wgt. mem. accesses**	3.11[1] (CONV) 0.57[1] (FCL) **including act. & wgt. mem. accesses**	2.2[1] (CONV) N/A (FCL) **including act. & wgt. mem. accesses**	4.31[1,5] (CONV) 0.829[1] (FCL) **including act. & wgt. mem. accesses**	11.99[1,6] (CONV) N/A (FCL) **excluding main wgt. mem. (off-chip) accesses**	5.06[7,8] (CONV) N/A (FCL) **excluding main wgt. mem. (off-chip) accesses**	3.76[9] (FFT CONV) 12.9[10] (FFT FCL) **excluding main wgt. mem. (off-chip) accesses**

[1]8-bit mult., [2]w/ 3 WHTs, [3]0.94 sparse wgt., [4]0.9 sparse act, [5]0.5 sparse act., [6]0.4 sparse act., [7]0.6 sparse act./wgt., [8]16-bit mult., [9]5-bit mult., [10]2-bit mult.

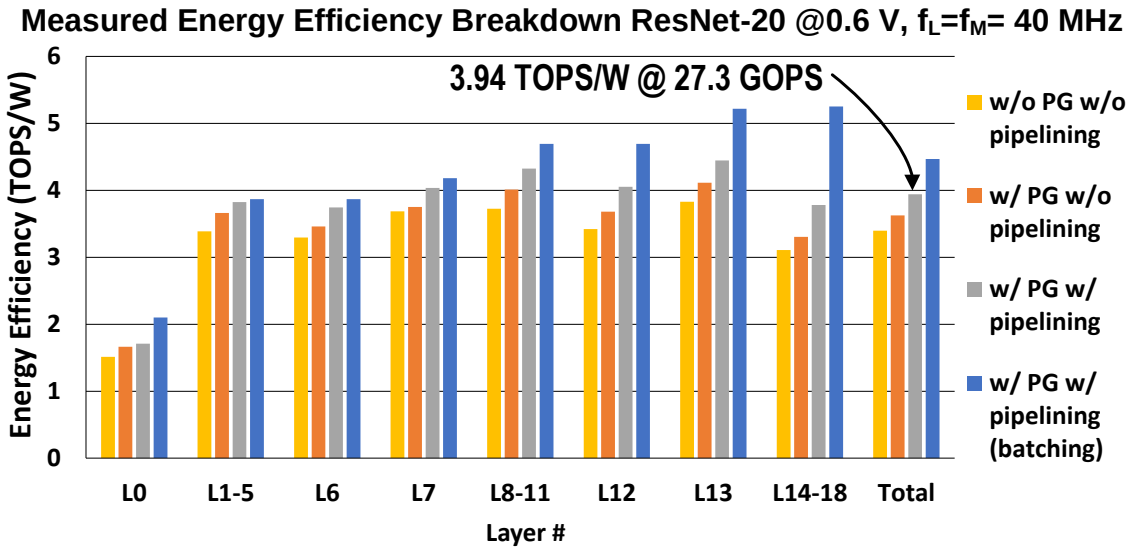


Figure 3.11: Measured efficiency layer breakdown at mid performance points for ResNet-20 HTNN.

Table 3.2: Measured impact of MRAM optimizations on performance and efficiency at high performance point for ResNet-20 HTNN.

Measurement @ 0.8 V		Performance	Energy Efficiency
f_L (MHz)	55	34.2 GOPS	2.023 TOPS/W
f_M (MHz)	55		
Pipelining & PG	**Disabled**		
f_L (MHz)	243	159.2 GOPS	2.917 TOPS/W
f_M (MHz)	55		
Pipelining & PG	**Enabled**		

in-class 9.51 and 22.86 TOPS/W peak energy efficiencies for sparse convolution and fully-connected layers, respectively, in a 22 nm CMOS process.

CHAPTER 4

One-Hot Multi-Level Leaky Integrate-and-Fire Spiking Neural Networks for Enhanced Accuracy-Latency Tradeoff

4.1 Introduction

Neural networks have become a fundamental technique for solving many important problems such as image classification, object detection, and face recognition [56, 2]. As neural network accuracy improves, models become increasingly complex, making their energy-efficient deployment on the edge a significant challenge. In order to reduce the computational complexity of these tasks, spiking neural networks (SNNs) [54] were proposed as an alternative to traditional artificial neural networks (ANNs) [56, 57] such as CNNs and HTNNs explored in Chapters 2 and 3. SNNs infer inputs across multiple timesteps while ANNs perform a one-shot inference, essentially over a single timestep. Neurons in SNNs differ from those in ANNs as they operate on sparse binary spike trains as opposed to non-binary 'analog' activations, resulting in the substitution of multiplications with energy-efficient additions [58].

To model spikes over time, SNNs employ various techniques, most notably the leaky integrate-and-fire (LIF) neuron model [113, 114]. Each neuron is characterized by two parameters: firing threshold and membrane leakage. During a timestep, a neuron either re-

mains silent or produces a spike if the membrane potential exceeds its firing threshold. The membrane potential can shrink over time depending on the membrane leakage and is reset if a spike is produced. Using such models, many training methods have emerged and can be categorized into two main approaches: ANN-SNN conversion and direct training. ANN-SNN conversion methods [115, 116, 117] convert the weights of a pre-trained ANN to an iso-architecture SNN. However, these methods can require a large number of timesteps (on the order of 1000 in some cases [118]) to achieve comparable or better accuracy than ANNs. Note that multi-timestep inference results in more memory storage/accesses which can dominate the compute cost [43]. Therefore, the reliance on multi-timestep processing in ANN-to-SNN conversion has been a key factor hindering the practical deployment of SNNs in energy-constrained edge scenarios.

Direct training using surrogate gradient-based optimization and back-propagation through time (BPTT) [119, 120, 121] has enabled training SNNs with significantly fewer timesteps, occasionally reducing them to just a single timestep [122]. However, these SNNs still lag ANNs in terms of accuracy. For image classification on static datasets [1, 56], SNNs cannot bridge this accuracy gap with ANNs without increasing the number of timesteps, which in turn reduces energy efficiency. Moreover, single-timestep SNNs such as [122] require iterative temporal pruning to converge, rendering training more time-consuming. For image classification, multi-timestep SNNs traditionally outperform ANNs on dynamic vision sensor [123] data but suffer from a sharp accuracy degradation [124] with a reduced number of timesteps.

As conventional LIF models employ binary-valued neuron spike outputs, traditional SNNs are restricted to solely scaling the temporal dimension T (*i.e.*, the number of timesteps) to achieve different accuracy-energy efficiency tradeoffs. Prior SNN works have explored the interaction between activation bit-width and timesteps using multi-spike [125, 126] and multi-threshold [127, 128] neurons to increase activation precision via uniform quantization. However, these approaches raise computational energy costs by disrupting the multiplication-free

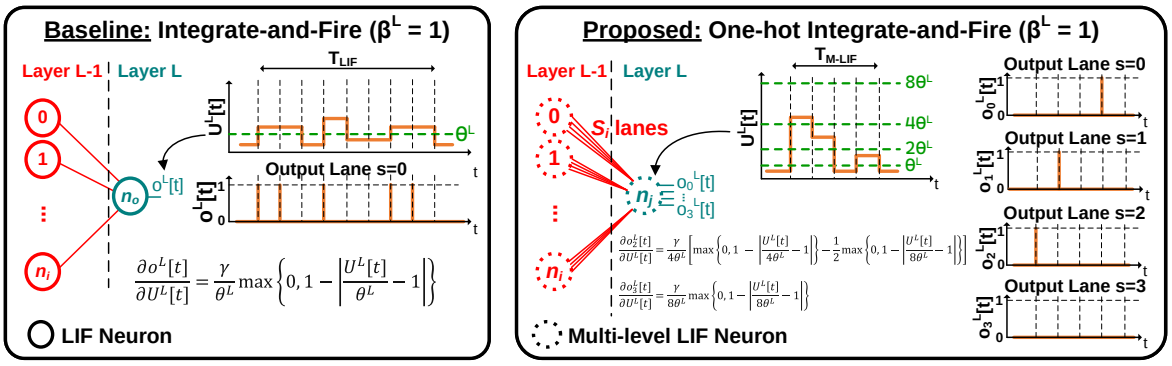


Figure 4.1: LIF neuron model (left) vs. one-hot M-LIF neuron model with $S_o = 4$ lanes (right).

nature of traditional SNNs. To address these limitations and enhance the accuracy-energy tradeoff space, this work extends the range of neuron spike outputs to include strictly powers of two by leveraging an additional dimension, S, and by using a novel one-hot multi-level leaky integrate-and-fire (M-LIF) neuron model as illustrated in Figure 4.1. The proposed one-hot M-LIF model has the following key properties: (1) it uses the S dimension to represent hidden layer outputs (inputs) as a set of S_o (S_i) binary-weighted spike lanes, and (2) it limits the simultaneous firing behavior of those S_o (S_i) spike lanes to only a single lane per timestep. These two properties of the proposed one-hot M-LIF model enable new accuracy-energy tradeoff points for SNNs while still only requiring additions (without multiplications) like the conventional LIF model. Furthermore, the proposed one-hot M-LIF model can be easily integrated into prior existing training frameworks. For static datasets such as CIFAR and ImageNet, this work demonstrates that one-hot M-LIF SNNs outperform conventional LIF SNN accuracy while achieving better or comparable energy efficiency on various architectures including the high-performance spike-driven transformer [121]. For dynamic vision datasets such as DVS-CIFAR-10 [129, 130], M-LIF SNNs using multi-level input layer encoding can achieve reduced timesteps (energy consumption) compared to LIF SNNs for comparable or better accuracy. To summarize, the main contributions are:

- Proposing a new direction to balance SNN energy efficiency and accuracy using a novel

one-hot M-LIF neuron model.

- Enabling a new tradeoff and show that one-hot M-LIF SNNs are more accurate than iso-architecture LIF SNNs while consuming comparable or lower energy with fewer timesteps.

- Demonstrating the benefit of one-hot M-LIF SNNs with dynamic vision sensor-based input compared to conventional SNNs while being the first SNN work to achieve a top-1 accuracy of 82.5% on DVS-CIFAR-10 using only 3 timesteps.

4.2 Background & Related Works

4.2.1 Leaky Integrate-and-Fire Model

A conventional spiking neural network (SNN) layer under the leaky integrate-and-fire (LIF) neuron model is described by

$$\mathbf{u}^l[t] = \beta^l \mathbf{u}^l[t-1] + \mathbf{W}^l \mathbf{o}^{l-1}[t] - \theta^l \mathbf{o}^l[t-1] \tag{4.1}$$

$$\mathbf{o}^l[t] = \begin{cases} 1, & \text{if } 1 < \frac{\mathbf{u}^l[t]}{\theta^l} \\ 0, & \text{otherwise} \end{cases} \tag{4.2}$$

where $\mathbf{W}^l$ is the weight matrix connecting layers $l-1$ and l, $\mathbf{u}$ is a vector containing the membrane potential of output neurons, $\beta \in [0,1]$ is the leakage factor, $\mathbf{o}$ is an output vector of binary spikes, $\theta > 0$ is the firing threshold, and $t \in \{0,1,2,...\}$ represents the discrete timestep. The first term in Equation (4.1) corresponds to the membrane leakage allowing the potential to shrink (leak) over time, and the final term accounts for resetting the potential to a specific value when an output binary spike given by Equation (4.2) is generated. All neurons in an input/hidden layer typically share the same leakage factor and firing threshold values. As for the final layer, adopting the LIF model without any

modifications can significantly impact the accuracy [120, 119]. Hence, the final output layer neurons only accumulate incoming inputs without any leakage and do not fire output spikes. Finally, the inference process is repeated for T timesteps from $t = 0$ to $T-1$, and the output of the last layer is averaged to produce the final result. The output spiking behavior of traditional LIF neurons in Equation (4.2) is depicted in Figure 4.1 (left) where the output range of the membrane potential $\mathbf{u}^L[t]$ is subdivided using a single decision boundary defined by the threshold θ^L.

4.2.2 ANN-SNN Conversion

ANN-SNN conversion methods [118, 115, 116, 117, 131, 132] convert the weights of a pre-trained ANN to an iso-architecture SNN. Specifically, these methods convert the output of a rectified linear unit neuron in an ANN into a sequence of binary spikes in the SNN over multiple timesteps. The primary challenge in this technique is determining the firing threshold in such a way that it balances the accuracy-latency tradeoff. In these methods, the firing thresholds are generally determined by profiling the pre-trained ANN and recording a certain percentile of layers' input activation distributions. However, these heuristic techniques can lead to a sub-optimal choice of firing threshold and can also require a large number of timesteps to achieve comparable or better accuracy than ANNs, thus further aggravating the accuracy-latency tradeoff. This multi-timestep processing requirement is a challenge for widespread SNN deployment as it primarily introduces more memory storage and accesses which can be significantly higher than compute cost [43].

4.2.3 Direct Training

An alternate approach to training SNNs is to use gradient-based techniques, such as back-propagation, either from scratch or from a pre-trained iso-architecture ANN [119, 120, 122, 133]. These approaches relate the temporal dimension of SNNs to that of recurrent neural networks, and perform back-propagation through time (BPTT) to learn weights across

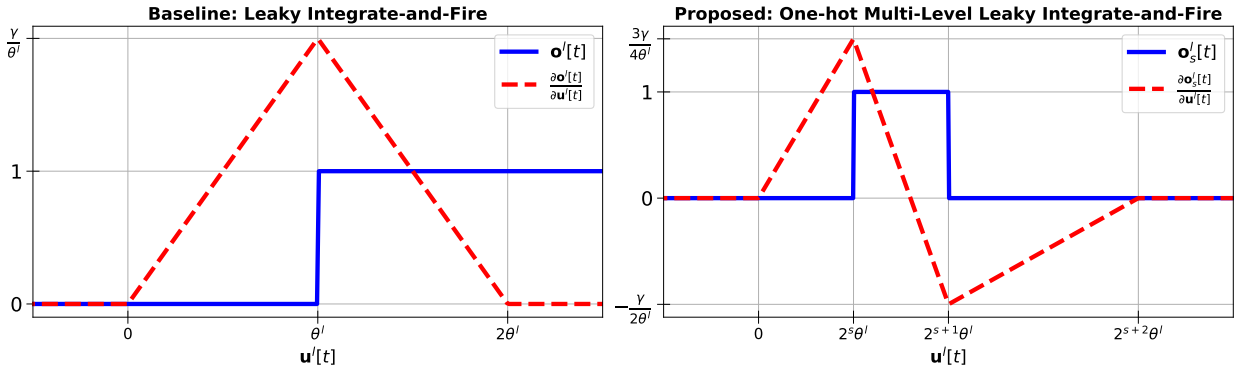


Figure 4.2: Output activation and surrogate gradient functions for conventional LIF neuron model (left) and one-hot multi-level LIF neuron model for $0 \leq s < S_o - 1$ (right).

multiple timesteps.

The cross-entropy loss L and gradients $\partial L/\partial \mathbf{W}^l$ are calculated by

$$L = -\sum_i y_i \log(\Phi(\mathbf{o}^L[T-1])_i) \tag{4.3}$$

$$\frac{\partial L}{\partial \mathbf{W}^l} = \sum_t \frac{\partial L}{\partial \mathbf{o}^l[t]} \frac{\partial \mathbf{o}^l[t]}{\partial \mathbf{u}^l[t]} \frac{\partial \mathbf{u}^l[t]}{\partial \mathbf{W}^l} \tag{4.4}$$

where L is the index of the final layer, $\Phi(\cdot)$ denotes the softmax function, and $\mathbf{y}$ is the one-hot encoded vector of the true label. The term $\partial \mathbf{o}^l[t]/\partial \mathbf{u}^l[t]$ in Equations (4.3) and (4.4) is the discontinuous gradient that is typically replaced by differentiable surrogate gradients. Prior works have explored the use of various surrogate gradient shapes such as triangular [134, 119], or the derivative of the sigmoid function [121] which are given below in Equations (4.5) and (4.6), respectively, where γ and α are constants used to scale the shapes of the gradients. The triangular surrogate gradient in Equation (4.5) used for the output spike of the traditional LIF model is depicted in Figure 4.2 (left) for the single activation channel case where γ is shown to scale the peak value of the gradient at $\mathbf{u}^L[t] = \theta^L$.

$$\frac{\partial \mathbf{o}^l[t]}{\partial \mathbf{u}^l[t]} = \text{diag}\left(\frac{\gamma}{\theta^l}\max\{0, 1 - |\frac{\mathbf{u}^l[t]}{\theta^l} - 1|\}\right) \tag{4.5}$$

$$\frac{\partial \mathbf{o}^l[t]}{\partial \mathbf{u}^l[t]} = \text{diag}\Bigg(\frac{\alpha}{\theta^l}\left(1 - \sigma\big(\alpha(\frac{\mathbf{u}^l[t]}{\theta^l} - 1)\big)\right) \cdot \sigma\big(\alpha(\frac{\mathbf{u}^l[t]}{\theta^l} - 1)\big)\Bigg) \tag{4.6}$$

Compared to ANN-SNN conversion techniques, direct training approaches typically achieve a better accuracy-latency tradeoff (higher accuracy using fewer timesteps overall) at the cost of more compute- and memory-intensive training [135]. To achieve competitive accuracy, most employ direct input encoding [136] and utilize the first layer as a spike generator by directly feeding pixel values as inputs to the network. Through gradient-based learning, many recent works have sought to optimize different aspects of this training methodology such as loss function definition [120], initialization and parameterization [119, 122], and extension to advanced network architectures such as spike-driven transformers [121].

Notably, the authors in [122] propose temporal pruning to gradually reduce the number of timesteps to successfully train SNNs with as little as a single timestep, despite a noticeable accuracy degradation (up to 4%). All these works differ from this work as they are restricted to solely scaling the temporal dimension T in order to achieve different accuracy-latency tradeoffs given a fixed neural network architecture. To address this limitation, the proposed approach leverages the S dimension using a one-hot multi-level LIF (M-LIF) neuron model to improve the accuracy-latency tradeoff space.

4.2.4 Quantized-Activation ANNs

As the number of timesteps converges to one, conventional SNNs become closely related to binary activated artificial neural networks (BNNs) [137, 138] as both use binary activations to perform an inference over a single timestep. The authors in [122] discuss that they are in

fact distinct. While SNNs quantize outputs to spikes (*i.e.*, $\{0, 1\}$), BNNs quantize outputs to be ± 1. Unlike BNNs which use non-linear activation functions where the firing threshold is zero, the firing threshold is learnable in SNNs. The authors in [122] observe that this allows SNNs to outperform BNNs in terms of accuracy and scale better to larger datasets such as ImageNet. Moreover, LIF enables SNNs to extend the same network for sequential processing unlike BNNs. Similarly, one-hot M-LIF SNNs become closely related to log-quantized-activation ANNs (LQ-ANNs) [44, 45] as timesteps converge to one, but remain distinct for analogous reasons. A discussion regarding similarities and differences is provided in Section 4.3.2.

4.2.5 Quantized-Activation SNNs

Prior SNN works have explored the interaction between activation bit-width and timesteps [125, 126, 127, 128, 139, 140]. This work's major distinction is that the one-hot M-LIF neuron model constrains outputs of neurons to be powers-of-two while enabling better accuracy with low spike rates in the low-latency (unit-timestep) regime.

Multi-threshold SNNs [127, 128] apply multiple thresholds to each membrane potential after integration, either in parallel or sequentially, and sum resulting spikes together to produce an output activation. As a result, multi-threshold neurons lead to a uniformly quantized output whose output range depends on the number of thresholds used. The proposed one-hot M-LIF neuron on the other hand uses dual-sided thresholding as shown later in Equation (4.8) to only produce a single spike on one of S_o weighted output spike lanes. As a result, this work restricts outputs to be strictly powers-of-two, a constraint which is accounted for during training. This enables efficient 32-bit floating-point (FP32) weight exponent updates via a single 8-bit integer (INT8) addition as discussed in Section 4.4, unlike the multiplication demands of uniformly quantized outputs. In [127, 128], the surrogate gradient of a traditional LIF neuron (e.g., Equations (4.5) or (4.6)) is applied to each spike lane while in the proposed approach, the surrogate gradient of the traditional LIF neuron

was adapted to handle dual-sided thresholding per spike lane. Finally, unlike [127, 128] which perform evaluations solely on convolutional neural network (CNN) architectures, the evaluations in this work span both CNNs and the more complex, high-performing spike-driven transformer architecture.

Burst-spike neuron models [139, 140] work by increasing the spike rate, in other words the number of spikes issued within a single timestep. This is unlike the proposed approach which emits at most a single spike per timestep independent of the number of spike lanes used per neuron. Moreover, evaluations for burst-spike neuron models show that resulting SNNs do not scale down to unit-timestep processing. For example, [140] reports results for at least $T \geq 32$ timesteps with a 14.2% and 18.2% accuracy drop at $T = 16$ for VGG16 and ResNet-20 on CIFAR-100, respectively. The work in [139] reports an accuracy of 70.61% for VGG16 on ImageNet with $T = 8$ (vs. 71.05% with $T = 1$ for one-hot M-LIF VGG16 SNN using $T = 1$ and $S = 3$) and lacks comprehensive reporting of energy estimation analyses for all experimental results.

Memory energy models for SNNs such as in [141] indicate that memory access energy of spikes and membrane potentials scales linearly with the number of timesteps, with single memory access costs often being 10× higher than single compute energy costs as discussed in [43] for a 45 nm CMOS technology. Unlike burst-spike neuron models that require multi-timestep processing, the proposed one-hot M-LIF SNN models achieve high accuracy with unit-timestep processing as reported later in Table 4.1 of this chapter along with comparisons against existing state-of-the-art unit-timestep SNNs [141, 122].

4.3 Proposed One-Hot Multi-level LIF-based SNNs

4.3.1 Multi-level LIF Model

The goal of this work is to reduce the number of timesteps T during SNN inference while improving accuracy and maintaining the low-spike rates of traditional SNNs. Reducing the

timesteps T while maintaining low spike rates can consequently decrease the energy overhead associated with multi-timestep processing. To do so, the proposed multi-level leaky integrate-and-fire (M-LIF) neuron model still uses a single membrane potential per output neuron while extending the range of neuron outputs to more than just binary representations. It employs a new dimension S to represent hidden layer outputs (inputs) as a set of S_o (S_i) binary-weighted spike lanes. This work denotes $\mathbf{o}_s^l[t]$ to be the output vector of binary spike lane s in layer l at timestep t. Each spike lane s is weighted by 2^s resulting in a non-binary output range for the neuron output $\mathbf{o}^l[t]$. As a result, Equation (4.1) needs to be modified to combine weighted spike lanes prior to updating the membrane potential as follows

$$\begin{aligned}\mathbf{u}^l[t] &= \beta^l \mathbf{u}^l[t-1] + \sum_{s=0}^{S_i-1} 2^s \mathbf{W}^l \mathbf{o}_s^{l-1}[t] - \sum_{s=0}^{S_o-1} 2^s \theta^l \mathbf{o}_s^l[t-1] \\ &= \beta^l \mathbf{u}^l[t-1] + \mathbf{W}^l \mathbf{o}^{l-1}[t] - \theta^l \mathbf{o}^l[t-1].\end{aligned} \tag{4.7}$$

This work also defines $\omega \leq S_o$ as the maximum number of simultaneously firing output spike lanes in any given timestep. With all spike lanes sharing the same firing threshold and membrane potential, it is non-trivial to devise a firing mechanism with $\leq \omega$ concurrent firing lanes. Section 4.3.2 discusses the practical one-hot case proposed for SNNs where $\omega = 1$.

4.3.2 One-hot Multi-level LIF Model

In the one-hot M-LIF model, $\omega = 1$ and *only one of the S_o output spike lanes fires in any given timestep*. The threshold mechanism is given by

$$\mathbf{o}_s^l[t] = \begin{cases} 1, & \text{if } \left(2^s < \frac{\mathbf{u}^l[t]}{\theta^l} \leq 2^{s+1} \wedge 0 \leq s < S_o - 1\right) \\ & \quad \vee \left(2^s < \frac{\mathbf{u}^l[t]}{\theta^l} \wedge s = S_o - 1\right) \\ 0, & \text{otherwise} \end{cases} \tag{4.8}$$

Figure 4.1 depicts the difference between the conventional LIF model (left) and the pro-

posed one-hot ($\omega = 1$) M-LIF model (right) given $S_o = 4$. Instead of having only one output (input) spiking signal, an M-LIF neuron has multiple ($S_o = 4$) binary-weighted output (input) spike lanes. With the one-hot constraint, only a single spike lane fires at any given timestep. In the example illustrated in Figure 4.1 (right), the membrane potential can increase by one of $S_o = 4$ possible levels in $\{\theta^l, 2\theta^l, 4\theta^l, 8\theta^l\}$ from one timestep to the next, as opposed to the single level θ^l in the conventional LIF model. The output spike lanes are one-hot, meaning that the output range is also subdivided into $S_o = 4$ non-overlapping decision boundaries using the binary weight of each spiking lane as shown in Equation (4.9). Therefore, the single activation channel case yields $\sum_{s=0}^{S_o-1} 2^s \mathbf{o}_s^l[t] = \mathbf{o}^l[t] \in \{0, 1, 2, 4, \ldots, 2^{S_o-1}\}$.

$$\begin{aligned}
\mathbf{o}_0^l[t] &= \begin{cases} 1, & \text{if } 1 < \frac{\mathbf{u}^l[t]}{\theta^l} \leq 2 \\ 0, & \text{otherwise} \end{cases} \quad \mathbf{o}_1^l[t] = \begin{cases} 1, & \text{if } 2 < \frac{\mathbf{u}^l[t]}{\theta^l} \leq 4 \\ 0, & \text{otherwise} \end{cases} \\
\mathbf{o}_2^l[t] &= \begin{cases} 1, & \text{if } 4 < \frac{\mathbf{u}^l[t]}{\theta^l} \leq 8 \\ 0, & \text{otherwise} \end{cases} \quad \mathbf{o}_3^l[t] = \begin{cases} 1, & \text{if } 8 < \frac{\mathbf{u}^l[t]}{\theta^l} \\ 0, & \text{otherwise} \end{cases}
\end{aligned} \tag{4.9}$$

Note that by setting $S_i = S_o = 1$ (*i.e.*, single lane), Equations (4.7) and (4.8) simplify to Equations (4.1) and (4.2). Therefore, binary spiking SNNs can be considered as a special case of the proposed one-hot M-LIF scheme.

4.3.2.1 Surrogate Gradient for Back-Propagation Training.

Given the update to $\mathbf{o}^l[t]$ in Equation (4.8), the surrogate gradient is extended from the LIF case to the proposed one-hot M-LIF neurons. For example, Figure 4.2 illustrates the differences in an updated triangular surrogate gradient for the single activation channel case when $0 \leq s < S_o - 1$. Each spike lane is now a window function of $\mathbf{u}^l[t]$ as opposed to a step function. This is equivalent to the difference of two step functions, the first of which is evaluated at the spike lane's threshold, and the second of which is evaluate at the next spike lane's threshold. As a result, the surrogate gradient becomes the difference of two triangular

sub-gradients, one for each of the rising and falling window edges, as shown in Equation (4.10). An example illustrating Equation (4.10) for $S_o = 4$ is provided in Figure 4.3.

$$\begin{aligned}
\mathbf{z}^l(s) &= \frac{\gamma}{2^s\theta^l}\max\{0, 1 - |\frac{\mathbf{u}^l[t]}{2^s\theta^l} - 1|\} \\
\frac{\partial \mathbf{o}^l[t]}{\partial \mathbf{u}^l[t]} &= \sum_{s=0}^{S_o-1} 2^s \frac{\partial \mathbf{o}_s^l[t]}{\partial \mathbf{u}^l[t]} \\
\frac{\partial \mathbf{o}_s^l[t]}{\partial \mathbf{u}^l[t]} &= \begin{cases} \operatorname{diag}\left(\mathbf{z}^l(s)\right), & \text{if } s = S_o - 1 \\ \operatorname{diag}\left(\sum_{a=0}^{1}(-1)^a \mathbf{z}^l(s+a)\right), & \text{if } 0 \le s < S_o - 1 \end{cases}
\end{aligned} \tag{4.10}$$

Adapting the derivative of sigmoid function σ' to one-hot M-LIF is performed in a similar fashion to the triangular surrogate gradient described in the main text. The surrogate gradient becomes the difference of σ' sub-gradients, one for each of the rising and falling window edges of the firing function, as shown in Equation (4.11).

$$\begin{aligned}
\mathbf{z}^l(s) &= \sigma\left(\alpha(\frac{\mathbf{u}^l[t]}{2^s\theta^l} - 1)\right) \\
\mathbf{q}^l(s) &= \frac{\alpha}{2^s\theta^l}\left(1 - \mathbf{z}^l(s)\right)\mathbf{z}^l(s) \\
\frac{\partial \mathbf{o}^l[t]}{\partial \mathbf{u}^l[t]} &= \sum_{s=0}^{S_o-1} 2^s \frac{\partial \mathbf{o}_s^l[t]}{\partial \mathbf{u}^l[t]} \\
\frac{\partial \mathbf{o}_s^l[t]}{\partial \mathbf{u}^l[t]} &= \begin{cases} \operatorname{diag}\left(\mathbf{q}^l(s)\right), & \text{if } s = S_o - 1 \\ \operatorname{diag}\left(\sum_{a=0}^{1}(-1)^a \mathbf{q}^l(s+a)\right), & \text{if } 0 \le s < S_o - 1 \end{cases}
\end{aligned} \tag{4.11}$$

4.3.2.2 Discussion.

As $T \to 1$ (*i.e.*, unit timestep inference), Equations (4.7) and (4.8) can be rewritten as

$$\mathbf{u}^l = \sum_{s=0}^{S_i-1} 2^s \mathbf{W}^l \mathbf{o}_s^{l-1} = \mathbf{W}^l \mathbf{o}^{l-1} \tag{4.12}$$

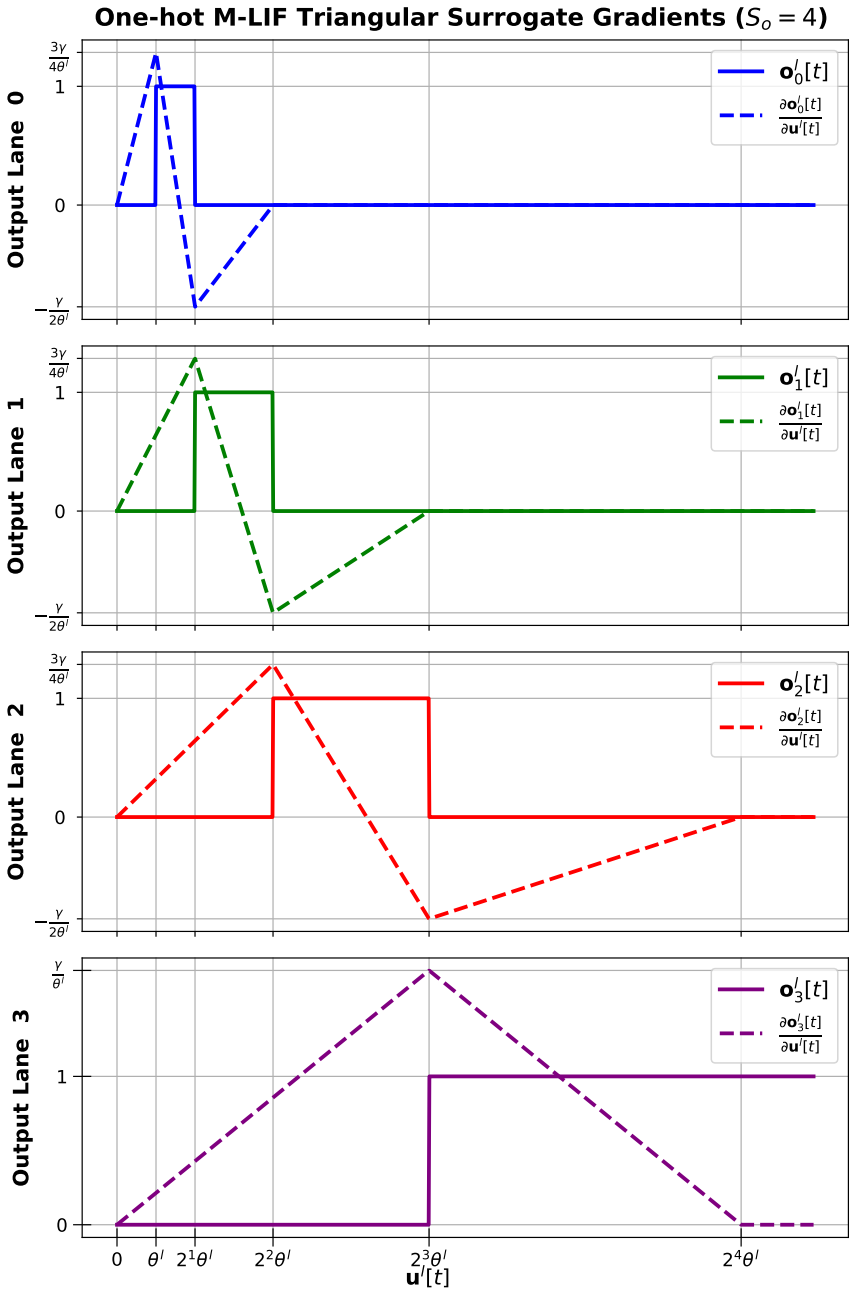


Figure 4.3: Example of triangular surrogate gradient for $S_o = 4$ one-hot M-LIF neuron output spike lanes.

where

$$\mathbf{o}^l = 2^{\text{clip}\left(\lfloor \log_2\left(\frac{\mathbf{u}^l}{\theta^l}\right)\rfloor, 0, S_o\right)}, \quad \text{clip}(x, v, z) = \begin{cases} -\infty, & \text{if } x \leq v \\ z - 1, & \text{if } x \geq z \\ x, & \text{otherwise} \end{cases}.$$

From this, an observable parallel can be drawn between the proposed one-hot M-LIF SNNs with unit timestep ($T = 1$) and log quantized-activation ANNs (LQ-ANNs) [142, 48]. While one-hot M-LIF-based SNNs are trained in a single phase, LQ-ANN training is performed in two phases per epoch. First, using the entire training dataset and full precision inference, a percentile value α of each layer's input activation distribution is recorded. Second, a straight-through estimator is typically applied to approximate the gradient with respect to quantized activations. Using b bits and assuming a ReLU activation function, the neuron output in an LQ-ANN is given by Equation (4.13). While both one-hot M-LIF SNNs with unit timestep ($T = 1$) and LQ-ANNs share commonalities, they remain slightly distinct. As highlighted by Equations (4.12) and (4.13), they namely differ in their choices of firing threshold and their final output value ranges. They also differ in their training methods and abilities to extend to sequential processing (*i.e.*, $T > 1$).

$$\mathbf{o}^l = \mathrm{ReLU}(\mathbf{W}^l \tilde{\mathbf{o}}^{l-1}), \qquad \tilde{\mathbf{o}}^{l-1} = \begin{cases} \alpha^l 2^{\mathrm{clip}\left(\lfloor \log_2\left(\frac{\mathbf{o}^{l-1}}{\alpha^l}\right) \rceil, 1-2^b, 1\right)}, & \text{if } \mathbf{o}^{l-1} \neq 0 \\ 0, & \text{otherwise} \end{cases} \tag{4.13}$$

4.4 Energy Consumption Estimation

The inference energy of the proposed approach was evaluated based on the approach in [122, 121]. In conventional SNNs, 32-bit floating-point (FP32) additions replace the FP32 multiply-and-accumulates (MACs) of ANNs except in the first layer which uses direct encoded inputs. For one-hot M-LIF SNNs ($\omega = 1$), inputs (outputs) are restricted to powers of two and, as illustrated in Figure 4.4 which provides details regarding the FP32 number format, multiplying by a power of two corresponds to adjusting the 8-bit integer (INT8) exponent of the FP32 multiplicand. Therefore, scaling the intermediate FP32 membrane potential by 2^s during integration in Equation (4.7) corresponds to increasing or decreasing its exponent in the INT8 format. According to [43], an INT8 addition consumes 30× less energy than a FP32 addition, hence the overhead of scaling in M-LIF SNNs is negligible

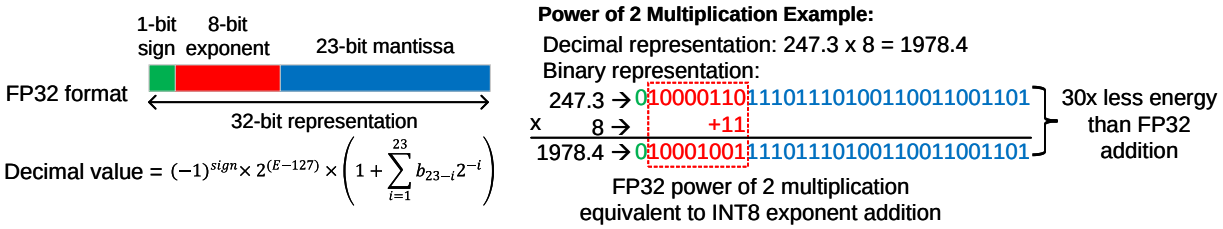


Figure 4.4: 32-bit floating-point format and power of two multiplication details where b_i is the i-th bit, $sign$ is the most significant bit, and E is the 8-bit exponent.

and FP32 additions dominate the energy consumption of one-hot M-LIF SNNs. In other words, M-LIF SNNs are indeed considered to leverage additions instead of multiplications like their LIF counterparts. The proposed approach is designed to be adaptable to various hardware architectures, including custom ASICs or FPGAs, where the operation depicted in Figure 4.4 can indeed be implemented efficiently. It is worth noting that modern commercial instruction set architectures, such as `x86`, include specific support for efficiently scaling floating-point numbers. For example, the `x87` floating-point unit (FPU) provides specialized instructions like `fscale` [143] designed to scale floating-point numbers by powers of two, rather than performing a general floating-point multiplication. This indeed suggests that the operation described in Figure 4.4 aligns with existing commercial hardware capabilities.

Due to the one-hot constraint, the overall spiking rate (and consequently, the number of additions) per layer per timestep in one-hot M-LIF SNNs is also not necessarily higher than that of conventional SNNs, even though one-hot M-LIF SNNs have multiple spiking lanes per neuron. This gives one-hot M-LIF SNNs the opportunity to learn more within a single timestep without increasing the computational complexity and energy compared to conventional SNNs.

It is also known that memory access energy can be significantly higher than compute energy [43, 23] and that a proportion of the number of memory accesses scales linearly with the number of timesteps in SNNs [122, 141]. However, estimating memory energy

improvements would depend on hardware architecture and system configuration. Therefore, as noted in [122], this work restricts its attention mainly to the computational energy benefits, δ defined in Equation (4.14) [122], of one-hot M-LIF SNNs and conventional SNNs over ANNs. As a result, this work considers δ to be an optimistic energy gain estimate when $T > 1$. Note that when $T = 1$, memory requirements are identical for both SNNs and full-precision ANNs. When an iso-architecture ANN does not exist as in the case of spike-driven transformers [121] due to unique mechanisms such as spike-driven attention, this work compares directly using the computational energy E.

As depicted in Figure 4.1, the one-hot M-LIF neuron model does not introduce any additional neurons to the existing set of neurons in iso-architecture SNNs and ANNs. As a result, ANNs and unit-timestep SNNs (both one-hot M-LIF and traditional LIF) have the same computational complexity. The computational complexity of multi-timestep SNNs, both one-hot M-LIF and traditional LIF, scales linearly with the number of timesteps as the inference process is repeated for T timesteps. Note however, that multi-timestep and unit-timestep SNNs, both one-hot M-LIF and traditional LIF, exhibit high activation sparsity r_l^t per layer at each timestep enabling linear reduction in computational complexity. In Equation (4.14), the computational complexities $\#\text{ANN}_{\text{ops},l}$ and $\#\text{SNN}_{\text{ops},l}$ of layers in ANNs and SNNs are given in Equations (4.15) and (4.16) for convolution layers and in Equations (4.17) and (4.18) for linear layers, respectively. Based on [43], the relative MAC and addition energy are set to 4.6pJ and 0.9pJ, respectively. C_l^i, C_l^o, k_l, H_l, and W_l denote the number of input channels, number of output channels, kernel size, feature map height, and feature map width, respectively. These computational complexities are employed along with Equation 4.14 to calculate the computational energy expenditure reported in Tables 4.1, 4.2, and 4.4 throughout Section 4.5. Finally, note that in one-hot M-LIF SNNs, only a single spike lane fires per timestep as explained in Section 4.3.2. The spiking activity is already accounted for in the term r_l^t. Therefore, the computational complexity does not explicitly depend on

the number of spike lanes per timestep.

$$\delta = \frac{E_{\text{ANN}}}{E_{\text{SNN}}} = \frac{\sum_{l=1}^{L} \#\text{ANN}_{\text{ops},l} \times 4.6}{\#\text{SNN}_{\text{ops},1} \times 4.6 + \sum_{l=2}^{L} \#\text{SNN}_{\text{ops},l} \times 0.9} \tag{4.14}$$

$$\#\text{ANN}_{\text{ops},l_{\text{conv}}} = H_{l_{\text{conv}}} \times W_{l_{\text{conv}}} \times C^{i}_{l_{\text{conv}}} \times C^{o}_{l_{\text{conv}}} \times k^{2}_{l_{\text{conv}}} \tag{4.15}$$

$$\#\text{SNN}_{\text{ops},l_{\text{conv}}} = \sum_{t=1}^{T} r^{t}_{l} \times \#\text{ANN}_{\text{ops},l_{\text{conv}}} \tag{4.16}$$

$$\#\text{ANN}_{\text{ops},l_{\text{linear}}} = C^{i}_{l_{\text{linear}}} \times C^{o}_{l_{\text{linear}}} \tag{4.17}$$

$$\#\text{SNN}_{\text{ops},l_{\text{linear}}} = \sum_{t=1}^{T} r^{t}_{l} \times \#\text{SNN}_{\text{ops},l_{\text{linear}}} \tag{4.18}$$

4.5 Experiments & Results

This work validates the proposed one-hot M-LIF model and compares the performance and inference energy of the proposed one-hot M-LIF SNNs with existing SNN works on both static and dynamic image classification tasks. The proposed neuron model can be integrated into existing SNN training methodologies. Hybrid training methods [122, 119] are compared against for static tasks (Section 4.5.1) and the temporal efficient training method [120] is compared against for dynamic vision tasks (Section 4.5.2). The impact of one-hot M-LIF on more complex SNN-based high performance models such as the spike-driven transformer [121] was also evaluated. As in prior works, this work employs direct input encoding for static tasks such that the input layer is fed with full-precision pixels. This work also fixes all layers to use the same number of (input) output spike lanes, S, as this reduces the number of hyper-parameters. The source code is available at: https://github.com/pabillam/one-hot-mlif-snn.

4.5.1 Static Image Classification

4.5.1.1 Implementation Details

CIFAR datasets [77] which consist of 50000 training and 10000 test images were employed. CIFAR-10 contains 10 classes while CIFAR-100 contains 100 classes. Standard data augmentation techniques were applied to training images such as padding by 4 pixels on each side, random horizontal flip and 32×32 crop by randomly sampling the padded image. During test, the original 32×32 images were used. The channel-wise mean and standard deviation of training images were calculated and used to normalize both training and test data. Contrary to other works such as [124], augmentation techniques such as Cutout [144] and AutoAugment [145] were not applied on CIFAR for CNN experiments in order to faithfully compare with results in [122]. The same augmentation techniques on CIFAR were applied when comparing with results in [121]. The ImageNet dataset [56] which contains 1000 classes and consists of more than 1250000 training and 50000 test images was also employed. The same augmentation techniques on ImageNet were applied when comparing with results in [121, 122].

The VGG16 and ResNet-20 architectures adopted were taken from [122]. Average pooling (2×2) is applied for all cases. The ResNet basic blocks use a 1×1 stride-2 convolutional layer shortcut path where the number of input and output channels are different. The architectures of each network are summarized where BB denotes a basic block [9], D denotes a dropout layer (probability ANN: 0, SNN: 0.2), AP2 denotes a (2×2) average pooling layer, (/2) denotes stride-2, oCk denotes a ($k \times k$) stride-1 convolutional layer with o output filters, oFC denotes a fully-connected layer with o output filers, and n denotes the number of classes. VGG16 is summarized as follows: {64C3, D, 64C3, AP2, 128C3, D, 128C3, AP2, 256C3, D, 256C3, D, 256C3, AP2, 512C3, D, 512C3, D, 512C3, AP2, 512C3, D, 512C3, D, 512C3, D, 4096FC, 4096FC, nFC}. ResNet-20 is summarized as follows: {64C3, D, 64C3, D, 64C3, AP2, 64BB, 64BB, 128BB (/2), 128BB, 256BB (/2), 256BB, 512BB (/2), 512BB,

nFC}.

Hybrid direct training was applied as described in [119, 122] to evaluate the accuracy of the one-hot M-LIF approach on CIFAR-10, CIFAR-100, and ImageNet using VGG16 and ResNet-20. An ANN with batch-normalization [6] was first trained and the batch-norm parameters were subsequently fused with the weights of the corresponding layer. The weights of the pre-trained ANN were then copied to an iso-architecture one-hot M-LIF SNN and the 90-th percentile of the input activation distribution was used as each layer's threshold θ^l. The SNN was then trained using BPTT but without temporal pruning.

To train ANNs, a stochastic gradient descent optimizer with weight decay of 0.0005 and momentum of 0.9 was employed. ANN weights were initialized using He initialization [146]. ResNet-20 and VGG16 ANNs were trained for CIFAR-10 and CIFAR-100 during 300 epochs with an initial learning rate of 0.01 that is divided by 10 at epochs 120, 180, and 240. VGG16 ANN was trained for ImageNet during 90 epochs with an initial learning rate of 0.01 that is divided by 5 at epochs 40, 62, and 81. Batch-normalization was also employed during ANN training [6]. A pre-trained ANN was used to initialize the parameters of a corresponding LQ-ANN and perform training using an Adam optimizer with a weight decay of 0.0005 for 50 epochs and an initial learning rate of 0.0001. The learning rate is divided by 5 at epochs 20, 30, and 40.

For iso-architecture M-LIF SNN initialization from a pre-trained ANN, the batch-norm parameters were fused with the corresponding layer's parameters similar to [122]. For static image classification with VGG16 and ResNet-20, a triangular surrogate gradient with $\gamma = 0.3$ was used and firing thresholds and membrane leakages were learned during training as per [122]. The learning rate was shared among weights, firing thresholds, and membrane leakages unless otherwise noted. Upon initialization, M-LIF SNNs were trained using an Adam optimizer with a weight decay of 0. ResNet-20 M-LIF SNNs were trained on CIFAR-10 for 150 epochs with an initial learning rate of 0.0001 that was divided by 5 at epochs 90, 120, and 135. ResNet-20 M-LIF SNNs were trained on CIFAR-100 for 600 epochs with a

fixed learning rate of 0.01 for firing thresholds and an initial learning rate of 0.0002 that was divided by 2 at epochs 120, 240, 360, 450, and 540 for the remaining parameters. VGG16 M-LIF SNNs were trained on CIFAR-10 for 200 epochs. During the first 10 epochs, the learning rate was increased linearly from 0.00001 to 0.0001 and subsequently divided by 5 at epochs 62 and 150. VGG16 M-LIF SNNs were trained on CIFAR-100 for 200 epochs with a fixed learning rate of 0.001 for firing thresholds and an initial learning rate of 0.0001 that is divided by 5 at epochs 62 and 150 for the remaining parameters. VGG16 M-LIF SNNs were also trained on ImageNet for 50 epochs. During the first 5 epochs, the learning rate was increased linearly from 0.00001 to 0.0001 and subsequently divided by 5 at epochs 30 and 40.

For spike-driven transformer, the proposed approach was evaluated on CIFAR-10, CIFAR-100, and ImageNet using Transformer-2-512 and Transformer-8-512 by replacing all LIF neurons with one-hot M-LIF neurons while using the same training methodology as in [121]. Note that Transformer-L-D represents a model with L encoder blocks and D channels. Transformer-L-D networks include spiking patch splitting followed by L spike-driven encoder blocks and a linear classification head. The encoder blocks consist of a spike-driven self attention layer followed by two multi-layer perceptron layers. D refers to the number of channels in the input to the encoder blocks. These networks were trained from scratch using BPTT without any pre-trained ANN initialization or batch norm fusion. The same hyperparameters as LIF spike-driven transformers in [121] were adopted when training their M-LIF counterparts for all experiments. A surrogate gradient based on the derivative of the sigmoid function with $\alpha = 4$ was used. In order to make use of the same hyperparameter configuration, the outputs of M-LIF neurons are scaled to be in the range $[0, 1]$ but continue to be powers of 2 (*i.e.,* $\in 1, \frac{1}{2}, \frac{1}{4}, \frac{1}{8}, ...$). Finally, similar to [121], a fixed threshold was used for all M-LIF neurons in spike-driven transformer experiments instead of learning the threshold as in experiments on other architectures.

Table 4.1: Iso-architecture comparison with SNNs for static image classification on CIFAR and ImageNet. * denotes self-implementation results.

Dataset	Architecture	Method	S	T	Accuracy (%)	δ	Comp Energy (μJ)
CIFAR-10	ResNet-20	ANN*	-	1	94.31	1.0	9.97E+02
		DIET-SNN [119]	1	5	91.78	6.3	1.58E+02
		Temporal Pruning [122]	1	1	91.10	16.32	6.11E+01
		1-hot M-LIF	3	1	**93.19**	**18.10**	**5.55E+01**
	VGG16	ANN*	-	1	94.43	1.0	1.56E+03
		DIET-SNN [119]	1	5	92.70	12.4	1.26E+02
		Temporal Pruning [122]	1	1	93.05	**33.0**	**4.72E+01**
		BANN [141]	1	1	**93.44**	25.08	6.22E+01
		1-hot M-LIF	3	1	93.34	29.73	5.24E+01
	Transformer-2-512	Spike-Driven Transformer [121]*	1	4	95.6	-	4.60E+02
		Spike-Driven Transformer [121]*	1	2	94.7	-	2.80E+02
		Spike-Driven Transformer [121]*	1	1	94.5	-	**1.92E+02**
		1-hot M-LIF	3	4	**95.9**	-	1.47E+03
		1-hot M-LIF	3	2	95.5	-	4.84E+02
		1-hot M-LIF	3	1	95.4	-	2.59E+02
CIFAR-100	ResNet-20	ANN*	-	1	67.10	1.0	9.97E+02
		DIET-SNN [119]	1	5	**64.07**	6.6	1.51E+02
		Temporal Pruning [122]	1	1	63.30	**15.35**	**6.50E+01**
		1-hot M-LIF	3	1	63.80	14.05	7.10E+01
	VGG16	ANN*	-	1	74.50	1.0	1.56E+03
		DIET-SNN [119]	1	5	69.97	12.1	1.29E+02
		Temporal Pruning [122]	1	1	70.15	**29.24**	**5.34E+01**
		1-hot M-LIF	3	1	**72.59**	23.63	6.60E+01
	Transformer-2-512	Spike-Driven Transformer [121]*	1	4	78.4	-	5.87E+02
		Spike-Driven Transformer [121]*	1	2	76.6	-	**1.90E+02**
		Spike-Driven Transformer [121]*	1	1	75.8	-	2.21E+02
		1-hot M-LIF	3	4	**78.9**	-	1.68E+03
		1-hot M-LIF	3	2	78.3	-	8.20E+02
		1-hot M-LIF	3	1	78.2	-	4.78E+02
ImageNet	VGG16	ANN*	-	1	72.56	1.0	7.12E+04
		DIET-SNN [119]	1	5	69.00	11.7	6.09E+03
		Temporal Pruning [122]	1	1	69.00	**24.61**	**2.89E+03**
		BANN [141]	1	1	68.00	20.94	3.40E+03
		1-hot M-LIF	3	1	**71.05**	20.20	3.37E+03
	Transformer-8-512	Spike-Driven Transformer [121]	1	1	71.68	-	**1.13E+03**
		Spike-Driven Transformer [121]	1	4	74.57	-	4.50E+03
		1-hot M-LIF	3	1	**75.33**	-	3.64E+03

4.5.1.2 Comparison with SNNs

Table 4.1 compares the accuracy and inference energy of one-hot M-LIF SNNs with iso-architecture conventional SNNs. While the proposed approach offers comparable or slightly lower energy benefits across most benchmarks, it consistently matches or exceeds conven-

tional SNNs in accuracy. The one-hot constraint ensures energy usage comparable to conventional SNNs in the worst-case (all neurons fire at each timestep) despite each M-LIF neuron having multiple spiking lanes, discovering new accuracy-energy tradeoff points. Prior work achieved 69% accuracy with a unit timestep on ImageNet using VGG16, while this work reached 71.05% with $S = 3$ spike lanes. For spike-driven transformers, M-LIF SNNs boost accuracy by up to 3% on ImageNet compared to LIF counterparts, consuming slightly more energy for a given T but achieving better tradeoffs. This is the case of ($T = 1$, $S = 3$) one-hot M-LIF spike-driven transformer, which achieves comparable or better accuracy to ($T = 4$) LIF on CIFAR-100 and ImageNet with $4\times$ less memory access energy (which can dominate overall energy as discussed in Sections 4.4 and 4.5.4) due to multi-timestep processing.

4.5.1.3 Comparison with LQ-ANNs

As discussed in Section 4.3.2, LQ-ANNs and unit timestep ($T = 1$) M-LIF SNNs remain distinct while both perform inference using a single timestep. Here, the accuracy and inference energy of b-bit LQ-ANNs and M-LIF-based SNNs using S spike lanes are contrasted in Table 4.2. M-LIF SNNs are observed to perform on par or better than LQ-ANNs in terms of accuracy and inference energy. For CIFAR-10, similar accuracy and energy benefits to LQ-ANNs are observed. On the other hand, for CIFAR-100, one-hot M-LIF SNNs are noted to be up to 54% more energy efficient than LQ-ANNs with comparable accuracy. Finally, the proposed approach scales much better on a large challenging dataset such as ImageNet yielding $> 3\%$ accuracy improvement. This gain can be primarily attributed to threshold parameter learning for SNNs.

4.5.1.4 Comparison with Quantized-Activation SNNs

As discussed in Section 4.2.5, prior works have explored the interplay between activation bit-width and timesteps in order to improve SNN accuracy. In this section, a quantitative comparison is provided in Table 4.3 with reported accuracies and energy estimates when

Table 4.2: Iso-architecture comparison with LQ-ANNs for static image classification on CIFAR and ImageNet. * denotes self implementation results.

Dataset	Architecture	Method	S	T	b	Accuracy (%)	δ	Comp Energy (μJ)
CIFAR-10	ResNet-20	LQ-ANN*	-	1	2	93.03	**21.60**	**4.61E+01**
			-	1	3	**93.59**	12.02	8.29E+01
		1-hot M-LIF	2	1	-	92.61	20.78	4.80E+01
			3	1	-	93.19	18.10	5.51E+01
CIFAR-100	VGG16	LQ-ANN*	-	1	2	71.24	25.28	6.17E+01
			-	1	3	**72.87**	15.29	1.02E+02
		1-hot M-LIF	2	1	-	71.32	**27.30**	**4.88E+01**
			3	1	-	72.59	23.63	5.66E+01
ImageNet	VGG16	LQ-ANN*	-	1	3	62.97	16.95	4.20E+03
			-	1	4	67.85	14.18	5.02E+03
		1-hot M-LIF	3	1	-	**71.05**	**20.20**	**3.37E+03**

available of burst-spike and multi-threshold methods on all evaluated static image classification datasets. Compared to burst-spike SNNs [139, 140], one-hot M-LIF SNNs significantly reduce the number of timesteps ($8-32\times$) while achieving better accuracy (up to $> 4\%$) on more challenging datasets such as ImageNet. Compared to multi-threshold SNNs [127, 128] one-hot M-LIF SNNs achieve better accuracy (up to $> 4\%$) while significantly reducing computational energy. This is due to the introduction of uniform activation quantization in multi-threshold SNNs which breaks the multiplication-free property of SNNs.

4.5.2 Dynamic Image Classification

4.5.2.1 Implementation Details

For the dynamic image classification task where SNN accuracy is generally superior than that of ANNs, temporal efficient training is applied similar to [120] using the proposed one-hot M-LIF neuron. Here, training was performed from scratch using BPTT without any pre-trained ANN initialization or batch normalization fusion. Experiments were conducted on DVS-CIFAR-10 [129] (converted from CIFAR-10) which is one of the most challenging mainstream dynamic vision datasets. It has 10000 images with size 128×128. Following

Table 4.3: Comparison with quantized-activation SNNs for static image classification on CIFAR and ImageNet.

Dataset	Method	Architecture	T	Accuracy (%)	Comp Energy (mJ)
CIFAR-10	MLF($K=3$) [128]	Spiking DS-ResNet	4	94.25	-
	Cascade-MT (4-bit) [127]	VGG9	1	95.17	-
	Burst-Spike+LIPooling [140]	VGG16	32	95.58	-
	1-hot M-LIF ($S=3$)	**Transformer-2-512**	**1**	**95.4**	**2.59E-01**
CIFAR-100	Parallel-MT (2-bit) [127]	VGG9	1	73.89	4.40E+02
	Cascade-MT (2-bit) [127]	VGG9	1	74.80	7.40E+02
	Burst-Spike+LIPooling [140]	VGG16	32	74.98	-
	1-hot M-LIF (S=3)	**Transformer-2-512**	**1**	**78.2**	**4.78E-01**
ImageNet	Parallel-MT (2-bit) [127]	ResNet-34	1	69.32	-
	Cascade-MT (4-bit) [127]	ResNet-34	1	72.17	-
	Burst-Spike ($\psi=4$) [139]	VGG16	8	70.61	-
	Burst-Spike+LIPooling [140]	VGG16	32	70.18	-
	1-hot M-LIF ($S=3$)	**Transformer-8-512**	**1**	**75.33**	**3.64E+00**

prior works, the spatial resolution was reduced to 48×48, and the dataset was split into 9000 training and 1000 test images [147]. Data augmentation techniques such as random horizontal flip and random roll within 5 pixels [148] were also applied. For all experiments, the VGGSNN architecture [120] was used with 300 epochs, the Adam optimizer with learning rate $\lambda = 0.001$, and a cosine annealing scheduler with 0 decay. All SNN surrogate gradient scaling used $\gamma = 1.0$ and firing thresholds and membrane leakages were fixed to 1.0 and 0.5, respectively, for all layers as per [120]. The VGGSNN architecture employed was taken from [120] and is described as follows: {64C3, 128C3 (/2), 256C3, 256C3 (/2), 512C3, 512C3 (/2), 512C3, 512C3 (/2), 10FC}.

4.5.2.2 Multi-level Input Layer Encoding

For DVS-CIFAR-10, direct input encoding is not applicable as the dataset consists of events recorded using a dynamic vision sensor. The adopted methodology described in [147] to prepare the data for SNN training is to split and convert the stream of T_E events into T_F binary frames as depicted in Figure 4.5 (top). In [120], a VGGSNN is trained with $T_F = 10$ and a top-1 accuracy of 83.17%. However, M-LIF SNNs are not limited to single spike lanes

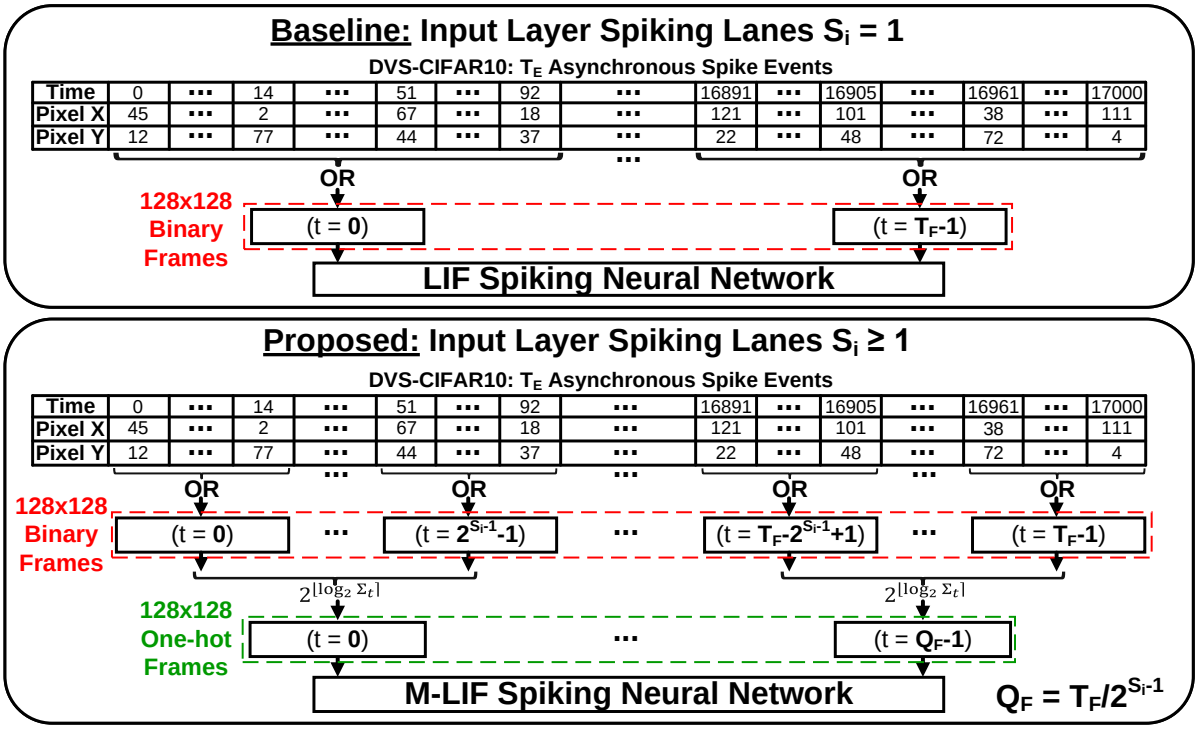


Figure 4.5: Workflow of multi-level input layer encoding for dynamic vision tasks.

at the input layer. Therefore, this work allows $T_F \neq 10$ and incorporates an additional data preparation step to perform multi-level input layer encoding as depicted at the bottom of Figure 4.5. After obtaining the T_F binary frames, every 2^{S_i-1} consecutive frames are combined into a one-hot frame resulting in $Q_F = T_F/2^{S_i-1}$ frames of one-hot S_i spike lanes. This enables M-LIF SNNs to limit accuracy degradation after reducing Q_F below the number of timesteps T_F. Figure 4.6 illustrates an example training image of a car in DVS-CIFAR-10 using the baseline input layer encoding with $T_F = 10$ binary frames and $S_i = 1$ spike lane (left) and the proposed multi-level input layer encoding with $Q_F = 5$ one-hot frames and $S_i = 3$ spike lanes (right).

4.5.2.3 Comparison with SNNs

Existing works on DVS-CIFAR-10 are compared against in Table 4.4. The compute energy is calculated using $E_{\text{SNN}} = \sum_{l=1}^{L} \#\text{SNN}_{\text{ops},l} \times 0.9$ pJ, where $\#\text{SNN}_{\text{ops},l}$ is defined in Section

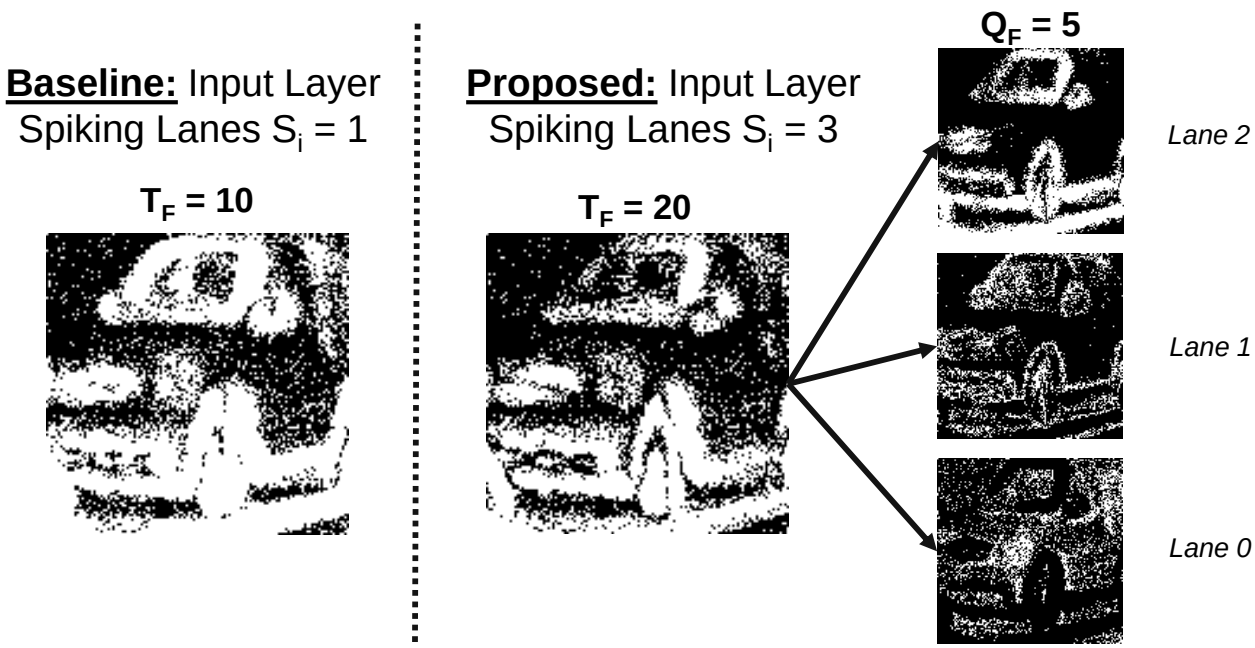


Figure 4.6: Example image in DVS-CIFAR-10 using baseline input layer encoding with $T_F = 10$ binary frames and $S_i = 1$ spike lane (left) and the proposed multi-level input layer encoding with $Q_F = 5$ one-hot frames and $S_i = 3$ spike lanes (right).

4.4 and 0.9 pJ is the energy of addition [43]. One-hot M-LIF VGGSNN achieves an accuracy of 84.7% using 10 timesteps and 4 spike lanes per neuron. This is also the first SNN work to achieve 82.5% accuracy on DVS-CIFAR-10 using 3 timesteps and 4 spike lanes compared to the best prior work [124] which can only achieve 82.6% using 4.5 timesteps and 77.6% using 2.5 timesteps. These improvements in accuracy stem primarily from introducing the S dimension. By reducing Q_F, not only does this work improve the computational energy by 3.45×, but also reduces membrane potential and spike memory access energy which scales linearly with timesteps and can be significantly higher than compute energy [122]. Table 4.4 also shows the impact of scaling Q_F and S on accuracy. By increasing S for a fixed Q_F, one-hot M-LIF SNNs are able to recover accuracy degradation unlike prior works which are limited by solely scaling T_F. By increasing S and Q_F, one-hot M-LIF neurons can scale the accuracy of SNNs to even higher than conventional SNNs.

Table 4.4: Comparison with prior works for dynamic image classification on DVS-CIFAR-10. † denotes data augmentation. $*$ denotes self-implementation results.

Method	Architecture	S	T_F	Q_F	Accuracy (%)	Comp Energy (μJ)
STBP-tdBN [149]	ResNet-19	1	10	-	67.8	-
Streaming Rollout [150]	DenseNet	1	10	-	66.8	-
Conv3D [151]	LIAF-Net	1	10	-	71.70	-
LIAF [151]	LIAF-Net	1	10	-	70.40	-
Dspike [152]	ResNet-18	1	10	-	75.4	-
RecDis-SNN [153]	ResNet-19	1	10	-	72.4	-
Spike-driven Transformer† [121]	Transformer-2-256	1	16	-	80.0	-
Burst-Spike ($\psi = 8$) [139]	ResNet-18	1	8	-	80.41	-
MLF ($K = 3$) [128]	Spiking DS-ResNet	1	10	-	70.36	-
PLIF [154]	VGGSNN	1	20	-	74.8	-
SEENN-II† [124]	VGGSNN	1	4.5	-	82.6	-
SEENN-I† [124]	VGGSNN	1	**2.5**	-	**77.6**	-
TET [120]	VGGSNN	1	10	-	77.3	-
TET*† [120]	VGGSNN	1	**10**	-	**83.1**	3.8E+02
TET*† [120]	VGGSNN	1	5	-	78.0	1.9E+02
TET*† [120]	VGGSNN	1	**3**	-	**74.7**	**1.2E+02**
1-hot M-LIF†	VGGSNN	4	-	**10**	**84.7**	**3.5E+02**
1-hot M-LIF†	VGGSNN	3	-	10	84.3	3.4E+02
1-hot M-LIF†	VGGSNN	4	-	5	83.3	1.8E+02
1-hot M-LIF†	VGGSNN	3	-	5	83.0	1.7E+02
1-hot M-LIF†	VGGSNN	4	-	**3**	**82.5**	**1.1E+02**
1-hot M-LIF†	VGGSNN	3	-	3	79.8	9.0E+01

4.5.3 Ablation Studies

An ablation study was conducted where the number of spike lanes S was varied while keeping the number of timesteps T fixed at 3 for VGGSNN on DVS-CIFAR-10. The results are provided in Table 4.5. The findings indicate that as the number of spike lanes increased from $S = 1$ to $S = 4$, there was a significant improvement in accuracy, after which performance saturates. Additionally, when the number of timesteps was increased to $T = 5$ in Table 4.5, a similar performance saturation around $S = 4$ timesteps was observed. The accuracy improvement due to spike lanes becomes less pronounced, but the overall accuracy ceiling improves. From these results, one can conclude that both timesteps and spike lanes contribute to better model accuracy with the caveat that multi-timestep processing introduces significant energy overhead, particularly with respect to memory access as noted in Section 4.4.

Table 4.5: Ablation study varying the number of spike lanes for one-hot M-LIF VGGSNN on DVS-CIFAR-10 with fixed timesteps $T = 3$ and $T = 5$.

S	Accuracy (%) for $T = 3$	Accuracy (%) for $T = 5$
1	74.7	78.0
2	78.6	81.5
3	79.8	83.0
4	82.5	83.3
5	82.5	82.9

4.5.4 Memory Access Energy Impact

As discussed in Section 4.4, memory accesses are indeed dependent on the hardware architecture. However, defining a precise memory architecture requires detailed assumptions regarding weight, input, and partial output reuse. Given that a hardware architecture study is outside the scope of this work, an analysis based on the memory energy model provided in [141, Appendix A.8.2] is included in this section. The energy model equations $SNN_{ME}^{T>1}$ and $SNN_{ME}^{T=1}$ for multi-timestep and unit-timestep SNNs, respectively, are given in Equations (4.19) and (4.20). C_l^i, C_l^o, k_l, H_l, W_l, $E_{\text{rd}}^{\text{wt}}$, $E_{\text{rd}}^{\text{inp}}$, $E_{\text{rd}}^{\text{sp}}$, $E_{\text{wr}}^{\text{sp}}$, $E_{\text{rd}}^{\text{mem}}$, $E_{\text{wr}}^{\text{mem}}$, and r_l^t denote the number of input channels, number of output channels, kernel size, feature map height, feature map width, read energy of weights from memory, read energy of full-precision first layer inputs from memory, read energy of spike inputs from memory, write energy of spike outputs to memory, read energy of full-precision membrane potentials from memory, write energy of full-precision membrane potentials to memory, and the activation sparsity of layer l at timestep t, respectively.

These equations were evaluated using the same 45nm CMOS technology for energy modeling [43], allowing for consistent energy comparisons, with $E_{\text{wr}}^{\text{mem}} = E_{\text{rd}}^{\text{mem}} = E_{\text{rd}}^{\text{wt}} = E_{\text{rd}}^{\text{inp}} = 50\text{pJ}$ and $E_{\text{rd}}^{\text{sp}} = 1.56\text{pJ}$. For one-hot M-LIF SNNs, input and output spikes are now replaced by power-of-two activations using $\log_2(S_o^l+1)$ bits, and read/write energies of spike inputs/outputs to/from memory are scaled by $\log_2(S_o^l + 1)$. Ablation studies in Section 4.5.3 show an accuracy performance saturation at around $S_o^l = 4$, which translates

to negligible 1 to 2 bit overhead considering the high activation sparsity and the much larger full-precision membrane potential read/write energies. Therefore, one-hot M-LIF SNNs do not introduce significant additional memory overhead compared to traditional SNNs using the same number of timesteps. Of course, memory overhead is not negligible in the case of multiple-timestep SNNs compared to ANNs (which effectively use a single timestep). In the unit-timestep case, one-hot M-LIF SNNs have the same number of memory accesses but benefit from smaller activation bit-widths.

$$\begin{aligned}\mathrm{SNN}_{\mathrm{ME}}^{T>1} = &\sum_{l=1}^{L} C_l^i C_l^o k_l^2 E_{\mathrm{rd}}^{\mathrm{wt}} + C_1^i C_1^o k_l^2 H_1 W_1 E_{\mathrm{rd}}^{\mathrm{inp}} \\ &+ \sum_{t=1}^{T} \left(\sum_{l=2}^{L} C_l^i C_l^o k_l^2 H_l W_l r_l^t E_{\mathrm{rd}}^{\mathrm{sp}} \right) \\ &+ \sum_{t=1}^{T} \left(\sum_{l=1}^{L} H_l W_l C_l^o (E_{\mathrm{wr}}^{\mathrm{sp}} + E_{\mathrm{rd}}^{\mathrm{mem}} + E_{\mathrm{wr}}^{\mathrm{mem}}) \right) \end{aligned} \tag{4.19}$$

$$\begin{aligned}\mathrm{SNN}_{\mathrm{ME}}^{T=1} = &\sum_{l=1}^{L} C_l^i C_l^o k_l^2 E_{\mathrm{rd}}^{\mathrm{wt}} + C_1^i C_1^o k_l^2 H_1 W_1 E_{\mathrm{rd}}^{\mathrm{inp}} \\ &+ \sum_{l=2}^{L} C_l^i C_l^o k_l^2 H_l W_l r_l^t E_{\mathrm{rd}}^{\mathrm{sp}} + \sum_{l=1}^{L} H_l W_l C_l^o E_{\mathrm{wr}}^{\mathrm{sp}} \end{aligned} \tag{4.20}$$

With this model, this work can estimate the impact of memory energy. However, the model factors in per layer activation sparsity, something which is not thoroughly provided for all prior works [122, 119, 141] due to lack of model checkpoints publicly available. A comparison for a model whose spike rates were provided across multiple prior works (VGG16 on CIFAR-10, CIFAR-100, and ImageNet; Transformer-8-512 on ImageNet) is included in this section. Self-implementation results using iso-accuracy one-hot M-LIF and traditional LIF spike-driven Transformer-2-512 on CIFAR-10 and CIFAR-100 were also employed.

Table 4.6 illustrates the impact of memory access energy on the overall energy comparison. For VGG16 on ImageNet, one-hot M-LIF SNN consumes 33× lower energy than ANNs, while maintaining higher accuracy (up to $> 3\%$) than prior unit-timestep SNN works. Multi-timestep processing introduces a noticeable energy overhead. As indicated in Equation

Table 4.6: Impact of memory access energy for static image classification. $*$ denotes self-implementation results.

Model	Dataset	Method	S	T	Accuracy (%)	Comp Energy (μJ)	Mem Energy (μJ)	Total Energy (μJ)
VGG16	CIFAR-10	ANN*	-	-	94.43	1.56E+03	1.89E+04	2.05E+04
		DIET-SNN [119]	1	5	92.7	1.26E+02	3.19E+03	3.32E+03
		Temporal Pruning [122]	1	1	93.05	4.72+01	2.15E+03	2.20E+03
		BANN [141]	1	1	93.44	6.22E+01	2.18E+03	2.24E+03
		1-hot M-LIF	**3**	**1**	**93.34**	**5.24E+01**	**2.18E+03**	**2.23E+03**
	CIFAR100	ANN*	-	-	74.5	1.56E+03	1.89E+04	2.05E+04
		DIET-SNN [119]	1	5	69.97	1.29E+02	3.01E+03	3.14E+03
		Temporal Pruning [122]	1	1	70.15	5.34E+01	2.16E+03	2.21E+03
		1-hot M-LIF	**3**	**1**	**72.59**	**6.60E+01**	**2.19E+03**	**2.26E+03**
	ImageNet	ANN*	-	-	72.56	7.12E+04	7.81E+05	8.52E+05
		DIET-SNN [119]	1	5	69.00	6.09E+03	5.88E+04	6.49E+04
		Temporal Pruning [122]	1	1	69.00	2.89E+03	1.55E+04	1.84E+04
		BANN [141]	1	1	68.00	3.40E+03	1.71E+04	2.05E+04
		1-hot M-LIF	**3**	**1**	**71.05**	**3.37E+03**	**2.21E+04**	**2.58E+04**
Transformer-2-512	CIFAR-10	Spike-Driven Transformer [121]*	1	4	95.6	4.60E+02	2.10E+03	2.56E+03
		1-hot M-LIF	**3**	**1**	**95.4**	**2.59E+02**	**1.47E+03**	**1.73E+03**
	CIFAR100	Spike-Driven Transformer [121]*	1	4	78.4	5.87E+02	2.90E+03	3.49E+03
		1-hot M-LIF	**3**	**1**	**78.2**	**4.78E+02**	**2.65E+03**	**3.13E+03**
Transformer-8-512	ImageNet	Spike-Driven Transformer [121]	1	4	74.57	4.50E+03	4.31E+04	4.76E+04
		1-hot M-LIF	**3**	**1**	**75.33**	**3.64E+03**	**1.69E+04**	**2.05E+04**

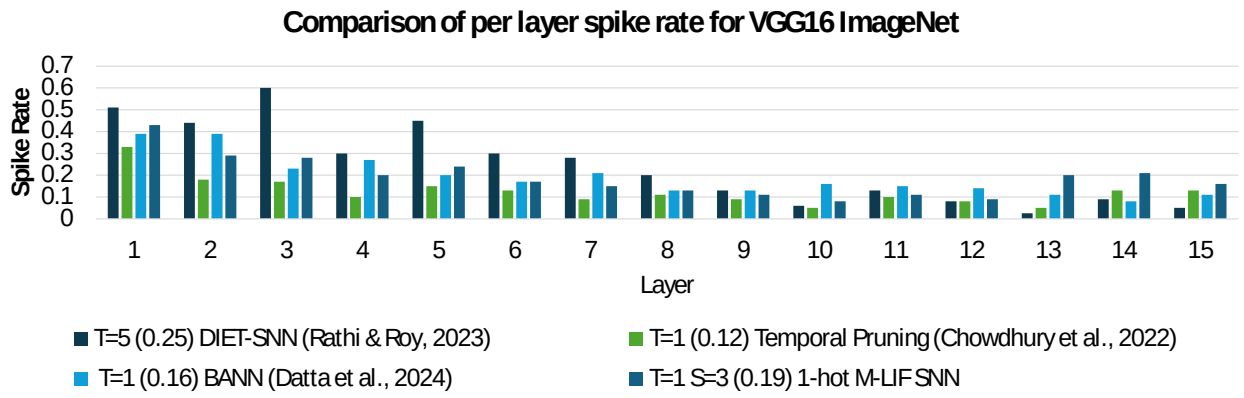


Figure 4.7: Comparison of per-layer spike rates for VGG16 on ImageNet.

(4.19), the model assumes a small number of timesteps is employed so as to reuse weights across all timestep calculations. As the number of timesteps increases, this assumption no longer holds as redundant weight accesses are performed. However, the memory access energy for membrane potentials and spikes scales linearly with the number of timesteps. For small feature map dimensions such as in CIFAR, the impact of this term is manageable. This observation matches the results shown in Table 4.6 for CIFAR. However, image dimensions in CIFAR (32×32) are not considered realistic for most real-world image classification tasks. Increasing the per layer feature map dimensions H_l and W_l to something more realistic as in the case of ImageNet (224×224), the memory access energy for membrane potentials and spikes starts to dominate the overall memory energy consumption for larger number of timesteps. In fact, for VGG16 on ImageNet, one-hot M-LIF SNNs achieve higher accuracy compared to ($T = 5$) [119] while consuming 2.5× less total energy, the majority of which stems from the memory energy overhead of multi-timestep ($T = 5$) membrane potential access. Figure 4.7 provides per layer spike rates for VGG16 ImageNet of the models compared in Table 4.6. Similarly, for Transformer-8-512 on ImageNet, one-hot M-LIF SNNs achieve higher accuracy compared to ($T = 4$) [121] while consuming 2.3× less total energy, the majority of which stems from the memory energy overhead of multi-timestep ($T = 4$) membrane potential access.

4.6 Future Directions

In this work, a novel algorithmic optimization to existing SNNs, namely the one-hot M-LIF neuron model, is introduced and evaluated. This approach distinguishes itself from other multi-threshold and burst-spike SNNs as well as unit-timestep SNNs and log-quantized ANNs, and this work demonstrates that one-hot M-LIF SNNs provide a better accuracy tradeoff in low-latency (unit-timestep) regimes on a wide range of network architectures for both static and dynamic image classification tasks. However, several areas remain unexplored.

Expanding experiments beyond image-based datasets (CIFAR-10, CIFAR-100, ImageNet, DVS-CIFAR-10) to include speech and text tasks, such as in[155, 156], would enhance generalizability. Robustness to noise and data distribution shifts is another critical aspect for real-world deployment. Integrating techniques from works solely focused on training robust SNNs such as[157, 158] into one-hot M-LIF SNN training could address this gap. Many prior works have also investigated the topic of hardware architectures and dataflow optimizations for existing SNN algorithms on ASIC or FPGA platforms [159, 160, 161, 162]. For example, [159] discusses the feasibility of leveraging systolic arrays to perform SNN inference using a small number of timesteps. The one-hot M-LIF neuron model was designed to be adaptable to many existing SNN hardware accelerator architectures. This work highlights the potential advantages of one-hot M-LIF SNNs over traditional SNNs through energy modeling. Future work could investigate the complete implementation and execution of one-hot M-LIF SNNs on ASIC or FPGA which would then further validate its hardware advantages. This aspect is also critical for practical deployment of one-hot M-LIF SNNs in low-power and real-time scenarios such as in edge computing, autonomous driving, and IoT systems.

4.7 Conclusion

SNNs hold promise as an energy-efficient alternative to traditional ANNs. However, achieving an optimal balance in the accuracy-energy tradeoff by adjusting latency remains a significant challenge for widespread deployment. To that end, this work introduces the dimension of spike lanes to conventional SNNs using a novel M-LIF neuron model without latency and computational complexity overhead. The proposed model represents the inputs and outputs of hidden layers as a set of one-hot binary-weighted spike lanes. Using the one-hot M-LIF neuron model, new and better tradeoff points can be found for both static and dynamic vision tasks. In particular, one-hot M-LIF-based SNNs achieve a top-1 accuracy of 71.05% on ImageNet using VGG16 and enhance the computational efficiency by 20×. One-hot M-LIF neurons also improve the accuracy-latency tradeoff for advanced network architectures such as spike-driven transformers ($> 3\%$ higher accuracy and 4× fewer timesteps on ImageNet). For dynamic vision tasks, such as image classification using dynamic vision sensor data, one-hot M-LIF SNNs retain higher accuracy (82.5%) when scaling down to fewer timesteps (3) on CIFAR-10-DVS thus providing better energy efficiency.

CHAPTER 5

Memory-Efficient Acceleration of Block Low-Rank Foundation Models on Resource Constrained GPUs

5.1 Introduction

Large-scale transformer-based foundation models have achieved remarkable success across language understanding, image classification, and generative tasks [14, 163, 4, 164, 165]. However, their rapid growth in size is increasingly outpacing the capacity of available hardware. For example, Llama-70B [163] requires over 140 GB of memory simply to load its weights in half-precision format, yet some of today's most powerful commercial GPUs provide only 80 GB. Beyond sheer memory constraints, the reliance of transformer models with dense matrix multiplications introduces significant computational and memory bandwidth bottlenecks during inference. These challenges limit the deployment of models at scale and also their accessibility on resource-constrained devices.

A widely adopted strategy to address these bottlenecks is to approximate weight matrices using low-rank factorizations [40, 166, 41]. By representing a dense weight matrix as the product of two smaller matrices, the computational and memory complexity of linear layers can be substantially reduced. However, traditional low-rank decompositions often exhibit sharp accuracy degradation at high compression ratios [167], which limits their prac-

ticality. To overcome this limitation, structured decompositions such as Monarch [39] and BLAST [38] have been proposed. They leverage block low-rank (BLR) structures to capture the underlying representation of weight matrices more effectively, thereby better preserving accuracy while offering memory savings and computational reduction.

Despite these algorithmic advances, the expected end-to-end speedups often fail to materialize in practice on GPUs. Performance on modern GPUs is governed by a roofline model [168, 169] that balances memory bandwidth, peak computational throughput, and arithmetic intensity (i.e., the ratio of operations to memory traffic). Figure 5.1 illustrates the roofline model of an NVIDIA A40 GPU for 16-bit brain floating-point (BF16) operations. While structured low-rank (LR) decompositions reduce the nominal compute requirements, this work first shows that they also introduce additional intermediate data movement, particularly in long-sequence scenarios such as the pre-fill stage of large language model (LLM) inference [170, 171]. This shift can move linear layers from the compute-bound regime into the memory-bound regime, creating a gap between algorithmic promise and system-level reality. The issue arises specifically for devices with limited memory subsystems, such as edge GPUs with small L2 caches (4–6 MB) and DRAM based on DDR technology (Jetson Orin Nano) as well as datacenter GPUs (A40). In such cases, (B)LR decompositions paradoxically degrade performance, despite reducing floating-point operations (FLOP) and model size.

In this work, the performance of LR, Monarch, and BLAST matrix multiplications for efficient transformer-based foundation model inference are analyzed, with a particular emphasis on long sequences. Key bottlenecks arising from data movement, suboptimal memory layouts, and compiler limitations are identified and characterized, and optimized implementations using Triton [71], an open-source intermediate language designed for writing efficient GPU kernels, are introduced. The proposed kernels exploit *partial fusion*, *operation reordering*, and *tailored memory layouts* to mitigate the overheads of (B)LR structured matrix multiplications in multi-token inference. Through extensive evaluation, this work demonstrates that these optimizations deliver substantial performance gains across diverse

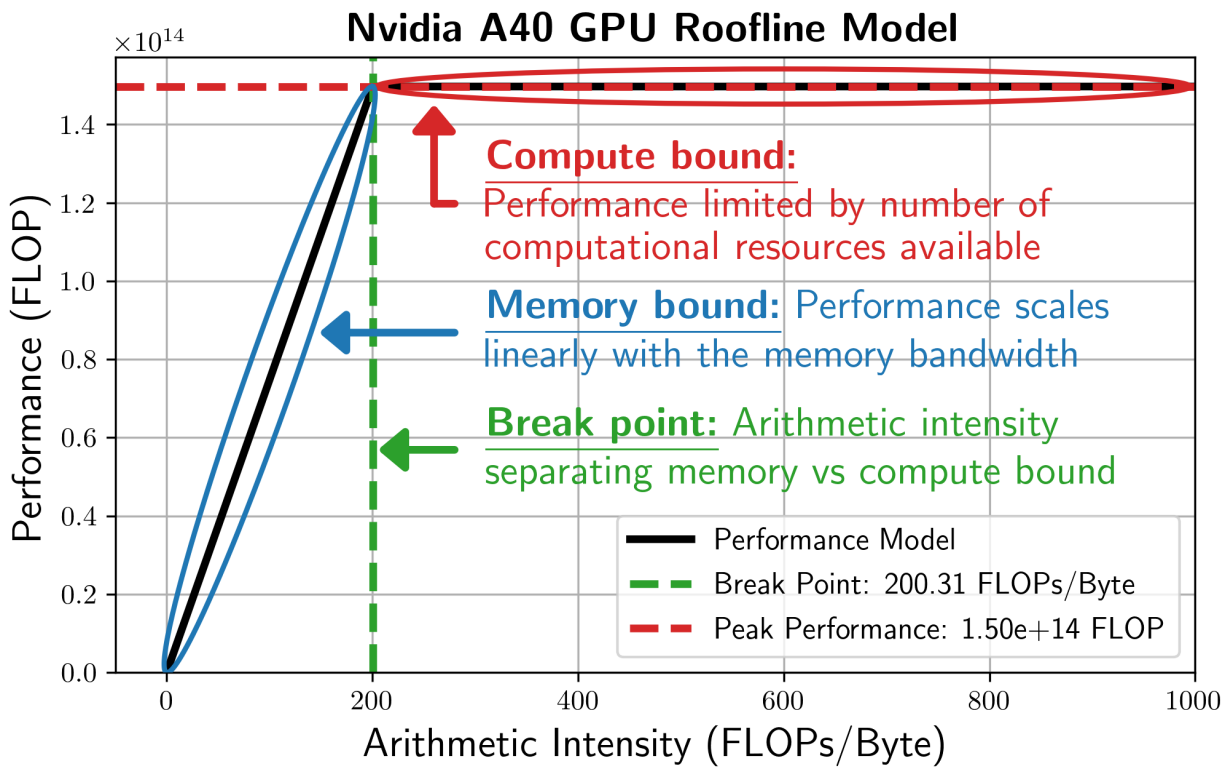


Figure 5.1: Roofline model of BF16 operations for NVIDIA A40 GPU.

transformer models, including GPT2-S, Llama-1/7B, and DiT-XL/2 on both server-grade and edge-class GPUs. The results establish that BLR-based model compression, when paired with hardware-aware optimizations, offers a viable path toward practical deployment of foundation models in resource-constrained environments. Overall, this work's contributions can be summarized as follows:

- The first systematic roofline analysis of BLR (Monarch, BLAST) matrix multiplications, showing that while they reduce FLOP, block structures introduce intermediate data movement and uncover PyTorch compiler limitations that push multi-token inference into the memory-bound regime compared to traditional low-rank and dense methods.

- Triton kernels designed with partial fusion, operation reordering, and tensor-core-friendly layouts that eliminate redundant data movement and restore efficiency to

BLR inference.

- Benchmarks on server and edge-grade GPUs, demonstrating up to 3.76× speedups and 3× model compression over PyTorch CUDA dense baselines with compiler optimizations, fostering reproducibility and rendering structured compression practical.

5.2 Background

5.2.1 Weight Structures

The weight matrix structures considered in this work are introduced, together with their computational properties and modeling capabilities. Let i, o, n, and r denote the number of input features, output features, sequence length, and rank, respectively. The input to all linear layers is $\boldsymbol{X} \in \mathbb{R}^{n \times i}$, and this work assumes $r \ll i, o$.

Dense A dense weight matrix $\boldsymbol{W} \in \mathbb{R}^{i \times o}$ has $i \times o$ parameters, and the corresponding linear layer $\boldsymbol{Y} = \boldsymbol{X}\boldsymbol{W}$ requires $n \times i \times o$ FLOP. Dense matrices can represent arbitrary linear maps and therefore provide the highest expressiveness for foundation models.

Low-Rank (LR) Dense weights can be factorized as $\boldsymbol{W} = \boldsymbol{V}\boldsymbol{U}$, where $\boldsymbol{V} \in \mathbb{R}^{i \times r}$ and $\boldsymbol{U} \in \mathbb{R}^{r \times o}$. Instead of materializing $\boldsymbol{W}$, the factorization is stored and used directly in computation. This reduces parameter count to $r(i+o)$ and computation to $nr(i+o)$ FLOP. The low-rank assumption restricts the transformation to a subspace of the input, which can cause accuracy degradation if the chosen rank r does not capture the true structure of $\boldsymbol{W}$. In practice, $r \ll i, o$ is selected such that $r(i+o) < io$, yielding both memory and computational savings [172, 173, 41].

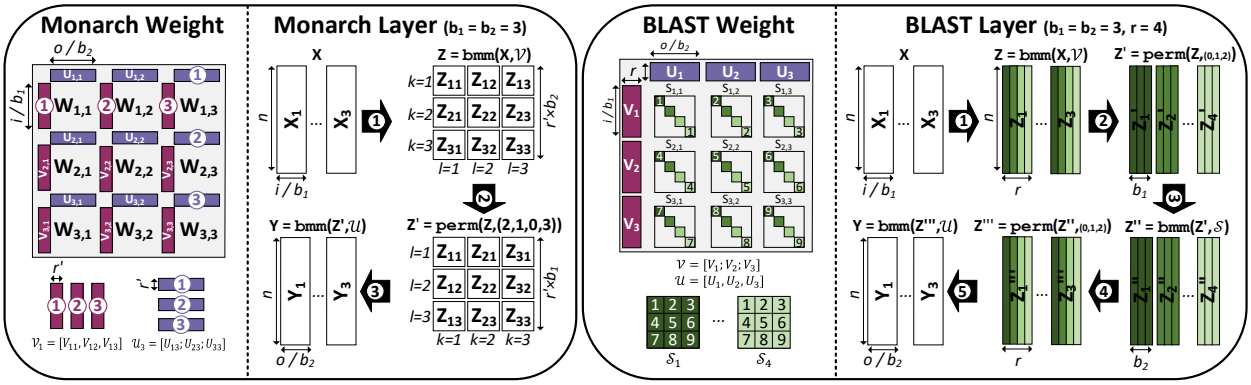


Figure 5.2: Monarch (left) and BLAST (right) weight parametrization and linear layer execution for $b_1 = b_2 = 3$ blocks and rank $r = 4$.

Monarch Introduced by [39], Monarch divides a dense weight into $b_2 \times b_1$ blocks, yielding a BLR representation[1] with uniform per-block rank r'. Each block $\boldsymbol{W}_{l,k}$ is factorized as

$$\boldsymbol{W}_{l,k} = \boldsymbol{V}_{l,k}\boldsymbol{U}_{l,k} \in \mathbb{R}^{p \times q}, \qquad l \in \{1, \ldots, b_1\},\ k \in \{1, \ldots, b_2\},\ p = i/b_1,\ q = o/b_2.$$

A Monarch layer has $b_1 b_2 r'(p+q)$ parameters. The k-th output block $\boldsymbol{Y}_k$ is computed as

$$\boldsymbol{Y}_k = \sum\nolimits_l \boldsymbol{X}_l \boldsymbol{W}_{l,k}, \qquad \boldsymbol{X}_l \in \mathbb{R}^{n \times p}, \qquad \boldsymbol{Y}_k \in \mathbb{R}^{n \times q},$$

which requires $nb_1 b_2 r'(p+q)$ FLOP in total.

Figure 5.2 (left) illustrates the execution of a Monarch layer for $b_1 = b_2 = 3$, where a permutation ($b_1 \Leftrightarrow b_2$) separates two batched matrix multiplications (`bmm`). In practice, the common setting $b_1 = b_2 = b$ with $r = r'b$ recovers the same complexity as low-rank layers, $r(i+o)$ parameters and $nr(i+o)$ FLOP. Monarch weights are stored as tensors : $\mathcal{V} \in \mathbb{R}^{b_1 \times (r' b_2) \times p}$ and $\mathcal{U} \in \mathbb{R}^{b_2 \times q \times (b_1 r')}$.

BLAST Introduced by [38], BLAST represents a weight matrix using a generalized BLR structure. Unlike Monarch, each block $\boldsymbol{W}_{l,k}$ shares a pair of matrices $\boldsymbol{V}_l$ and $\boldsymbol{U}_k$ while

[1] [39] introduces a "transposed" permutation at the output which is omitted here for simplicity.

retaining a unique diagonal matrix $\boldsymbol{S}_{l,k}$ such that the block factorization becomes

$$\boldsymbol{W}_{l,k} = \boldsymbol{V}_l \boldsymbol{S}_{l,k} \boldsymbol{U}_k, \qquad \boldsymbol{V}_l \in \mathbb{R}^{p \times r},\ \boldsymbol{S}_{l,k} \in \mathbb{R}^{r \times r},\ \boldsymbol{U}_k \in \mathbb{R}^{r \times q}.$$

This structure generalizes multiple families of structured low-rank matrices. In particular, low-rank and Monarch layers can be recovered by setting the values of $\boldsymbol{S}_{l,k}$ appropriately for all l and k.

A BLAST layer has $r(p+q+b_1 b_2)$ parameters. The k-th output block $\boldsymbol{Y}_k$ is computed as

$$\boldsymbol{Y}_k = \Big(\sum\nolimits_l (\boldsymbol{X}_l \boldsymbol{V}_l) \boldsymbol{S}_{l,k}\Big) \boldsymbol{U}_k, \qquad \boldsymbol{X}_l \in \mathbb{R}^{n \times p}, \qquad \boldsymbol{Y}_k \in \mathbb{R}^{n \times q},$$

which requires $nr(p+q+b_1 b_2)$ FLOP in total.

Typically, $b_1 = b_2 = b \leq 16$, which yields $r(i+o+b^2)$ parameters and $nr(i+o+b^2)$ FLOP. Since $b \ll i, o$, BLAST achieves the same asymptotic savings as low-rank and Monarch layers using a slightly higher compression ratio. Figure 5.2 (right) illustrates the execution of a BLAST layer for $b_1 = b_2 = 3$ and $r = 4$. In practice, BLAST parameters are stored as $\mathcal{V} \in \mathbb{R}^{b_1 \times p \times r}$, $\mathcal{S} \in \mathbb{R}^{b_1 \times b_2 \times r}$, and $\mathcal{U} \in \mathbb{R}^{b_2 \times r \times q}$.

Accuracy [38] evaluate the impact of replacing dense linear layers in foundation models with low-rank, Monarch, and BLAST layers. Results are reported on language and vision tasks using Llama-7/1B [163], GPT2-S [174], ViT-B [14], and DiT-XL/2 [165]. Language models are evaluated with WikiText-103/2 perplexity and zero-shot classification accuracy on common sense reasoning benchmarks [175, 176, 177, 178, 179, 180]. Vision models are evaluated on ImageNet [78] classification accuracy. Diffusion models are evaluated by generating images with a DDPM sampler [181] and computing FID, sFID, and IS against 50,000 ImageNet validation images (step size 250) to quantify generation quality. Table 5.1 summarizes the results where Monarch mostly improves upon low-rank, while BLAST achieves the best accuracy at the same compression factor (CF).

Method	Small		Medium					Large	
	ViT-B (CF = 3×)	GPT2-S (CF = 1.85×)	DiT-XL/2 (CF = 2×)			Llama-3.2-1B (CF = 2×)		Llama-7B (CF = 2×)	
	ImageNet Accuracy (%)	WikiText-103 Perplexity (↓)	FID (↓)	sFID (↓)	IS (↑)	WikiText-2 Perplexity (↓)	Avg. 0-shot Accuracy (%)	WikiText-2 Perplexity (↓)	Avg. 0-shot Accuracy (%)
Dense	78.7	20.2	9.62	6.85	121.50	11.57	56.54	9.37	66.07
BLAST	79.3	20.7	10.45	6.72	111.05	20.10	46.37	14.21	56.23
Monarch	79.2	21.1	-	-	-	22.17	44.35	19.54	49.78
Low-Rank	78.9	21.7	48.07	11.44	26.09	21.92	44.71	26.33	48.40

Table 5.1: Accuracy of foundation models using different (B)LR model compression factors (CF).

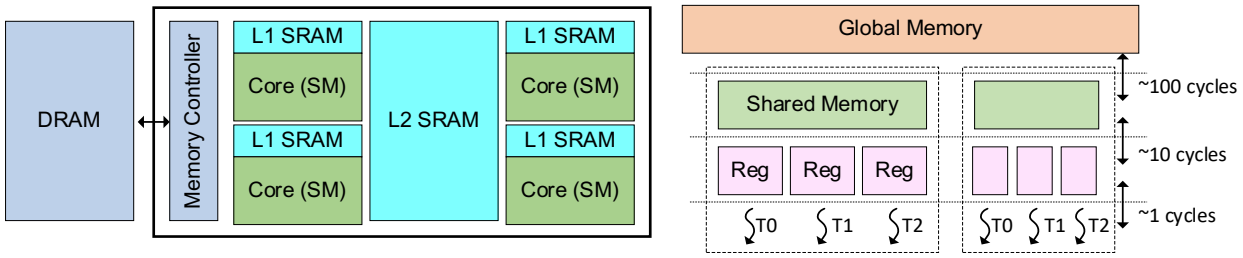


Figure 5.3: Basic GPU architecture (left) and execution model (right).

5.2.2 GPU Performance

This section reviews the fundamental aspects of GPU hardware and software.

Hardware Architecture Figure 5.3 (left) shows a simplified GPU architecture. GPUs integrate many CUDA cores for general-purpose parallelism and tensor cores specialized for matrix operations. The memory hierarchy spans high-capacity but high-latency off-chip DRAM and smaller on-chip caches (L1/L2) [182]. On resource-constrained devices such as the Jetson Orin Nano, the L2 cache is only a few MB [183], so the large activations of foundation models often spill to DRAM between kernels rather than being reused from cache. Even some data center GPUs like the A40 provide just a 6 MB shared L2 cache [184]. While L2 is not directly programmable, developers can leverage L1 and shared memory to improve locality and hide latency in software [185].

Execution Model and Programming Figure 5.3 (right) illustrates the GPU execution model. Computation is done by many parallel threads organized into thread blocks

and scheduled across streaming multiprocessors (SMs). Threads access a hierarchical memory system: high-latency global memory (DRAM), lower-latency but non-programmable L2 cache, explicitly managed shared memory (L1/SRAM) per thread block, and per-thread registers. A typical kernel loads data from global memory into registers or shared memory, performs computations, and writes results back. Kernels are usually written in CUDA, offering fine-grained control but requiring detailed knowledge of GPU architecture to achieve high performance [186]. For example, uncoalesced accesses reduce bandwidth efficiency, making coalesced loads a key optimization [187]. While NVIDIA's proprietary libraries often outperform custom CUDA kernels, recent alternatives like OpenAI's Triton [71] allow developers to write efficient GPU kernels in a Python-like syntax, with the compiler handling optimizations such as shared memory management, warp scheduling, and memory coalescing [188, 189].

Performance Characteristics GPU kernels are broadly categorized as compute-bound or memory-bound depending on their arithmetic intensity, α (operations per byte of memory accessed). The roofline model in Figure 5.1 captures this tradeoff, with the breakpoint $\tilde{\alpha}$ distinguishing compute-bound workloads ($\alpha \geq \tilde{\alpha}$) from memory-bound ones ($\alpha < \tilde{\alpha}$).

5.3 Profiling and Characterizing Bottlenecks

This work first conducts a case study on Llama-7B layers to examine how GPUs with limited L2 caches and memory bandwidths exhibit bottlenecks when executing inference with BLR approaches. Using empirical measurements and roofline modeling, this work identifies when and why these bottlenecks occur, providing insights that motivate this work's proposal for memory-efficient kernels. Separate discussions for single-token and multi-token inference are presented.

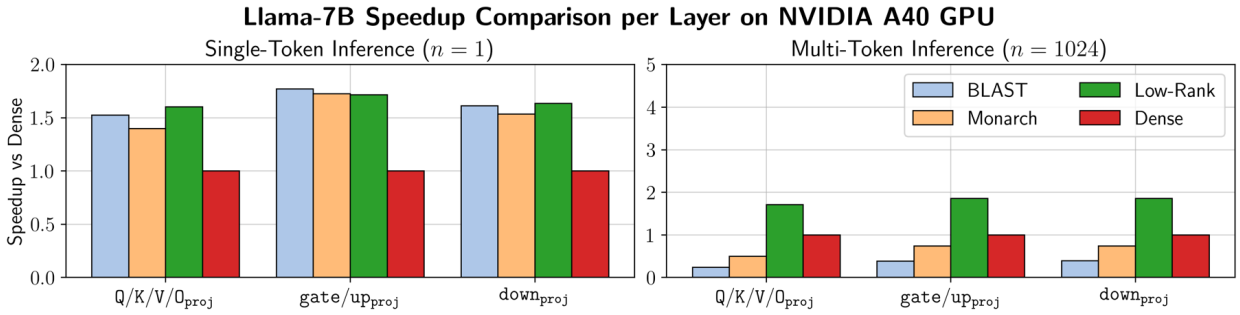


Figure 5.4: Performance of Llama-7B layers with low-rank methods versus dense, for single-token (left) and multi-token (right) inference on an NVIDIA A40 GPU.

5.3.1 Single-Token Inference

During single-token inference, typical in the decoding stage of LLMs where $n = 1$, the problem becomes memory-bound [190]. In this setting, memory traffic is dominated by weight movement rather than activations. As a result, compressing weights (e.g., by 2×) can nearly double throughput. This trend is evident in Llama-7B layers on the A40 GPU (Figure 5.4, left). Here, (B)LR methods such as BLAST, Monarch, and traditional low-rank all achieve similar performance compared to dense since the bottleneck lies in weight data movement rather than compute.

5.3.2 Multi-Token Inference

Unlike the single-token case, multi-token inference operates on larger input matrices where activation traffic grows with sequence length. This shift exposes a key weakness of (B)LR approaches: all of them generate intermediate outputs absent in the dense baseline. For traditional low-rank, the intermediate is an $n \times r$ matrix; for Monarch, it expands to $b \times n \times r$ with b as large as 16 in Llama-7B; and for BLAST, two such intermediates appear ($b_1 = b_2 = b$). Each of these tensors adds data movement, eroding the theoretical memory and compute savings. Blocked methods are hit especially hard, since the block dimension implies a `bmm`, and both Monarch and BLAST further require permutations on the innermost (contiguous) dimension, creating uncoalesced accesses and throttling memory bandwidth.

Method	FLOP	Memory (bytes)
Dense (D)	$2 \times nio$	$2 \times (ni + io + no)$
Low-Rank (LR)	$2 \times nr(i + o)$	$2 \times (ni + ir + ro + no + \underline{2nr})$
Monarch (M)	$2 \times nr(i + o)$	$2 \times (ni + ir + ro + no + \underline{4bnr})$
BLAST (B)	$2 \times nr(i + o + b^2)$	$2 \times (ni + ir + ro + rb^2 + no + \underline{8bnr})$

Table 5.2: FLOP and memory traffic for linear layers under different BF16 weight structures.

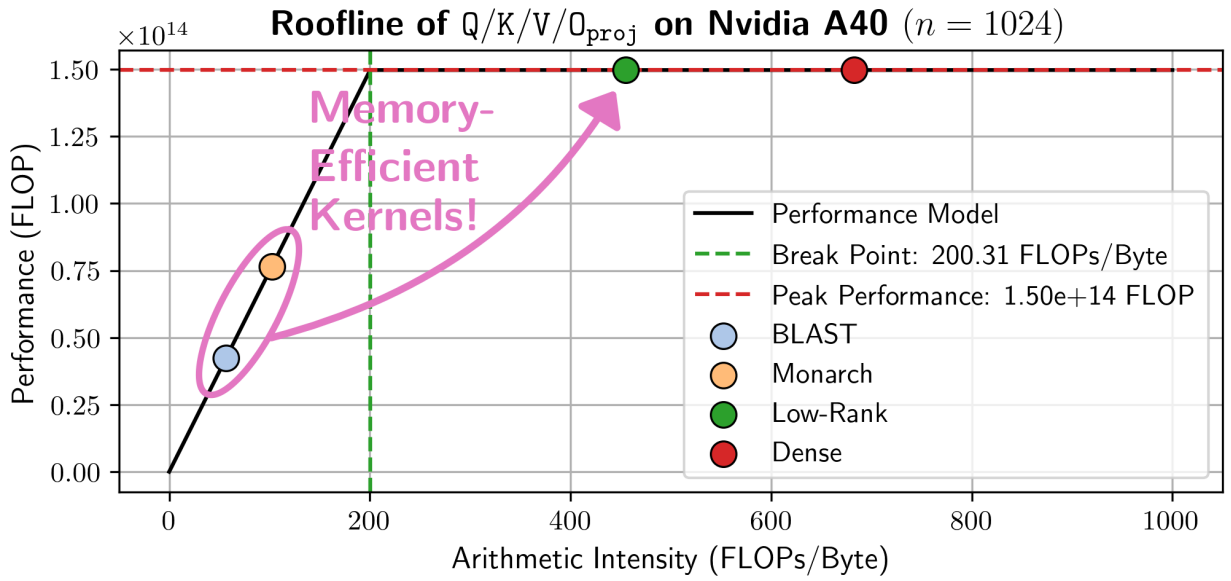


Figure 5.5: Roofline of Q/K/V/O$_{proj}$ during multi-token inference on A40.

Figure 5.4 (right) shows the resulting degradation. Traditional low-rank runs at 0.53–0.59× the runtime of dense, roughly consistent with its 2× compression. Monarch slows down, taking 1.14–1.68× longer than dense, while BLAST takes 2.63–4.31× longer.

To interpret these results, the FLOP and memory traffic for each method summarized in Table 5.2 were considered. The arithmetic intensity α = FLOP/Memory can then be computed directly from these values for a Q/K/V/O$_{proj}$ layer during multi-token inference ($b = 16$, $n = 1024$, $r = 1024$, $i = o = 4096$). Figure 5.5 overlays the resulting arithmetic intensity values on the roofline model. Dense (α_D) and traditional low-rank (α_{LR}) lie above the roofline breakpoint and are compute-bound, while Monarch (α_M) and BLAST (α_B) fall below it and are memory-bound, mirroring the empirical results. Therefore, multi-token

inference exposes a fundamental limitation: BLR methods are undermined by intermediate data movement and poor memory access patterns. To mitigate these issues, *fused* and *memory-efficient* kernels were proposed in Triton that avoid redundant memory trips and reorganize computation to exploit better layouts of intermediate tensors.

5.4 Memory-Efficient Kernels

Full Fusion Kernel fusion integrates consecutive kernels into one. For Monarch and BLAST, this work first explored fully fusing the `bmm` kernels, building on prior work for low-rank layers [191, 192]. In Triton, matrix multiplications are parallelized via 2-D output tiling, where each thread block iterates over the inner dimension and loads operand tiles into shared memory for tensor core computation via the `dot`() operator. In the context of full fusion, this tiling leads to redundant weight loads and recomputation of intermediates, often making fusion slower than launching separate kernels. Using 1-D tiles avoids redundancy but restricts rank and parallelism, yielding speedups only for very small ranks (e.g., ≤ 128) that correspond to extreme compression ratios (e.g., $\geq 8\times$ for $\texttt{Q}/\texttt{K}/\texttt{V}/\texttt{O}_{\texttt{proj}}$ in Llama-7B). Figure 5.6 illustrates these tradeoffs. The proposed Triton implementation of fully fused low-rank highlights the strong dependence of performance on r: for $r = 256$ (Figure 5.6, right), full fusion is consistently slower than dense across all feature dimensions, while for $r = 128$ (Figure 5.6, left), speedups appear but only for small output dimensions. As output dimension grows, the baseline low-rank implementation increasingly benefits from parallelism across the second dimension, which is sacrificed under 1-D tiling. Consequently, fully fused low-rank underperforms in these regimes.

Given these limitations, this work turns to *partial fusion*, where only permutations or subsets of `bmm` are fused to reduce memory traffic. This work next outlines kernel-specific optimizations for Monarch and BLAST, whose different parameterizations motivate distinct strategies.

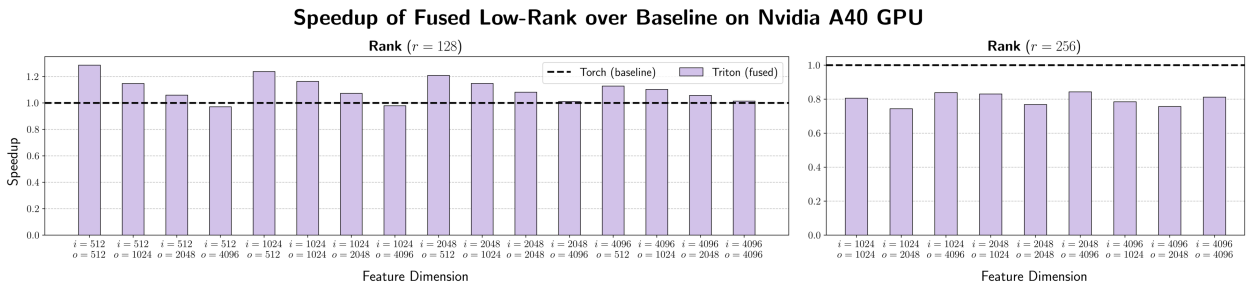


Figure 5.6: Speedup of Triton fused low-rank matrix multiplication over the PyTorch low-rank baseline on NVIDIA A40 GPU. Results are shown across different input/output feature dimensions for two fixed ranks: $r = 128$ (left) and $r = 256$ (right).

```
parfor b_1 in range (0, b_1):
  parfor n in range(0, n, t_n):
    parfor r in range(0, r, t_r):
      # Each thread block executes below
      b_2 = (r * t_r + [0:1:t_r - 1]) // r' (★)
      r' = (r * t_r + [0:1:t_r - 1]) % r' + b_1 * r' (★)
      acc = zeros((t_n, t_r))
      for p in range(0, p, t_p):
        x = X[b_1, n * t_n : (n + 1) * t_n, p * t_p : (p + 1) * t_p]
        v = V[b_1, p * t_p : (p + 1) * t_p, r * t_r: (r + 1) * t_r]
        acc += dot(x, v)
      Z'[n * t_n : (n + 1) * t_n, b_2 * n * r + r'] = acc (★)
```

Figure 5.7: Pseudo-code for fused permutation and `bmm` Monarch kernel (❷).

Monarch Optimizations ❶, ❷, and ❸ described below are intended to be employed together to provide additive efficiency benefits to Monarch linear layers during inference.

❶ *Re-layout of* $\mathcal{V}$. In the original code, $\mathcal{V}$ is stored as a $(b_1, r'b_2, p)$ tensor with the middle dimension contiguous along b_2 then r'. Meanwhile, $\mathcal{U}$ is stored as a $(b_2, q, b_1 r')$ tensor with the innermost dimension contiguous along r' then b_1. Multiplying the input batches with $\mathcal{V}$ after transposing its last two dimensions produces a $(b_1, n, r'b_2)$ tensor, after which two permutations become necessary: $r' \leftrightarrow b_2$ first, then $b_2 \leftrightarrow b_1$. In practice, this results in two separate kernel launches, each cloning the tensor into a different layout and incurring uncoalesced loads because it targets the innermost (contiguous) dimension. The first optimization is therefore to modify the memory layout of $\mathcal{V}$ so the middle dimension is

contiguous along r' first, then b_2. Since $\mathcal{V}$ is a static weight, this re-layout can be performed once before inference, eliminating the unnecessary $r' \leftrightarrow b_2$ permutation.

❷ *Permutation fusion.* After optimally re-laying out $\mathcal{V}$, the $b_2 \leftrightarrow b_1$ permutation is fused with the first `bmm` in a single Triton kernel. The fused kernel computes $b_1 \times t_n \times t_r$ output tiles, where t_n and t_r are the tile sizes along n and $r = r'b_2$, respectively. The permutation is implemented by first calculating the index `b2` (★, see Figure 5.7), then adjusting the innermost index `r'` by the offset of the corresponding block indexed at `b1` (★), and writing out the output using the swapped indices (★). These three steps are highlighted in the pseudo-code shown in Figure 5.7.

❸ *Avoiding the final permutation.* After computing the Monarch linear layer, a final permutation is applied to the output, transforming its shape from (b_2, n, q) to (n, q, b_2). Unlike the simpler stride change to (n, b_2, q), this transformation requires a full kernel launch. The additional permutation is unavoidable if the output of the Monarch linear layer is consumed by a residual connection or split into multiple heads. However, if the output is immediately multiplied by a static weight, one can pre-permute the rows of that weight offline and avoid running this kernel at inference time.

BLAST Optimizations ❹ and ❺ are applied separately as they represent distinct strategies to improve the efficiency of BLAST linear layers during inference.

❹ *Partial fusion of* `bmm`. This optimization eliminates both the intermediate permutation between $\mathcal{V}$ and $\mathcal{S}$ and the materialization of the first `bmm` output in global memory. Instead of assigning each thread block to a separate `b1` batch, the b_1 dimension is looped over within each thread block to compute an output tile of $\boldsymbol{Z}$ (★, see Figure 5.8). This restructuring is required because the second `bmm` reduces along b_1. If b_1 were distributed across multiple thread blocks, the threads would not be able to share the data needed for the reduction. Within each `b1` loop iteration, a (b_2, t_r) tile of $\mathcal{S}$, stored here as a (b_1, b_2, r) tensor, is loaded. It is reshaped to $(b_2, 1, t_r)$ and broadcast it with the $(1, t_n, t_r)$ output from the first `bmm`

```
parfor n in range(0, n, t_n):
  parfor r in range(0, r, t_r):
    # Each thread block executes below
    z''= zeros((b_2, t_n, t_r))
    for b_1 in range(0, b_1): (★)
      s = expand_dims(S[b_1, :, r * t_r: (r + 1) * t_r], 1) (★)
      z' = zeros((t_n, t_r))
      for p in range(0, i/b_1, t_p):
        x = X[b_1, n * t_n : (n + 1) * t_n, p * t_p: (p + 1) * t_p]
        v = V[b_1, p * t_p : (p + 1) * t_p, r * t_r : (r + 1) * t_r]
        z' += dot(x, v)
      z' = expand_dims(z', 0) (★)
      z'' += s * z' (★)
    Z''[:, n * t_n : (n + 1) * t_n, r * t_r : (r + 1) * t_r] = z'' (★)
```

Figure 5.8: Pseudo-code for BLAST partial fusion (❹).

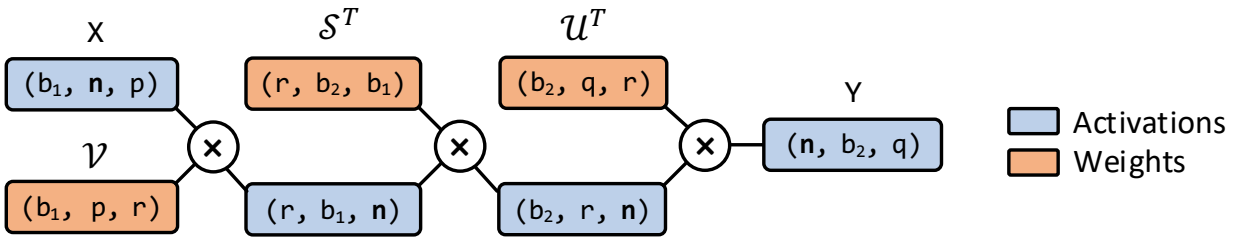


Figure 5.9: Compute diagram for BLAST permutation-only fusion (❺).

(★). This allows the second `bmm` to be expressed as an accumulated batched outer product across t_r (★). The results are accumulated in Z'' with shape (b_2, t_n, t_r), avoiding the large intermediate tensor required in the baseline (★). The trade-off however is that tensor cores cannot be used for the second `bmm`. The overall procedure is highlighted in Figure 5.8.

❺ *Permutation-only fusion with tensor core optimization.* The earlier strategy (❹) mapped the second `bmm` to CUDA cores rather than tensor cores, sacrificing up to 16× higher throughput [193]. This tradeoff could negate the FLOP savings from BLAST. To preserve tensor-core execution, one alternative is to eliminate only the costly permutations, but this is challenging because BLAST swaps the outer and innermost dimensions. Directly writing into the target layout of the next `bmm` leads to uncoalesced stores, as the batch dimen-

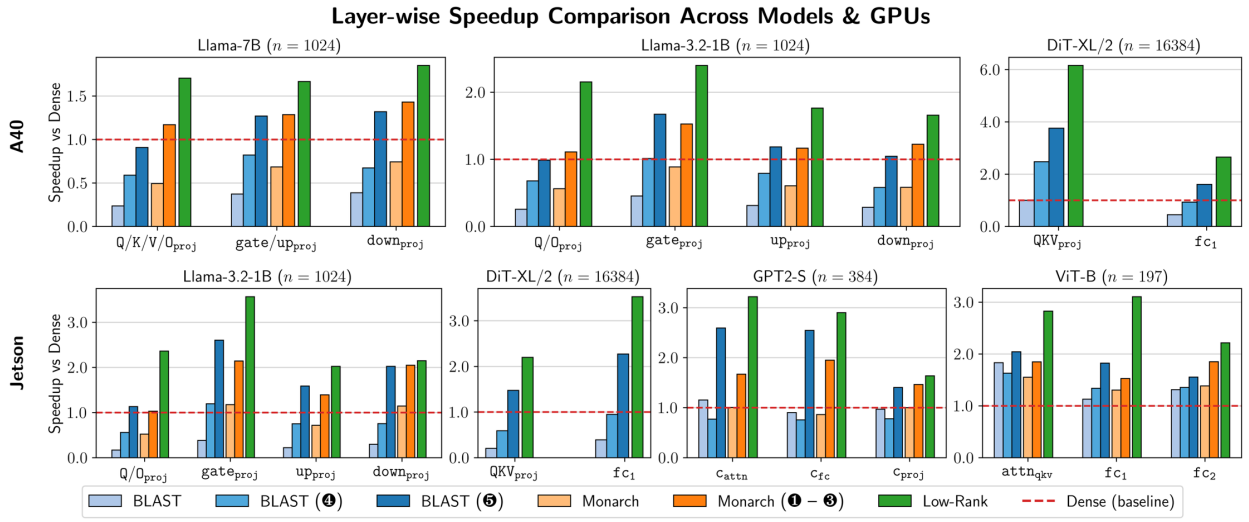


Figure 5.10: Layer-wise performance comparison across BLR methods and GPUs.

sion split across thread blocks cannot share data. This work's key insight is that transposing the `dot()` output before storing is inexpensive in Triton. This work therefore reorders the computation as shown in Figure 5.9. Instead of right-multiplying by $\mathcal{S}$ and $\mathcal{U}$, their first and last dimensions are transposed to obtain $\mathcal{S}^T$ and $\mathcal{U}^T$, multiply from the left, and transpose intermediate output tiles within each kernel. This keeps n contiguous, while r, b_1, and b_2 are successively exposed as the batch dimensions across three kernels, each implementing a transposed `bmm` with outer-dimension reordering. Reordering eliminates permutation overhead and maintains high tensor-core utilization via Triton's `dot()`, a level of efficiency that neither `einsum` nor PyTorch compiler-guided methods can match.

5.5 Experiments and Results

Evaluation Setup This work evaluated the proposed memory-efficient kernels against baseline implementations from the BLAST [38] and Monarch [39] repositories. Prior work focused on accuracy across a variety of language and vision domain tasks. As shown in Table 5.1, low-rank performs worst, Monarch improves upon low-rank, and BLAST achieves the highest accuracy with the same model compression factor (CF, ranging from 1.8 to 3×).

This work's evaluation complements these results with detailed performance benchmarking on the same set of models while additionally including Llama-3.2-1B for broader coverage. Details regarding Llama-3.2-1B training are provided in Section 5.5 with accuracies reported in Table 5.1.

Experiments were conducted on two hardware platforms. For mid to large-scale models (Llama-7B, DiT-XL/2, Llama-3.2-1B), this work uses an NVIDIA A40 with 40GB of memory. For mid to small-scale models (Llama-3.2-1B, DiT-XL/2, GPT2-S, ViT-B), this work evaluates on the Jetson Orin Nano with 8GB of memory. Note that Llama-7B does not fit on the Jetson device. All experiments use batch size of 1, with n determined by the model and application. Models are evaluated in BF16, and results are validated against original PyTorch implementations. Baselines leverage both Triton's auto-tuner and `torch.compile`() for fair comparison. For language models (GPT2-S, Llama), prefill throughput was reported, for diffusion models, inference at a single step was benchmarked, and for vision models, standard forward inference was measured.

Experiments were conducted using Python `3.12.8`, PyTorch `2.8.0`, Triton `3.4.0`, and CUDA `12.6.3` on the NVIDIA A40 GPU. For the Jetson Orin Nano 8GB, JetPack `6.2` with L4T `36.4.3`, CUDA `12.6.11`, PyTorch `2.6.0`, and Triton `3.2.0` were used. Latency of individual layers was measured with Triton's `do_bench`() utility, which executes the targeted layer multiple times under controlled conditions and reports averaged runtime. End-to-end model inference latency was measured using PyTorch's benchmarking utilities. To eliminate cold-start effects such as kernel compilation and cache population, several warm-up passes were first performed. During timing, inference ran under `torch.no_grad`() to disable gradient tracking, and `torch.cuda.synchronize`() was invoked to account for asynchronous CUDA execution. Measurements were collected with `torch.utils.benchmark.Timer`(), which repeatedly executes the forward pass for a specified number of iterations.

As illustrated by the pseudo-code in Figures 5.6 and 5.7, GPU kernels define tile sizes that partition the problem into parallelizable chunks. Tile sizes are typically chosen as powers of

two (commonly between 32 and 256) to align with hardware constraints. Beyond tile size, other hyperparameters such as the number of threads per block and the number of pipelining stages significantly affect kernel performance, memory requirements, and scheduling. Triton provides an `autotune` decorator that explores candidate configurations for these hyperparameters. The autotuner sweeps through different values, executes each configuration once to evaluate performance, and caches the best-performing settings for subsequent runs, conditioned on a sensitivity list of parameters. If any of these parameters change, the autotuner is re-run.

Detailed configurations for all layers that were replaced with (B)LR counterparts in the evaluated models are provided in Table 5.3. This includes the rank, number of blocks, input/output feature dimensions, and the number of occurrences of each layer type within the network. Note that for DiT-XL/2, $\texttt{adaLN}_{\texttt{proj}}$ was replaced with a (B)LR counterpart to compress the model, but it was not included in the layer-wise benchmarking results reported in Section 4.5, as it processes a single token rather than the 16K tokens used in other layers.

Llama-3.2-1B Compression and Re-training This work employed the Llama-3.2 [194] model with 1.24B parameters as one of the representative medium-sized models. The model was compressed by 50% using low-rank, Monarch, and BLAST weight parameterizations. Specifically,the query and output projection weights in the attention modules were replaced, as well as the up projection, gate projection, and down projection weights in the feed-forward network modules.

The rank and number of blocks used in each experiment are reported in Table 5.3. For low-rank compression, the weights were factorized via singular value decomposition (SVD). For Monarch compression, block-wise SVD was applied, where each block had rank $r' = \frac{r}{b}$. Following [38], BLAST compression was obtained by applying 300 steps of preconditioned gradient descent to factorize the weights into BLAST factors. All three methods reduced the model size to 0.6B parameters.

Model	Layer	Input (i)	Output (o)	Indices	Method	(r, b)
Llama-7B	Q/K/V/O$_{\text{proj}}$	4096	4096	0 – 31	Low-Rank	(1024, –)
					Monarch	(1024, 16)
					BLAST	(1024, 16)
	gate/up$_{\text{proj}}$	4096	11008	0 – 31	Low-Rank	(1488, –)
					Monarch	(1536, 16)
					BLAST	(1488, 16)
	down$_{\text{proj}}$	11008	4096	0 – 31	Low-Rank	(1488, –)
					Monarch	(1536, 16)
					BLAST	(1488, 16)
Llama-3.2-1B	Q/O$_{\text{proj}}$	2048	2048	0 – 31	Low-Rank	(256, –)
					Monarch	(256, 16)
					BLAST	(256, 16)
	gate$_{\text{proj}}$	2048	8192	0 – 31	Low-Rank	(512, –)
					Monarch	(512, 16)
					BLAST	(512, 16)
	up$_{\text{proj}}$	2048	8192	0 – 31	Low-Rank	(768, –)
					Monarch	(768, 16)
					BLAST	(768, 16)
	down$_{\text{proj}}$	8192	2048	0 – 31	Low-Rank	(768, –)
					Monarch	(768, 16)
					BLAST	(768, 16)
GPT2-S	c$_{\text{attn}}$	768	2304	0 – 11	Low-Rank	(192, –)
					Monarch	(192, 4)
					BLAST	(192, 6)
	c$_{\text{fc}}$	768	3072	0 – 11	Low-Rank	(192, –)
					Monarch	(192, 4)
					BLAST	(192, 6)
	c$_{\text{proj}}$	3072	768	0 – 11	Low-Rank	(192, –)
					Monarch	(192, 4)
					BLAST	(192, 6)
ViT-B	attn$_{\text{qkv}}$	768	2304	0 – 11	Low-Rank	(128, –)
					Monarch	(128, 4)
					BLAST	(128, 3)
	fc$_1$	768	3072	0 – 11	Low-Rank	(128, –)
					Monarch	(128, 4)
					BLAST	(128, 3)
	fc$_2$	3072	768	0 – 11	Low-Rank	(128, –)
					Monarch	(128, 4)
					BLAST	(128, 3)
DiT-XL/2	QKV$_{\text{proj}}$	1152	3456	0 – 27	Low-Rank	(384, –)
					BLAST	(384, 9)
	fc$_1$	1152	4608	0 – 27	Low-Rank	(256, –)
					BLAST	(256, 9)
	adaLN$_{\text{proj}}$	1152	6912	0 – 27	Low-Rank	(256, –)
					BLAST	(256, 9)

Table 5.3: Layer configurations for dense layers replaced by low-rank, Monarch, and BLAST counterparts across evaluated models.

As in [38], the compressed models were re-trained. Specifically, the compressed weights were fine-tuned for 4000 steps on a subset[2] of the SlimPajama dataset [195], using a learning rate of 8×10^{-4} with linear decay scheduling.

[2] https://huggingface.co/datasets/DKYoon/SlimPajama-6B

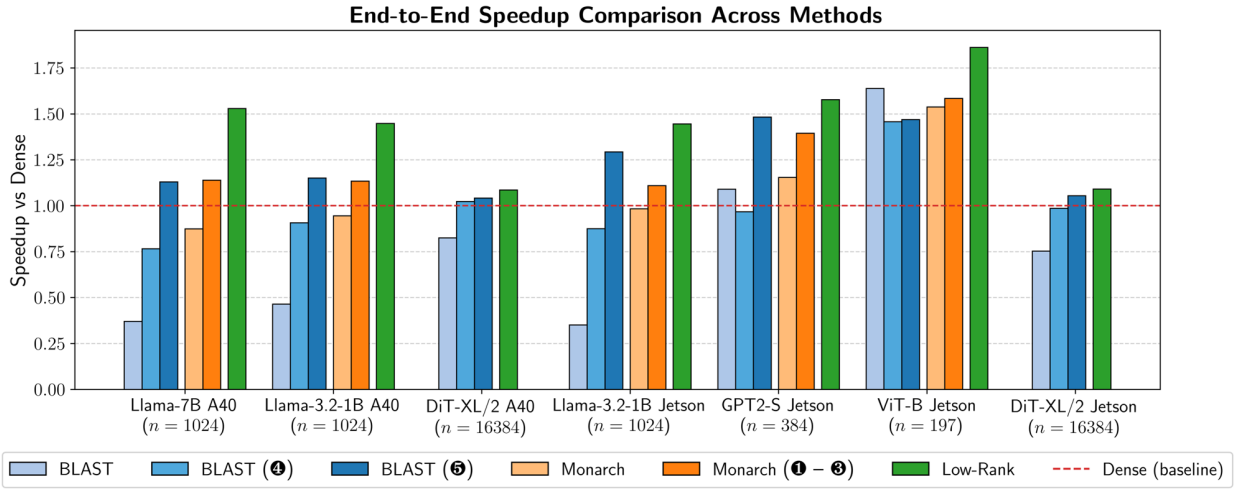


Figure 5.11: End-to-end inference performance across models and platforms.

Layer-wise Breakdown Figure 5.10 summarizes layer-wise speedups. The optimized BLAST kernel (❺) consistently outperforms both the BLAST and Monarch baselines across all architectures. Notably, ❺ delivers up to 7.15× speedup over its baseline for the $\texttt{QKV}_{\texttt{proj}}$ layer of DiT-XL/2 on Jetson, and up to 2.95× over Monarch for the $\texttt{c}_{\texttt{fc}}$ layer of GPT2-S on Jetson. The optimized Monarch kernel (❶ – ❸) also provides meaningful gains, achieving 1.46–2.37× speedups across layers relative to its baseline. Since BLAST also achieves higher accuracy than Monarch overall, ❺ represents the best balance of accuracy and efficiency. Most importantly, ❺ outperforms dense by 1.13-3.76× in $> 90\%$ of cases where its baseline falls short, proving competitive with highly tuned dense kernels as well as other BLR baselines. In addition, BLAST kernels employing optimization ❹ are consistently worse than those employing ❺, and in some cases worse than baseline BLAST, such as GPT2-S on Jetson, because the second `bmm` in ❹ is mapped as batched outer product running on CUDA cores while ❺ leverages tensor cores for this `bmm`.

End-to-End Comparison Figure 5.11 reports end-to-end inference results with dense linear layers replaced by BLAST, Monarch, or low-rank layers, using either baseline or optimized BLR implementations. To minimize CPU overhead and improve scheduling effi-

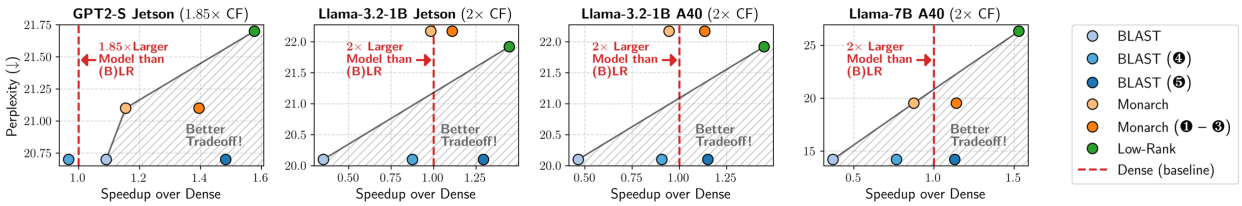


Figure 5.12: Tradeoff between perplexity and speedup over dense compared across methods.

ciency, this work applied `torch.compile()` to the entire network in each case, enabling CUDA graph execution. Since BLR linear layers account for only part of the total runtime, overall speedups are naturally smaller than layer-wise gains, especially in long-sequence workloads such as DiT-XL/2 with 16K tokens where attention dominates.

Nevertheless, BLAST-based models using ❺ achieve substantial acceleration over models that use its baseline and the Monarch baseline. Relative to the BLAST baseline, ❺ provides 3.05× speedup on Llama-7B/A40, 2.48× on Llama-3.2-1B/A40, 3.68× on Llama-3.2-1B/Jetson, and 1.36× on GPT2-S/Jetson end-to-end. Even in settings where attention dominates like DiT-XL/2 [196], ❺ reduces overall inference time by 1.4× on the Jetson compared to the BLAST baseline. Some cases such as ViT-B show a different trend where most of the gain comes from applying `torch.compile()` to the entire model, which already yields speedups over the dense baseline comparable to those of low-rank. In this setting, ❺ does not provide additional benefit and can even regress performance, while ❶ – ❸ offer little improvement. The configuration itself (with only b=3 blocks for BLAST, b=4 for Monarch, and rank 128) is small and particularly well-suited for `torch.compile()` to produce an optimized baseline.

Finally, ❺ not only outperforms dense across all tested models by up to 1.48× but also approaches the speed of low-rank while maintaining higher accuracy. This establishes it as the most effective option when, for instance, end-to-end gains from ❶ – ❸ are comparable. This is made evident in Figure 5.12 where both ❺ and ❶ – ❸ provide a better tradeoff than low-rank and BLR baselines for language models between perplexity and speedup over dense.

5.6 Conclusion

This work shows that while prior studies of BLR foundation models focused on modeling accuracy and single-token inference performance, their speedup benefits often vanish in multi-token settings, especially on resource-constrained GPUs. Through a detailed roofline analysis and memory-efficient Triton kernels, this work bridges the gap between reduced FLOP and realized speedups using techniques such as partial fusion, operation re-ordering, and optimized memory layouts. This in turn enables practical deployment of BLR-compressed foundation models at a level that current PyTorch compiler-guided implementations cannot achieve.

CHAPTER 6

Conclusion

6.1 Summary of Contributions

This dissertation presents research aimed at bridging the gap between algorithmic innovation and practical deployment by tightly coupling neural network design with the capabilities of the underlying hardware. As deep learning models continue to grow in complexity, traditional hardware often fails to fully exploit the benefits of algorithmic optimizations. In many cases, realizing these benefits requires not only model simplification but also careful mapping of those optimizations to the hardware's architectural strengths to avoid performance bottlenecks. Through a co-design approach, this thesis demonstrates how aligning techniques such as sparsity, compression, and quantization with both specialized accelerators and general-purpose platforms leads to significant gains in metrics such as energy efficiency and throughput.

Chapter 2 introduces SONA, a novel energy-efficient accelerator optimized for HTNNs. It employs an HTNN-specific output-stationary dataflow and optimized memory organization to reduce transform-domain computation overhead. SONA also integrates a CBB sparsity pattern for FCLs, enabling efficient support for both sparse transform-domain convolutions and FCLs while maintaining high hardware utilization. The BS-CSD-MAC unit further boosts efficiency by replacing multiplications with bit-shifts and additions. Through simulator-based evaluations on CIFAR-10/100 and ImageNet, SONA demonstrated substan-

tial improvements in energy efficiency and performance over prior dense and sparse CNN accelerators by up to 5.1× and 2.4× for convolution layers compared to prior dense and sparse CNN accelerators, respectively, while improving performance by 5.2× and 2.1×. For sparse FCLs, SONA achieves a 2.4× reduction in inference energy and a 2.1× performance gain. These results highlight the potential of HTNNs when coupled with specialized hardware, paving the way for future research in energy-efficient deep learning accelerators.

Chapter 3 presents the first silicon implementation of a structured-sparse HTNN processor designed to accelerate both convolutional and fully connected layers while operating entirely on-chip. By leveraging a dense, retentive MRAM for weight storage, the processor eliminates external memory access, reducing data transfer overhead and improving energy efficiency. However, the inherent active and leakage energy consumption of MRAM posed a significant challenge, which was mitigated through a layer-wise dynamic power gating mechanism and an optimized weight caching strategy. Additionally, the slow read speed of MRAM introduced a 3.5× increase in memory access latency, accounting for up to 43% of total execution time. To address this, the processor incorporated a double-buffered weight cache with pipelined prefetching and decoupled MRAM/compute clocks, effectively hiding memory access latency. These optimizations enabled a 4.9× increase in throughput, reaching a peak performance of 195 GOPS. Fabricated in a 22nm CMOS process, the proposed HTNN processor demonstrated a 40% reduction in energy consumption and achieved a peak efficiency of 9.51 TOPS/W for convolutional layers and 22.86 TOPS/W for fully connected layers. These results highlight the feasibility of deploying structured-sparse HTNNs in energy-constrained environments and open new possibilities for efficient neural network acceleration in edge applications.

Chapter 4 introduces a novel approach to improving the accuracy-energy tradeoff in SNNs through the one-hot multi-level leaky integrate-and-fire neuron model. Unlike conventional LIF models, the proposed M-LIF model uses strictly power-of-two spike values while maintaining the multiplication-free nature of SNNs. This method allows SNNs to achieve higher

accuracy with fewer timesteps, reducing energy consumption while preserving efficiency. Experimental results demonstrate that one-hot M-LIF SNNs outperform standard LIF SNNs on static datasets like CIFAR and ImageNet while maintaining comparable or better energy efficiency. Furthermore, on the DVS-CIFAR10 dynamic vision dataset, the proposed model achieves state-of-the-art accuracy of 82.5% using only 3 timesteps, marking a significant reduction in computational cost. These findings highlight the potential of one-hot M-LIF neurons to advance energy-efficient and accurate SNN deployment in edge applications.

Chapter 5 explores the efficiency of block low-rank (BLR) foundation model inference. In this chapter, roofline analysis was employed to show that, although BLR methods achieve theoretical savings and practical speedups for single-token inference, multi-token inference often becomes memory-bound in practice, increasing latency despite compiler-level optimizations in PyTorch. To address this, custom Triton kernels were introduced with partial fusion and memory layout optimizations for both Monarch and BLAST. On memory-constrained NVIDIA GPUs such as Jetson Orin Nano and A40, the proposed kernels deliver up to 3.76× speedups and 3× model size compression over PyTorch dense baselines using CUDA backend and compiler-level optimizations, while supporting various models including Llama-7/1B, GPT2-S, DiT-XL/2, and ViT-B, paving the way for practical deployment of structured compression techniques in large-scale AI systems.

6.2 Future Work

The research presented in Chapters 2 and 3 introduced an accelerator architecture and its silicon implementation for heterogeneous transform-domain neural networks (HTNNs). While conventional convolutional neural networks such as VGG and ResNet were adapted to the HTNN framework by replacing 3×3 stride-1 convolutions with 4×4 stride-2 convolutions, future work could further expand the applicability of HTNNs. In particular, extending the heterogeneous sparse-orthogonal WHT-domain convolution operation to additional convo-

lution types such as pointwise and depthwise separable convolutions would enable the use of HTNNs in popular edge models like MobileNetV1/V2 and EfficientViT. Although the current SONA architecture supports pointwise and depthwise separable convolutions, it is not yet optimized for MobileNet-style workloads. For instance, when applied to 3×3 depthwise separable convolutions in MobileNetV1/V2 [79, 80], HTNN can theoretically reduce the number of MAC operations by approximately 48% and 56%, respectively, for those layers, as sparse orthogonality between kernels is not utilized and the benefit of HTNN arises solely from the element-wise multiplications in a single WHT domain. Beyond convolution operation extensions, another promising direction is dual-sided sparsity. The current co-design primarily exploits structured sparsity on the weight side, leaving activation sparsity largely untapped. Exploring activation-aware transformations and hardware support for sparse activations within the HTNN framework could unlock multiplicative efficiency gains, similar to what has been observed in other sparse accelerators, but tailored to heterogeneous transform-domain computation.

In Chapter 4, the one-hot multi-level leaky integrate-and-fire (M-LIF) neuron demonstrated that power-of-two quantization of activations can reduce computational complexity without sacrificing accuracy. Future research could extend this approach by integrating quantization-aware training (QAT) with reduced weight precision. Such joint optimization of activation and weight quantization may further enhance energy efficiency, though the resulting accuracy–efficiency trade-off warrants careful study. Beyond algorithmic optimization, expanding experiments beyond image-based datasets (CIFAR-10, CIFAR-100, ImageNet, DVS-CIFAR10) to include speech and text tasks—such as those explored in [155, 156]—would strengthen the generality of the one-hot M-LIF model. Another important direction is improving robustness to noise and data distribution shifts, which are critical for real-world deployment. Integrating techniques from robustness-focused SNN training methods [157, 158] into the one-hot M-LIF framework could enhance stability and performance under varying input conditions. Finally, validating the hardware implications of one-hot M-

LIF neurons remains a key step toward deployment. Prior studies have explored hardware architectures and dataflow optimizations for SNN inference on ASIC and FPGA platforms [159, 160, 161, 162]. Realizing one-hot M-LIF SNNs on ASIC or FPGA platforms would provide concrete validation of their energy advantages and enable their practical use in low-power, real-time applications such as edge computing, autonomous driving, and IoT systems.

For block low-rank (BLR) foundation model inference (Chapter 5), the optimized BLAST and Monarch kernels still lag behind traditional low-rank in most multi-token inference scenarios due to intermediate activation overheads introduced by their blocked structure. This limitation could be addressed through intermediate activation quantization or mixed-precision tensor core utilization, provided that accuracy is maintained. Recent developments in activation quantization [197, 198, 199], both co-designed during training and through post-training calibration, offer promising pathways for such integration. Additionally, extending the scope of re-training beyond the limited 400–4000 steps used in current experiments to longer fine-tuning schedules (e.g., $> 10^6$ steps) could help close the remaining accuracy gap with dense baselines for billion-parameter models.

Looking ahead, the principles developed in this dissertation can be extended to emerging AI architectures and pipelines such as state-space models (e.g., Mamba [200]) and retrieval-augmented generation (RAG) systems [201]. Mamba's causal convolution introduces new computational bottlenecks, while RAG expands inference complexity through vector-based retrieval and longer contextual windows. As these workloads evolve, adapting co-design strategies such as structured compression, transform-domain processing, and mixed-precision inference to these new computation and memory patterns will be essential. Ultimately, future work should continue to unify algorithmic efficiency and hardware specialization to enable scalable, energy-efficient AI across next-generation architectures.

BIBLIOGRAPHY

[1] Waseem Rawat and Zenghui Wang. Deep convolutional neural networks for image classification: A comprehensive review. *Neural Computation*, 29(9):2352–2449, 2017.

[2] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016.

[3] A Vaswani. Attention is all you need. *Advances in Neural Information Processing Systems*, 2017.

[4] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[5] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1–9, 2015.

[6] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, pages 448–456. pmlr, 2015.

[7] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. *arXiv preprint arXiv:1512.00567*, 2015.

[8] Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, and Alexander Alemi. Inception-v4, inception-resnet and the impact of residual connections on learning. In *Proceedings of the AAAI conference on artificial intelligence*, volume 31, 2017.

[9] 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 (CVPR)*, June 2016.

[10] Saining Xie, Ross B. Girshick, Piotr Dollár, Zhuowen Tu, and Kaiming He. Aggregated residual transformations for deep neural networks. In *Proceedings of the IEEE/CVF*

Conference on Computer Vision and Pattern Recognition (CVPR), pages 1492–1500. IEEE, 2017.

[11] Irwan Bello, William Fedus, Xianzhi Du, Ekin D. Cubuk, Aravind Srinivas, Tsung-Yi Lin, Jonathon Shlens, and Barret Zoph. Revisiting resnets: Improved training and scaling strategies. *arXiv preprint arXiv:2103.07579*, 2021.

[12] Mingxing Tan and Quoc Le. Efficientnet: Rethinking model scaling for convolutional neural networks. In *International Conference on Machine Learning*, pages 6105–6114. PMLR, 2019.

[13] Mingxing Tan and Quoc V. Le. Efficientnetv2: Smaller models and faster training. *arXiv preprint arXiv:2104.00298*, 2021.

[14] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale. *International Conference on Learning Representations (ICLR)*, 2021.

[15] Ze Liu, Yutong Lin, Yue Cao, Han Hu, Yixuan Wei, Zheng Zhang, Stephen Lin, and Baining Guo. Swin transformer: Hierarchical vision transformer using shifted windows. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 10012–10022, 2021.

[16] Hugo Touvron, Matthieu Cord, Matthijs Douze, Francisco Massa, Alexandre Sablayrolles, and Hervé Jégou. Training data-efficient image transformers & distillation through attention. *arXiv preprint arXiv:2012.12877*, 2020.

[17] Li Yuan, Yunpeng Chen, Tao Wang, Weihao Yu, Yujun Shi, Zihang Jiang, Francis EH Tay, Jiashi Feng, and Shuicheng Yan. Tokens-to-token vit: Training vision transformers from scratch on imagenet. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 558–567, 2021.

[18] Haiping Wu, Bin Xiao, Noel Codella, Mengchen Liu, Xiyang Dai, Lu Yuan, and Lei Zhang. Cvt: Introducing convolutions to vision transformers. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 22–31. IEEE, 2021.

[19] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. *OpenAI Blog*, 1(8):9, 2019.

[20] Martin Andrews and Sam Witteveen. Unsupervised natural question answering with a small model. *arXiv preprint arXiv:1911.08340*, 2019.

[21] NVIDIA. Nvidia ampere architecture whitepaper. https://www.nvidia.com/en-us/data-center/ampere-architecture/, 2020.

[22] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, et al. Pytorch distributed: Experiences on accelerating data parallel training. *arXiv preprint arXiv:2006.15704*, 2020.

[23] Song Han, Huizi Mao, and William J. Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding, 2016.

[24] Tien-Ju Yang, Yu-Hsin Chen, and Vivienne Sze. Designing energy-efficient convolutional neural networks using energy-aware pruning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, July 2017.

[25] Wei Wen, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. Learning structured sparsity in deep neural networks. In D. Lee, M. Sugiyama, U. Luxburg, I. Guyon, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 29. Curran Associates, Inc., 2016.

[26] Andrew Lavin and Scott Gray. Fast algorithms for convolutional neural networks. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 4013–4021, 2016.

[27] Michael Mathieu, Mikael Henaff, and Yann LeCun. Fast training of convolutional networks through ffts. *arXiv preprint arXiv:1312.5851*, 2013.

[28] Vinay Verma, Nikita Agarwal, and Nitin Khanna. Dct-domain deep convolutional neural networks for multiple jpeg compression classification. *Signal Processing: Image Communication*, 67:22–33, 2018.

[29] Alan V. Oppenheim, Alan S. Willsky, and S. Hamid Nawab. *Signals and Systems*. Prentice Hall, 2 edition, 1996.

[30] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cudnn: Efficient primitives for deep learning. *CoRR*, abs/1410.0759, 2014.

[31] Tianyun Zhang, Shaokai Ye, Kaiqi Zhang, Jian Tang, Wujie Wen, Makan Fardad, and Yanzhi Wang. A systematic dnn weight pruning framework using alternating direction method of multipliers. In *Proceedings of the European Conference on Computer Vision (ECCV)*, September 2018.

[32] 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. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, page 27–40, New York, NY, USA, 2017. Association for Computing Machinery.

[33] Xuda Zhou, Zidong Du, Qi Guo, Shaoli Liu, Chengsi Liu, Chao Wang, Xuehai Zhou, Ling Li, Tianshi Chen, and Yunji Chen. Cambricon-s: Addressing irregularity in

sparse neural networks through a cooperative software/hardware approach. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 15–28, 2018.

[34] Ashish Gondimalla, Noah Chesnut, Mithuna Thottethodi, and T. N. Vijaykumar. Sparten: A sparse tensor accelerator for convolutional neural networks. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, page 151–165, New York, NY, USA, 2019. Association for Computing Machinery.

[35] Baoyuan Liu, Min Wang, Hassan Foroosh, Marshall Tappen, and Marianna Pensky. Sparse convolutional neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2015.

[36] Yu Chen, Bowen Liu, Pierre Abillama, and Hun-Seok Kim. Htnn: Deep learning in heterogeneous transform domains with sparse-orthogonal weights. In *2021 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, pages 1–6, 2021.

[37] Zhi-Gang Liu, Paul N. Whatmough, Yuhao Zhu, and Matthew Mattina. S2ta: Exploiting structured sparsity for energy-efficient mobile cnn acceleration. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 573–586, 2022.

[38] Changwoo Lee, Soo Min Kwon, Qing Qu, and Hun-Seok Kim. Blast: Block-level adaptive structured matrices for efficient deep neural network inference. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang, editors, *Advances in Neural Information Processing Systems*, volume 37, pages 14996–15027. Curran Associates, Inc., 2024.

[39] Tri Dao, Beidi Chen, Nimit S Sohoni, Arjun Desai, Michael Poli, Jessica Grogan, Alexander Liu, Aniruddh Rao, Atri Rudra, and Christopher Re. Monarch: Expressive structured matrices for efficient and accurate training. In Kamalika Chaudhuri, Stefanie Jegelka, Le Song, Csaba Szepesvari, Gang Niu, and Sivan Sabato, editors, *Proceedings of the 39th International Conference on Machine Learning*, volume 162 of *Proceedings of Machine Learning Research*, pages 4690–4721. PMLR, 17–23 Jul 2022.

[40] Minyoung Huh, Hossein Mobahi, Richard Zhang, Brian Cheung, Pulkit Agrawal, and Phillip Isola. The low-rank simplicity bias in deep networks. *arXiv preprint arXiv:2103.10427*, 2021.

[41] Soo Min Kwon, Zekai Zhang, Dogyoon Song, Laura Balzano, and Qing Qu. Efficient low-dimensional compression of overparameterized models. In Sanjoy Dasgupta, Stephan Mandt, and Yingzhen Li, editors, *Proceedings of The 27th International Conference on Artificial Intelligence and Statistics*, volume 238 of *Proceedings of Machine Learning Research*, pages 1009–1017. PMLR, 02–04 May 2024.

[42] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in neural information processing systems*, 35:16344–16359, 2022.

[43] Mark Horowitz. 1.1 computing's energy problem (and what we can do about it). In *2014 IEEE international solid-state circuits conference digest of technical papers (ISSCC)*, pages 10–14. IEEE, 2014.

[44] Shuchang Zhou, Yuxin Wu, Zekun Ni, Xinyu Zhou, He Wen, and Yuheng Zou. Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients. *arXiv preprint arXiv:1606.06160*, 2016.

[45] Penghang Yin, Jiancheng Lyu, Shuai Zhang, Stanley Osher, Yingyong Qi, and Jack Xin. Understanding straight-through estimator in training activation quantized neural nets. *arXiv preprint arXiv:1903.05662*, 2019.

[46] Eunhyeok Park, Sungjoo Yoo, and Peter Vajda. Value-aware quantization for training and inference of neural networks. In *Proceedings of the European Conference on Computer Vision (ECCV)*, September 2018.

[47] Haoran You, Xiaohan Chen, Yongan Zhang, Chaojian Li, Sicheng Li, Zihao Liu, Zhangyang Wang, and Yingyan Lin. Shiftaddnet: A hardware-inspired deep network. *Advances in Neural Information Processing Systems*, 33:2771–2783, 2020.

[48] Edward H Lee, Daisuke Miyashita, Elaina Chai, Boris Murmann, and S Simon Wong. Lognet: Energy-efficient neural networks using logarithmic computation. In *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 5900–5904. IEEE, 2017.

[49] Xinlin Li, Bang Liu, Rui Heng Yang, Vanessa Courville, Chao Xing, and Vahid Partovi Nia. Denseshift: Towards accurate and efficient low-bit power-of-two quantization. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 17010–17020, October 2023.

[50] Dominika Przewlocka-Rus, Syed Shakib Sarwar, H. Ekin Sumbul, Yuecheng Li, and Barbara De Salvo. Power-of-two quantization for low bitwidth and hardware compliant neural networks, 2022.

[51] Jorge Albericio, Alberto Delmás, Patrick Judd, Sayeh Sharify, Gerard O'Leary, Roman Genov, and Andreas Moshovos. Bit-pragmatic deep neural network computing. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-50 '17, page 382–394, New York, NY, USA, 2017. Association for Computing Machinery.

[52] Alberto Delmas Lascorz, Patrick Judd, Dylan Malone Stuart, Zissis Poulos, Mostafa Mahmoud, Sayeh Sharify, Milos Nikolic, Kevin Siu, and Andreas Moshovos. Bit-tactical: A software/hardware approach to exploiting value and bit sparsity in neural

networks. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 749–763, New York, NY, USA, 2019. Association for Computing Machinery.

[53] Hyochan An, Yu Chen, Zichen Fan, Qirui Zhang, Pierre Abillama, Hun-Seok Kim, David Blaauw, and Dennis Sylvester. 29.3 an 8.09 tops/w neural engine leveraging bit-sparsified sign-magnitude multiplications and dual adder trees. In *2023 IEEE International Solid-State Circuits Conference (ISSCC)*, pages 422–424. IEEE, 2023.

[54] Wolfgang Maass. Networks of spiking neurons: the third generation of neural network models. *Neural networks*, 10(9):1659–1671, 1997.

[55] Pierre Abillama, Changwoo Lee, Andrea Bejarano-Carbo, Qirui Zhang, Dennis Sylvester, David Blaauw, and Hun-Seok Kim. One-hot multi-level leaky integrate-and-fire spiking neural networks for enhanced accuracy-latency tradeoff. *IEEE Access*, 13:38163–38180, 2025.

[56] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C.J. Burges, L. Bottou, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012.

[57] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[58] Bing Han, Abhronil Sengupta, and Kaushik Roy. On the energy benefits of spiking deep neural networks: A case study. In *2016 International Joint Conference on Neural Networks (IJCNN)*, pages 971–976, 2016.

[59] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, page 1–12, New York, NY, USA, 2017. Association for Computing Machinery.

[60] NVIDIA. Nvidia deep learning accelerator (nvdla) open source project. http://nvdla.org, 2018.

[61] Google. Google edge tpu. https://cloud.google.com/edge-tpu/, 2020.

[62] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 367–379, 2016.

[63] Zichen Fan, Qirui Zhang, Hyochan An, Boxun Xu, Li Xu, Chien-Wei Tseng, Yimai Peng, Andrea Bejarano-Carbo, Pierre Abillama, Ang Cao, Bowen Liu, Changwoo Lee, Zhehong Wang, Hun-Seok Kim, David Blaauw, and Dennis Sylvester. Aimmi: Audio and image multi-modal intelligence via a low-power soc with 2-mbyte on-chip mram for iot devices. *IEEE Journal of Solid-State Circuits*, 59(10):3488–3501, 2024.

[64] Qirui Zhang, Zichen Fan, Hyochan An, Zhehong Wang, Ziyun Li, Guanru Wang, Pierre Abillama, Hun-Seok Kim, David Blaauw, and Dennis Sylvester. Robovisio: A micro-robot vision domain-specific soc for autonomous navigation enabling fully-on-chip intelligence via 2-mb emram. *IEEE Journal of Solid-State Circuits*, 2024.

[65] Pierre Abillama, Qirui Zhang, Zichen Fan, Hyochan An, Kuan-Yu Chen, Yu Chen, David Blaauw, Dennis Sylvester, and Hun-Seok Kim. A 22nm 9.51 tops/w neural engine with 2mb mram leveraging sparse-orthogonal walsh-hadamard transform computations and dynamic power gating. In *2024 IEEE European Solid-State Electronics Research Conference (ESSERC)*, pages 25–28, 2024.

[66] Kartik Prabhu, Robert M Radway, Jeffrey Yu, Kai Bartolone, Massimo Giordano, Fabian Peddinghaus, Yonatan Urman, Win-San Khwa, Yu-Der Chih, Meng-Fan Chang, et al. Minotaur: A posit-based 0.42–0.50-tops/w edge transformer inference and training accelerator. *IEEE Journal of Solid-State Circuits*, 2025.

[67] Clifford E Cummings. Simulation and synthesis techniques for asynchronous fifo design. In *SNUG 2002 (Synopsys Users Group Conference, San Jose, CA, 2002) User Papers*, volume 281. Citeseer, 2002.

[68] Hailin Jiang, M. Marek-Sadowska, and S.R. Nassif. Benefits and costs of power-gating technique. In *2005 International Conference on Computer Design*, pages 559–566, 2005.

[69] Takashi Ohsawa, Hiroki Koike, Sadahiko Miura, Hiroaki Honjo, Keizo Kinoshita, Shoji Ikeda, Takahiro Hanyu, Hideo Ohno, and Tetsuo Endoh. A 1 mb nonvolatile embedded memory using 4t2mtj cell with 32 b fine-grained power gating scheme. *IEEE Journal of Solid-State Circuits*, 48(6):1511–1520, 2013.

[70] Qing Wu, M. Pedram, and Xunwei Wu. Clock-gating and its application to low power design of sequential circuits. *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*, 47(3):415–420, 2000.

[71] Philippe Tillet, H. T. Kung, and David Cox. Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, MAPL 2019, page 10–19, New York, NY, USA, 2019. Association for Computing Machinery.

[72] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A. Horowitz, and William J. Dally. Eie: Efficient inference engine on compressed deep neural network. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 243–254, 2016.

[73] Shijin Zhang, Zidong Du, Lei Zhang, Huiying Lan, Shaoli Liu, Ling Li, Qi Guo, Tianshi Chen, and Yunji Chen. Cambricon-x: An accelerator for sparse neural networks. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12, 2016.

[74] Sheng R. Li, Jongsoo Park, and Ping Tak Peter Tang. Enabling sparse winograd convolution by native pruning. *CoRR*, abs/1702.08597, 2017.

[75] Xingyu Liu, Jeff Pool, Song Han, and William J. Dally. Efficient sparse-winograd convolutional neural networks. In *International Conference on Learning Representations*, 2018.

[76] William K Pratt, Julius Kane, and Harry C Andrews. Hadamard transform image coding. *Proceedings of the IEEE*, 57(1):58–68, 1969.

[77] Alex Krizhevsky and Geoffrey Hinton. Learning multiple layers of features from tiny images. Technical report, University of Toronto, 2009.

[78] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pages 248–255, 2009.

[79] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *CoRR*, abs/1704.04861, 2017.

[80] Mark Sandler, Andrew G. Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Inverted residuals and linear bottlenecks: Mobile networks for classification, detection and segmentation. *CoRR*, abs/1801.04381, 2018.

[81] Reid M Hewlitt and ES Swartzlantler. Canonical signed digit representation for fir digital filters. In *2000 IEEE Workshop on SiGNAL PROCESSING SYSTEMS. SiPS 2000. Design and Implementation (Cat. No. 00TH8528)*, pages 416–426. IEEE, 2000.

[82] Subhankar Pal, Jonathan Beaumont, Dong-Hyeon Park, Aporva Amarnath, Siying Feng, Chaitali Chakrabarti, Hun-Seok Kim, David Blaauw, Trevor Mudge, and Ronald

Dreslinski. Outerspace: An outer product based sparse matrix multiplication accelerator. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 724–736, 2018.

[83] Daehyeon Baek, Soojin Hwang, Taekyung Heo, Daehoon Kim, and Jaehyuk Huh. Innersp: A memory efficient sparse matrix multiplication accelerator with locality-aware inner product processing. In *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 116–128, 2021.

[84] Sumanth Gudaparthi, Surya Narayanan, Rajeev Balasubramonian, Edouard Giacomin, Hari Kambalasubramanyam, and Pierre-Emmanuel Gaillardon. Wire-aware architecture and dataflow for cnn accelerators. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, page 1–13, New York, NY, USA, 2019. Association for Computing Machinery.

[85] Wenyan Lu, Guihai Yan, Jiajun Li, Shijun Gong, Yinhe Han, and Xiaowei Li. Flexflow: A flexible dataflow accelerator architecture for convolutional neural networks. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 553–564, 2017.

[86] Taoran Xiang, Yujing Feng, Xiaochun Ye, Xu Tan, Wenming Li, Yatao Zhu, Meng Wu, Hao Zhang, and Dongrui Fan. Accelerating cnn algorithm with fine-grained dataflow architectures. In *2018 IEEE 20th International Conference on High Performance Computing and Communications; IEEE 16th International Conference on Smart City; IEEE 4th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, pages 243–251, 2018.

[87] Angshuman Parashar, Priyanka Raina, Yakun Sophia Shao, Yu-Hsin Chen, Victor A. Ying, Anurag Mukkara, Rangharajan Venkatesan, Brucek Khailany, Stephen W. Keckler, and Joel Emer. Timeloop: A systematic approach to dnn accelerator evaluation. In *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 304–315, 2019.

[88] isakedo. Dnnsim. https://github.com/isakedo/DNNsim, 2020.

[89] Xuan Yang, Mingyu Gao, Qiaoyi Liu, Jeff Setter, Jing Pu, Ankita Nayak, Steven Bell, Kaidi Cao, Heonjae Ha, Priyanka Raina, Christos Kozyrakis, and Mark Horowitz. Interstellar: Using halide's scheduling language to analyze dnn accelerators. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 369–383, New York, NY, USA, 2020. Association for Computing Machinery.

[90] Hyochan An, Sam Schiferl, Siddharth Venkatesan, Tim Wesley, Qirui Zhang, Jingcheng Wang, Kyojin D. Choo, Shiyu Liu, Bowen Liu, Ziyun Li, Luyao Gong, Hengfei Zhong, David Blaauw, Ronald Dreslinski, Hun Seok Kim, and Dennis Sylvester. An ultra-low-power image signal processor for hierarchical image recognition with deep neural networks. *IEEE Journal of Solid-State Circuits*, 56(4):1071–1081, 2021.

[91] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *CVPR*, 2016.

[92] Nagadomi. Code for kaggle-cifar10 competition, 5th place. https://github.com/nagadomi/kaggle-cifar10-torch7, 2014.

[93] Jost Tobias Springenberg, Alexey Dosovitskiy, Thomas Brox, and Martin A. Riedmiller. Striving for simplicity: The all convolutional net. *CoRR*, abs/1412.6806, 2015.

[94] Sayeh Sharify, Alberto Delmas Lascorz, Mostafa Mahmoud, Milos Nikolic, Kevin Siu, Dylan Malone Stuart, Zissis Poulos, and Andreas Moshovos. Laconic deep learning inference acceleration. In *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, pages 304–317, 2019.

[95] Jorge Albericio, Patrick Judd, Tayler Hetherington, Tor Aamodt, Natalie Enright Jerger, and Andreas Moshovos. Cnvlutin: Ineffectual-neuron-free deep neural network computing. In *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA '16, page 1–13. IEEE Press, 2016.

[96] Kartik Hegde, Hadi Asghari-Moghaddam, Michael Pellauer, Neal Crago, Aamer Jaleel, Edgar Solomonik, Joel Emer, and Christopher W. Fletcher. Extensor: An accelerator for sparse tensor algebra. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, page 319–333, New York, NY, USA, 2019. Association for Computing Machinery.

[97] Vinayak Gokhale, Aliasger Zaidy, Andre Xian Ming Chang, and Eugenio Culurciello. Snowflake: An efficient hardware accelerator for convolutional neural networks. In *2017 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–4, 2017.

[98] Yongming Shen, Michael Ferdman, and Peter Milder. Escher: A cnn accelerator with flexible buffering to minimize off-chip transfer. In *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 93–100, 2017.

[99] Yongming Shen, Michael Ferdman, and Peter Milder. Maximizing cnn accelerator efficiency through resource partitioning. *SIGARCH Comput. Archit. News*, 45(2):535–547, jun 2017.

[100] Alessandro Aimar, Hesham Mostafa, Enrico Calabrese, Antonio Rios-Navarro, Ricardo Tapiador-Morales, Iulia-Alexandra Lungu, Moritz B. Milde, Federico Corradi, Alejandro Linares-Barranco, Shih-Chii Liu, and Tobi Delbruck. Nullhop: A flexible convolutional neural network accelerator based on sparse representations of feature maps. *IEEE Transactions on Neural Networks and Learning Systems*, 30(3):644–656, 2019.

[101] Patrick Judd, Jorge Albericio, Tayler Hetherington, Tor M. Aamodt, and Andreas Moshovos. Stripes: Bit-serial deep neural network computing. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12, 2016.

[102] Jinmook Lee, Changhyeon Kim, Sanghoon Kang, Dongjoo Shin, Sangyeob Kim, and Hoi-Jun Yoo. Unpu: An energy-efficient deep neural network accelerator with fully variable weight bit precision. *IEEE Journal of Solid-State Circuits*, 54(1):173–185, 2019.

[103] Chunhua Deng, Siyu Liao, Yi Xie, Keshab K. Parhi, Xuehai Qian, and Bo Yuan. Permdnn: Efficient compressed dnn architecture with permuted diagonal matrices. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 189–202, 2018.

[104] Caiwen Ding, Siyu Liao, Yanzhi Wang, Zhe Li, Ning Liu, Youwei Zhuo, Chao Wang, Xuehai Qian, Yu Bai, Geng Yuan, Xiaolong Ma, Yipeng Zhang, Jian Tang, Qinru Qiu, Xue Lin, and Bo Yuan. Circnn: Accelerating and compressing deep neural networks using block-circulant weight matrices. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-50 '17, page 395–408, New York, NY, USA, 2017. Association for Computing Machinery.

[105] Pierre Abillama, Zichen Fan, Yu Chen, Hyochan An, Qirui Zhang, Seungkyu Choi, David Blaauw, Dennis Sylvester, and Hun-Seok Kim. Sona: An accelerator for transform-domain neural networks with sparse-orthogonal weights. In *2023 IEEE 34th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 18–26, 2023.

[106] Francesco Conti, Davide Rossi, Gianna Paulin, Angelo Garofalo, Alfio Di Mauro, Georg Rutishauer, Gian marco Ottavi, Manuel Eggimann, Hayate Okuhara, Vincent Huard, Olivier Montfort, Lionel Jure, Nils Exibard, Pascal Gouedo, Mathieu Louvat, Emmanuel Botte, and Luca Benini. 22.1 a 12.4tops/w @ 136gops ai-iot system-on-chip with 16 risc-v, 2-to-8b precision-scalable dnn acceleration and 30In *2023 IEEE International Solid-State Circuits Conference (ISSCC)*, pages 21–23, 2023.

[107] Zichen Fan, Qirui Zhang, Hyochan An, Boxun Xu, Li Xu, Chien-Wei Tseng, Yimai Peng, Andrea Bejarano-Carbo, Pierre Abillama, Ang Cao, Bowen Liu, Changwoo Lee, Zhehong Wang, Hun-Seok Kim, David Blaauw, and Dennis Sylvester. Aimmi: Audio and image multi-modal intelligence via a low-power soc with 2-mbyte on-chip mram for iot devices. *IEEE Journal of Solid-State Circuits*, 59(10):3488–3501, 2024.

[108] Kartik Prabhu, Albert Gural, Zainab F. Khan, Robert M. Radway, Massimo Giordano, Kalhan Koul, Rohan Doshi, John W. Kustin, Timothy Liu, Gregorio B. Lopes, Victor Turbiner, Win-San Khwa, Yu-Der Chih, Meng-Fan Chang, Guénolé Lallement, Boris Murmann, Subhasish Mitra, and Priyanka Raina. Chimera: A 0.92-tops, 2.2-tops/w edge ai accelerator with 2-mbyte on-chip foundry resistive ram for efficient training and inference. *IEEE Journal of Solid-State Circuits*, 57(4):1013–1026, 2022.

[109] Vikram Jain, Sebastian Giraldo, Jaro De Roose, Linyan Mei, Bert Boons, and Marian Verhelst. Tinyvers: A tiny versatile system-on-chip with state-retentive emram for ml inference at the extreme edge. *IEEE Journal of Solid-State Circuits*, 58(8):2360–2371, 2023.

[110] Wenyu Sun, Xiaoyu Feng, Chen Tang, Shupei Fan, Yixiong Yang, Jinshan Yue, Huazhong Yang, and Yongpan Liu. A 28nm 2d/3d unified sparse convolution accelerator with block-wise neighbor searcher for large-scaled voxel-based point cloud network. In *2023 IEEE International Solid-State Circuits Conference (ISSCC)*, pages 328–330, 2023.

[111] Jie-Fang Zhang, Ching-En Lee, Chester Liu, Yakun Sophia Shao, Stephen W. Keckler, and Zhengya Zhang. Snap: An efficient sparse neural acceleration processor for unstructured sparse deep neural network inference. *IEEE Journal of Solid-State Circuits*, 56(2):636–647, 2021.

[112] Jinshan Yue, Yongpan Liu, Ruoyang Liu, Wenyu Sun, Zhe Yuan, Yung-Ning Tu, Yi-Ju Chen, Ao Ren, Yanzhi Wang, Meng-Fan Chang, Xueqing Li, and Huazhong Yang. Sticker-t: An energy-efficient neural network processor using block-circulant algorithm and unified frequency-domain acceleration. *IEEE Journal of Solid-State Circuits*, 56(6):1936–1948, 2021.

[113] Eric Hunsberger and Chris Eliasmith. Spiking deep networks with lif neurons. *arXiv preprint arXiv:1510.08829*, 2015.

[114] Anthony N Burkitt. A review of the integrate-and-fire neuron model: I. homogeneous synaptic input. *Biological cybernetics*, 95:1–19, 2006.

[115] Bodo Rueckauer, Iulia-Alexandra Lungu, Yuhuang Hu, Michael Pfeiffer, and Shih-Chii Liu. Conversion of continuous-valued deep networks to efficient event-driven networks for image classification. *Frontiers in neuroscience*, 11:682, 2017.

[116] Peter U Diehl, Daniel Neil, Jonathan Binas, Matthew Cook, Shih-Chii Liu, and Michael Pfeiffer. Fast-classifying, high-accuracy spiking deep networks through weight and threshold balancing. In *2015 International joint conference on neural networks (IJCNN)*, pages 1–8. ieee, 2015.

[117] Tong Bu, Wei Fang, Jianhao Ding, PengLin Dai, Zhaofei Yu, and Tiejun Huang. Optimal ann-snn conversion for high-accuracy and ultra-low-latency spiking neural networks. *arXiv preprint arXiv:2303.04347*, 2023.

[118] Abhronil Sengupta, Yuting Ye, Robert Wang, Chiao Liu, and Kaushik Roy. Going deeper in spiking neural networks: Vgg and residual architectures. *Frontiers in neuroscience*, 13:95, 2019.

[119] Nitin Rathi and Kaushik Roy. Diet-snn: A low-latency spiking neural network with direct input encoding and leakage and threshold optimization. In *IEEE Transactions on Neural Networks and Learning Systems*, pages 3174–3182, 2023.

[120] Shikuang Deng, Yuhang Li, Shanghang Zhang, and Shi Gu. Temporal efficient training of spiking neural network via gradient re-weighting. In *International Conference on Learning Representations*, 2022.

[121] Man Yao, JiaKui Hu, Zhaokun Zhou, Li Yuan, Yonghong Tian, Bo Xu, and Guoqi Li. Spike-driven transformer. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, editors, *Advances in Neural Information Processing Systems*, volume 36, pages 64043–64058. Curran Associates, Inc., 2023.

[122] Sayeed Shafayet Chowdhury, Nitin Rathi, and Kaushik Roy. Towards ultra low latency spiking neural networks for vision and sequential tasks using temporal pruning. In *Computer Vision – ECCV 2022: 17th European Conference, Tel Aviv, Israel, October 23–27, 2022, Proceedings, Part XI*, page 709–726, Berlin, Heidelberg, 2022. Springer-Verlag.

[123] Juan Antonio Leñero-Bardallo, Teresa Serrano-Gotarredona, and Bernabé Linares-Barranco. A 3.6μs latency asynchronous frame-free event-driven dynamic-vision-sensor. *IEEE Journal of Solid-State Circuits*, 46(6):1443–1455, 2011.

[124] Yuhang Li, Tamar Geller, Youngeun Kim, and Priyadarshini Panda. Seenn: Towards temporal spiking early exit neural networks. *Advances in Neural Information Processing Systems*, 36, 2024.

[125] Rong Xiao, Qiang Yu, Rui Yan, and Huajin Tang. Fast and accurate classification with a multi-spike learning algorithm for spiking neurons. In *IJCAI*, pages 1445–1451, 2019.

[126] Yu Miao, Huajin Tang, and Gang Pan. A supervised multi-spike learning algorithm for spiking neural networks. In *2018 international joint conference on neural networks (IJCNN)*, pages 1–7. IEEE, 2018.

[127] Xiaoting Wang and Yanxiang Zhang. Mt-snn: enhance spiking neural network with multiple thresholds. *arXiv preprint arXiv:2303.11127*, 2023.

[128] Lang Feng, Qianhui Liu, Huajin Tang, De Ma, and Gang Pan. Multi-level firing with spiking ds-resnet: Enabling better and deeper directly-trained spiking neural networks. *arXiv preprint arXiv:2210.06386*, 2022.

[129] Hongmin Li, Hanchao Liu, Xiangyang Ji, Guoqi Li, and Luping Shi. Cifar10-dvs: an event-stream dataset for object classification. *Frontiers in neuroscience*, 11:309, 2017.

[130] Garrick Orchard, Ajinkya Jayawant, Gregory K Cohen, and Nitish Thakor. Converting static image datasets to spiking neuromorphic datasets using saccades. *Frontiers in neuroscience*, 9:437, 2015.

[131] Shikuang Deng and Shi Gu. Optimal conversion of conventional artificial neural networks to spiking neural networks. *arXiv preprint arXiv:2103.00476*, 2021.

[132] Yuhang Li, Shikuang Deng, Xin Dong, Ruihao Gong, and Shi Gu. A free lunch from ann: Towards efficient, accurate spiking neural networks calibration. In *International conference on machine learning*, pages 6316–6325. PMLR, 2021.

[133] Emre O Neftci, Hesham Mostafa, and Friedemann Zenke. Surrogate gradient learning in spiking neural networks: Bringing the power of gradient-based optimization to spiking neural networks. *IEEE Signal Processing Magazine*, 36(6):51–63, 2019.

[134] Guillaume Bellec, Darjan Salaj, Anand Subramoney, Robert Legenstein, and Wolfgang Maass. Long short-term memory and learning-to-learn in networks of spiking neurons. *Advances in neural information processing systems*, 31, 2018.

[135] Lei Deng, Yujie Wu, Xing Hu, Ling Liang, Yufei Ding, Guoqi Li, Guangshe Zhao, Peng Li, and Yuan Xie. Rethinking the performance comparison between snns and anns. *Neural networks*, 121:294–307, 2020.

[136] Weihao Tan, Devdhar Patel, and Robert Kozma. Strategy and benchmark for converting deep q-networks to event-driven spiking neural networks. In *Proceedings of the AAAI conference on artificial intelligence*, volume 35, pages 9816–9824, 2021.

[137] Peisong Wang, Xiangyu He, Gang Li, Tianli Zhao, and Jian Cheng. Sparsity-inducing binarized neural networks. In *Proceedings of the AAAI conference on artificial intelligence*, volume 34, pages 12192–12199, 2020.

[138] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. In *European conference on computer vision*, pages 525–542. Springer, 2016.

[139] Ziqing Wang, Yuetong Fang, Jiahang Cao, and Renjing Xu. Bursting spikes: Efficient and high-performance snns for event-based vision. *arXiv preprint arXiv:2311.14265*, 2023.

[140] Yang Li and Yi Zeng. Efficient and accurate conversion of spiking neural network with burst spikes. *arXiv preprint arXiv:2204.13271*, 2022.

[141] Gourav Datta, Zeyu Liu, and Peter Anthony Beerel. Can we get the best of both binary neural networks and spiking neural networks for efficient computer vision? In *The Twelfth International Conference on Learning Representations*, 2024.

[142] Daisuke Miyashita, Edward H Lee, and Boris Murmann. Convolutional neural networks using logarithmic data representation. *arXiv preprint arXiv:1603.01025*, 2016.

[143] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A, Section 3-437*, 2024. Available at: https://cdrdv2.intel.com/v1/dl/getContent/671200.

[144] Terrance DeVries and Graham W Taylor. Improved regularization of convolutional neural networks with cutout. *arXiv preprint arXiv:1708.04552*, 2017.

[145] Ekin D Cubuk, Barret Zoph, Dandelion Mane, Vijay Vasudevan, and Quoc V Le. Autoaugment: Learning augmentation strategies from data. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 113–123, 2019.

[146] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034, 2015.

[147] Ali Samadzadeh, Fatemeh Sadat Tabatabaei Far, Ali Javadi, Ahmad Nickabadi, and Morteza Haghir Chehreghani. Convolutional spiking neural networks for spatio-temporal feature extraction. *Neural Processing Letters*, pages 1–17, 2023.

[148] Yuhang Li, Youngeun Kim, Hyoungseob Park, Tamar Geller, and Priyadarshini Panda. Neuromorphic data augmentation for training spiking neural networks. In *European Conference on Computer Vision*, pages 631–649. Springer, 2022.

[149] Hanle Zheng, Yujie Wu, Lei Deng, Yifan Hu, and Guoqi Li. Going deeper with directly-trained larger spiking neural networks. In *Proceedings of the AAAI conference on artificial intelligence*, volume 35, pages 11062–11070, 2021.

[150] Alexander Kugele, Thomas Pfeil, Michael Pfeiffer, and Elisabetta Chicca. Efficient processing of spatio-temporal data streams with spiking neural networks. *Frontiers in Neuroscience*, 14:439, 2020.

[151] Zhenzhi Wu, Hehui Zhang, Yihan Lin, Guoqi Li, Meng Wang, and Ye Tang. Liaf-net: Leaky integrate and analog fire network for lightweight and efficient spatiotemporal information processing. *IEEE Transactions on Neural Networks and Learning Systems*, 33:6249–6262, 2021.

[152] Yuhang Li, Yufei Guo, Shanghang Zhang, Shikuang Deng, Yongqing Hai, and Shi Gu. Differentiable spike: Rethinking gradient-descent for training spiking neural networks. *Advances in Neural Information Processing Systems*, 34:23426–23439, 2021.

[153] Yufei Guo, Xinyi Tong, Yuanpei Chen, Liwen Zhang, Xiaode Liu, Zhe Ma, and Xuhui Huang. Recdis-snn: Rectifying membrane potential distribution for directly training spiking neural networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 326–335, 2022.

[154] Wei Fang, Zhaofei Yu, Yanqi Chen, Timothée Masquelier, Tiejun Huang, and Yonghong Tian. Incorporating learnable membrane time constant to enhance learning of spiking neural networks. In *Proceedings of the IEEE/CVF international conference on computer vision*, pages 2661–2671, 2021.

[155] Kexin Wang, Jiahong Zhang, Yong Ren, Man Yao, Di Shang, Bo Xu, and Guoqi Li. Spikevoice: High-quality text-to-speech via efficient spiking neural network. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 7927–7940, 2024.

[156] Xingrun Xing, Boyan Gao, Zheng Zhang, David A Clifton, Shitao Xiao, Li Du, Guoqi Li, and Jiajun Zhang. Spikellm: Scaling up spiking neural network to large language models via saliency-based spiking. *CoRR*, 2024.

[157] Jianhao Ding, Tong Bu, Zhaofei Yu, Tiejun Huang, and Jian Liu. Snn-rat: Robustness-enhanced spiking neural network through regularized adversarial training. *Advances in Neural Information Processing Systems*, 35:24780–24793, 2022.

[158] Jianhao Ding, Zhiyu Pan, Yujia Liu, Zhaofei Yu, and Tiejun Huang. Robust stable spiking neural networks. In Ruslan Salakhutdinov, Zico Kolter, Katherine Heller, Adrian Weller, Nuria Oliver, Jonathan Scarlett, and Felix Berkenkamp, editors, *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pages 11016–11029. PMLR, 21–27 Jul 2024.

[159] Jeong-Jun Lee and Peng Li. Reconfigurable dataflow optimization for spatiotemporal spiking neural computation on systolic array accelerators. In *2020 IEEE 38th International Conference on Computer Design (ICCD)*, pages 57–64. IEEE, 2020.

[160] Jeong-Jun Lee, Wenrui Zhang, and Peng Li. Parallel time batching: Systolic-array acceleration of sparse spiking neural computation. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 317–330. IEEE, 2022.

[161] Surya Narayanan, Karl Taht, Rajeev Balasubramonian, Edouard Giacomin, and Pierre-Emmanuel Gaillardon. Spinalflow: An architecture and dataflow tailored for spiking neural networks. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 349–362. IEEE, 2020.

[162] Jindong Li, Guobin Shen, Dongcheng Zhao, Qian Zhang, and Yi Zeng. Firefly v2: Advancing hardware support for high-performance spiking neural network with a spatiotemporal fpga accelerator. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 43(9):2647–2660, 2024.

[163] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

[164] Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, et al. Training compute-optimal large language models. *arXiv preprint arXiv:2203.15556*, 2022.

[165] William Peebles and Saining Xie. Scalable diffusion models with transformers. *arXiv preprint arXiv:2212.09748*, 2022.

[166] Can Yaras, Peng Wang, Wei Hu, Zhihui Zhu, Laura Balzano, and Qing Qu. The law of parsimony in gradient descent for learning deep linear networks. *arXiv preprint arXiv:2306.01154*, 2023.

[167] Changwoo Lee and Hun-Seok Kim. Differentiable learning of generalized structured matrices for efficient deep neural networks. In *The Twelfth International Conference on Learning Representations*, 2024.

[168] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.

[169] Chi Yang, Sara Seif Baghsorkhi, Karthik Muralidharan, and John Cavazos. An empirical roofline methodology for gpus: Analyzing performance portability. In *Proceedings of the ACM International Conference on Computing Frontiers*, pages 1–10. ACM, 2013.

[170] Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. A survey on large language models for code generation. *arXiv preprint arXiv:2406.00515*, 2024.

[171] Masahiro Kaneko and Naoaki Okazaki. Reducing sequence length by predicting edit spans with large language models. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 10017–10029, 2023.

[172] Xin Wang, Yu Zheng, Zhongwei Wan, and Mi Zhang. SVD-LLM: Truncation-aware singular value decomposition for large language model compression. In *International Conference on Learning Representations (ICLR)*, 2025.

[173] Yerlan Idelbayev and Miguel A Carreira-Perpinán. Low-rank compression of neural nets: Learning the rank of each layer. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 8049–8059, 2020.

[174] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. *OpenAI Blog*, 1(8):9, 2019.

[175] Yonatan Bisk, Rowan Zellers, Jianfeng Gao, Yejin Choi, et al. Piqa: Reasoning about physical commonsense in natural language. In *Proceedings of the AAAI conference on artificial intelligence*, volume 34, pages 7432–7439, 2020.

[176] Rowan Zellers, Ari Holtzman, Yonatan Bisk, Ali Farhadi, and Yejin Choi. Hellaswag: Can a machine really finish your sentence? *arXiv preprint arXiv:1905.07830*, 2019.

[177] Keisuke Sakaguchi, Ronan Le Bras, Chandra Bhagavatula, and Yejin Choi. Winogrande: An adversarial winograd schema challenge at scale. *Communications of the ACM*, 64(9):99–106, 2021.

[178] Christopher Clark, Kenton Lee, Ming-Wei Chang, Tom Kwiatkowski, Michael Collins, and Kristina Toutanova. Boolq: Exploring the surprising difficulty of natural yes/no questions. *arXiv preprint arXiv:1905.10044*, 2019.

[179] Todor Mihaylov, Peter Clark, Tushar Khot, and Ashish Sabharwal. Can a suit of armor conduct electricity? a new dataset for open book question answering. *arXiv preprint arXiv:1809.02789*, 2018.

[180] Peter Clark, Isaac Cowhey, Oren Etzioni, Tushar Khot, Ashish Sabharwal, Carissa Schoenick, and Oyvind Tafjord. Think you have solved question answering? try arc, the ai2 reasoning challenge. *arXiv preprint arXiv:1803.05457*, 2018.

[181] Jonathan Ho, Ajay Jain, and Pieter Abbeel. Denoising diffusion probabilistic models. In *Proceedings of the 34th International Conference on Neural Information Processing Systems*, NIPS '20, Red Hook, NY, USA, 2020. Curran Associates Inc.

[182] Zhe Jia, Marco Maggioni, Benjamin Staiger, and Daniele P Scarpazza. Dissecting the nvidia volta gpu architecture via microbenchmarking. *arXiv preprint arXiv:1804.06826*, 2018.

[183] NVIDIA. Jetson orin nano developer kit, 2024.

[184] NVIDIA. Nvidia a40 data center gpu, 2022.

[185] Kyoshin Choo, William Panlener, and Byunghyun Jang. Understanding and optimizing gpu cache memory performance for compute workloads. In *2014 IEEE 13th International Symposium on Parallel and Distributed Computing*, pages 189–196, 2014.

[186] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, and Kevin Skadron. A performance study of general-purpose applications on graphics processors using cuda. *Journal of parallel and distributed computing*, 68(10):1370–1380, 2008.

[187] Shane Ryoo, Christopher I. Rodrigues, Sara S. Baghsorkhi, Sam S. Stone, David B. Kirk, and Wen-mei W. Hwu. Optimization principles and application performance evaluation of a multithreaded gpu using cuda. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, page 73–82, New York, NY, USA, 2008. Association for Computing Machinery.

[188] Keren Zhou, Mario Lezcano, Adam Goucher, Akhmed Rakhmati, Jeff Niu, Justin Lebar, Pawel Szczerbuk, Peter Bell, Phil Tillet, Thomas Raoux, et al. Linear layouts: Robust code generation of efficient tensor computation using $\mathbb{F}_2$. *arXiv preprint arXiv:2505.23819*, 2025.

[189] Jianling Li, Shangzhan Li, Zhenye Gao, Qi Shi, Yuxuan Li, Zefan Wang, Jiacheng Huang, Haojie Wang, Jianrong Wang, Xu Han, Zhiyuan Liu, and Maosong Sun. Tritonbench: Benchmarking large language model capabilities for generating triton operators, 2025.

[190] Zhihang Yuan, Yuzhang Shang, Yang Zhou, Zhen Dong, Zhe Zhou, Chenhao Xue, Bingzhe Wu, Zhikai Li, Qingyi Gu, Yong Jae Lee, Yan Yan, Beidi Chen, Guangyu Sun, and Kurt Keutzer. Llm inference unveiled: Survey and roofline model insights, 2024.

[191] Wei Sun, Ang Li, Sander Stuijk, and Henk Corporaal. How much can we gain from tensor kernel fusion on gpus? *IEEE Access*, 12:126135–126144, 2024.

[192] Nader Al Awar, Muhammad Hannan Naeem, James Almgren-Bell, George Biros, and Milos Gligoric. Dynamically fusing python hpc kernels. *Proc. ACM Softw. Eng.*, 2(ISSTA), June 2025.

[193] Tri Dao. Flashattention-2: Faster attention with better parallelism and work partitioning, 2023.

[194] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

[195] Daria Soboleva, Faisal Al-Khateeb, Robert Myers, Jacob R Steeves, Joel Hestness, and Nolan Dey. SlimPajama: A 627B token cleaned and deduplicated version of RedPajama, June 2023.

[196] Zhihang Yuan, Hanling Zhang, Lu Pu, Xuefei Ning, Linfeng Zhang, Tianchen Zhao, Shengen Yan, Guohao Dai, and Yu Wang. Ditfastattn: Attention compression for diffusion transformer models. *Advances in Neural Information Processing Systems*, 37:1196–1219, 2024.

[197] Saleh Ashkboos, Amirkeivan Mohtashami, Maximilian L. Croci, Bo Li, Pashmina Cameron, Martin Jaggi, Dan Alistarh, Torsten Hoefler, and James Hensman. Quarot: Outlier-free 4-bit inference in rotated llms. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024. arXiv preprint arXiv:2404.00456.

[198] Zechun Liu, Changsheng Zhao, Igor Fedorov, Bilge Soran, Dhruv Choudhary, Raghuraman Krishnamoorthi, Vikas Chandra, Yuandong Tian, and Tijmen Blankevoort. Spinquant: Llm quantization with learned rotations. In *International Conference on Learning Representations (ICLR)*, 2025. arXiv preprint arXiv:2405.16406.

[199] Zechun Liu, Barlas Oguz, Changsheng Zhao, Ernie Chang, Pierre Stock, Yashar Mehdad, Yangyang Shi, Raghuraman Krishnamoorthi, and Vikas Chandra. Llm-qat: Data-free quantization aware training for large language models. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 467–484, Bangkok, Thailand, 2024.

[200] Albert Gu and Tri Dao. Mamba: Linear-time sequence modeling with selective state spaces. In *First Conference on Language Modeling*, 2024.

[201] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in neural information processing systems*, 33:9459–9474, 2020.