**ΔΗΜΟΚΡΙΤΕΙΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΡΑΚΗΣ**

**ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ**

**ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ**

# Κυκλώματα με Παραλληλία στα Δεδομένα για Αλγόριθμους Μηχανικής Μάθησης

**ΔΙΔΑΚΤΟΡΙΚΗ ΔΙΑΤΡΙΒΗ**

Χριστόδουλος Πελτέκης, 70397:

Επιβλέπων Καθηγητής: Γεώργιος Δημητρακόπουλος, Καθηγητής,
Τμήμα Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών, Δ.Π.Θ.

*Ξάνθη, 2025*

**ΔΗΜΟΚΡΙΤΕΙΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΡΑΚΗΣ**
**ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ**
**ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ**

# Κυκλώματα με Παραλληλία στα Δεδομένα για Αλγόριθμους Μηχανικής Μάθησης

## ΔΙΔΑΚΤΟΡΙΚΗ ΔΙΑΤΡΙΒΗ

Χριστόδουλος Πελτέκης, 70397:

Η παρούσα διδακτορική διατριβή υποβλήθηκε στο Τμήμα Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών του Δημοκρίτειου Πανεπιστημίου Θράκης για την απόκτηση του διδακτορικού τίτλου σπουδών.

*Ξάνθη, 2025*

**DEMOCRITUS UNIVERSITY OF THRACE**
**SCHOOL OF ENGINEERING**
**DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING**

# Data Parallel Hardware Accelerators for Machine Learning Algorithms

## DOCTORAL THESIS

Christodoulos Peltekis, 70397:

A thesis submitted in partial fulfilment of the requirements for the degree of Doctor of Philosophy (Ph.D.), Department of Electrical and Computer Engineering, Democritus University of Thrace.

*Xanthi, 2025*

# Περίληψη

Η μηχανική μάθηση (Machine Learning – ML) έχει μετασχηματίσει καθοριστικά πολλά πεδία εφαρμογών όπως η υπολογιστική όραση, η επεξεργασία φυσικής γλώσσας και η ρομποτική. Ωστόσο, οι υπολογιστικές απαιτήσεις των σύγχρονων μοντέλων βαθιάς μάθησης υπερβαίνουν κατά πολύ τις δυνατότητες των συμβατικών επεξεργαστών. Αυτή η πρόκληση έχει οδηγήσει στην ανάπτυξη εξειδικευμένων επιταχυντών, με κυριότερο παράδειγμα τους συστολικούς πίνακες (systolic arrays), οι οποίοι υπερέχουν στον υπολογισμό πράξεων μεταξύ πινάκων που κυριαρχούν στον υπολογισμό των νευρωνικών δικτύων. Παρόλα αυτά, η επίτευξη υψηλής απόδοσης και ενεργειακής αποδοτικότητας στα σύγχρονα μοντέλα απαιτεί την αποτελεσματική αντιμετώπιση αραιών μοντέλων μηχανικής μάθησης καθώς και την σχεδίαση υλικού που είναι ανθεκτικό σε σφάλματα. Η παρούσα διατριβή αντιμετωπίζει αυτές τις προκλήσεις προτείνοντας ένα σύνολο βελτιστοποιήσεων υλικού που ενισχύουν την απόδοση και την αξιοπιστία των συστολικών πινάκων και των επεκτάσεών τους για αραιούς πίνακες καλύπτοντας έτσι ένα ευρύ φάσμα εφαρμογών.

Αρχικά, βελτιστοποιούμε τόσο την μικροαρχιτεκτονική όσο και τη φυσική υλοποίηση πυκνών συστολικών πινάκων. Προτείνουμε το *ArrayFlex*, μια αρχιτεκτονική παραμετροποιήσιμης διασωλήνωσης (pipelining), που προσαρμόζει δυναμικά το βάθος του pipeline ανάλογα με τα χαρακτηριστικά της συνέλιξης που καλείται να υπολογίσει, με σκοπό την ελαχιστοποίηση της καθυστέρησης και της κατανάλωσης ισχύος. Συμπληρωματικά, προτείνουμε μία μεθοδολογία ασύμμετρης χωροθέτησης του συστολικού πίνακα που λαμβάνει υπόψη την εγγενή ανισορροπία στο πλάτος των οριζόντιων και κάθετων διασύνδεσων, οδηγώντας σε μετρήσιμες μειώσεις της κατανάλωσης ισχύος. Επιπλέον, με την κατάλληλη κωδικοποίηση των δεδομένων που εισέρχονται στο συστολικό πίνακα και την αναδιάταξη του πετυχαίνουμε σημαντική μείωση των συνολικών μεταβάσεων με αποτέλεσμα την περαιτέρω μείωση της κατανάλωσης ισχύος.

Στη συνέχεια, ενισχύουμε τη μικροαρχιτεκτονική των αραιών συστολικών πινάκων που εκτελούν κλαδεμένα (pruned) μοντέλα μηχανικής μάθησης, τα οποία χρησιμοποιούνται για τη μείωση των απαιτήσεων σε μνήμη. Πιο συγκεκριμένα, προτείνουμε το DeMM, μια νέα αρχιτεκτονική πολλαπλασιασμού αραιού με πυκνό πίνακα που υποστηρίζει χαλαρή δομημένη αραιότητα διατηρώντας παράλληλα τη κανονικότητα του υλικού. Επίσης, προτείνουμε το DSSTA, έναν τανυστικό πίνακα με διαγώνια μετατεθειμένη ροή δεδομένων (diagonal-permuted dataflow) που εξαλείφει το κόστος ολίσθησης και αποθήκευσης κατά την είσοδο των δεδομένων στο συστολικό πίνακα. Και οι δύο αρχιτεκτονικές επιτυγχάνουν σημαντικές μειώσεις στην καθυστέρηση και την κατανάλωση ενέργειας για σύγχρονα φορτία συνελκτικών δικτύων. Για την ενίσχυση της αξιοπιστίας του υλικού, εισάγουμε δύο συμπληρωματικούς μηχανισμούς ανοχής σε σφάλματα: μια μέθοδο βασισμένη σε αθροίσματα ελέγχου (checksum) προσαρμοσμένη στη δομημένη αραιότητα, και ένα περιοδικό πλαίσιο ελέγχου που ανιχνεύει κατά τη διάρκεια της λειτουργίας μόνιμα σφάλματα με ελάχιστο κόστος. Από κοινού, οι τεχνικές αυτές ενισχύουν την αξιοπιστία των επιταχυντών για εφαρμογές μηχανικής μάθησης με κλαδεμένα μοντέλα.

Οι μικροαρχιτεκτονικές βελτιώσεις επεκτείνονται επίσης σε επιταχυντές υλικού για Γραφο-Συνελικτικά Δίκτυα (Graph Convolutional Networks – GCNs). Πιο συγκεκριμένα, προτείνουμε το FusedGCN, μια συστολική αρχιτεκτονική τριπλού πολλαπλασιασμού μητρώων (triple-matrix) που μειώνει σημαντικά τον χρόνο εξαγωγής (inference latency) σε σχέση με προηγμένες λύσεις. Επίσης, για την ενίσχυση της αξιοπιστίας, προτείνουμε το μηχανισμό GCN-ABFT, έναν προσαρμοσμένο μηχανισμό ανοχής σε σφάλματα για GCNs που υπολογίζει το άθροισμα ελέγχου ολόκληρου του επιπέδου GCN χωρίς να απαιτείται ξεχωριστός έλεγχος για κάθε στάδιο πολλαπλασιασμού πινάκων.

Επιπλέον, προτείνουμε μια μονάδα υλικού softmax βασισμένη στην παραλληλία στα δεδομένα, η οποία υποστηρίζει ευέλικτα τόσο τον υπολογισμό softmax ενός πλήρους διανύσματος όσο και τμηματικών υπολογισμών softmax για υποδιανύσματα. Ο σχεδιασμός αυτός επιτρέπει στη μονάδα να υπολογίζει είτε το πλήρες softmax ενός διανύσματος είτε πολλαπλούς μικρότερους υπολογισμούς softmax, των οποίων τα ενδιάμεσα αποτελέσματα μπορούν να επαναχρησιμοποιηθούν για την υλοποίηση διαφόρων συναρτήσεων ενεργοποίησης, όπως GELU, sigmoid ή tanh. Μέσω αυτής της επαναχρησιμοποίησης, μια μοναδική μονάδα SIMD softmax μπορεί να λειτουργήσει ως ενιαίο μπλοκ υλικού για πολλαπλές συναρτήσεις ενεργοποίησης.

Συνολικά, στη διατριβή αυτή συνδυάζοντας τη μικροαρχιτεκτονική καινοτομία με την επίγνωση του φυσικού σχεδιασμού και των χαρακτηριστικών των εφαρμογών, οδηγηθήκαμε σε νέες συστολικές αρχιτεκτονικές υψηλότερης απόδοσης και αξιοπιστίας. Οι προτεινόμενες μέθοδοι καλύπτουν τόσο το πυκνό όσο και το αραιό υπολογιστικό καθεστώς των σύγχρονων μοντέλων ML, θέτοντας τα θεμέλια για την επόμενη γενιά υλικού που εξισορροπεί ταχύτητα, ενέργεια και ανθεκτικότητα σε όλο και πιο απαιτητικά περιβάλλοντα εφαρμογών.

**Λέξεις-κλειδιά:** Συστολικοί Πίνακες, Αραιοί Συστολικοί Πίνακες, Δομημένη Αραιότητα, Επιταχυντές Υλικού, Ενεργειακή Αποδοτικότητα, Ανοχή σε Σφάλματα, Συνελικτικά Νευρωνικά Δίκτυα , Συνελικτικά Νευρωνικά Δίκτυα Γράφων, Transformers, Μηχανική Μάθηση

# Abstract

Machine learning (ML) has rapidly evolved into a transformative technology, driving advances across diverse domains such as computer vision, natural language processing, and robotics. However, the computational demands of modern deep learning models far exceed the capabilities of conventional processors. This challenge has motivated the development of domain-specific accelerators, particularly systolic arrays, which excel at the dense and regular matrix operations that dominate neural network workloads. Achieving high performance and energy efficiency on modern models, however, also requires addressing challenges such as sparsity, fault tolerance, and workload irregularity. This thesis tackles these challenges by proposing a set of hardware optimizations that improve the performance, efficiency, and reliability of systolic arrays and their tensor-based extensions for a broad range of machine learning applications.

First, we investigate microarchitectural and physical-design-oriented optimizations for dense systolic arrays. We introduce ArrayFlex, an architecture featuring a configurable pipeline that dynamically adjusts its depth across convolutional layers to minimize latency and energy consumption during convolutional neural network execution. Complementing this, we propose an asymmetric floorplanning methodology that accounts for the inherent imbalance in horizontal and vertical interconnect demands, resulting in measurable reductions in interconnect power. Building on these foundations, we further lower switching activity through dynamic encoding and weight reordering, leveraging workload-aware data characteristics to achieve significant power savings with minimal hardware overhead.

Second, we enhance the microarchitecture of sparse systolic tensor arrays designed to execute pruned machine learning models, which are often employed to reduce memory and storage requirements. We propose DeMM, a disaggregated matrix multiplication engine that supports relaxed sparsity patterns while maintaining hardware regularity, and DSSTA, a tensor array featuring a diagonal-permuted dataflow that eliminates skewing overhead. Both architectures achieve substantial reductions in latency and power consumption for modern CNN workloads. To further ensure reliability, we introduce two complementary fault-tolerance mechanisms: a checksum-based ABFT method adapted for structured sparsity, and a periodic online testing framework that detects permanent faults with minimal overhead. Together, these techniques enhance the robustness of sparse tensor accelerators for safety-critical machine learning applications.

Microarchitecture improvements are also extended to hardware accelerators for Graph Convolutional Networks. More specifically, we propose FusedGCN, a fused triple-matrix systolic architecture that significantly reduces inference latency compared to state-of-the-art accelerators. To enhance reliability, we also introduce GCN-ABFT, a custom fault-tolerance mechanism for GCNs that computes checksum for the whole GCN layer without relying on separate checksums for each matrix multiplication step.

Also, we propose a SIMD-based softmax hardware unit that flexibly supports both full-vector and segmented softmax computations. This design enables the unit to compute a complete softmax over an input vector or multiple smaller softmax operations over its sub-vectors. The intermediate results from these sub-vector computations can be efficiently reused to implement various activation functions, such as GELU, sigmoid or tanh. By leveraging this reuse, a single SIMD softmax unit can serve as a unified hardware block for multiple activation functions, reducing hardware redundancy and improving overall resource efficiency.

Overall, this work demonstrates that by combining micro-architectural innovation, physical implementation awareness, and application-driven co-design, systolic and tensor array accelerators can be significantly improved in terms of performance, energy efficiency, and reliability. The proposed methods address both the dense and sparse computation regimes of modern ML models, laying the foundation for next-generation hardware that balances speed, energy, and robustness in increasingly demanding application contexts.

**Keywords:** Systolic Arrays, Sparse Tensor Arrays, Structured Sparsity, Hardware Accelerators, Energy Efficiency, Fault-Tolerance, Convolutional Neural Networks (CNNs), Graph Convolutional Networks (GCNs), Transformers, Machine Learning Inference

# Acknowledgements

First and foremost, I would like to express my deepest gratitude to my advisor, Professor Giorgos Dimitrakopoulos. His guidance, encouragement, and constant support have been invaluable to me, not only during my Ph.D. studies, but even before I embarked on this journey. His dedication, depth of knowledge, and passion for research have inspired me, and I feel truly fortunate to have had the chance to work under his supervision. Beyond research, his mentorship has helped me grow as both an engineer and as a person, something for which I will always remain grateful.

I would also like to thank the members of my advisory and defense committee Professors Georgios Sirakoulis, Vasileios Paliouras, Chrysostomos Nicopoulos, Georgios Zervakis, Ioannis Vourkas, and Lazaros Papadopoulos for taking the time to review my work and for offering constructive comments that helped improve this thesis. I owe special thanks to Professor Chrysostomos Nicopoulos for our close collaboration and for his meaningful contribution to my published research.

My heartfelt thanks also go to my labmates and friends, Dionysis, Vasilis, Kosmas, Giorgos and Angelos, for all the discussions, the laughs, and the moments we shared inside and outside the lab. I am especially thankful to Dionysis for our excellent collaborations and the papers we worked on together. I also want to thank my fellow Ph.D. friends, Vasilis, Kostas, Aggelos, and Manolis, whose company during quick coffee breaks and long evenings out provided a welcome balance to the demands of research life.

Finally, I would like to thank my family. My parents, Kostas and Afroditi, and my brother, Nikolas, have been my biggest source of strength throughout all these years. Their unconditional love, endless support, and constant encouragement have carried me through the hardest moments and made this achievement possible. This thesis is as much theirs as it is mine.

# Contents

# Chapter 1

# Introduction

The past few years have witnessed the remarkable ascent of machine learning (ML) as a transformative force impacting a vast and expanding array of domains. From revolutionizing computer vision [1–3] and natural language processing [4, 5] to driving advancements in healthcare [6] and accelerating scientific discovery, ML's influence is undeniable. At the heart of this profound shift lies the continuous development of increasingly sophisticated neural network architectures. These intricate models have demonstrated extraordinary capabilities in discerning complex patterns and extracting meaningful insights from massive datasets. However, this exponential growth in model complexity, the sheer volume of training data, and the escalating demands for real-time inference have introduced significant computational and memory bottlenecks, creating a critical and urgent need for more efficient hardware and intelligent software solutions.

A pivotal strategy in addressing these burgeoning performance requirements is the adoption of data parallelism, a fundamental computational paradigm that leverages the inherently parallelizable structure of many machine learning workloads. In data-parallel computation, large datasets or input batches are strategically partitioned and distributed across multiple processing elements or computational cores. Each core then independently performs the same set of operations on its assigned subset of the data. This approach enables significant and often linear performance gains, particularly when training or deploying models on modern multi-core Central Processing Units (CPUs), Graphics Processing Units (GPUs), Field-Programmable Gate Arrays (FPGAs), or specialized domain-specific accelerators. The efficacy of data parallelism is intrinsically linked to the underlying architecture of the model and the specific nature of the computations involved, making it a crucial consideration throughout both the algorithmic design and the underlying hardware architecture.

## 1.1 Machine Learning Workloads

Machine learning has rapidly evolved into a cornerstone of modern computing, with diverse models being developed to address an equally diverse set of applications. At the core of this progress lies a family of neural network architectures that, while sharing common computational foundations, differ significantly in structure, dataflow, and workload characteristics. Artificial Neural Networks (ANNs) provide the fundamental building blocks for learning non-linear mappings between inputs and outputs.

Convolutional Neural Networks (CNNs) extend this paradigm to exploit spatial correlations in data such as images and videos, enabling breakthroughs in computer vision. Graph Neural Networks (GNNs) generalize learning to irregular, non-Euclidean domains, capturing relationships across graph-structured data in applications like social networks and recommendation systems. More recently, Transformers have emerged as the dominant architecture for sequence modeling, leveraging attention mechanisms to capture long-range dependencies and powering the latest generation of large language models. Understanding the computational demands and structural differences of these workloads is essential for the design of efficient hardware accelerators, as it provides the foundation for identifying common bottlenecks, opportunities for parallelism, and optimization strategies.

### 1.1.1 Artificial Neural Networks (ANNs)

Artificial Neural Networks (ANNs) are computational models inspired by the structure and function of the human brain. They serve as the foundational architecture in machine learning, particularly in supervised learning tasks such as classification, regression, and pattern recognition. An ANN consists of layers of interconnected processing units (neurons) that transform input data through a series of linear and non-linear operations to produce an output.

ANNs have been successfully applied in every-day applications like *handwritten digit recognition* [7] (e.g., MNIST dataset), *financial forecasting* [8] and *medical diagnosis and risk prediction* [9]. While more complex architectures like CNNs and Transformers have surpassed ANNs in many domains, the simplicity and generality of ANNs continue to make them a useful baseline for both research and hardware benchmarking.

#### Basic Structure and Components

A standard feedforward ANN, as shown in Figure 1.1 comprises three types of layers: (i) the input layer, which receives the raw input features; (ii) the hidden layers, that perform transformations on the input via learned parameters; and (iii) the output layer, producing the final predictions. Each ANN layer consists of one or more artificial neurons, the basic structural unit of ANNs, shown in Figure 1.2. Each neuron in a layer computes a weighted sum of its inputs and applies a non-linear activation function:

$$z = \mathbf{w}^T \mathbf{x} + b, \qquad y = \phi(z) \tag{1.1}$$

Here, $\mathbf{x} \in \mathbb{R}^n$ is the input vector, $\mathbf{w} \in \mathbb{R}^n$ is the weight vector, $b \in \mathbb{R}$ is the bias term, $z$ is the pre-activation value, and $\phi(\cdot)$ is a non-linear activation function such as the Rectified Linear Unit (ReLU), sigmoid or tanh.

The forward pass through an ANN can be expressed using a series of matrix-vector multiplications:

$$\mathbf{y}^{(l)} = \phi\left(\mathbf{W}^{(l)} \mathbf{y}^{(l-1)} + \mathbf{b}^{(l)}\right), \quad l = 1, 2, \ldots, L \tag{1.2}$$

where $L$ is the number of layers, $\mathbf{W}^{(l)}$ and $\mathbf{b}^{(l)}$ are the weights and biases for layer $l$, and $\mathbf{y}^{(l-1)}$ are the activations from the previous layer.

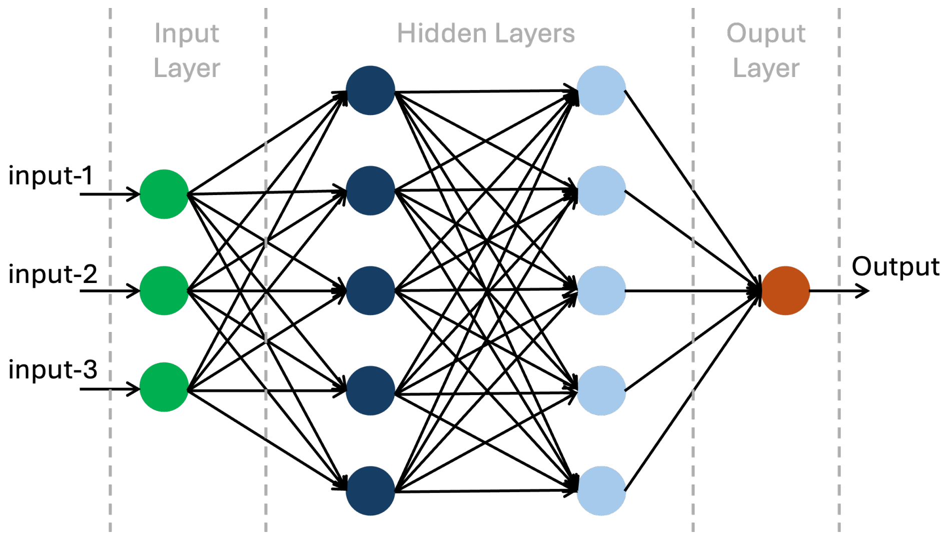


Figure 1.1: Architecture of a basic feedforward artificial neural network with two hidden layers.

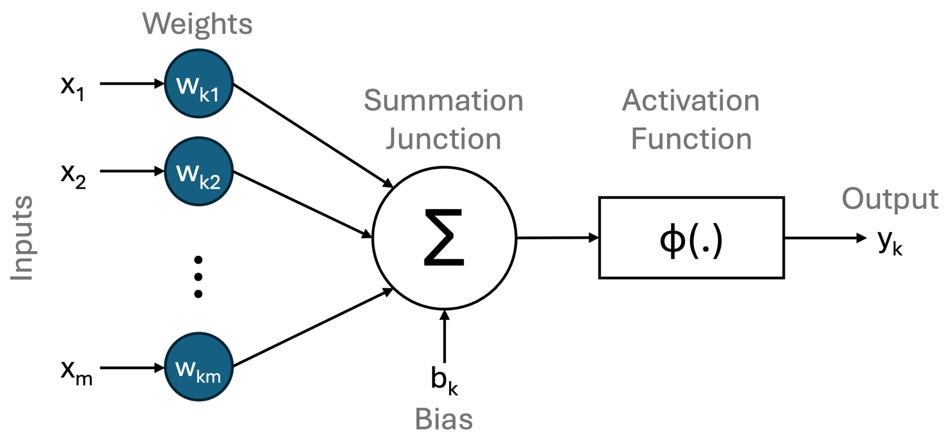


Figure 1.2: Computation performed by a single artificial neuron, including weighted sum and non-linear activation.

### Training with Backpropagation

The process of training an ANN involves adjusting the network's parameters $\{\mathbf{W}^{(l)}, \mathbf{b}^{(l)}\}$ to minimize a loss function $\mathcal{L}$ over a dataset. This is typically done using stochastic gradient descent (SGD) [10] and its variants (e.g., Adam [11], RMSProp [12]), which rely on the backpropagation algorithm. Backpropagation computes gradients of the loss function with respect to each parameter via the chain rule:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(l)}} = \delta^{(l)} \left(\mathbf{y}^{(l-1)}\right)^T \tag{1.3}$$

where $\delta^{(l)}$ is the error signal at layer $l$. The backward pass involves traversing the network in reverse, computing derivatives at each layer, as abstractly depicted in Figure 1.3.

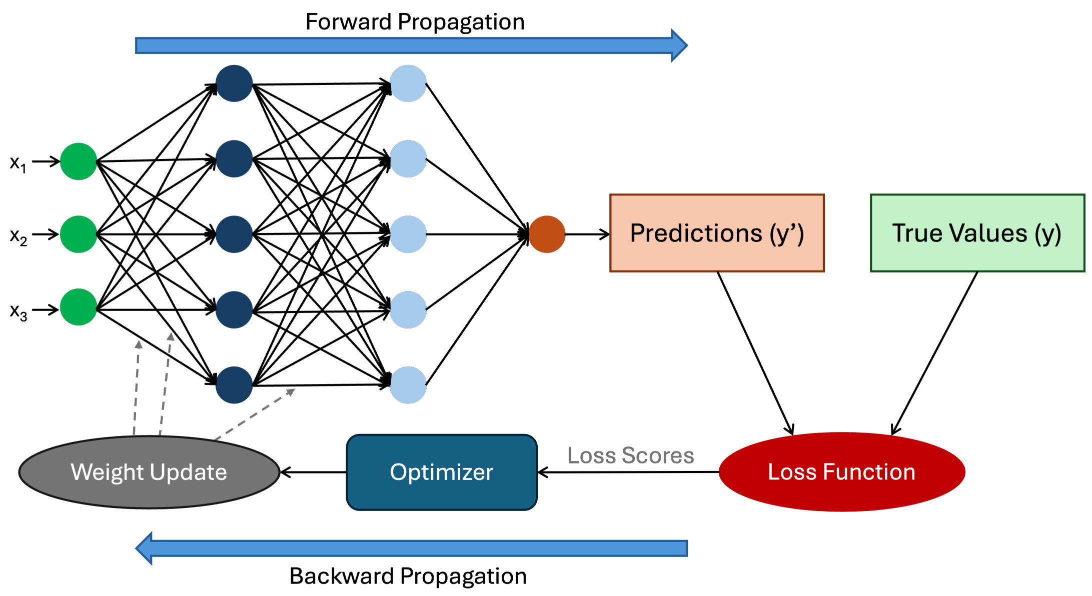


Figure 1.3: Forward and backward propagation in a multi-layer neural network during training.

### Computational Characteristics

Artificial Neural Networks (ANNs) exhibit several computational characteristics that make them particularly well-suited for efficient hardware implementation. Their operations are marked by high arithmetic intensity, primarily dominated by multiply-accumulate (MAC) computations. The mathematical formulation of ANNs relies heavily on dense linear algebra, with matrix-matrix and matrix-vector multiplications forming the backbone of both inference and training stages. Additionally, ANNs inherently support layer-wise parallelism, as neurons within the same layer operate independently. These properties collectively make ANNs ideal candidates for data-parallel architectures such as graphics processing units (GPUs) and tensor processing units (TPUs). The deterministic dataflow and the absence of control-flow divergence within layers further facilitate simplified and efficient hardware accelerator design.

Due to this regular and parallel computation structure, ANNs are also highly amenable to acceleration using specialized architectures, particularly systolic arrays (SAs). SAs are well-suited for efficiently executing the dense matrix operations involved in both the forward and backward passes of neural networks [13]. They offer **high data reuse**, as weight and activation matrices can be streamed through the array and reused, depending on the selected dataflow. Furthermore, SAs provide **predictable and structured data movement**, which supports compiler-level optimizations. Finally, their **scalability** allows them to be adapted to various matrix dimensions and silicon area constraints. However, achieving optimal performance also depends on careful consideration of memory hierarchy design, bandwidth allocation, and support for various activation functions and training modes.

To sum up, ANNs represent the conceptual and computational groundwork upon which modern deep learning architectures are built. Their regular, dense, and parallel computational structure makes them ideal candidates for specialized acceleration, particularly through SAs and other parallel dataflow architectures. Understanding ANNs provides essential context for exploring the hardware challenges and opportunities posed by more advanced ML models.

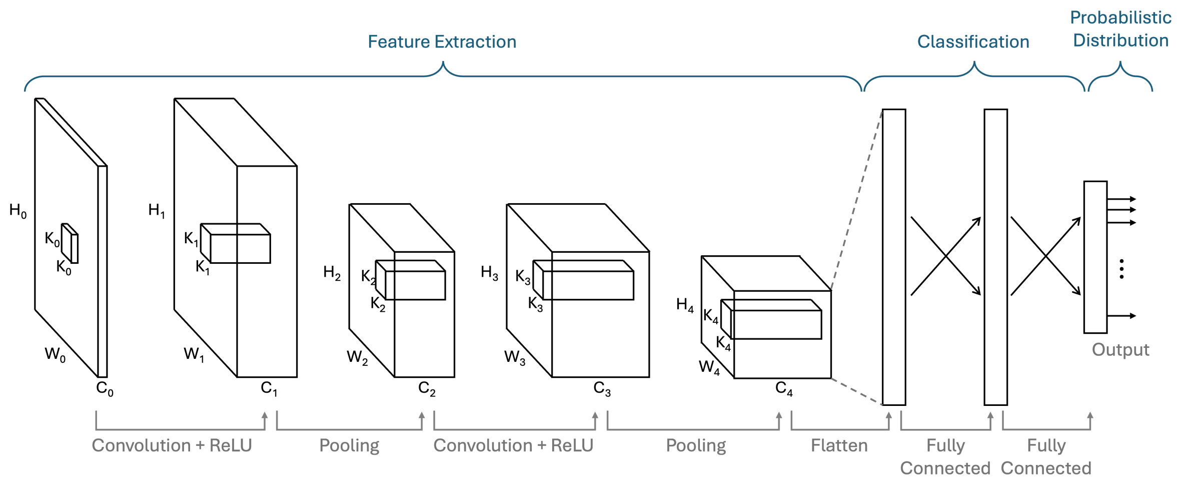


Figure 1.4: A typical CNN architecture for image classification, highlighting key layers and data transformations.

## 1.1.2 Convolutional Neural Networks (CNNs)

Convolutional Neural Networks (CNNs) are a specialized class of neural networks designed to process data with a known grid-like topology, such as images, audio spectrograms, or volumetric data. Originally proposed for image recognition [14], CNNs have become the dominant architecture in computer vision tasks and have also found applications in video analysis, speech recognition, and medical imaging.

Unlike fully connected networks where each neuron is connected to all neurons in the preceding layer, CNNs exploit spatial hierarchies by using local connections and shared weights. This design not only reduces the number of parameters significantly but also makes the network more efficient in learning local patterns such as edges, textures, and shapes.

### Core Components and Operations

A typical CNN architecture is composed of multiple layers including convolutional layers, activation functions, pooling layers, and fully connected layers, as depicted in Figure 1.4. The most distinctive component is the *convolutional layer*, which performs feature extraction through spatial filtering.

At the heart of Convolutional Neural Networks (CNNs) lies the convolution operation, a spatially localized linear transformation that enables the network to extract patterns from structured data such as images, audio, or videos. Unlike fully connected layers that operate globally over the entire input, convolutional layers focus on local receptive fields, enabling the model to detect low-level features like edges and textures, which can later be composed into high-level representations such as shapes and objects.

The core mathematical operation of a 2D convolution can be defined between a 3D input tensor $x \in \mathbb{R}^{H \times W \times C}$ and a bank of convolutional filters $w \in \mathbb{R}^{K \times K \times C \times N}$, where $H$ and $W$ denote the input height and width, $C$ is the number of input channels, $K$ is the spatial size of each kernel (typically 3 or 5), and $N$ is the number of output channels (or filters). The result of this operation is a 3D output tensor

$y \in \mathbb{R}^{H' \times W' \times N}$, representing the learned feature maps. The convolution is computed as follows:

$$y_{i,j,k} = \sum_{m=0}^{K-1} \sum_{n=0}^{K-1} \sum_{l=0}^{C-1} x_{i+m,j+n,l} \cdot w_{m,n,l,k} \tag{1.4}$$

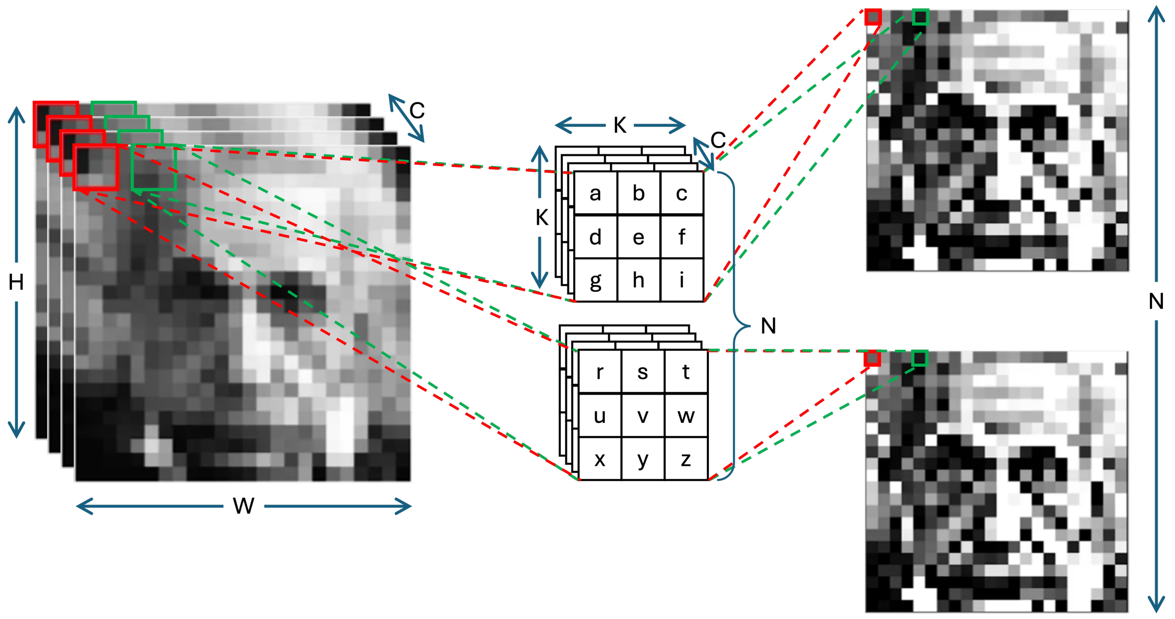


Figure 1.5: Convolution of a 3D input tensor with a learnable kernel. Each filter operates across all channels in the input. At each location, a dot product is computed between the kernel and input patch.

This equation captures the dot product between a local patch of the input and a corresponding filter at every spatial location $(i, j)$, as shown in red or green in Figure 1.5. For each output channel $k$, a distinct kernel $w_{\cdot,\cdot,\cdot,k}$ is used, which spans the entire depth $C$ of the input. The result $y_{i,j,k}$ thus encodes the presence of a particular learned feature in the spatial vicinity of position $(i, j)$.

To intuitively understand this process, imagine a small filter sliding across the input image, computing the weighted sum of local pixel neighborhoods at each location. This sliding mechanism ensures that the same filter parameters are reused throughout the spatial domain of the input, a property known as *parameter sharing*. As a result, convolutional layers dramatically reduce the number of parameters compared to fully connected layers, especially when dealing with high-dimensional data like images. This not only reduces the risk of overfitting but also allows the model to generalize patterns across different spatial locations.

Beyond this linear transformation, convolutional layers are typically followed by a non-linear activation function such as the Rectified Linear Unit (ReLU), defined as $\text{ReLU}(x) = \max(0, x)$. The activation introduces non-linearity into the model, allowing it to learn complex decision boundaries. Without such activation, a deep network would effectively collapse into a single-layer linear model, regardless of depth. The activation function operates element-wise, preserving the spatial structure of the output while enhancing the network's expressive power.

To manage the dimensionality and improve the efficiency of computation, convolutional layers are often interleaved with *pooling layers*, which perform spatial downsampling. The most common types

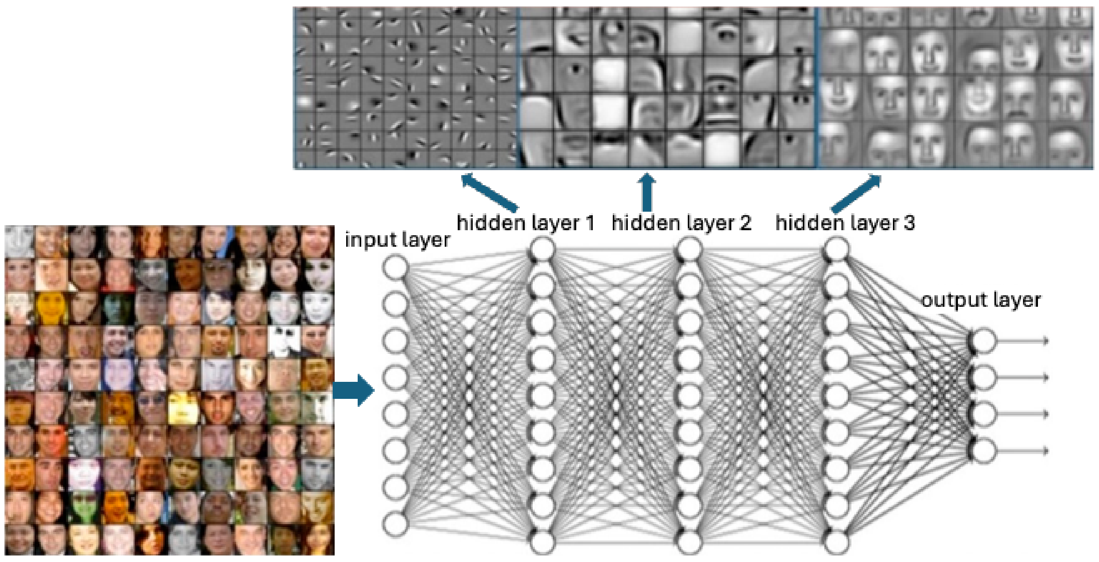


Figure 1.6: Typical processing pipeline in a CNN layer, composed of convolution, activation, and pooling operations.

are max pooling and average pooling, which respectively select the maximum or average value within a local neighborhood. For instance, a $2 \times 2$ max-pooling operation with a stride of 2 reduces the spatial resolution by half in each dimension. Pooling serves multiple purposes: (i) it introduces *translational invariance* to small shifts in the input; (ii) reduces the *number of computations* in deeper layers; and (iii) *mitigates the risk of overfitting* by enforcing spatial smoothness.

The compositionality of CNNs allows for the hierarchical learning of increasingly abstract representations. In early layers, filters may capture simple patterns such as edges or corners, as shown on the top left of Figure 1.6 for the first hidden layer. As data flows deeper into the network, these patterns are combined to detect more complex features, including object parts and entire objects, as depicted in the top middle and right of Figure 1.6 for the second and third hidden layers. This hierarchical feature learning mirrors the visual processing pipeline in biological systems and is a key reason for the success of CNNs in tasks such as image classification, object detection, and semantic segmentation.

### Mapping Convolution to Hardware

To facilitate efficient hardware execution, particularly on architectures optimized for dense matrix operations such as GPUs and Systolic arrays (SAs), the convolution operation is often transformed into a matrix multiplication using a method known as *im2col* [15] (image-to-column). This transformation unfolds the input tensor into a two-dimensional matrix by extracting overlapping patches from the input feature map and arranging them as rows. Specifically, for a convolutional layer with a kernel of size $K \times K$, each local receptive field in the input that the kernel slides over is flattened into a row vector, shown in blue in Figure 1.7. Consequently, an input tensor of shape $H \times W \times C$ is transformed into a matrix of shape $H' \cdot W' \times K^2 \cdot C$, where $H'$ and $W'$ are the spatial dimensions of the output feature map.

Simultaneously, the convolutional filters, originally represented as a 4D tensor of shape $K \times K \times C \times N$, are reshaped into a matrix of shape $K^2 \cdot C \times N$, where each column corresponds to a flattened kernel, shown in brown in Figure 1.7. The convolution operation is then performed as a general matrix

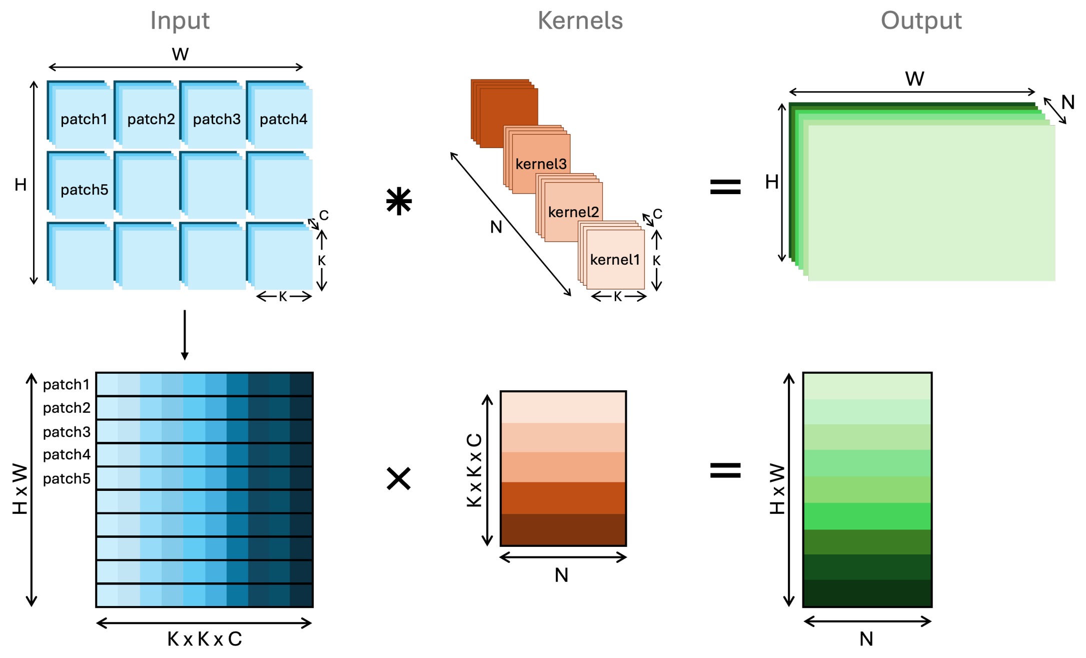


Figure 1.7: Transformation of convolution into matrix multiplication using im2col [15] and GEMM, facilitating hardware acceleration.

multiplication (GEMM) between the unfolded input and the reshaped filter matrix, yielding an output matrix of shape $H' \cdot W' \times N$. This result is subsequently reshaped into the final output tensor $y \in \mathbb{R}^{H' \times W' \times N}$. While the *im2col* approach introduces memory redundancy due to overlapping patches, it allows convolutional computations to leverage highly optimized GEMM libraries and enhances data reuse, memory coalescing, and computational throughput on modern hardware accelerators.

Together, the convolution, activation, and pooling operations form the backbone of CNN-based models, enabling them to extract and propagate discriminative features through multiple layers of abstraction.

Overall, CNNs have established themselves as a cornerstone of modern deep learning, particularly in domains involving spatially structured data. Their core operations (convolution, ReLU activation and pooling) are highly parallel and data-reuse friendly. These traits make CNNs an excellent match for data-parallel hardware accelerators, including systolic arrays, which can exploit their regular dataflow and computational structure for the convolution, which is the most computational intensive operation.

### 1.1.3 Graph Neural Networks (GNNs)

Graph-structured data naturally arise in numerous domains, such as social networks, molecular chemistry, knowledge graphs, and recommendation systems [16–19]. Unlike regular grid-based data structures like images or sequences, graphs are inherently non-Euclidean and irregular: the number of neighbors for each node varies, node connectivity is sparse, and there is no canonical ordering of nodes. Traditional deep learning architectures, such as Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs), struggle to directly operate on such structures. To address this challenge, Graph Neural Networks (GNNs) have emerged as a powerful class of models capable of learning over graph data by exploiting both node features and graph topology.

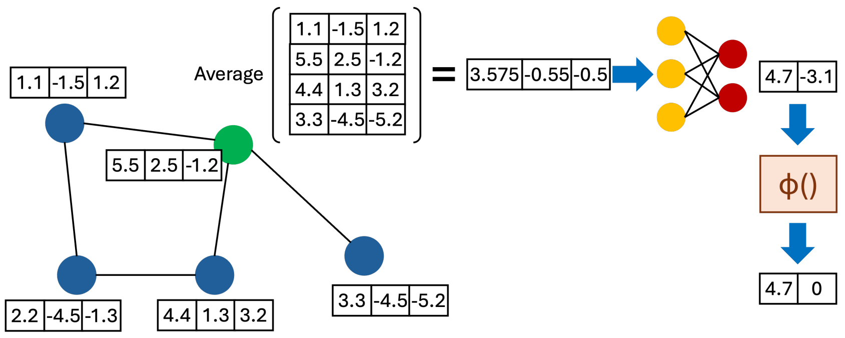


Figure 1.8: Example of the computations took place in a GCN layer.

GNNs extend the principles of neural computation to graphs through iterative message passing, a framework in which node representations (or embeddings) are updated by aggregating information from their neighbors. Formally, the computation in a single GNN layer for node $v$ at layer $k$ can be described as follows:

$$h_v^{(k)} = \text{UPDATE}^{(k)} \left( h_v^{(k-1)}, \text{AGGREGATE}^{(k)} \left( h_u^{(k-1)} : u \in \mathcal{N}(v) \right) \right) \tag{1.5}$$

where $h_v^{(k)}$ is the feature representation of node $v$ at the $k$-th layer, $\mathcal{N}(v)$ denotes the set of neighboring nodes of $v$, and the AGGREGATE and UPDATE functions define how neighborhood information is collected and integrated. Common aggregation functions include element-wise mean, sum, max, and attention-weighted sums, while update functions typically involve simple nonlinear transformations such as feed-forward layers with ReLU activation.

### Variants and Architectures

Several architectures fall under the umbrella of GNNs, differing primarily in how they perform the aggregation and update steps. Graph Convolutional Networks (GCNs), for instance, use a spectral or spatial interpretation of convolution to aggregate normalized features from immediate neighbors. Graph Attention Networks (GATs) introduce attention mechanisms to learn importance weights for each neighbor dynamically. GraphSAGE samples fixed-size neighborhoods and aggregates them using learnable functions, facilitating scalable training on large graphs. Other models, like Gated Graph Neural Networks (GGNNs) and Message Passing Neural Networks (MPNNs), introduce recurrent and more expressive update mechanisms to capture temporal or multi-step dependencies.

Figure 1.8 illustrates a simple example of the forward propagation process for one layer of a Graph Convolutional Network. Next to each node, its initial feature vector is shown. Let's consider the computations for the green node. We calculate the average of its own features and the features of its neighbors. This average is then fed into a neural network, which in our case consists of a fully-connected layer, to obtain the output feature vector of the green node. A non-linear activation function $\phi()$, such as ReLU or softmax(Eq. 1.6), is applied before the final result is obtained. This same process is repeated for all nodes using the same neural network with the same weights.

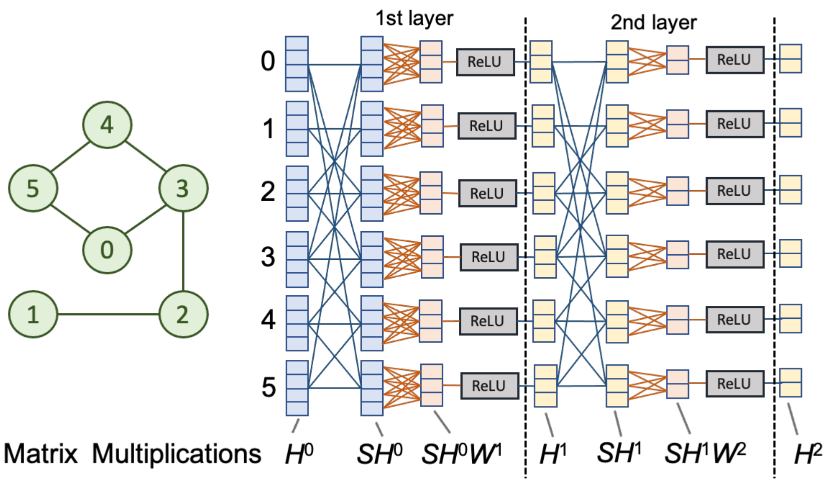


Figure 1.9: Abstract diagram of a two-layer Graph Convolutional Network

$$softmax_i(z) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \tag{1.6}$$

To construct deeper Graph Convolutional Networks, multiple layers like the one described above can be stacked sequentially. In this case, the output features of one layer serve as the input features for the next layer.

Figure 1.9 shows the overall high-level diagram of a two-layer Graph Convolutional Network. Here, $H^{(l)}$ represents the feature matrix of the $l$-th layer, $A$ is the adjacency matrix of the graph indicating the neighbors of each node whose features should be aggregated, and $W^{(l)}$ is the weight matrix of the neural network used in the $l$-th layer.

To compute the sum of the feature vectors of each node's neighbors, it suffices to multiply the adjacency matrix $A$ of the graph by the feature matrix $H$. In the resulting matrix from the product $AH$, each row corresponds to the sum of the feature vectors of the neighbors of the respective node. To include the feature vector of the node itself in the sum, we add the identity matrix $I$ to $A$ before computing the product. Thus, we have the product $\tilde{A}H$, where $\tilde{A} = A + I$

However, we aim to calculate the average of the feature vectors to feed into the neural network. To achieve this, we need to normalize the product $\tilde{A}H$ with respect to the degree of the nodes. This normalization can be done by multiplying with the inverse of the diagonal degree matrix $D$, where each element on its main diagonal contains the degree of the corresponding node. Kipf and Welling in [20] apply a symmetric normalization of $\tilde{A}$ with respect to its rows and columns, essentially multiplying by $D^{-\frac{1}{2}}$ twice. Therefore, we arrive at computing the average of the feature vectors of each node with the product $D^{-\frac{1}{2}}\tilde{A}HD^{-\frac{1}{2}}$

Any modification and normalization of the adjacency matrix A can be performed in a preprocessing stage. For simplicity, therefore, in the remainder of this thesis, when we refer to the adjacency matrix $A$, we will mean the normalized adjacency matrix. The equation 1.5 for one layer of a Graph Convolutional Network is then written as:

$$H^{(l+1)} = \phi\left(AH^{(l)}W^{(l)}\right) \tag{1.7}$$

Where $l$ is the layer number, $\phi()$ is a non-linear activation function, A is the normalized adjacency matrix (i.e., $D^{-\frac{1}{2}}\tilde{A}HD^{-\frac{1}{2}}$), $H^{(l)}$ is the input feature matrix, $H^{(l+1)}$ is the output feature matrix, and $W^{(l)}$ is the weight matrix of the neural network used in the $l$-th layer.

#### Computational Characteristics

Despite their conceptual elegance, GNNs present unique computational challenges due to the irregular structure of graph data. Unlike CNNs, which benefit from dense and regular memory access patterns and highly parallelizable computations, GNNs operate on sparse and dynamic structures. The key computational challenges include (i) **irregular memory accesses**, since each node reads from a variable-sized set of neighbors, leading to non-coalesced and scattered memory access patterns; (ii) **sparse data structures** to represent adjacency matrices, which require indirect indexing and result in underutilization of compute resources; and (iii) **dynamic neighborhood aggregation**, because the number and identity of neighbors differ across nodes, making parallelism difficult and necessitating dynamic scheduling.

These characteristics hinder the efficient execution of GNNs on traditional hardware accelerators such as GPUs and TPUs, which are optimized for regular, high-throughput dense matrix operations. Sparse matrix multiplications (SpMM) involved in GNNs result in low arithmetic intensity and irregular compute-to-memory ratios, reducing hardware utilization.

Overall, GNNs have demonstrated state-of-the-art performance in diverse tasks such as node classification, link prediction, graph classification, and recommendation systems [18, 19]. In molecular biology [17], GNNs are used to predict molecular properties and simulate protein interactions. In social networks [16], they enable user profiling and community detection. As the scale of graph data continues to grow, the development of scalable and hardware-efficient GNN models is becoming increasingly important.

### 1.1.4 Transformers and Large Language Models (LLMs)

Transformers have become the cornerstone architecture for a wide array of modern machine learning tasks, particularly in natural language processing (NLP), computer vision and, more recently, multimodal and scientific domains. Originally introduced by Vaswani et al. in *"Attention Is All You Need"* [21], the Transformer architecture discards recurrence and convolution in favor of a pure attention mechanism that allows modeling long-range dependencies with greater computational efficiency and parallelism. Transformers form the foundation of large language models (LLMs), which scale this architecture to billions or even trillions of parameters, unlocking emergent capabilities in understanding, generation, reasoning and translation.

#### Transformer Architecture Overview

At its core, the Transformer architecture is composed of an encoder-decoder structure, as shown in Figure 1.10, although most modern LLMs (e.g., BERT, GPT) utilize only the encoder or decoder. The

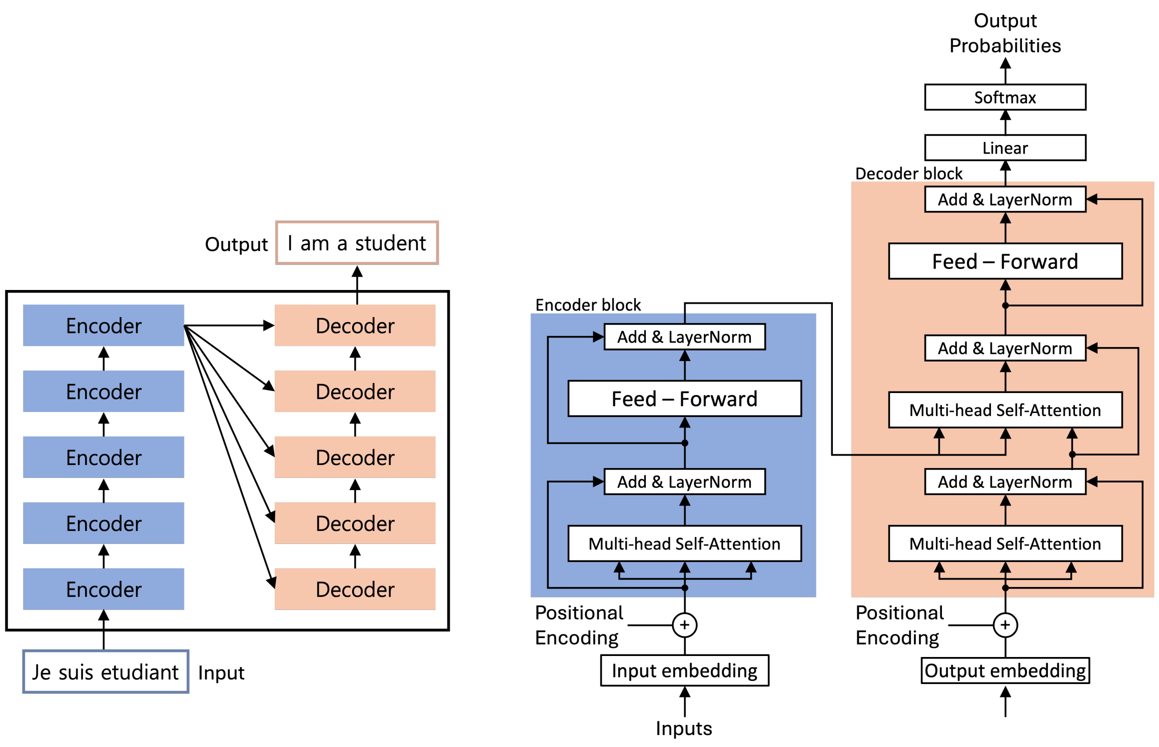


Figure 1.10: The encoder-decoder structure originally used in Transformers.

encoder processes the input sequence to generate contextual embeddings, while the decoder generates an output sequence by attending to both the previously generated tokens and the encoder's outputs.

Each Transformer block consists of two main sublayers, highlighted in Figure 1.11: multi-head self-attention and a position-wise fully connected feedforward network. Residual connections and layer normalization are applied around each sublayer to stabilize training. Given an input sequence represented as a matrix $X \in \mathbb{R}^{n \times d}$, where $n$ is the sequence length and $d$ is the hidden dimensionality, the self-attention mechanism computes the output as:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right) V \tag{1.8}$$

where the input $X$ is linearly projected into queries $Q$, keys $K$, and values $V$. The dot-product attention weights determine how much focus each token should have on every other token in the sequence. Multi-head attention allows the model to jointly attend to information from different representation subspaces.

The feedforward sublayer applies two linear transformations with a non-linearity (typically GELU) in between:

$$\text{FFN}(x) = \text{GELU}(xW_1 + b_1)W_2 + b_2. \tag{1.9}$$

To preserve the order of the sequence, Transformers add positional encodings to the input embeddings, which can be either learned or based on fixed sinusoidal functions.

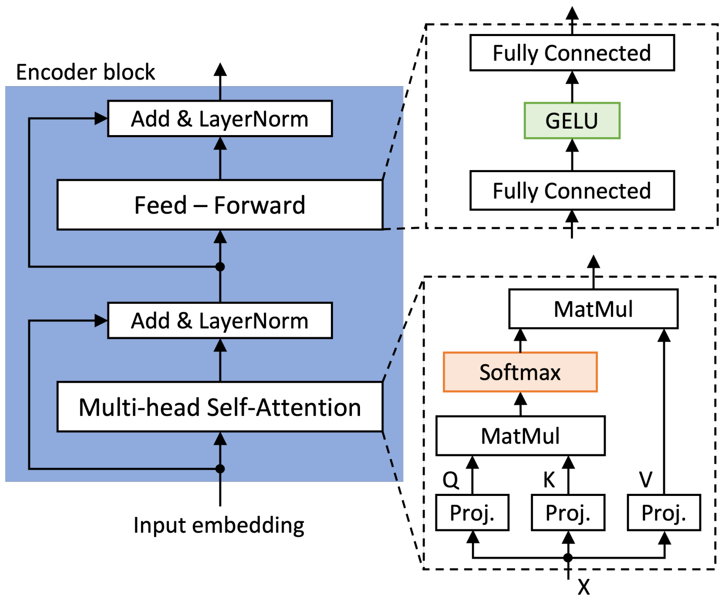


Figure 1.11: The Transformer architecture showing multi-head self-attention, position-wise feedforward layers, and residual connections.

### Large Language Models (LLMs)

Large language models extend the Transformer architecture by massively scaling the number of parameters, training data, and compute budgets. Examples include GPT-4 [22], PaLM [23] and LLaMA [24]. These models are typically trained on internet-scale corpora using self-supervised learning, where the objective is to predict the next token given previous ones (causal language modeling) or to fill in masked tokens (masked language modeling).

LLMs exhibit emergent properties that are not evident in smaller models, such as few-shot and zero-shot learning, code generation, and instruction following. These capabilities arise due to the combination of large-scale training and the expressive power of attention-based architectures. As a result, LLMs have revolutionized tasks like translation, summarization, question answering, and more.

### Computational Characteristics and Hardware Acceleration

The computational workload of Transformers is dominated by matrix multiplications, particularly in the attention and feedforward layers. The self-attention mechanism has quadratic complexity with respect to sequence length, i.e., $O(n^2 \cdot d)$, due to the pairwise interactions between all tokens. Feedforward layers, which often use intermediate dimensionalities 2–4x larger than the input, further increase arithmetic intensity.

Despite their regular dataflow and high parallelism potential, Transformers also pose several hardware challenges. First, they require **high memory bandwidth**, due to their large intermediate activations and gradient storage. Transformers also have **limited data reuse**, especially in self-attention where queries, keys, and values are dynamically computed.

For efficient inference and training, techniques such as operator fusion, activation quantization, weight pruning, and memory-efficient attention mechanisms have been proposed [25–27]. Additionally, distributed training and model parallelism are crucial to scaling LLMs, as single-device training becomes infeasible.

Transformers are generally more amenable to hardware acceleration compared to GNNs due to their dense and regular structure. Their reliance on large-scale matrix operations makes them ideal candidates for systolic array-based accelerators like TPUs and custom ASICs. The attention and feedforward layers map naturally to matrix-matrix multiplications, and batching across tokens allows for highly parallel execution. However, challenges remain in optimizing memory movement, handling variable sequence lengths, and supporting dynamic computation paths (e.g., during beam search).

As LLMs continue to grow in size and capability, several trends are shaping their future development. These include efficient training strategies (e.g., mixture of experts), sparse attention mechanisms, model compression for on-device inference, and multimodal extensions that incorporate images, audio, and code. Hardware-aware algorithm design, where models are tailored to the underlying accelerator capabilities, will be critical to ensure the sustainability and accessibility of LLMs.

## 1.2 Application Characteristics Guiding Machine Learning Accelerator Design

The design of hardware accelerators for machine learning (ML) is fundamentally shaped by specific application characteristics that influence computational patterns, memory access behaviors, and data precision requirements. Among these, quantization plays a pivotal role in optimizing the trade-off between performance and accuracy. **Quantization** involves mapping high-precision data, such as 32-bit floating point numbers, into lower-precision representations like 8-bit or even binary formats. This significantly reduces memory bandwidth requirements and allows more efficient utilization of on-chip resources such as SRAM and register files. Moreover, it enables simpler and smaller arithmetic units, directly impacting energy efficiency and area cost. For models like CNNs and Transformers, which involve large-scale matrix operations, quantization not only improves throughput but also allows hardware reuse across different layers and architectures.

Closely related to quantization is the use of reduced precision floating-point arithmetic. Many modern accelerators support mixed-precision computing, such as FP16 or bfloat16 [28], which offer a compromise between numeric range and computational efficiency. Reduced precision is especially effective in training or inference phases where exact arithmetic is less critical, and statistical properties of neural networks enable resilience to small perturbations in values. Architectures such as NVIDIA's Tensor Cores [29] and Google's TPU [30] are explicitly designed to exploit such numeric formats. For GCNs, where vertex-level computations can be bandwidth-bound, reduced precision can significantly alleviate bottlenecks in memory hierarchy, especially when paired with aggressive caching strategies.

Data **sparsity**, both in activations and weights, is another key application characteristic that guides architectural decisions. In many ML models, especially pruned CNNs [31–33] or attention layers in Transformers [34, 35], a large fraction of weights or activations are zero. Specialized hardware units, such as sparse matrix multipliers or zero-skipping execution engines [36–39], are therefore designed to exploit this sparsity by avoiding unnecessary computations. For instance, accelerators like the SCNN(Sparse CNN Accelerator) [39] leverage compressed storage formats and dedicated scheduling mechanisms to bypass zero-valued data, significantly reducing energy consumption and improving throughput. This could be particularly beneficial in GCNs, where graph sparsity at the adjacency matrix level can be exploited to optimize memory access and compute workloads.

Matrix multiplication, often in the form of General Matrix Multiplication (GEMM), lies at the heart of virtually all deep learning models. Convolutional layers in CNNs are typically reformulated as matrix multiplications, and Transformers rely heavily on matrix multiplications for attention mechanisms and feedforward layers. This centrality makes matrix multiplication operators a cornerstone of ML accelerator design. Efficient implementation of GEMM operations involves data tiling, double buffering, and systolic array architectures to maximize compute density and minimize latency. For example, systolic arrays, as used in Google's TPU [30, 40], allow high throughput at low power by orchestrating data movement in a pipelined manner, thus exploiting spatial locality and reducing control overhead.

The **evaluation of non-linear functions**, such as ReLU, sigmoid, tanh, softmax and GELU, is also a critical consideration in ML accelerator design. These functions are computationally intensive when implemented naively, especially for activation-heavy architectures like CNNs or the multi-head attention mechanisms in Transformers. Hardware accelerators address this by employing approximation techniques such as piecewise linear interpolation, lookup tables (LUTs), or specialized approximation circuits like CORDIC units. These methods provide a balance between accuracy and computational efficiency, ensuring that non-linear transformations do not become bottlenecks in deep pipelines.

**Memory hierarchy** and **bandwidth** play a defining role in shaping ML accelerator performance. With the increasing size of ML models, especially Transformers with hundreds of millions of parameters, memory traffic becomes a dominant concern. Application-aware memory systems leverage data reuse patterns, such as weight sharing in CNNs or key-value caching in Transformers, to design multi-level memory hierarchies, including local scratchpads, eDRAM buffers, and DMA-controlled DRAM access. These systems are tightly coupled with scheduling algorithms that manage dataflow and workload partitioning across processing elements, ensuring high utilization and minimizing idle cycles.

Another increasingly important characteristic is **workload heterogeneity**, especially in emerging models like GCNs, where computation varies across graph nodes and edges. Unlike CNNs or Transformers with regular computation graphs, GCNs introduce irregular data access patterns [18, 20] due to their graph-structured nature. This irregularity poses challenges for SIMD or systolic architectures, which rely on uniform operations. To address this, modern accelerators incorporate flexible execution units, such as MIMD cores or domain-specific processors with runtime-adaptable pipelines. These designs support efficient execution of irregular workloads while still maintaining acceleration benefits for regular operations.

Lastly, **programmability** and **scalability** are critical for supporting evolving ML workloads. With the rapid pace of innovation in model architectures, accelerators must support a wide range of operations and data types while allowing reconfiguration and future-proofing. This has led to the development of hardware-software co-design frameworks, where compiler-level optimizations inform hardware scheduling, tiling, and instruction generation. Architectures such as VTA(Versatile Tensor Accelerator) [41] exemplify this trend, providing a flexible ISA and programmable micro-ops to adapt to different models efficiently. As such, application characteristics not only drive hardware microarchitecture but also influence the surrounding software ecosystem, shaping the overall accelerator stack.

## 1.3 Thesis Contribution

This thesis makes several contributions toward improving the performance, energy efficiency, and robustness of systolic and tensor array accelerators for ML inference. We first investigate architectural and physical layout optimizations for classical dense systolic arrays. The thesis introduces *ArrayFlex*, a novel architecture that supports configurable transparent pipelining. Unlike rigidly pipelined arrays, ArrayFlex allows pipeline depth to be dynamically adjusted across convolutional layers. This flexibility minimizes overall execution time by balancing cycle count reduction with modest increases in clock period, ultimately reducing energy consumption while incurring negligible hardware overhead. ArrayFlex demonstrates that adaptive micro-architectural techniques can significantly improve systolic array utilization across diverse workloads.

Complementary to architectural pipelining, this work also introduces a physical-level optimization: asymmetric floorplanning of processing elements (PEs). Traditional systolic arrays use a symmetric floorplan that does not reflect the inherent asymmetry of horizontal versus vertical data movement. By tailoring PE aspect ratios to the relative width and switching activity of interconnects in these two directions, the proposed methodology reduces wirelength and interconnect power without altering computational throughput. Post-layout implementations confirm measurable energy savings, underscoring the importance of physical design awareness in accelerator optimization.

The thesis further explores methods to reduce dynamic power by minimizing switching activity within systolic arrays. Techniques such as dynamic input encoding and weight reordering are proposed to exploit workload characteristics and reduce unnecessary transitions in datapaths. These approaches deliver notable power savings with minimal hardware overhead, complementing layout-level optimizations and demonstrating that data-activity reduction can be systematically integrated into accelerator design.

Modern CNNs often employ pruning to introduce structured sparsity in their weights, reducing both storage and compute requirements. To efficiently exploit this property, we propose *DeMM*, a disaggregated matrix multiplication engine that supports relaxed structured sparsity patterns while maintaining hardware regularity. By decoupling memory elements from multiply-accumulate units, DeMM provides flexibility in handling fine-grained sparsity and delivers substantial latency reductions and power savings compared to state-of-the-art structured-sparse architectures. This contribution highlights the effectiveness of disaggregation in supporting diverse sparsity formats.

In addition to DeMM, we propose *DSSTA*, a sparse systolic tensor array that eliminates the need for skewed input scheduling. By adopting a diagonal and permuted-weight dataflow, DSSTA removes shift registers and reduces wirelength while maintaining synchronization across PEs. Experimental evaluations show significant reductions in area, power, and interconnect cost, along with latency improvements. This demonstrates that rethinking dataflow at the tensor level can yield both simpler and more efficient hardware.

Recognizing the increasing importance of reliability in ML accelerators, particularly in safety-critical domains, this thesis introduces two complementary fault-tolerance schemes for sparse systolic tensor arrays. First, we adapt Algorithm-Based Fault Tolerance (ABFT) to structured sparsity by incorporating checksum logic along the periphery of the array and leveraging digit-serial arithmetic for efficient checksum accumulation. This enables online error detection with low area and power overhead. Second, we propose a periodic online testing methodology that proactively detects permanent faults before execution begins, using only a small set of test vectors and reusing stored model weights. Together, these techniques ensure robust operation of sparse tensor accelerators with minimal overhead.

The thesis also extends systolic array design to irregular graph-based workloads by introducing *FusedGCN*, the first systolic architecture capable of fusing three-matrix multiplication into a single operation. Unlike state-of-the-art accelerators that treat aggregation and combination as separate phases, FusedGCN executes the entire graph convolution in a fused manner, supporting both sparse adjacency matrices and potentially sparse input features. Evaluation shows that FusedGCN reduces inference latency relative to existing designs while requiring only modest hardware overhead, thereby providing a scalable solution for accelerating GCNs.

To address reliability in graph convolution, we further propose *GCN-ABFT*, a fault-tolerance scheme tailored to the three-matrix product structure of GCNs. Instead of verifying each multiplication separately, GCN-ABFT computes a fused checksum for the entire operation. This reduces verification cost by over 20% compared to baseline ABFT and achieves strong fault detection with negligible performance impact. Experimental fault injection confirms its robustness, making it a practical mechanism for error detection in GCN inference pipelines.

Finally, we propose a SIMD-based softmax hardware unit that supports both full-vector and segmented softmax computations. The intermediate results from segmented operations are efficiently reused to evaluate various activation functions by mapping them to softmax. This unified approach allows a single unit to implement multiple nonlinear functions, reducing hardware redundancy and complexity while improving efficiency in Transformer inference.

## 1.4 Thesis Organization

The remainder of this thesis is organized as follows:

**Chapter 2** presents physical-design-aware microarchitectural optimizations for systolic array architectures aimed at reducing power consumption. Specifically, it introduces ArrayFlex, a systolic array with a configurable pipeline that selects the optimal pipeline depth for each CNN layer to minimize inference latency and power usage. In addition, an asymmetric physical layout for systolic arrays is proposed to further enhance energy efficiency.

**Chapter 3** introduces an SA architecture that employs application-aware dynamic encoding and offline weight reordering to reduce the switching activity of the data entering the systolic array. By exploiting salient attributes of state-of-the-art CNNs, such as the value distribution of the weights, the proposed SA applies bus-invert coding and zero-value clock gating, only to the data that exhibits high switching activity.

**Chapter 4** presents two new sparse SA architectures: DeMM and DSSTA. Both architectures are tailored to operate on structured block sparse data. More specifically, DeMM ecouples the memory part of the systolic array from the multiply-add units, aiming to accelerate CNN inference on models with relaxed structured sparsity. While, DSSTA is a structured sparse systolic tensor array with diagonal flow and permuted weights, which eliminates the need for input skewing by adopting a novel dataflow that ensures synchronized computation with reduced routing complexity.
**Chapter 5** focuses on fault-tolerance in sparse systolic arrays. Two novel testing mechanisms are presented. A low-cost ABFT methodology tailored for sparse systolic tensor arrays and a periodic online testing mechanism that relies on functional test vectors.
**Chapter 6** discusses GCN specific optimizations: (a) FusedGCN, which is a fused triple matrix multiplication systolic architecture, that reduces execution times compared to state-of-the-art architectures for computing representative GCN application; (b) GCN-ABFT, a fault-tolerant methodology, that directly calculates a checksum for the entire three-matrix product within a single GCN layer, providing a cost-effective approach for error detection in GCN accelerators.
**Chapter 7** introduces a novel SIMD-based softmax hardware unit that supports both full-vector and segmented computations, reusing intermediate results to evaluate multiple activation functions by mapping them to softmax.
Finally, a summary of our work including also its future research aspects is covered in **Chapter 8**.

# Chapter 2

# Microarchitecture and Physical Design Optimization of Systolic Arrays

Matrix multiplications are at the heart of deep learning algorithms and, in hardware, they naturally map onto Systolic Arrays (SA) [42]. Systolic arrays have a long history of wide applicability [42], while, recently, the SA paradigm has regained interest due to the large volume of rapidly emerging machine learning applications. Tensor Processing Units (TPUs) [40] and other related architectures [43–46] are characteristic examples of newly designed SA architectures/derivatives.

The typical SA hardware structure consists of an array of Processing Elements (PEs), as depicted in Figure 2.1. Each PE consists of a multiplier and an adder and necessary registers to appropriately pipeline the streaming operation, regardless of the selected dataflow. The SA is fed by local memory banks placed on the West and North edges of the array, while the output results are collected on the South or East edge, depending on the employed dataflow. When the sizes of the matrices are larger than the size of the SA, matrix multiplication is executed in tiles, where the size of each tile (i.e., sub-matrix) matches the size of the SA.

The *dataflow* type employed by the SA determines the internal structure of the PEs and how the matrix multiplication, $A \times B$, is executed. Specifically, in so called *weight-stationary* dataflow implementations, matrix $B$ (the 'weights') is pre-loaded in the SA, while matrix $A$ (the 'inputs') is fed into the SA from the West side, as shown in Figure 2.1(a). The results are accumulated at the South end of each column. In this setup, matrix $B$ is stationary. On the other hand, in *input-stationary* dataflow implementations, matrix $A$ is pre-loaded in the SA, while matrix $B$ is fed into the SA from the North side, as shown in Figure 2.1(b). The results are accumulated at the East end of each row. In this setup, matrix $A$ is stationary. Finally, in *output-stationary* dataflow implementations, the matrices $A$ and $B$ arrive synchronously from the West and North borders of the SA, respectively, and the result is accumulated locally within each PE, as illustrated in Figure 2.1(c). Here, it is the result (the 'outputs') that remains stationary. To increase applicability, flexibility, and PE utilization, configurable SAs can support various dataflow types [47–49].

Assuming a WS dataflow, as shown in Figure 2.1(a), matrix $B$ is first pre-loaded in the SA by loading a new row per cycle. Thus, $R$ cycles are needed to complete the loading. Once $B$ is loaded, matrix $A$ is streamed in from the left edge of the array. The first arriving element would reach the rightmost column of the SA after $C - 1$ cycles. After the top row is filled with the incoming data elements, it takes $R - 1$

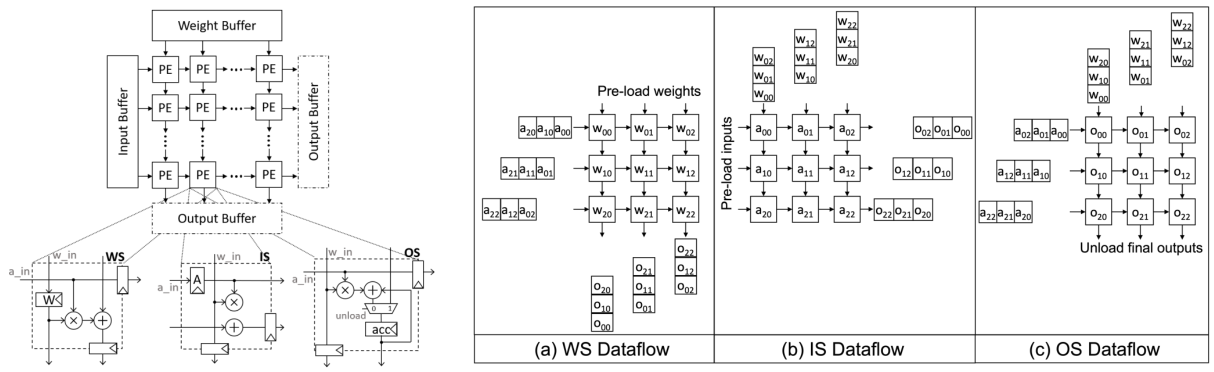


Figure 2.1: A systolic array architecture, the organization of the processing elements under Weight-Stationary (WS), Input-Stationary (IS), and Output-Stationary (OS) and their respective dataflows.

cycles to reduce the result of all PEs of the same column. For the reduction operation, the result of the multiplication and addition in each PE is first registered at the borders of the PE and it then moves downwards to the next PE of the same column. The SA becomes empty when the reduction is finished on the rightmost column for all incoming skewed columns of $A$. Since $A$ consists of $T$ rows, the overall latency $L$ of computing matrix multiplication equals:

$$L = 2R + C + T - 2 \tag{2.1}$$

When the size of matrices $A$ and $B$ is larger than the size of the SA, i.e., $N > R$ and/or $M > C$, matrix multiplication is executed in tiles, where each tile (sub-matrix) matches the size of the SA. The partial sums for each tile that reach the bottom of the SA, get accumulated into the corresponding output accumulator located below the SA. According to [43], the latency of tiled matrix multiplication is equal to

$$L_{\text{total}} = L \times \left\lceil \frac{N}{R} \right\rceil \times \left\lceil \frac{M}{C} \right\rceil . \tag{2.2}$$

$L$ is the latency for computing the product of two tiles $A^{\text{sub}}_{T,R} \times B^{\text{sub}}_{R,C}$ and it is given by (2.1), and $\lceil \frac{N}{R} \rceil \times \lceil \frac{M}{C} \rceil$ represents the total number of tiles.

Despite their advantages, the practical deployment of systolic arrays is constrained by several design challenges that stem from the interaction between logical architecture and physical implementation. On the micro-architectural side, the choice of pipelining strategy directly affects the achievable operating frequency, energy efficiency, and overall scalability of the array. Rigid pipelining structures can limit adaptability across different workloads and process technologies. On the physical design side, the spatial arrangement of processing elements (PEs) and their interconnects significantly impacts wirelength, switching activity, and ultimately power consumption. Conventional symmetric floorplanning, while simple to implement, may not align with the inherently asymmetric dataflows of common ML workloads, leading to inefficiencies in interconnect power and area utilization.

The growing demand for flexible, high-performance accelerators has motivated research into layout optimizations that span both the architectural and physical design levels. From the architectural perspective, innovations such as configurable transparent pipelining aim to provide adaptive control over pipeline depth, enabling systolic arrays to trade off frequency, latency, and energy consumption depending on application needs or technology constraints. From the physical design perspective, asymmetric floorplanning techniques optimize the aspect ratio of PEs in accordance with workload-induced dataflow asymmetries, reducing interconnect wirelength and dynamic power without altering the logical function of the array.

These complementary approaches underscore the importance of cross-layer co-design in the evolution of systolic arrays. By simultaneously considering the architectural flexibility of the compute fabric and the efficiency of its physical layout, it is possible to significantly improve the performance, energy efficiency, and scalability of ML accelerators. The remainder of this chapter explores these two lines of research in detail, highlighting how micro-architectural adaptability and physical-level optimizations can be leveraged to advance the state of systolic array design.

## 2.1 ArrayFlex: A Systolic Array Architecture with Configurable Transparent Pipelining

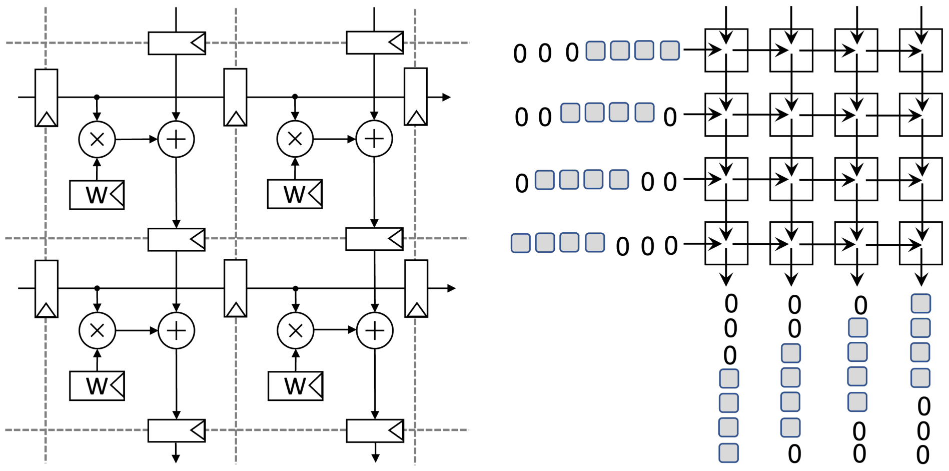
In this section, we focus on customizing the *pipeline* structure of SAs with the goal being to reduce the execution latency of matrix multiplication. Throughput can always be increased by adding more processing elements and increasing the input and output bandwidth of the SA. On the contrary, optimizing the execution *latency* requires architectural re-organization of the SA that should also adjust to the structure of each CNN layer. Reducing latency is important in applications executed at the edge [50] and a necessity for applications that also require real-time responses [51]. Moreover, for small batch sizes, reducing the latency can also reduce the time to the final result. This is critical in RNNs, which are harder to batch than CNNs [52].

The proposed SA architecture, named *ArrayFlex*, can configure its pipeline structure between normal and various shallow pipeline depths. In shallow mode, two or more adjacent pipeline stages are joined by bypassing intermediate pipeline stage(s) [53, 54]. This merging effectively reduces the number of cycles needed to complete matrix multiplication. On the other hand, the clock frequency is reduced to avoid timing violations due to the increased logic depth. This double-faceted tradeoff allows us to identify the best possible configuration per CNN layer that minimizes the total execution latency in absolute time. Overall, the contributions of this architecture can be summarized as follows:

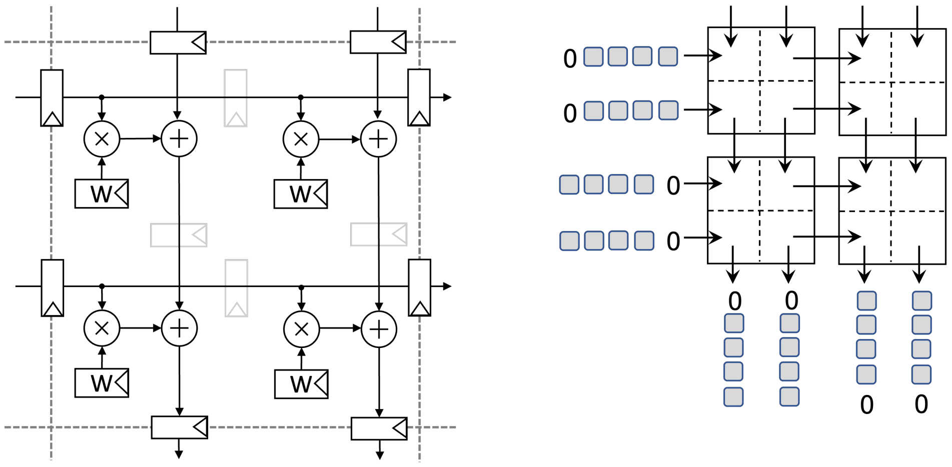
- ArrayFlex introduces a configurable pipeline architecture for SAs that can adjust its pipeline depth to the size of the corresponding matrix multiplication while aiming to minimize the total execution latency.
- When shallow pipeline mode is beneficial, power is equivalently reduced, since transparent registers remain clock-gated and the design as a whole operates at a lower clock frequency.
- Extensive evaluations using state-of-the-art CNN applications demonstrate that the proposed architecture reduces the latency by 11%, on average, while also consuming 13%–23% less power, as compared to SAs with a fixed pipeline organization. This amounts to a substantial improvement in overall energy efficiency.

### 2.1.1 A Systolic Array with Configurable Transparent Pipelining

ArrayFlex aims to adjust the pipeline depth of the vertical and horizontal pipelines of the SA, in order to optimally calibrate them to the size of the systolic array ($R$ and $C$), and the size $T$ of matrix $A$, which are crucial in the overall latency of computation.



(a) Normal pipeline; $k = 1$



(b): Shallow pipeline; $k = 2$

Figure 2.2: (a) In normal pipeline mode, each data item moves to the next PE, either horizontally or vertically, in one clock cycle. (b) In shallow pipeline mode, the dataflow of every $k = 2$ PEs is merged in a single-cycle operation. Merging is possible by bypassing the intermediate pipeline registers. The input and output dataflow skew is altered to match the shallower pipeline structure.

To reduce the $R-1$ cycles spent in the reduction operation in each column of the SA, we can configure the vertical reduction pipeline to operate in shallow mode, where two or more adjacent pipeline stages are merged by making the intermediate registers transparent. The registers in transparent mode, bypass the input data to the next stage, thereby joining two adjacent combinational logic circuits into one pipeline stage. Up to $k$ registers can be joined in the vertical direction.

Pipeline collapsing is also performed in the horizontal dataflow. Instead of letting the input stream move one column to the right in each cycle, we allow it to broadcast to $k$ columns when operating in shallow pipeline mode [55].

The normal pipeline mode that corresponds to the case of $k = 1$ is shown in Figure 2.2(a). Similarly, Figure 2.2(b) depicts an example of shallow pipeline operation assuming $k = 2$. In this case, the result of the top row of PEs is added transparently to the result of the second row of PEs in the same clock cycle. The same operation occurs for every two adjacent PEs.

To align the shallow pipelines of the SA with the arrival of the input data, their arrival skew should be altered. The first (and last) elements of matrix $A$ arrive in batches of $k$ words. It should be stressed that this change does not fundamentally alter the operation of the systolic array, since the required input and output bandwidth remains the same and it is equal to $R$ and $C$ words per cycle (i.e., equal to the number of rows and columns of the SA).

### Latency vs. clock frequency tradeoff

Using this approach, and assuming that we can collapse/merge $k$ intermediate PEs into the same pipeline stage in both the vertical and the horizontal directions, the number of cycles spent in the reduction operation reduces from $R - 1$ to $\frac{R}{k} - 1$, and the number of cycles spent in the broadcast of the first data element to the rightmost column of the SA reduces from $C - 1$ to $\frac{C}{k} - 1$. Thus, the overall latency of computing a matrix product $A^{\text{sub}}_{T,R} \times B^{\text{sub}}_{R,C}$ (as needed by each sub-matrix of the original $A \times B$ product) becomes

$$L(k) = R + \frac{R}{k} + \frac{C}{k} + T - 2 \tag{2.3}$$

The number of cycles needed for all tiles is then equal to

$$L_{\text{total}}(k) = L(k) \times \left\lceil \frac{N}{R} \right\rceil \times \left\lceil \frac{M}{C} \right\rceil . \tag{2.4}$$

Overall, the higher the amount of collapsing (i.e., higher value of $k$), the larger the reduction in the number of cycles needed to complete matrix multiplication.

On the other hand, to enable pipeline collapsing within the SA, one must slow down the clock frequency to avoid timing violations due to increased logic depth. Column collapsing only affects the delay marginally. However, row collapsing requires $k$ additions to be performed in series in the same clock cycle. Therefore, for each shallow pipeline configuration, there is an equivalent throttling of the operating clock frequency to adjust the clock period to the logic depth of the selected configuration. Our goal is to select the best possible $k$ that minimizes the total execution latency in absolute time (i.e., clock cycles $\times$ clock period), given the size of the systolic array $R \times C$ and the size of the matrix multiplication, as determined by $N$, $M$, and $T$.

### The Organization of Configurable PEs

The minimum clock period that the design can operate at is determined by the maximum logic delay between any two pipeline registers, plus any clocking overhead (sum of the register clock-to-Q delay and the setup time of the flip-flops). In the baseline case ($k = 1$), the maximum combinational delay remains inside the borders of one PE and is equal to the delay of the multiplier and the delay of the adder.

When collapsing $k$ pipeline stages into one, the maximum combinational delay again involves the

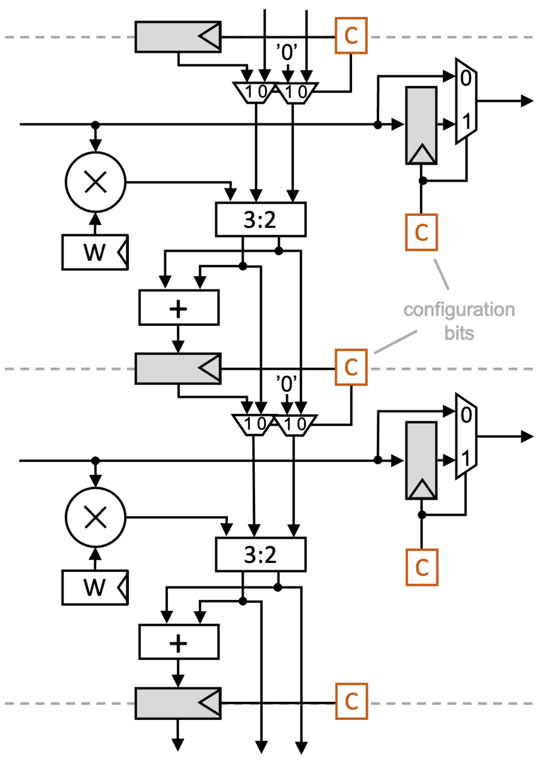


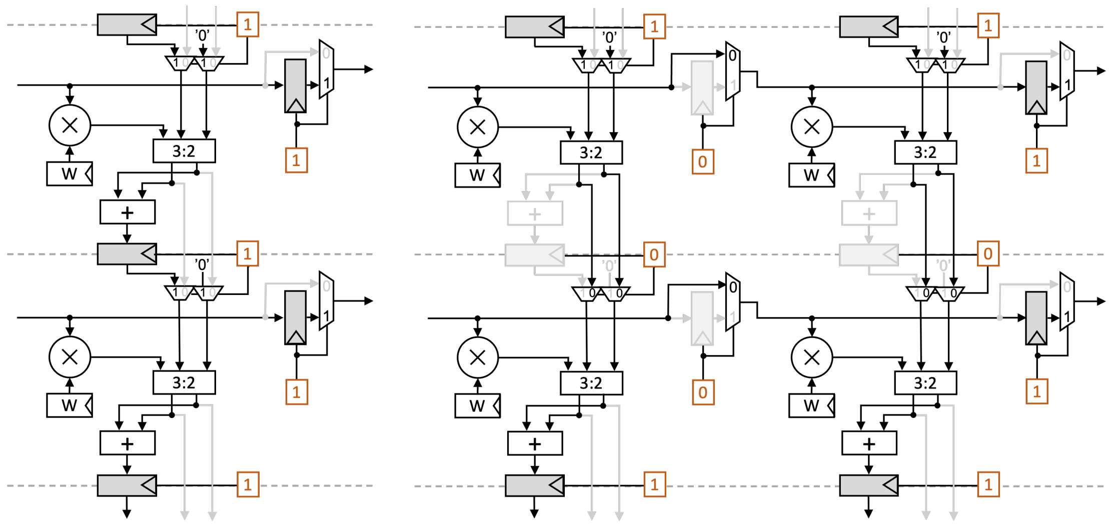
Figure 2.3: The organization of two enhanced, configurable PEs of the same column. Registers are bypassed in the vertical and horizontal directions according to the pipeline configuration. In shallow pipeline mode, reduction is performed using a series of 3:2 carry-save adders ending with a carry-propagate adder.

delay of one multiplier plus the delay of $k$ carry-propagate adders in series plus the delay of bypass multiplexers. To avoid this significant delay overhead when collapsing adjacent pipeline stages, we augment the PEs of the SA with an additional 3:2 carry-save stage that is only enabled during shallow pipeline mode. The organization of two enhanced PEs of the same column are shown in Figure 2.3. The 3:2 carry-save adder is composed of parallel full-adders, with each one placed at each bit position.

When PEs are collapsed, the registers placed in the horizontal direction are bypassed (and clock gated) by additional multiplexers controlled by configuration bits loaded in parallel to the weights of matrix $B$. In the vertical direction, to add the $k$ products produced by the multipliers of each PE, we utilize $k$ carry-save adder stages. The carry-save adders are connected in series through additional bypass multiplexers placed in the vertical direction. At the last stage, where pipeline collapsing ends and the result needs to be saved in the corresponding pipeline register, the result of the carry-save adders is transformed to one operand using the carry-propagate adder of each PE.

This configuration is shown in Figure 2.4(b) for $k = 2$. The products of the first and the second row in each column are added using carry-save adders. The final sum is produced by the carry propagate adders of the last row. The carry propagate adders of the top row are not used, while the registers that are bypassed are clock-gated to save power.

Each PE needs two configuration bits that configure independently the transparency (bypassing) of the pipeline registers in each direction. Separate configuration bits per PE are needed since each PE can play a different role depending on the selected pipeline mode.



(a) Normal pipeline (b) Shallow pipeline; $k = 2$

Figure 2.4: Example of active paths for (a) a normal pipeline ($k = 1$), and (b) a shallow pipeline ($k = 2$).

The incoming input and the weight stored in each PE have the same bit width. However, the vertical connections, including the carry-save adders and the carry propagate adders, have double the bit width, in order to accommodate the full product of the multiplier.

The 3:2 carry-save adder and the bypass multiplexers participate in the operation of each PE even when configured for a normal pipeline, i.e., $k = 1$. As shown in Figure 2.4(a), the product of each multiplier is first added to the result of the previous PE of the same column through the carry-save adder before finalizing the addition with the carry-propagate adder. This extra hardware placed in series between the multiplier and the adder inevitably affects the minimum delay that can be achieved by a conventional PE that does not offer any pipeline reconfiguration. The experimental results show that this delay overhead is marginal and does not limit the applicability of the proposed approach.

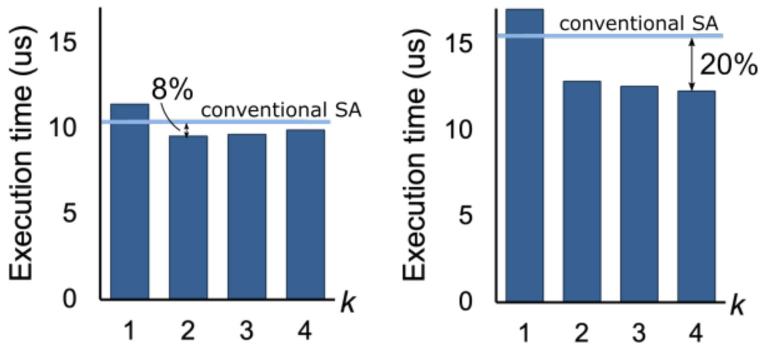
### Minimizing the total execution time

To identify the optimal value of $k$ that best fits the examined configuration, we first need to develop for ArrayFlex a rough model of how the clock period is affected with respect to $k$.

When collapsing $k$ pipeline stages of the SA, the maximum combinational delay involves the delay of $k$ bypass multiplexers in the horizontal direction, the delay of the multiplier ($d_{\text{mul}}$) of the rightmost PE of the collapsed pipeline block, plus the delay of $k$ cascaded 3:2 carry-save adders ($d_{\text{CSA}}$) and bypass multiplexers ($d_{\text{mux}}$) in the vertical direction. In this delay, we should add the delay of the final carry propagate adder ($d_{\text{add}}$) of the last row of the collapsed pipeline block and any flip-flop clocking overhead ($d_{\text{FF}}$). Overall, we can roughly estimate that the minimum clock period that can be achieved by a $k$-collapsed pipeline is:

$$T_{clock}(k) = d_{\text{FF}} + d_{\text{mul}} + d_{\text{add}} + k(d_{\text{CSA}} + 2d_{\text{mux}}) \tag{2.5}$$

In practice, the design supports a maximum pipeline collapsing depth $k_{\text{max}}$. When collapsing fewer than $k_{\text{max}}$ pipeline stages, the combinational paths that still exist in the design but are not used are con-



(a) ResNet-34 Layer 20 (b) ResNet-34 Layer 28

Figure 2.5: The execution time of computing layers (a) 20 and (b) 28 of ResNet-34 [2] as an equivalent matrix multiplication using a configurable SA under various pipeline collapsing depths $k$. The execution time of the conventional (non-configurable) SA that operates only in normal pipeline mode under the highest clock frequency is depicted as a straight line.

sidered false paths. We provide this information explicitly to the static timing analyzer.

The latency in absolute time $T_{abs}(k)$ of computing a complete matrix multiplication using an SA with $k$-collapsible pipeline is the product of the latency in clock cycles $L_{\text{total}}(k)$ given in Equation (2.4) and the minimum clock period $T_{\text{clock}}(k)$ that corresponds to each $k$ (as given by Equation (2.5)):

$$T_{\text{abs}}(k) = L_{\text{total}}(k) \times T_{\text{clock}}(k) \tag{2.6}$$

To explore the interesting interplay between computation latency in cycles and the configurable pipeline depth of the columns of the systolic array, we performed a simple experiment. We measured the execution time required to compute layers 20 and 28 of ResNet-34 CNN [2] as matrix multiplications using a configurable SA that consists of $(R, C) = (132, 132)$ rows and columns. The values of $R$ and $C$ were selected to be divisible by all the examined values of $k$, i.e., 1, 2, 3, and 4. The sizes of the corresponding matrix multiplications for computing layers 20 and 28 are $(M, N, T) = (256, 2304, 196)$ and $(M, N, T) = (512, 2304, 49)$, respectively. In both cases, we examined various pipeline collapsing depths. The obtained results in each case are depicted in Figure 2.5.

For each pipeline collapsing depth $k$, we scaled the clock frequency accordingly to match the combinational delay for each case. The execution latencies that correspond to a conventional (non-configurable) SA are shown as straight lines in both cases of Figure 2.5. The conventional SA operates using a normal pipeline at the highest clock frequency, since it does not suffer any delay overhead associated with configurability.

According to Figure 2.5(a), the execution time for layer 20 is minimized at $k = 2$. In this case, the reduction of clock cycles and the increase of the clock period find their optimal match. Collapsing the pipeline deeper, i.e., $k = 3$, still reduces the execution time, relative to a conventional SA, but the savings are less. For layer 28, as depicted in Figure 2.5(b), deeper pipeline collapsing offers the best execution time. In this case, utilizing a pipeline collapse depth of $k = 4$ is the best choice.

To identify the optimal pipeline depth $\hat{k}$ that minimizes $T_{\text{abs}}(k)$, we take the derivative of $T_{\text{abs}}(k)$

with respect to $k$ and set it equal to zero. This leads to:

$$\hat{k} = \sqrt{\left(\frac{R+C}{R+T-2}\right)\left(\frac{d_{\text{FF}} + d_{\text{mul}} + d_{\text{add}}}{d_{\text{CSA}} + 2d_{\text{mux}}}\right)} \tag{2.7}$$

Even though $k$ is a discrete variable, Eq. (2.7) gives a simple analytical model that leads to one interesting conclusion: the pipeline depth of the SA should be judged not only on the delay profile of the adder and the multiplier hardware blocks, but also on the size of the matrix multiplication (dimension $T$) relative to the size of the SA.

For instance, the first layers of the CNN try to identify on the input coarse features using a wide search area. This leads to large values for dimension $T$ on the corresponding matrix multiplication. As a result, $\hat{k}$ cannot easily reach values larger than one. This means that the best choice is to use an architecture with a normal pipeline, i.e., with $k = 1$. On the contrary, in the last CNN layers, it is common the size of the input features to decrease and their number to increase [56, 57]. Effectively, in these layers the value of $T$ drops and using shallow pipelining (higher $k$) is a better choice. Allowing for pipeline collapse, which also reduces the clock frequency, not only reduces the overall execution time, but it also saves dynamic power. Under shallow pipelining, the clocking power is also reduced, since more registers remain clock-gated.

### 2.1.2 Evaluation

In the experimental evaluation, the goal is to highlight: (a) when SAs operating in shallow mode make sense, (b) the latency and power savings in these cases, and (c) the area overhead incurred in offering pipeline-depth reconfigurability.

To answer these questions, we developed parameterized models of a conventional SA and ArrayFlex in SystemVerilog RTL. Both SAs operate on 32-bit integer quantized inputs and weights executing single-batch inference of various CNNs that consist of matrix multiplications of different sizes. The additions in each column of the SAs are performed at 64 bits.

The SAs were implemented using Cadence's digital implementation flow using a 28 nm standard-cell library. Conventional SAs operating only with a normal pipeline in a non-configurable manner can reach a clock frequency of 2 GHz. The proposed configurable SAs support one normal and two shallow pipeline modes. In normal pipeline mode ($k = 1$), the proposed SA operates at 1.8 GHz. The two shallow pipeline modes allow for collapsing $k = 2$ or $k = 4$ pipeline stages. In these cases, the clock frequency is configured at 1.7 GHz and 1.4 GHz, respectively. Collapsing three pipeline stages is not supported, since three does not divide exactly with the size of the SA, which is a power-of-two in both dimensions.

To estimate the area cost of reconfigurability, Figure 2.6 highlights the physical layout of a conventional SA, relative to the ArrayFlex design, using 8×8 PEs. From the physical layout of both SAs, it is evident that the area of ArrayFlex is increased in both dimensions. The area overhead per PE for this design is approximately 16%. This extra area is consumed by the carry-save adder and the bypass multiplexers, while some marginal area is consumed by the two configuration bits per PE.

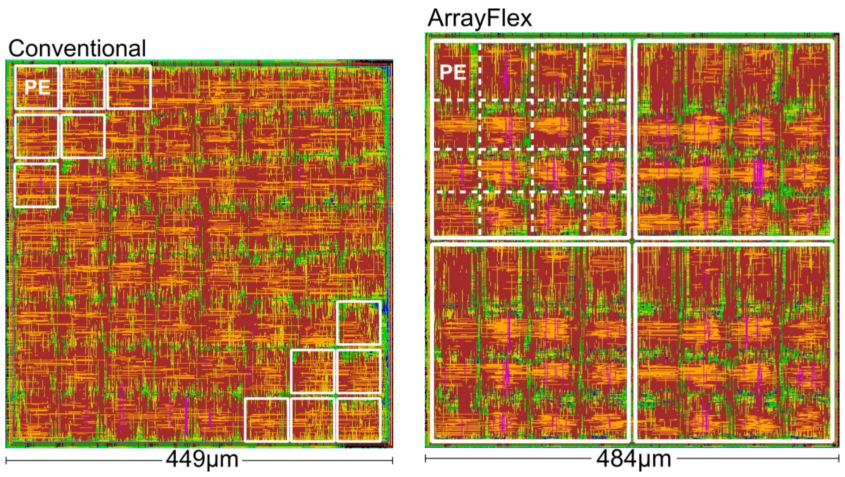


Figure 2.6: The physical layouts of 8×8 PEs using the conventional SA (left) and the proposed ArrayFlex design (right).

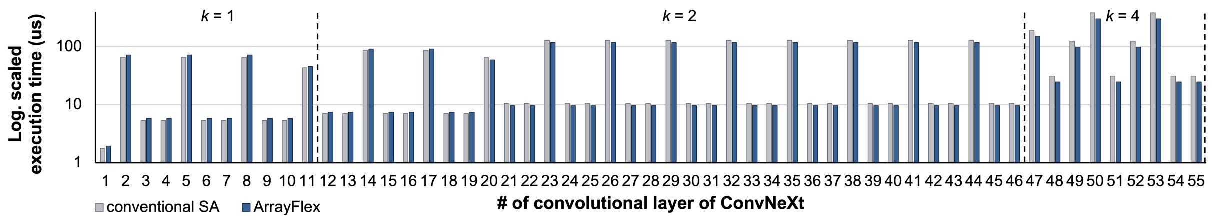
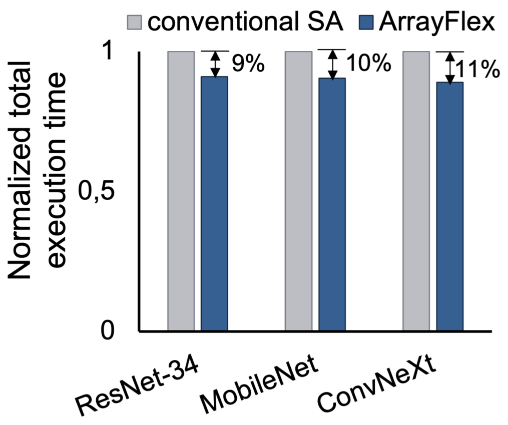


Figure 2.7: The execution time of each CNN layer of ConvNeXt [56] using the conventional and the proposed ArrayFlex SAs. Size of both SAs: 128×128 PEs.

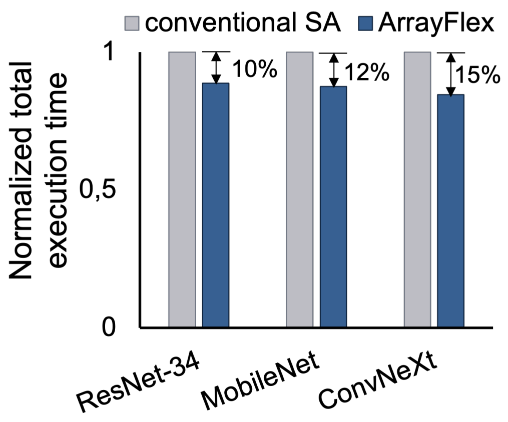
## Performance evaluation

Initially, the aim is to highlight the effectiveness of configuring the pipeline depth per CNN layer in a way that minimizes the total execution time. Figure 2.7 illustrates the execution time per CNN layer of ConvNeXt [56] using SAs that consist of 128×128 PEs. The proposed ArrayFlex SA selects the optimal pipeline depth based on the structure of each CNN layer. For the first 11 layers, it is advantageous to operate under normal pipeline mode. This means that both the conventional SA and ArrayFlex require the same number of cycles to finish the matrix multiplication of each layer. Thus, since the conventional SA operates at a higher clock frequency, it finishes earlier in these cases. For layers 12–46, the proposed SA works optimally under a shallow pipeline mode of $k = 2$, while, for layers 47–55, $k = 4$ is the best configuration. In those cases, the execution time required by ArrayFlex is less than the execution time on the conventional SA. Interestingly, the best pipeline organization per CNN layer is approximated fairly accurately (assuming continuous values) by Equation (2.7). For ArrayFlex, the execution time savings per layer range between 1.5% and 26%, while the *total execution time for all layers* is 11% less than the time required by the conventional SA.

Similar behavior is observed under other CNN models and different SA sizes. Figure 2.8 depicts the normalized total execution time of three CNNs, ResNet-34 [2], MobileNet [57], and ConvNeXt [56], using 128×128 and 256×256 SAs. In all cases, the proposed ArrayFlex design, which configures the pipeline depth and the corresponding clock frequency to the characteristics of each CNN layer, achieves



(a) 128×128 SAs



(b) 256×256 SAs

Figure 2.8: The normalized execution times for *complete runs* (i.e., execution of all layers) for three CNNs using (a) 128×128 and (b) 256×256 SAs. The times are normalized for visual clarity, since the execution time of ConvNeXt is significantly higher than the execution times of the other two CNNs.

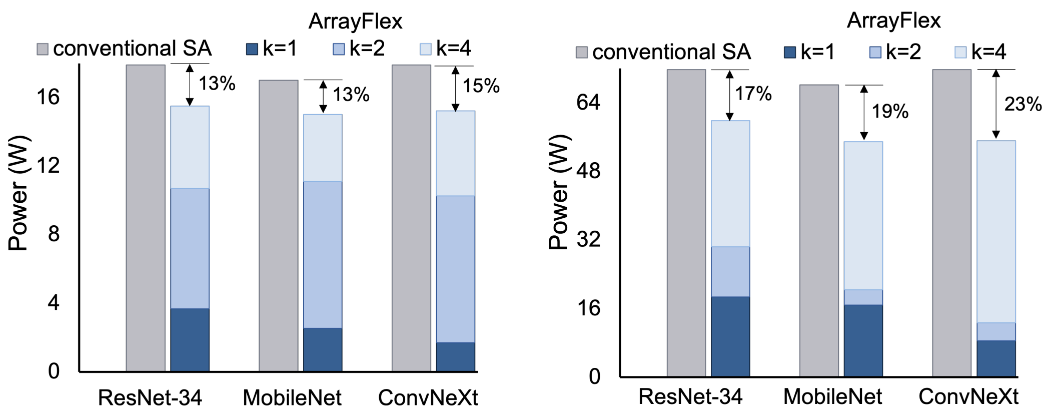
lower execution latency, ranging between 9% and 11%. The savings increase for larger SAs, since more CNN layers prefer a shallow pipeline configuration with $k = 4$. This behavior is in line with Equation (2.7) that ”predicts” higher values for $\hat{k}$ when the size of the SA increases, i.e., with larger values of $R$ and $C$.

### Power consumption evaluation

One other equally important attribute of the proposed ArrayFlex architecture is that it reduces execution time *without* increasing power.

ArrayFlex has larger switched capacitance than a conventional SA, due to the extra hardware required to enable pipeline-depth configurability. Furthermore, it operates at a lower clock frequency than a conventional SA in all pipeline modes. The latter property partially amortizes the power cost of the additional hardware. However, in normal pipeline mode, ArrayFlex still consumes more power than a conventional SA. This behavior changes when in shallow pipeline mode, whereby the clock frequency is further reduced and additional power is saved by the clock gating of the bypassed registers. Therefore, the power profile of ArrayFlex strongly depends on the selected pipeline mode, which is decided independently for each CNN layer.

Figure 2.9 depicts the average power consumption of both SAs under comparison when executing inference on the ResNet-34 [2], MobileNet [57], and ConvNeXT [56] CNNs. For ArrayFlex, the power cost of each pipeline mode is shown separately. ArrayFlex operates in shallow pipeline mode in the majority of the CNN layers of each application. Consequently, this behavior translates to *overall power savings* that range between 13% and 15% for SAs of size 128×128 PEs, and increase to 17%–23% for SAs of size 256×256 PEs. The combined effect of reduced power and less execution time makes ArrayFlex 1.4×–1.8× more efficient in terms of energy-delay-product than a conventional SA.



(a) 128×128 SAs (b) 256×256 SAs

Figure 2.9: The power of the SAs for *complete runs* (i.e., execution of all layers) for three CNNs using (a) 128×128 and (b) 256×256 SAs. The power of the SRAMs and any other peripheral circuitry outside the SAs is omitted.

## 2.2 The Case for Asymmetric Systolic Array Floorplanning

The dynamic power consumption of a SA when computing a matrix multiplication consists of three main components: (a) Input- and weight-loading in the horizontal and vertical directions; (b) Computation power, i.e., the dynamic power consumed for the actual computations of multiplications and additions; and (c) Sum accumulation/unloading that involves the power cost of moving the partial, or final, sums within the columns of the SA.

In this sectcion proposed a floorplanning method, that tackles the *interconnect-related part* of the first and third of the aforementioned components; namely, the consumption attributed to the wires routed in the horizontal and vertical directions across the PEs of the SA and used for (a) the loading of the data and weights and (b) to transfer the partial sums of the reduction operation. Our goal is to select the aspect ratio of the PEs of the SA in a way that minimizes the average dynamic power consumption in said data buses.

The proposed optimization leverages *the inherent asymmetry* in the width of the horizontal and the vertical data buses used in SAs, as well as the inherent differences in the switching activity profiles of the data objects that pass through these buses.

The outcome of this optimization is to design *asymmetric* SAs that are more energy-efficient than their typical counterparts, which tend to use 'square' processing elements for maximum regularity. Experimental results derived from the physical implementation of various SAs at 28 nm technology demonstrate that the proposed approach yields 9.1% reduction in the total interconnect power of the design. In turn, this translates into 2.1% lower overall power consumption within the SA, when executing selected layers of state-of-the-art CNNs.

### 2.2.1 Matrix Multiplication on Systolic Arrays

Under the WS dataflow, a chain of multiply-add operations is computed in each column of the array. Such consecutive additions inevitably increase the dynamic range of the output result, as compared to the dynamic range of the inputs and weights. For integer-arithmetic implementations, this implies that the adder's output width should be more than double the width of the input (or weights) to retain all accuracy.

Similarly, in floating point (FP) implementations, the FP multiply-add units in each PE have a fused/ cascaded structure [58,59], whereby the product of the multiplication is passed directly to the adder, without intermediate normalization and rounding. To avoid precision loss, the intermediate results produced at the South output of each PE use double-width precision [30]. For instance, for Bfloat16 inputs, the reduction that occurs in the vertical direction is implemented with FP32 arithmetic.

Therefore, even if SAs follow an architecturally regular and symmetric structure, the amount of wiring required in the vertical direction, at least for SAs that support the widely used WS dataflow, is significantly higher than the wiring of the horizontal direction.

A similar asymmetry is observed in the switching activity profiles of the inputs that flow horizontally within the rows of the SA and the partial sums that flow vertically in each column of the SA. The input data does not follow a certain statistical pattern and the input values are highly dependent on the selected activation function. The only consistent attribute that is observed for input data is the abundance of zero values generated by the Rectified Linear Unit (ReLU) activation function in each layer. On the contrary, the partial sums that flow in the vertical direction of a SA exhibit more random behavior and possibly higher switching activity, since either positive, or negative, or zero products may need to be accumulated in each step and passed to the next row of the SA.

### 2.2.2 Optimization of the SA floorplan

Our goal is to leverage (a) the innate asymmetry in the wiring of the horizontal and vertical directions in a typical SA, and (b) the difference in the switching activity profiles of the data flowing in each direction, in order to design energy-efficient SAs by optimizing their floor-plan design.

The SA consists of $R$ rows and $C$ columns of PEs. The area of each PE is determined by the area of its constituent components, i.e., the multiplier, the adder (or fused multiply-add units in the case of FP-arithmetic implementations), and the necessary pipeline registers. As shown in Figure 2.10(a), for a fixed input data width of $B_h$ (in the horizontal direction) and an output data width of $B_v$ (in the vertical direction), we can assume that each PE has a constant area of $A\ \mu m^2$. Let us also assume that each PE has height $H$ and width $W$. Since the area of each PE is constant, then $HW = A$.

The aspect ratio $W/H$ of the physical layout of each PE can be optimized to minimize the power consumption associated with the wide data buses that run horizontally and vertically inside the SA and connect the PEs in each direction.

#### Minimizing the total wire length

The input signals arriving at the West side of each row of the SA are broadcast to all columns of the SA. In practice, those wires are interrupted by pipeline registers at the borders of each PE. However, the inclusion of these registers does not change their overall length. Thus, the bus of $B_h$ wires in each row of the SA crosses $C$ columns of PEs of $W\mu m$ width each. The same is true for all rows of the SA.

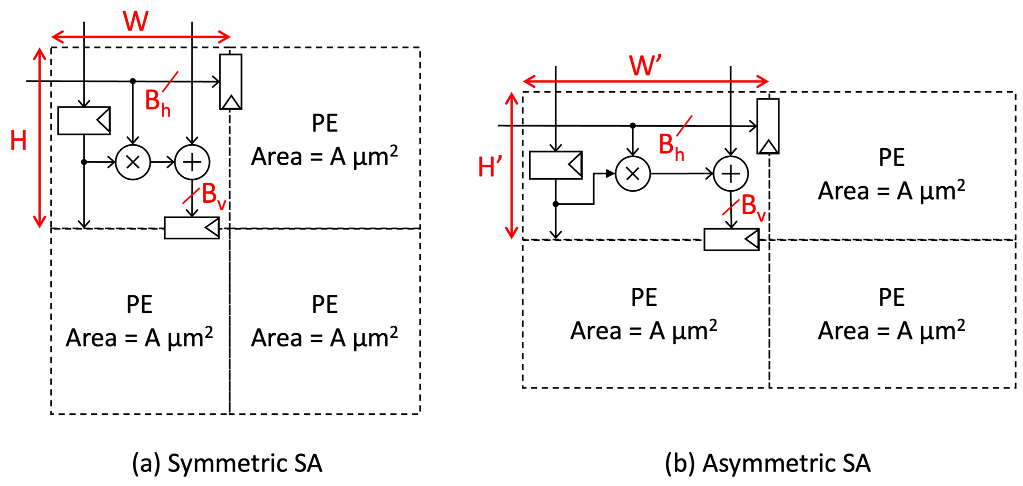


Figure 2.10: The salient dimensions and bus widths in each PE of (a) a *symmetric*, and (b) an *asymmetric* SA organization.

Consequently, the total wire-length of the horizontal input connections of all rows is equal to:

$$WL_h = RC(WB_h) \tag{2.8}$$

Similarly, in the vertical direction of each column of the SA, a bus of $B_v$ wires runs in the North-to-South direction. This bus connects to the adders of each PE and the corresponding output registers. In each column, this bus spans $R$ PEs with a height of $H\mu m$. Thus, for all columns of the SA, the total wire-length of the vertical connections is equal to:

$$WL_v = RC(HB_v) \tag{2.9}$$

Adding the wire-lengths of the horizontal and vertical connections/buses, we can calculate the *total* wire-length $WL$:

$$WL = WL_h + WL_v = RC(WB_h + HB_v) \tag{2.10}$$

Substituting $A/H$ for $W$ in (2.10),

$$WL = RC\left(\frac{AB_h}{H} + HB_v\right) \tag{2.11}$$

To identify the optimal aspect ratio that minimizes $WL$ for a PE of constant area $A$, one must take the derivative of $WL$ with respect to $H$ and set it equal to zero. This leads to:

$$\frac{AB_h}{H^2} = B_v \xrightarrow{A=HW} \frac{WH}{H^2} = \frac{B_v}{B_h} \rightarrow \frac{W}{H} = \frac{B_v}{B_h} \tag{2.12}$$

Hence, the aspect ratio $W/H$ that minimizes the total wire-length follows the ratio of the width of the vertical and horizontal buses. Since the vertical connections that carry partial sums are, by construction, wider than the horizontal buses that transfer input data, the presented analysis indicates that the PEs should *not be square*. Instead, they should adopt a *rectangular shape* with smaller height than width, as depicted in Figure 2.10(b), where $H' < W'$.

In summary, this analysis demonstrates that the aspect ratio of each PE's floorplan should be *asymmetric*, if one wishes to optimize the energy efficiency. This result holds for *all* SAs, irrespective of their size, i.e., the number of rows and columns.

**Optimal floorplan aspect ratio**

Instead of dealing only with the minimization of the wire length, the above analysis could be enhanced to also include the average switching activity profiles of (a) the input data flowing in the horizontal direction and (b) the partial sums flowing in the vertical direction. In this way, the optimization of the SA floorplan could also target the dynamic power consumption within the interconnects that traverse the PEs in both directions.

To include the average switching per bit in the analysis, it suffices to scale the bit widths of the horizontal and vertical busses, $B_h$ and $B_v$, respectively, with the average switching activity observed in each direction (i.e., $a_h$ and $a_v$) when executing representative ML matrix multiplication workloads. In this case, $B_h$ is scaled by $a_h$ and $B_v$ by $a_v$. After applying such scaling factors, we arrive at the following optimal aspect ratio for each PE:

$$\frac{W}{H} = \frac{B_v a_v}{B_h a_h} \tag{2.13}$$

Note that this result does not change the main outcome of the analysis presented in Section 2.2.2 above. Each PE should still have a smaller height than width, since $a_h$ is expected to be smaller, or at least equal to, $a_v$. Therefore, the fact that $B_v > B_h$ still determines the optimal aspect ratio.

### 2.2.3 Experimental Evaluation

To highlight the benefits of asymmetric SA floorplanning, we designed two 32×32 SA-based accelerators in SystemVerilog RTL. The SAs were implemented with the Cadence digital implementation flow using a 28 nm standard-cell library. The first accelerator has a fully symmetric layout, where each PE is placed as a square, while the second SA adopts the proposed asymmetric floorplan with rectangular PEs.

Both SA designs operate at 1 GHz with 16-bit integer quantized inputs and weights, and they both execute single-batch inference on the ResNet50 [2] CNN layers, which consist of matrix multiplications of different sizes. The additions in each column of the SAs are performed at a width of 37 bits. This particular output bit-width is required to accommodate the dynamic range when adding 32 products of 32 bits each. Therefore, for the selected configuration, $B_h$=16 and $B_v$=37.

The switching activity information was collected after feeding the CNN model with sample images from ImageNet [60]. The average switching activity observed after executing all layers of ResNet50 [2] is $a_h$=0.22 and $a_v$=0.36. It is important to note that the horizontal switching activity is an average of the input observed in all layers of ResNet50. This means that layers with denser inputs have higher switching activity and layers with sparser input (i.e., more zero values observed due to the ReLU activation function) have lower switching activity. The increased average switching activity observed in the vertical direction (output of the adder of each PE) is mostly due to the signed arithmetic that causes many bits to change value when moving from positive to negative numbers. On the contrary, the inputs in the horizontal direction are, by construction, positive integers.

These switching activities are merely used as indicative examples. For a real design, one needs to take into account the switching profiles of many applications, in order to arrive at a solution that is efficient in various different application scenarios.

Based on the width of the horizontal and vertical data buses, and the observed switching activities (that represent the average behavior of many CNN models), we selected an aspect ratio of $W/H = 3.8$ for the proposed asymmetric SA design. Figure 2.11 shows the physical layouts of a conventional SA with square PEs and the proposed asymmetric SA design with rectangular PEs, assuming arrays with $8\times8$ PEs.

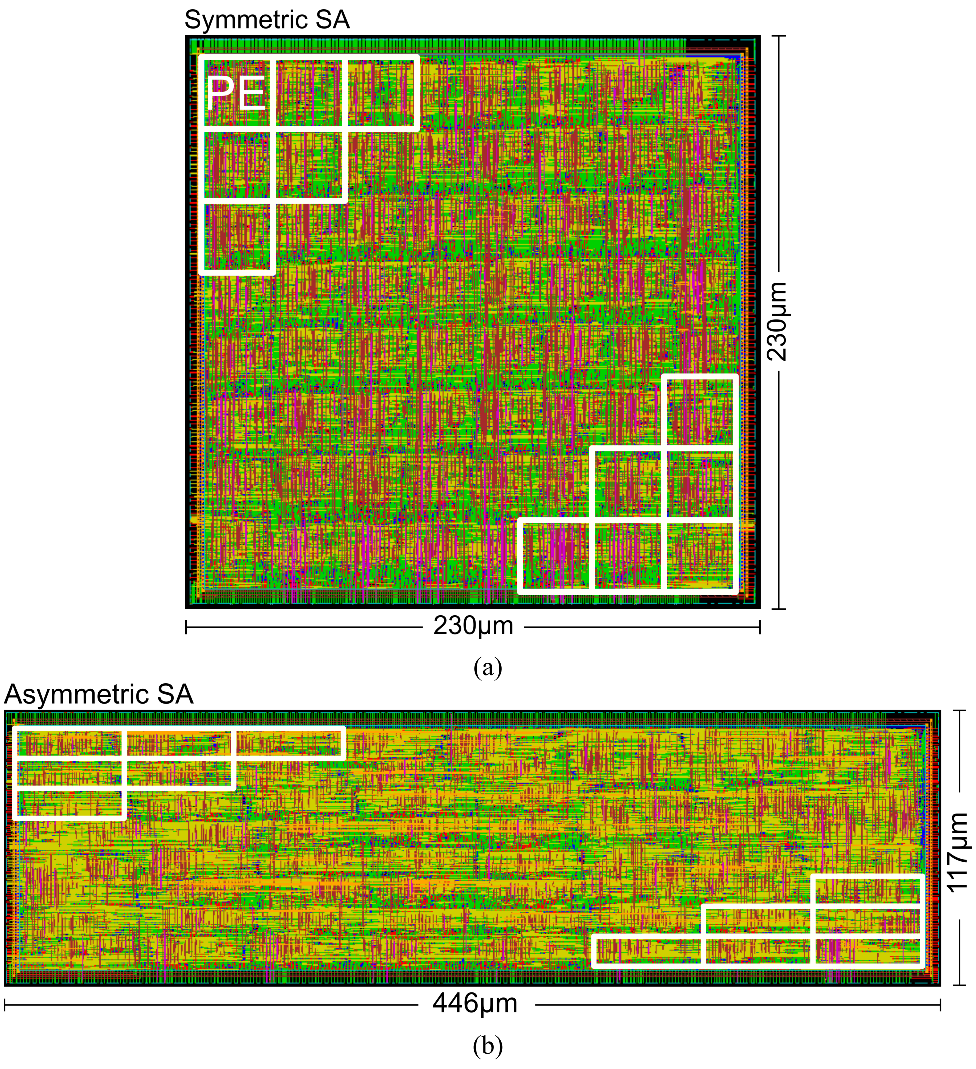


Figure 2.11: Physical layouts of the $8\times8$ (a) symmetric SA, and (b) asymmetric SA. Both designs were implemented using a 28 nm standard-cell library and operate at a clock frequency of 1 GHz.

Figure 2.12 depicts the power consumed on the interconnects of the symmetric and asymmetric SA configurations, for 6 selected convolutional layers of ResNet50 [2]. The average per-layer power consumption for ResNet50 is also shown in the figure. The corresponding dimensions of the 6 selected layers are summarized in Table 2.1. As shown in Figure 2.12, the proposed asymmetric layout reduces the total

Table 2.1: Selected convolutional layers of ResNet50 [2] and their respective attributes. Parameter K is the kernel size, H is the output height, W is the output width, C is the number of input channels, and M is the number of output channels.

| Name | Attributes |
|---|---|
| L1 | K=1, H=56, W=56, C=256, M=64 |
| L2 | K=3, H=28, W=28, C=128, M=128 |
| L3 | K=1, H=28, W=28, C=128, M=512 |
| L4 | K=1, H=14, W=14, C=512, M=256 |
| L5 | K=1, H=14, W=14, C=1024, M=256 |
| L6 | K=3, H=14, W=14, C=256, M=256 |

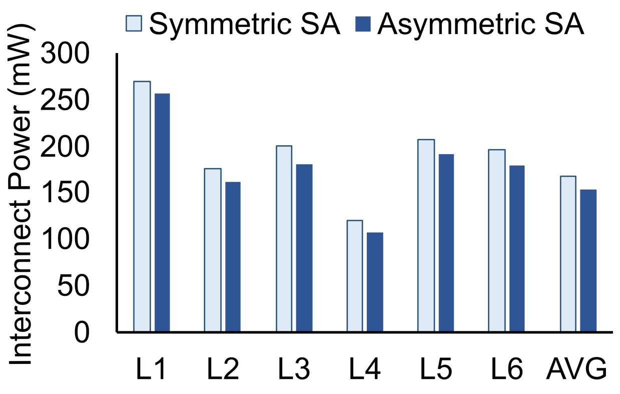


Figure 2.12: Interconnect power consumption for selected layers of ResNet50 [2].

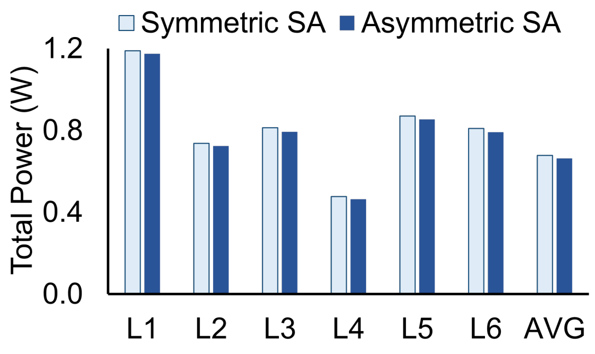


Figure 2.13: Total power consumption for selected layers of ResNet50 [2].

interconnect power consumption by 9.1%, as compared to the symmetric layout. This interconnect power reduction translates into a *total* average power reduction of 2.1%, as illustrated in Figure 2.13.

Even if the overall power savings are small, they are reaped without *any* performance trade-off whatsoever. The proposed approach simply requires a customization of the design's floorplan during its physical synthesis flow.

## 2.3 Conclusion

This chapter highlights two complementary directions for improving the efficiency of systolic arrays in the context of machine learning acceleration. From the micro-architectural perspective, merging the pipeline stages of an SA introduces a tradeoff between the reduction in the number of execution cycles for matrix multiplication and the increase in clock period due to deeper combinational paths. By employing carry-save adders in parallel with the multiply-add units of the processing elements, ArrayFlex effectively mitigates frequency degradation. Tailored specifically for each convolutional layers, the reduction in cycles outweighs the modest increase in clock period, enabling overall reductions in execution time and power consumption with only marginal hardware overhead.

From the physical design perspective, systolic arrays exhibit a natural asymmetry in their interconnect requirements, with horizontal (input) wiring differing significantly from vertical (output) wiring. Exploiting this asymmetry, asymmetric floorplanning aligns the physical aspect ratio of processing elements with workload-induced dataflow characteristics, thereby reducing interconnect wirelength and switching activity. This optimization lowers interconnect-related dynamic power while remaining fully complementary to other data-driven low-power techniques proposed for systolic arrays [61].

Together, these two approaches demonstrate the importance of cross-layer optimization in systolic array design. By addressing either micro-architectural adaptability or physical-level layout efficiency, it is possible to achieve substantial gains in execution time, energy efficiency, and overall accelerator performance. Such advances are particularly valuable for power-constrained applications, including edge computing scenarios and ambient assisted living systems, where energy-efficient neural network accelerators are essential.

# Chapter 3

# Data Encoding for Low-Power Systolic Arrays

The optimizations presented in Chapter 2 underscore the importance of tailoring both the micro-architectural pipeline structure and the physical floorplan of systolic arrays to reduce execution latency and power consumption. While these techniques significantly enhance efficiency at the architectural level, an equally critical source of dynamic power stems from the movement and switching activity of data within the array. Specifically, inputs, weights, and partial sums exhibit diverse switching characteristics that can be exploited to further curtail power dissipation. This observation motivates the focus of this chapter, where we shift from structural considerations to encoding strategies and data reordering techniques. By intelligently shaping the representation and flow of data, these methods complement the layout-level optimizations and provide an additional layer of energy savings, thereby contributing to a holistic approach for low-power systolic array design.

More specifically, the dynamic power consumption of an SA when computing a matrix multiplication consists of three main components:

**Input and weight loading:** This involves the loading of inputs and weights in the horizontal and vertical directions, respectively. In output-stationary dataflow, the data loading occurs in parallel to the actual computation, from the West and the North side of the SA, in an orchestrated manner. On the contrary, in the weight-stationary and input-stationary dataflow types, the weights and the inputs, respectively, can be loaded beforehand, as previously explained. The cost of data loading involves the power expended in the clocked elements (pipeline registers) and the wires that connect them, and it is directly related to the switching-activity profile of the incoming data.

**Computation power:** The dynamic power consumed for the actual computation, i.e., multiplication and addition. This part is the major power consumer and depends mainly on the size and the arithmetic format of the processed data and the complexity of the datapath logic. Quantization for Deep Neural Networks (DNNs) retains the inference-time model quality using integer-only arithmetic. State-of-the-art Machine Learning (ML) models, such as transformers, still require floating-point arithmetic, even if this refers to low-precision 8-/16-bit formats [30].

**Sum accumulation/unloading:** The power cost of moving the partial or final sums. In weight-stationary and input-stationary dataflows, this occurs in parallel to the computation in the vertical and

Table 3.1: The types of data that continuously move during matrix multiplication under the three main SA dataflows.

| Dataflow | Inputs | Weights | Partial Sums |
|---|---|---|---|
| Weight-Stationary (WS) | HIGH | LOW | HIGH |
| Input-Stationary (IS) | LOW | HIGH | HIGH |
| Output-Stationary (OS) | HIGH | HIGH | LOW |

horizontal directions, respectively. Instead, in output-stationary dataflow, this occurs only once, at the end of the computation phase. The dynamic power associated with the movement of partial (or final) sums involves the power consumed in pipeline registers and the wires that connect them.

The focus of this optimization is on the reduction of dynamic power consumption within the SA, which is directly proportional to the amount of activity flowing through the system.

To reduce the data movement activity, one must assess the switching attributes of the various data types involved in the matrix multiplication calculations. Table 3.1 indicates which of the three key data types, i.e., inputs, weights, and partial sums, move continuously under the three main SA dataflows. Out of the three data types, the partial sums exhibit totally unpredictable switching behavior. Their values change continuously due to the ongoing accumulations, thereby manifesting unpredictable numerical attributes. In contrast, the numerical behaviors of the weights and the inputs are more predictable. The weights are determined during training and remain unchanged during inference, while the inputs are – by construction – sparse (i.e., they contain many zero values), due to the application of activation functions, such as Rectified Linear Units (ReLU). Hence, those two data types are inherently amenable to techniques that can leverage numerical attributes to reduce the data movement activity. Consequently, without loss of generality, we *focus on the OS dataflow* for two reasons: (1) *both* the inputs and the weights exhibit continuous movement, thus increasing the potential to reap benefit; (2) the partial sums, whose attributes are unpredictable and difficult to exploit, remain stationary with minimal movement, due to local accumulation. Note that the decision to focus on the OS dataflow is not limited in scope, since said dataflow type enjoys widespread use in existing accelerators [36, 46].

Targeting systolic arrays, we combine for the first time, to the best of our knowledge, dynamic (data encoding) and static (data reordering) techniques to reduce the dynamic power consumption involved in the input-and-weight loading process. The goal is to *optimally* apply data encoding and reordering using a *targeted* approach that minimizes data switching activity in a cost-effective manner. Overall, the contributions of this mechanism can be summarized as follows:

- We explore the value distribution of the *weights* of modern CNN applications to identify the data segments within the weights that exhibit high switching activity. This information allows for optimal application of encoding; data segments that exhibit low switching activity are simply not encoded. Hence, the data switching activity is minimized at the minimum possible hardware cost.
- The statistical analysis of the weight values constitutes a first attempt to identify and exploit useful

numerical attributes in the new and increasingly prevalent floating-point arithmetic formats, such as Bfloat16 [28,59]. Reduced precision floating-point arithmetic is rapidly gaining traction in deep learning, where sufficient inference accuracy is not always achieved using only integers [30,62].
- Zero-value clock gating is selectively applied only to the *inputs* of the CNN layers to exploit the presence of zero values generated by the ReLU activation function. Unnecessary operations involving zeros are skipped altogether within the SA.
- In conjunction with the aforementioned *dynamic* techniques, an *offline* reordering of the weight matrix further ensures that the weight-loading process induces minimum switching activity within the SA. The reordering problem is formulated as a Traveling Salesman Problem (TSP) and the generated shortest cycle is translated into a row permutation of the weight matrix.

Extensive evaluations using complete CNN applications driven with test images from the ImageNet database [60] demonstrate the efficacy and efficiency of the proposed architecture. The switching activity is reduced by 38%, on average, as compared to SAs with unencoded data values and no weight reordering, which translates into total dynamic power savings of 20.2%, on average, for an SA of size $32\times32$, in ASIC hardware implementations. These power savings improve to 32.1% for an array of size $64\times64$, while they remain limited to 11% for a $16\times16$ SA.

## 3.1 Dynamic Techniques for Low-Power Data Movement

To decrease the dynamic power consumption due to data movement, the ultimate goal is to minimize the switching activity within the horizontal and vertical register pipelines of the array. While there is a wealth of proposed approaches in the literature to lower switching activity, this approach is the first – to the best of our knowledge – to apply such techniques within the SA context and quantitatively assess their effectiveness when executing real CNN applications.

### 3.1.1 Dynamically reducing the switching activity

The proposed architecture minimizes the switching activity within the SA by employing both dynamic (i.e., run-time) and static (i.e., offline) techniques that operate cohesively and in tandem.

There are various dynamic mechanisms, employed at the hardware level, that can be utilized to lower the switching activity. The three most relevant techniques are: (a) data-driven clock gating, (b) bus-invert coding, and (c) zero-value clock gating.

The first one is the most fine-grained mechanism for clock gating, since it is deployed in flip-flops at the gate level. The clock signal driving a flip-flop is disabled (gated) if the flip-flop state would remain unchanged in the next clock cycle [63]. To reduce overhead, several flip-flops may be grouped together and driven by the same clock gating logic generated by ORing the enabling signals of the individual flip-flops. Of course, this coarser granularity tends to lower the disabling effectiveness. Therefore, it is beneficial to group flip-flops whose switching activities are highly correlated. This constraint renders this approach infeasible for CNN applications, because it is very difficult to find groups of bits that remain the same. Consequently, the aggressive bit-level application of this technique would incur undue overhead, whereas the coarser group-level implementation would not be effective for CNNs.

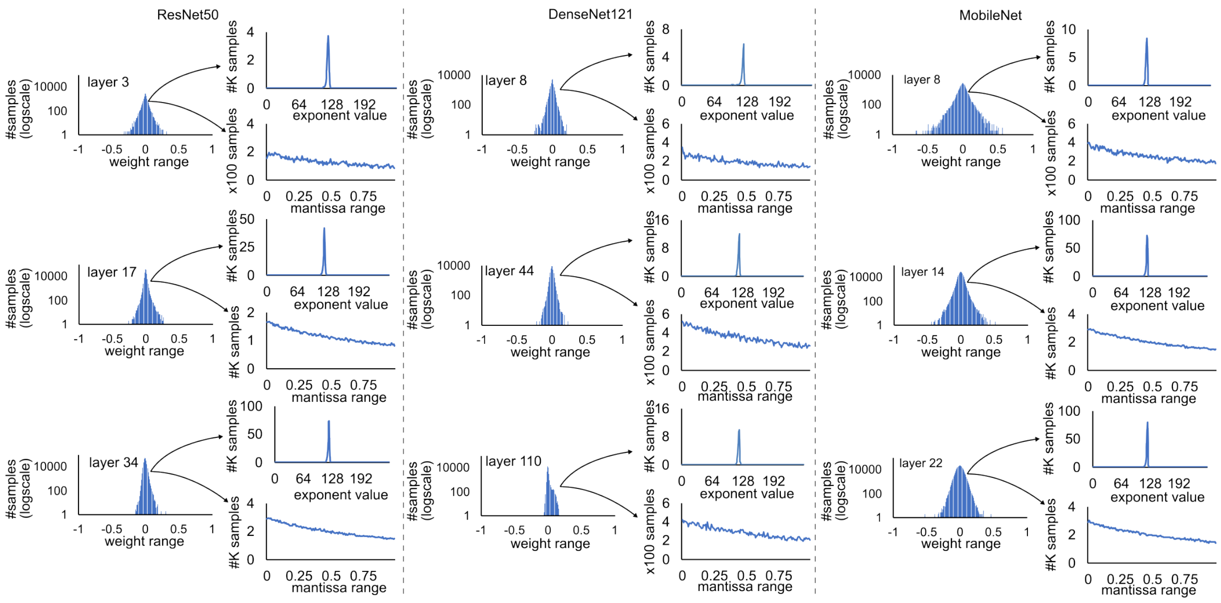


Figure 3.1: The observed value distributions of the weights in three randomly selected layers of ResNet50 [2], DenseNet121 [68], and MobileNet [57]. The exponent values of the corresponding Bfloat16 numbers are highly concentrated, while the mantissa values are almost uniformly distributed in the entire available dynamic range.

Hence, we focus on the remaining two techniques that are well suited for a systolic array setting.

**(1) Bus-Invert Coding (BIC)**: This well-known technique [64] lowers the bus activity by reducing the number of bit transitions. The algorithm computes the hamming distance between the present and next bus values, i.e., the number of bits in which they differ. If this number is more than half of the bus width, then the *complement* of the next datum is transmitted instead. Alongside the transmitted data, the BIC scheme also transmits a single bit ('inv-bit'), indicating whether the transmitted data is inverted or not. A variant of the BIC technique, called **Segmented Bus-Invert Coding**, can be applied to different segments of the bus separately [65]. For instance, BIC could be applied to the mantissa (fraction) and the exponent fields of a floating-point number *independently*. Other more elaborate coding techniques that are useful when driving large buses are not considered [66, 67], since their encoding/decoding overhead diminishes any savings in switching activity at the PE level.

**(2) Zero-Value Clock Gating**: This technique can be used when the input or/and the weight have zero value. Since the result of any multiplication involving at least one zero value is zero, there is no need to perform the multiplication. Therefore, upon detecting a zero input value, the pipeline is 'frozen' (amounting to inserting a bubble) and an 'is-zero' bit is asserted. This bit is subsequently used to clock-gate the registers and to bypass the result of the multiplication, which is known a priori to be equal to zero.

### 3.1.2 Systolic Arrays with Selective Coding and Clock Gating

To cost-effectively apply the above two techniques within an SA, it is imperative to identify, and account for, some nuances of CNNs, which constitute our primary target applications.

For instance, in CNNs, the weights of each layer are determined during training and remain unchanged

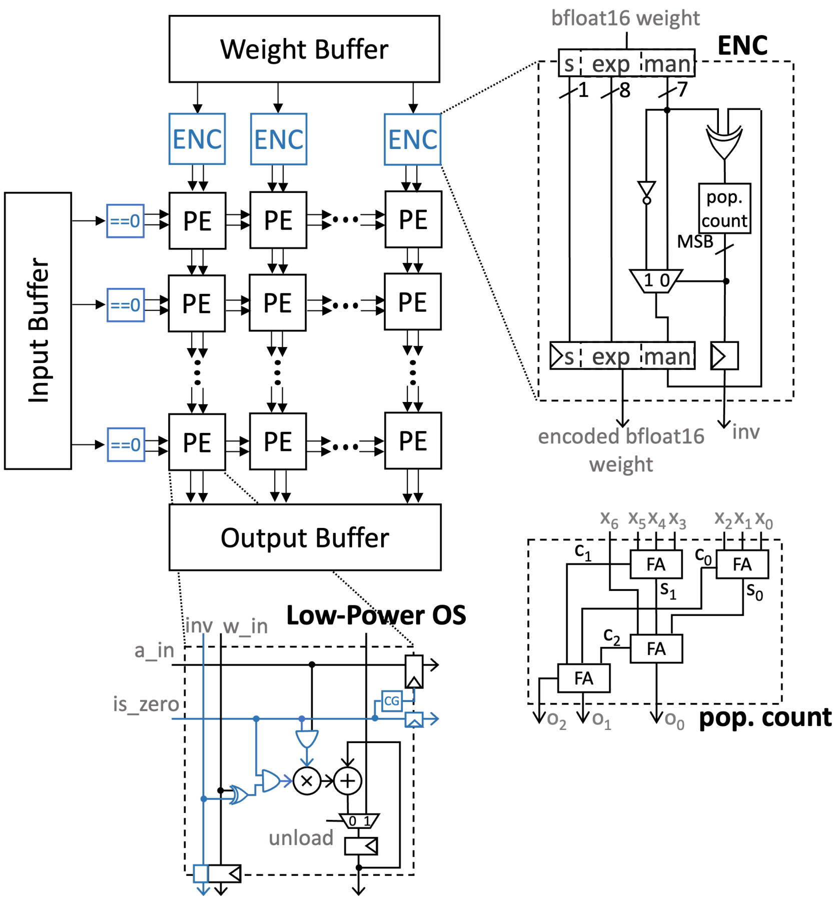


Figure 3.2: The proposed low-power SA architecture that combines two power-saving techniques to reduce the switching activity as data flows through the array of PEs. The inputs pass through zero-detection logic at the West edge of the SA. The weights are encoded at the North edge, prior to entering the array, and the actual values are recovered within each PE for the ensuing calculations.

throughout the inference phase. The weights encountered in CNNs have some interesting attributes that can be exploited by the BIC technique. In an attempt to identify useful, in terms of encoding potential, numerical attributes exhibited by the weights of modern CNN applications, we perform a detailed statistical analysis of the observed value distributions when using the widely utilized Bfloat16 floating-point format. Figure 3.1 illustrates the value distributions of the weights of three randomly selected layers for each one of three state-of-the-art CNNs: ResNet50 [2], DenseNet121 [68] and MobileNet [57]. Specifically, the histogram of the weight values is further analyzed into the distributions of the *exponent* and *mantissa* values. These weight values are bounded to the range $[-1, 1]$ from the training step. More importantly, the trends illustrated in Figure 3.1 are observed across all the layers of the three analyzed CNNs; i.e., all layers exhibit very similar behavior in terms of the value distribution of the weights.

As shown in Figure 3.1, the weight values are highly concentrated around zero, i.e., their absolute values are very small. As a result, when the weights are represented as Bfloat16 numbers, their exponent values are highly concentrated close to the bias value. This high concentration implies that consecutive exponent values have very few bits that are different, thereby rendering the BIC technique *non-beneficial for the exponents*. On the other hand, the values of the *mantissa* field of the CNN weights are almost uniformly distributed, thus making them amenable to BIC. Hence, for the *weights*, we use BIC targeting *only the mantissa* field.

Note that the observed trends in the exponent and mantissa destributions of the weight values are *not* due to a single/few dominating layers in each of the CNNs. This behavior is observed in *all* the layers of the CNNs under investigation. Hence, the targeted application of BIC to solely the mantissa field of the weights is expected to yield benefits consistently in all the layers of the CNN.

On the other hand, the input data is different for each input image and is highly dependent on the selected activation function. Therefore, there is no clear statistical attribute that can be exploited for the inputs, other than the abundant zero values generated by the ReLU activation function in each layer. Hence, the zero-value clock gating technique need *only be applied to the inputs* of the CNN layers. This ensures that no power is wasted on redundant operations. Note that the abundance of zeros can be artificially increased in the weights, too, by enabling weight pruning techniques. However, such approaches are out of the scope for the proposed mechanism.

The proposed SA architecture, which effectively combines the BIC and zero-value clock gating techniques, is shown in Figure 3.2. As compared to a baseline SA (i.e., without power-saving functionality), the additional new logic (shown in color blue) is primarily found on the North and West edges of the SA within the *Encoding* ('ENC') modules and the zero-value checkers, respectively. As illustrated in Figure 3.2, each Encoding module implements the BIC technique only on the mantissas of the weights, as previously explained. Specifically, BIC is performed by XORing the current and the previously transmitted mantissa bits, and then counting the number of 1s in the result of the XOR operation. This gives the number of bits that are different between the two mantissa values. The counting of the number of 1s is facilitated by the 'pop. count' module shown in Figure 3.2, which consists of a 7-to-3 reduction tree to produce the final addition result. However, if the number of bits that are different is greater than half of the mantissa bits, the Most Significant Bit (MSB) of the 'pop. count' is 1. Thus, we only need the MSB of the 'pop. count,' which means that the logic synthesizer could optimize the aforementioned 7-to-3 reduction tree. If the MSB of the 'pop. count' is 1, the mantissa bits are inverted, as per the BIC algorithm, and the 'inv' bit is high.

This targeted/selective BIC approach reduces the incurred area overhead and avoids redundant power consumption. In the proposed architecture, there is also some lightweight new logic within each PE. Specifically, XOR gates are added to recover the original value of the mantissa of each weight, if it was inverted by the BIC technique. Latch-based clock gating cells that are triggered by the 'is-zero' bit are automatically added to the input datapath during logic synthesis. The same 'is-zero' bit is also used to data-gate the multiplier, in order to eliminate wasted power in multiplications with zero.

It should be noted that the zero-value clock gating mechanism has been applied to SAs in the past [36]. However, in this case, we use it in *conjunction* with BIC to reap the extra benefits of the *synergistic* effect of both techniques operating in tandem. More importantly, the employed *targeted* encoding approach selectively applies the most appropriate scheme to each data type (inputs vs. weights) to reap the most power savings with the minimum hardware cost.

## 3.2 Offline Weight Reordering for Low-Power Data Loading

As previously described, the matrix multiplication operation in a systolic array requires the two matrices to ‘slide’ inside from the West and North edges in a staggered and synchronized/orchestrated fashion. Recall that, within the context of CNN applications, the matrix entering the SA from the West includes the *input* data, which constantly varies. On the contrary, the matrix entering from the North includes the *weights* of each layer, which are calculated a priori and remain unchanged throughout the inference phase. Under output-stationary dataflow, these fixed/non-changing weights will be repeatedly loaded in the SA for each multiplication.

Hence, to decrease the dynamic power consumption associated with the weight-loading process, the *flow* of the individual weights inside the SA must be generated in such a way as to minimize the Hamming distance between consecutive values. Since the weight values enter from the North of the SA in *columns* of numbers, their flow can be altered by reordering the *rows* of the weight matrix. However, in any two-matrix multiplication, if the rows of the second matrix are reordered, then the columns of the first matrix must be reordered in the exact same manner to retain the correctness of the multiplication result. Consequently, the reordering of the rows of the weight matrix imposes, by default, a similar reordering to the columns of the input matrix entering the SA from the West edge.

This reordering of the weights aiming to reduce the Hamming distance between consecutive values of each weight-loading flow could be formulated as a Traveling Salesman Problem (TSP), where we try to identify the ‘visiting’ order of weight values that minimizes the total switching activity.

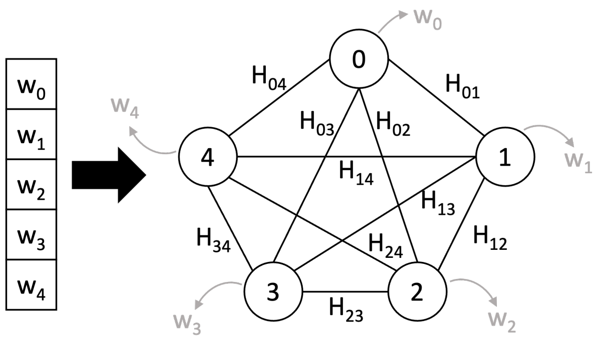


Figure 3.3: Representation of a column vector of weight values as a fully-connected graph to facilitate TSP formulation.

To understand how the reordering of the two-dimensional weight matrix can be effectively formulated as a TSP, let us first start with the one-dimensional case. Let us assume that we have a row vector of input data (e.g., activations) and a column vector of weights that flow through a single PE as the multiplication is performed. The weight vector could be represented as a graph, with each node corresponding to a value of the weight vector (i.e., column) and each edge representing the Hamming distance between the two values, as shown in Figure 3.3. The edge designation $H_{01}$ is the Hamming distance between values $w_0$ and $w_1$. Note that the generated graph is fully connected, because, after the reordering, every pair of weight values is possible to be consecutive. Figure 3.4 depicts an example of how the TSP formulation yields the shortest cycle, highlighted in red, connecting all the nodes of a random graph. The permutation

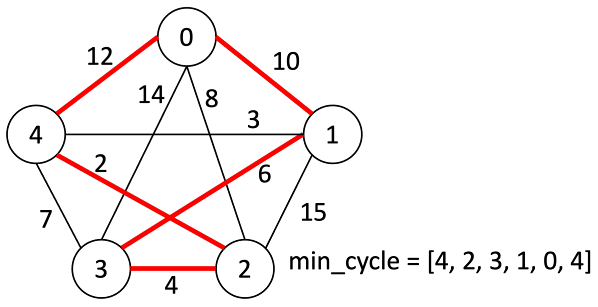


Figure 3.4: An example of the TSP formulation identifying the shortest cycle connecting all the nodes of a a random graph. The shortest cycle (aka ‘min_cycle’) is the node visiting order that creates a cycle in the graph with the minimum total distance. This is a solution of the TSP problem.

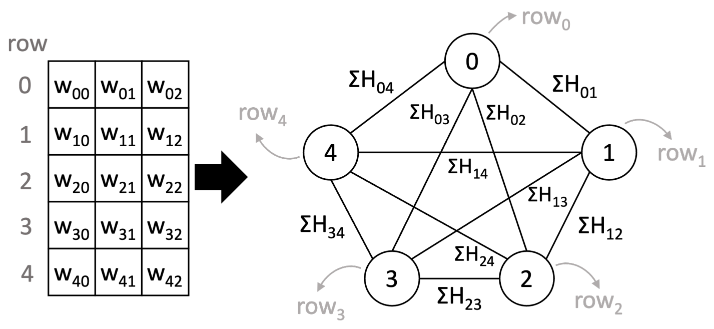


Figure 3.5: Representation of a 2D weight matrix as a fully-connected graph to facilitate TSP formulation. The nodes now represent entire *rows* of the 2D weight matrix, instead of individual weight values.

designated as *‘min_cycle’* in Figure 3.4 is the node visiting order that creates a cycle with the minimum total distance for the given graph. In our application scenario, instead of a *cycle*, only a *path* is needed that connects all the nodes. Thus, we have to choose the starting node that gives the shortest/minimum path among all possible column paths. To do so, we test each node in the column as being the start of the path and choose the node that forms the path with the smallest total distance. If the weights are reordered following the permutation produced by the TSP formulation, the resulting flow of the weights would have the minimum Hamming distance between consecutive values. In turn, this would generate the minimum switching activity between consecutive values.

Extending this approach from a 1D weight vector to a 2D weight matrix is bound by only one restriction: the exact same row permutation must be applied to *all* columns of the weight matrix, i.e., we no longer reorder single values in a column vector, but entire rows of values in a matrix. This restriction allows for the corresponding appropriate reordering of the columns of the input matrix, which is a necessary condition to ensure correctness of the matrix multiplication. In this scope, each node in the generated graph now corresponds to a row of the weight matrix and each edge represents the sum of the Hamming distances of the corresponding elements of the two connected rows. Let us assume a $5 \times 3$ weight matrix, as shown on the left of Figure 3.5. The generated fully-connected graph, as depicted on the right of Figure 3.5, has one node for each of the 5 rows of the weight matrix, and each edge's value is the sum of the Hamming distances of the values of the two connected rows. For instance, edge $e(0, 1)$

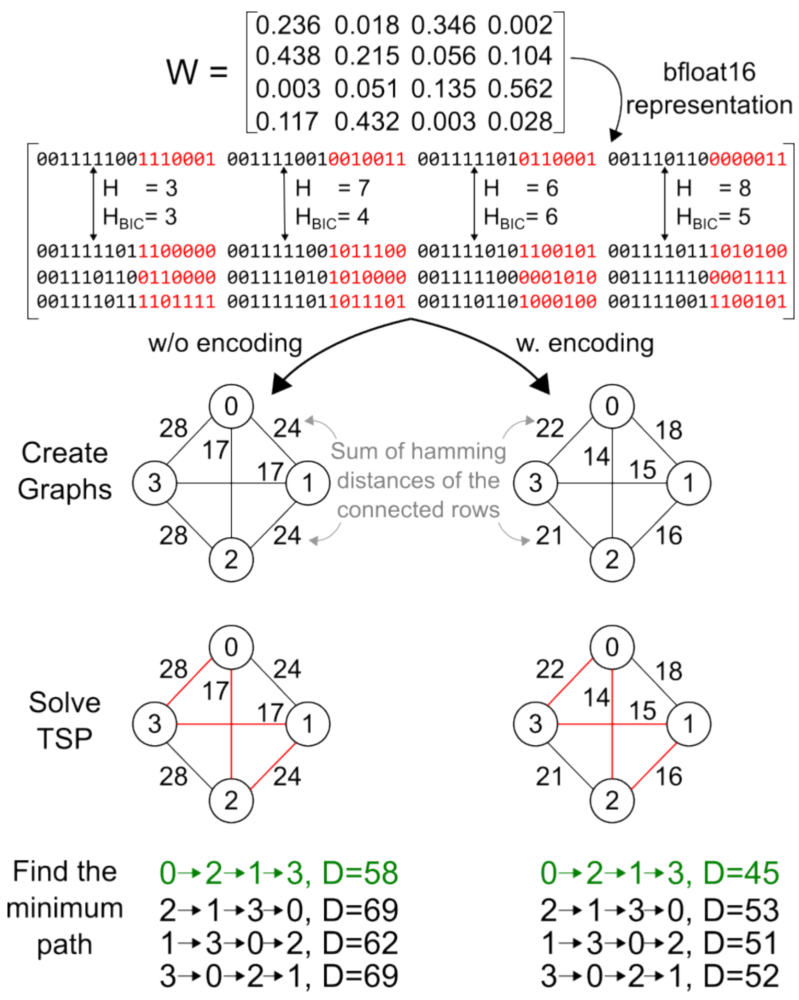


Figure 3.6: A simple walk-through example of the proposed weight reordering technique, which can be applied either alone (left column), or in combination with BIC (right column).

has value:

$$\begin{aligned}\Sigma H_{01} = \Sigma H\left(w_{0i}, w_{1i}\right) =& H\left(w_{00}, w_{10}\right) + H\left(w_{01}, w_{11}\right) \\ &+ H\left(w_{02}, w_{12}\right),\end{aligned}$$

where $H\left(w_{00}, w_{10}\right)$ is the Hamming distance between values $w_{00}$ and $w_{10}$.

To further aid the reader's understanding of the proposed weight reordering technique, we conclude this section with a simple walk-through example. Let us assume that we wish to reorder the rows of a $4 \times 4$ weight matrix, $W$, as per the previously described methodology. Figure 3.6 illustrates all the pertinent steps required to identify the final row order that minimizes the switching activity during the weight-loading process. First, the weight values within matrix $W$ are converted into their binary Bfloat16 format. The 'exponent' and 'sign' fields of each weight value are shown in black, while the 'mantissa' field is shown in red. The step-by-step procedure in Figure 3.6 is presented in two columns: the left column assumes that no BIC is applied to the weight values, while the right column assumes that BIC is applied dynamically at the hardware level.

In both cases, the next step, after converting to the Bfloat16 format, is to calculate the Hamming distance between every pair of values in each of the four columns of $W$. In the case when BIC is applied (right column), we assume that the mantissa (red) bits are transmitted using the BIC hardware logic presented in Section 3.1.2. Indicatively, the calculated Hamming distances between the four corresponding elements of rows 0 and 1 in the $W$ matrix of Figure 3.6 are shown next to the arrows connecting the elements, for both examined cases, i.e., when no BIC is applied (‘$H$’) and when BIC is applied (‘$H_{BIC}$’). This calculation must be performed for every possible pairing of the rows of matrix $W$. The calculated Hamming distances are subsequently used in the fully-connected graph that is used in the TSP formulation. The four nodes of this graph represent the four rows of weight matrix $W$, while the edges’ values represent the sum of the Hamming distances between all corresponding elements of the two connected rows. For example, if no BIC is employed (left graph), the value of the edge connecting nodes 0 and 1 is simply the addition of the four ‘$H$’ values: $3 + 7 + 6 + 8 = 24$. If BIC is employed at the hardware level (right graph), the value of the edge connecting nodes 0 and 1 is the addition of the four ‘$H_{BIC}$’ values: $3 + 4 + 6 + 5 = 18$.

The brute-force TSP formulation provides the best quality of results (i.e., the minimum path). However, its complexity scales exponentially with the size of the graph, thereby making it computationally infeasible, even for relatively small graphs. Consequently, various heuristics are commonly used to solve the TSP for bigger graphs. In our case, the Simulated-Annealing heuristic is employed, which provides satisfactory solutions with reasonable run-times, even for quite large graphs.

The solution of the TSP provides the cycle connecting all the nodes with the minimum cost (i.e., minimum Hamming distance, in our case). Since we do not want a *cycle*, but a *path* connecting all the nodes, we must examine all four possibilities for a starting node, i.e., we test each node in the graph as being the start of the path, as shown at the bottom of Figure 3.6. Of the four examined paths, we choose the one (highlighted in green) that results in the minimum total distance. This path, derived as $0 \rightarrow 2 \rightarrow 1 \rightarrow 3$ in our example, is the new order of the *rows* of the weight matrix $W$, and the corresponding new order of the *columns* of the input matrix. Note that the minimum Hamming distance is significantly smaller (45 vs. 58) in the case where BIC is applied in conjunction with reordering. This observation reinforces our assertion that the *combined* effect of BIC and weight reordering would yield improved power savings, as will be demonstrated shortly.

The steps of the weight reordering technique described in this Section are summarized in Algorithm 1.

In principle, the weight reordering methodology described above could be implemented as a standalone offline technique, without any reliance on a dynamic BIC mechanism at the hardware level (see Section 3.1.2). Nevertheless, in the proposed mechanism, the overarching goal is to combine various techniques that synergistically reduce the switching activity in the SA to a minimum. Thus, the proposed offline (static) weight reordering approach assumes the presence of (dynamic) BIC in the hardware implementation, thereby combining the effects of both mechanisms to yield even higher power savings during weight loading. To reap these combined benefits, the formulated TSP is modified to take into consideration the application of BIC. Specifically, instead of using the *actual* Hamming distance between consecutive values as edges on the graph, we use the Hamming distance *after* the application of BIC on the mantissas of the weight values.

To isolate the individual contribution of each of the two techniques (weight reordering and BIC), the

**Algorithm 1** Weight reordering

```
Input: weight matrix $W[K][M]$
Output: reordered weight matrix $\widehat{W}[K][M]$

//Step 1: Create graph with adjacency matrix $A$
for i=0 to K-1 do
    for j=0 to K-1 do
        for m=0 to M-1 do
            wa = BIC($W[i][m]$)
            wb = BIC($W[j][m]$)
            $A[i][j]$+=HammingDist(wa, wb)
        end for
    end for
end for

//Step 2: Solve the TSP
min_cycle = TSP($A$)

//Step 3: Find the source node that gives the shortest path
minDistance = inf
minSrc = None
for i=0 to K-1 do
    distance = 0
    for v=0 to K-1 do
        distance += $A[min\_cycle[v]][min\_cycle[v+1]]$
    end for
    if $distance < minDistance$ then
        minDistance = distance
        minSrc = i
    end if
    RotateLeftOnePosition(min_cycle)
end for

//Step 4: Reorder weight matrix
RotateToPosition(min_cycle,minSrc)//begin from minSrc
for i=0 to K-1 do
    for j=0 to M-1 do
        $\widehat{W}[i][j]$ = $W[min\_cycle[i]][j]$
    end for
end for
```

experimental evaluation will also present results with the mechanisms operating independently/alone. In this way, the benefit of their *combined* application will be further elucidated.

## 3.3 Experimental results

In this section, we demonstrate the effectiveness of combining various techniques to reduce the data switching activity and, consequently, the overall power consumption. Since this mechanism proposes both *offline* and *dynamic* techniques, it is imperative – for insightful analysis – (a) to isolate the benefits reaped from each of the two different approaches, and (b) to demonstrate the additional benefit obtained when the dynamic and static mechanisms are combined.

Toward this goal, the following experiments include a total of four different setups:

1. A conventional SA with no power-saving features (‘Conventional’).

2. A conventional SA that employs only the presented offline weight reordering technique (‘Conventional+WR’). This setup represents an *offline-only* approach.
3. The proposed architecture employing only the two *dynamic* techniques, i.e., zero-value clock gating for the inputs and BIC for the weights (‘Proposed_Dynamic_Only’). This is the design in Figure 3.2.
4. The proposed architecture employing the two dynamic techniques *and* the offline weight reordering technique (‘Proposed_Dynamic+WR’). This last setup represents the design that combines all the proposed *dynamic+offline* techniques to achieve the most power savings.

All the architectures under comparison were implemented in C++ and synthesized to Verilog RTL using Catapult HLS, using a commercial-grade 45-nm standard-cell library. Both SA designs (i.e., proposed and conventional) operate at a clock frequency of 500 MHz. The Bfloat16 multiply and add operators were implemented using Catapult’s built-in floating-point data types. The final timing/area results were derived from the Oasys logic synthesis tool. The power consumption was estimated after synthesis using the PowerPro power analysis and optimization tool.

The hardware area overhead incurred by the extra logic in the proposed design is a minimal 3.2% for a 32×32 SA size. It is important to note that this percentage overhead will, in fact, decrease with larger SA sizes, because the number of additional encoders scales *linearly* with the size of the SA, whereas the number of additional PEs scales *quadratically*. For instance, the area overhead for an SA of size 64×64 is 1.2%. Note that the weight reordering technique is a purely offline technique that does not require any hardware support; i.e., no hardware area overhead is incurred.

In any case, the small area overhead due to the two dynamic techniques is fully amortized by the larger reaped power savings, as illustrated in Figures 3.7, 3.8, and Table 3.2. The power consumption numbers are the average of 100 randomly picked images from the ImageNet database [60].

Figure 3.7 reports the power consumption in selected layers of three state-of-the-art CNNs: ResNet50 [2], DenseNet121 [68], and MobileNet [57], for a 32×32 SA. As illustrated, the proposed techniques – in all presented setups – effectively reduce the power consumption. As expected, the reduction is more pronounced when the dynamic and static techniques are combined. More specifically, weight reordering, by itself, yields per-layer power savings of 1.5%–23% in the ‘Conventional+WR’ setup. This is the best that the offline-only approach can achieve. The proposed architecture employing only the two dynamic techniques of zero-value clock gating and BIC (‘Proposed_Dynamic_Only’) yields per-layer power savings of 3.6%–35.7%. The combination of the two dynamic techniques and offline weight reordering (‘Proposed_Dynamic+WR’) yields 10.8%–44.6% per-layer power savings, as compared to the conventional SA (‘Conventional’). The obtained per-layer savings translate to the *overall power reductions* reported in Table 3.2, for the three state-of-the-art CNNs examined.

The power consumption in each layer is dependent on the number of zero-value inputs. In the layers where the percentage of zero-value inputs is high, the power consumption in the proposed design is – in most cases – much lower than in the conventional SA, due to the extensive use of the zero-value clock gating technique. Nevertheless, when the number of zero values becomes very high, there are more cases of multiple consecutive zero inputs. Naturally, these cases also benefit the power consumption of the conventional SA design.

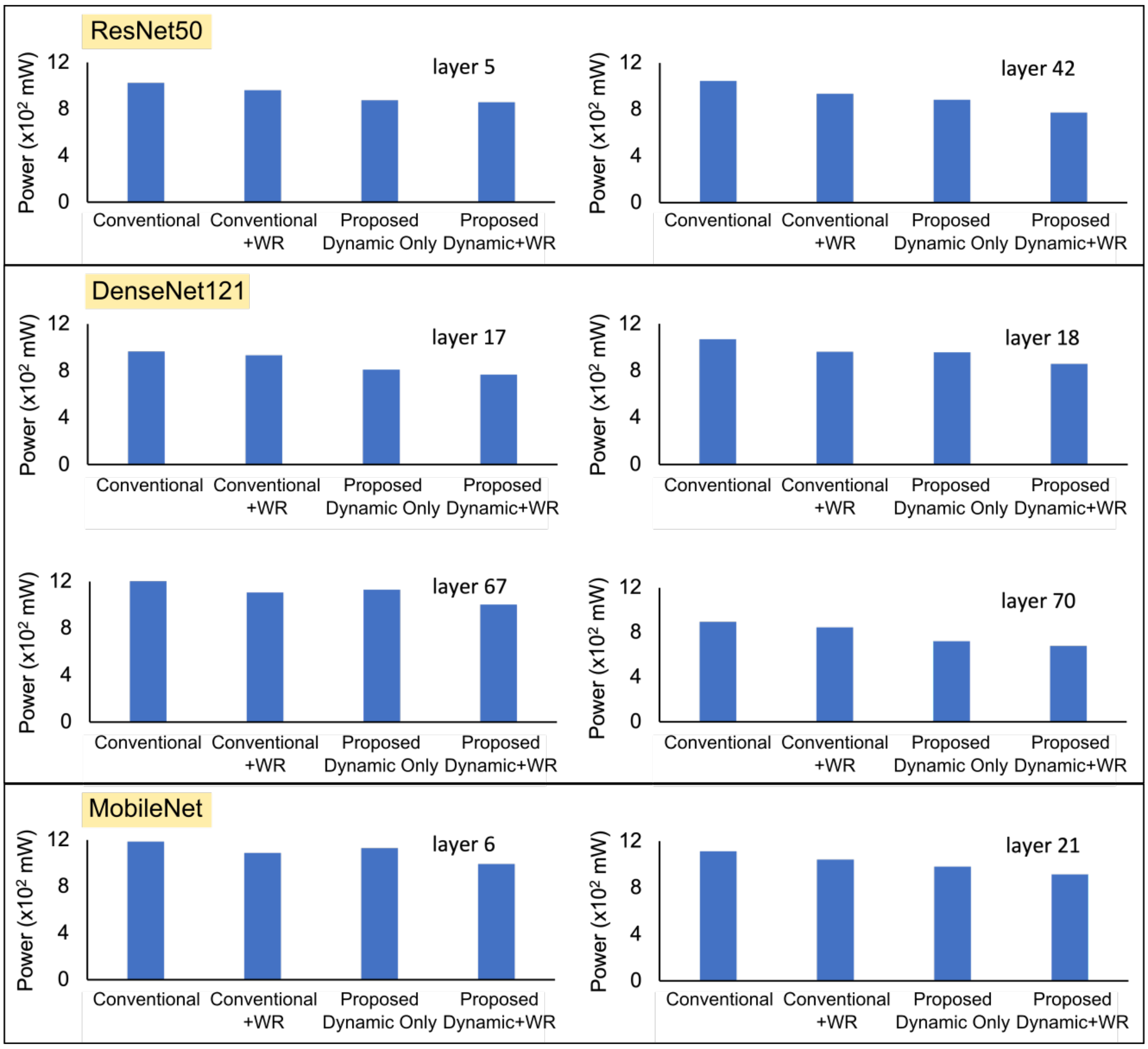


Figure 3.7: Power consumption for selected layers of ResNet50 [2], DenseNet121 [68], and MobileNet [57]

Furthermore, the reordering of the columns of the input matrix caused by the corresponding reordering of the rows of the weight matrix (to retain the correctness of the matrix multiplication, as explained in Section 3.2) inevitably alters the switching activity of the inputs. Thus, this column reordering in the input matrix may, in fact, cause an increase in the switching activity of the input, which may lead to limited overall power savings in some layers. As illustrated in Figure 3.8, for smaller arrays (16×16), the reaped power reduction is diminished, since the power overheads of the encoding mechanisms themselves begin to outweigh the benefits from the reduction of the data switching activity. As already noted, the number of additional encoders scales linearly with the size of the array, while the number of PEs scales quadratically, yielding a smaller $\frac{Encoding\ Logic}{PE\ Logic}$ ratio. Thus, for larger SAs (32×32 and 64×64), the obtained power reductions are larger, thereby indicating that the proposed techniques scale well with SA size and provide increasing benefits with increasing array sizes. The average per-layer power consumption is reduced in all setups of the proposed design, with the most benefits reaped when combining all three techniques (‘Proposed_Dynamic+WR’), i.e., zero-value clock gating for the inputs and BIC+weight-reordering for the weights.

Table 3.2: The overall power savings obtained by each setup of the proposed architecture, as compared to a conventional SA, when executing three state-of-the-art CNNs on a 32×32 SA.

| | Conventional + WR | Proposed Dynamic Only | Proposed Dynamic + WR |
|---|---|---|---|
| ResNet50 [2] | 8.8% | 11.8% | 20.5% |
| DenseNet121 [68] | 8.9% | 15.5% | 23% |
| MobileNet [57] | 9.3% | 8.7% | 17.1% |

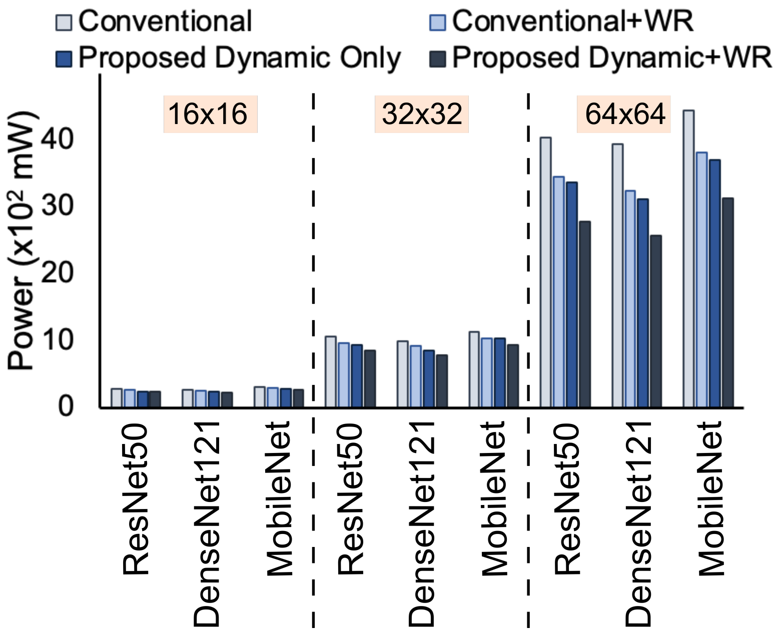


Figure 3.8: Average per-layer power consumption when executing the ResNet50 [2], DenseNet121 [68], and MobileNet [57] CNNs, for three SA sizes: 16×16, 32×32, and 64×64.

## 3.4 Related work

There is a rich body of previous work aiming to improve the power consumption by either applying data reordering schemes, or data encoding techniques. These prior efforts mostly targeted data buses and streaming data accesses. Instead, the proposed mechanism is **the first**, to the best of our knowledge, that combines dynamic (data encoding) and static (data reordering) techniques and applies them synergistically **on *systolic arrays*** to reduce the power consumed by the data flowing through the array.

The first set of related work aims to minimize the switching activity of data movement. A collection of high-level synthesis techniques have been proposed to reduce the power consumed within Digital Signal Processing (DSP) units [69]. One of these techniques is the so called *"operand reordering,"* which tries to minimize the switching activity of the inputs. In a similar vein, the work in [70] introduces novel cost functions related to the ordering of operations. These power costs are used as edge weights to build an operation graph, in an effort to determine the operation order with the minimum cost. This is achieved using the TSP algorithm, and the resulting minimum-cost operation order is used to synthesize low-power sum-of-products computations. After analyzing the effects of floating-point data values on switching activity, the authors of [71] introduce a new model that relates the data characteristics to the switching activity of a datapath. Additionally, they propose a heuristic algorithm that lower switching activity during high-level design space exploration. Similarly, the work in [72] reorders sparse weights derived from pruned neural networks to minimize the power consumed in data accesses in GPUs.

The second set of related prior work focuses on data encoding. Encoding of values transmitted over buses is a very common methodology to reduce the power consumption and several techniques have been proposed [73]. One approach is to transmit on the bus the difference between the current and the previous values [74]. Thus, if the consecutive values to be transmitted over the bus have the same offset, the actually transmitted value remains constant. Similarly, consecutive data similarity can be exploited to transmit the bitwise-XOR operation between the current and the previously transmitted values [75], to minimize the number of ones in the transmitted data. Following the same philosophy, sequential addressing patterns can benefit from an encoding scheme that eliminates the transmission of consecutive addresses by merely sending an *increment* value [76]. Consequently, for consecutive addresses, the transmitted value remains constant and the decoder knows that the previous value must be incremented by a known stride. In a more complex scheme, the authors of [77] combine the encoding technique of [76] with BIC and selectively apply these on instruction and data addresses transmitted over multiplexed buses, to maximize the potential power improvement. Instead of targeting the *address* bus, the work in [78] reduces the power consumption associated with the actual *data* transmission. Said work leverages the small number of distinct *frequent values* that are observed to be frequently transmitted over data buses. Frequent values are thereby transmitted over the bus in an encoded one-hot word, while non-frequent values are transmitted in their original form. In this way, the maximum hamming distance between every pair of frequent values is two. Finally, an approximate encoding technique has also been proposed, which exploits the inherent error resilience of the target applications to reduce the power consumed in DRAM channels [79].

The third set of related work targets low-power SA configurations. HeSA [80] tackles the PE underutilization caused by the application of compact CNN models on larger SAs. The power consumption is reduced by improving the SA utilization with a heterogeneous architecture. On the other hand, Gemmini [81] provides an SA generator that improves the power consumption by alternating the pipeline stages between the neighboring PEs. In a similar vein, ArrayFlex [49] proposes an on-the-fly reconfigurable SA architecture that reduces the power consumption by selectively fusing the pipeline stages in the SA.

## 3.5 Conclusions

Even though data- and weight-loading constitute only a portion of the total power consumed in a systolic array, a reduction in switching activity while traversing the array still yields significant overall power savings. The proposed SA architecture reduces the overall power consumption by employing a combination of dynamic and static techniques. On the dynamic side, the proposed architecture appropriately encodes the data flowing through the array to reduce the switching activity. By exploiting key attributes of modern CNNs, such as the value distribution of the weights and the prevalence of zero values in the inputs, the new design applies the most appropriate scheme to each data type to maximize the power savings with the minimum hardware cost. To further reduce the switching activity, the weight matrix is reordered offline to minimize the Hamming distance between consecutive values, as the weights are repeatedly loaded into the array. The combination of targeted, dynamic application-aware encoding and offline weight reordering yields overall power savings of 17.1%–23% in state-of-the-art CNN applications running on an SA of size 32×32. More importantly, these power savings are demonstrated to scale with the array size, obtaining 29.7%–35.4% power savings in an array of size 64×64.

# Chapter 4

# Architectures for Sparse Systolic Arrays

The success of convolutional neural networks (CNNs) across vision and language tasks has driven widespread deployment of deep learning models on mobile and embedded platforms. However, the high computational demands of these models pose challenges for resource-constrained environments. In particular, general matrix multiplication (GEMM), which serves as the backbone of convolutional and fully connected layers, requires significant energy and memory bandwidth. To address these challenges, pruning has emerged as a promising approach, reducing both computation and storage overhead by eliminating redundant weights. There are two main types of sparsity. Under *unstructured* sparsity [33], there are no constraints on the distribution of the non-zero elements in the input, as abstractly illustrated in Figure 4.1(a). Consequently, extensive meta-data and multiple indexes are required to encode the non-zero locations. On the contrary, *structured* sparsity [31, 32] alleviates the meta-data cost and complexity by enforcing a constraint on the maximum possible number of non-zero elements present in every fixed-size block of consecutive input elements. Structured sparsity is typically defined by the notation $N{:}M$, which indicates that in each block of $M$ consecutive elements, at most $N$ may be non-zero. Figure 4.1(b) shows an example of a 2:4 structured sparse input matrix. The positions of the non-zero elements within each block is decided by the stored 4-bit masks. Building upon this idea, Liu et al. propose the *Systolic Tensor Array* (STA) [37] and its extension, the *Sparse Systolic Tensor Array* (S2TA) [36], as energy-efficient accelerators that exploit structured sparsity for mobile CNN inference.

The core principle behind systolic tensor arrays is to generalize the traditional systolic array architecture used in dense GEMM operations, while incorporating mechanisms to skip unnecessary computation induced by structured sparsity. In dense systolic arrays, each processing element (PE) performs multiply-accumulate (MAC) operations in a regular, cycle-by-cycle manner, leading to high utilization under dense workloads. However, when sparsity is introduced, many of these MAC operations become redundant. STA addresses this by mapping structured-sparse GEMM operations onto a tensorized systolic array, shown in Figure 4.2, which can effectively bypass inactive computations while retaining the regular dataflow patterns that make systolic arrays hardware-friendly. This design balances the tradeoff between sparsity exploitation and hardware regularity, achieving energy savings without sacrificing high throughput.

A sparse systolic tensor array performs the matrix multiplication $C = AW$ between a dense input matrix $A$ and a structured-sparse weight matrix $W$. Architecturally, it follows the same systolic array

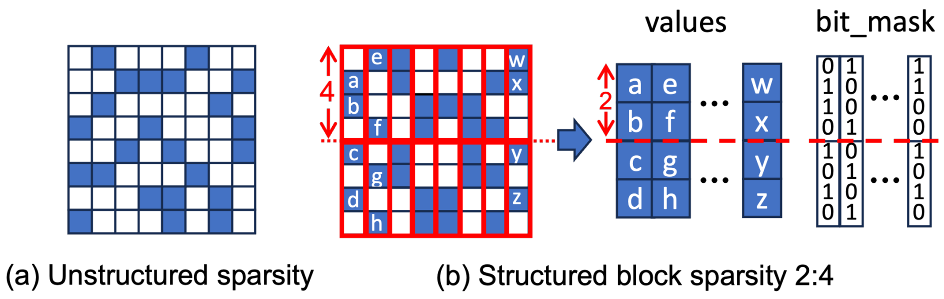


Figure 4.1: Example of (a) unstructured sparsity; and (b) structured block sparsity of 2:4 (i.e., up to 2 non-zero elements in every 4 consecutive elements in each column) and their respective packed storage with their associated bit masks.

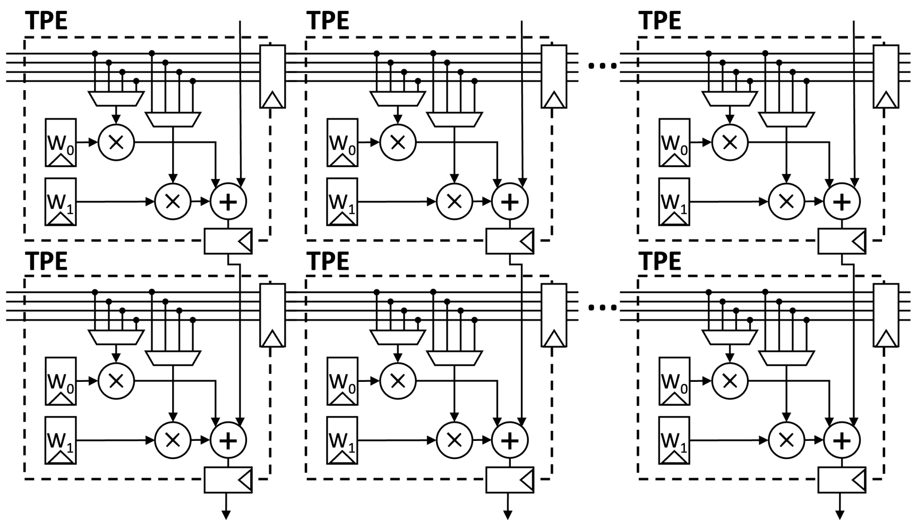


Figure 4.2: A sparse systolic tensor array that employs the weight-stationary dataflow; i.e., the weights are pre-loaded into the Tensor Processing Elements (TPE) and remain stationary during the operations. The inputs and outputs flow in the horizontal (west-to-east) and vertical (north-to-south) directions, respectively.

organization as a conventional *dense* array; however, instead of employing simple *scalar* Processing Elements (PEs) as its fundamental units, it utilizes more sophisticated *Tensor* PEs (TPEs) [37]. In a traditional weight-stationary (WS) dataflow, scalar PEs accept a single input activation per cycle and multiply it with a single locally stored weight, performing one multiply-accumulate (MAC) operation per cycle. By contrast, Tensor PEs process an entire *block* of $M$ consecutive input activations simultaneously, and can store up to $N$ local weights, thereby performing as many as $N$ multiplications in each MAC operation. This enhanced microarchitecture enables support for structured block sparsity with an $N$:$M$ ratio. Figure 4.2 illustrates the design of a two-row sparse tensor array, which is configurable to support both 2:4 and 1:4 sparsity patterns.

As depicted in the figure, each TPE contains two weight registers that hold up to two non-zero weights selected from every group of four pruned weight elements. These registers are populated during a distinct weight-loading phase, consistent with the WS dataflow. During the subsequent computation phase, four consecutive input activations from the same row of $A$ are streamed into each TPE from the west. The TPE then selects up to two of these four inputs—based on the indices of the nonzero weights—using two 4-to-1 multiplexers. Each selected activation is multiplied with its corresponding locally stored weight, and the resulting products are accumulated vertically across the columns of the tensor array. When the 1:4 sparsity pattern is applied, only one multiplexer and multiplier pair is active within each TPE, further reducing computation. This process is repeated iteratively across all rows of $A$, with each TPE receiving input blocks of four consecutive elements, while the stationary weights determine which values participate in the MAC operations.

The follow-up work, S2TA [36], further refines this approach by introducing architectural mechanisms to more aggressively exploit structured sparsity for mobile CNN acceleration. S2TA integrates specialized control logic within the systolic array to detect and skip zero blocks dynamically, while maintaining pipeline regularity. Moreover, it introduces a block-level compression scheme that reduces memory bandwidth requirements by storing only the nonzero tensor tiles. This compressed representation is directly compatible with the systolic dataflow, minimizing decompression overhead. As a result, S2TA achieves both computational and memory-level efficiency, making it highly suitable for energy-constrained mobile devices.

Beyond their immediate contributions, STA and S2TA also provide a blueprint for future accelerator designs targeting sparsity. The tensorized systolic array abstraction allows for the seamless integration of structured pruning techniques developed in the machine learning community, while the hardware-friendly dataflow ensures scalability and energy efficiency. As models continue to grow in size, the importance of sparsity-aware accelerators will only increase. In particular, structured sparsity offers a sustainable path forward, enabling efficient deployment of state-of-the-art CNNs on resource-limited platforms such as smartphones, wearables, and edge devices.

## 4.1 DeMM: A Decoupled Matrix Multiplication Engine Supporting Relaxed Structured Sparsity

In most practical applications that use structured sparsity [31, 36, 37], blocks are small and *fine-grained* $N$:$M$ sparsity patterns of 1:2, 1:4 or 2:4 are supported, where each block of $M$ elements may contain up to $N$ non-zero elements. Nevertheless, while fine-grained structured sparsity promises high performance and low storage overhead, it may also lead to less accurate ML models [82, 83]. This possible weakness is attributed to the constraints imposed during the fine-grained sparsification, where a fixed amount of non-zero elements is required for all consecutive small blocks.

To increase the flexibility during model training, sparsity could refer to much coarser blocks [38, 84]. For instance, a *relaxed* (coarser) structured sparsity of 8:128 allows the presence of at most 8 non-zero elements in every 128 consecutive elements. Figure 4.3 shows an example of 4:16 relaxed structured sparsity, together with the packed representation of each row, which contains the non-zero element values and their corresponding column indexes. Moving to coarser blocks complicates the operation of the corresponding hardware modules, e.g. systolic arrays, that operate optimally on well-structured data with small block sizes.

To effectively reconcile these two conflicting attributes of relaxed (coarser) sparsity vs. hardware

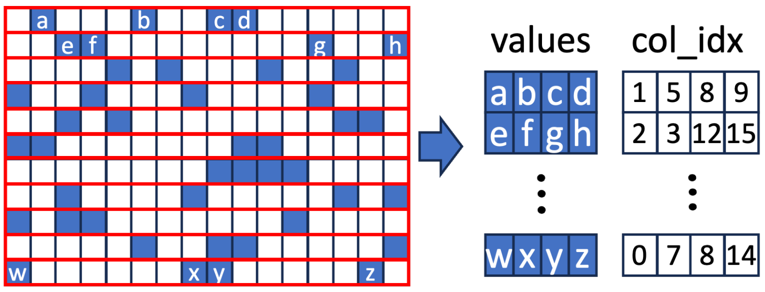


Figure 4.3: Example of relaxed structured sparsity 4:16, and the corresponding packed representation of the non-zero elements. A blue square indicates a non-zero element.

complexity, in this section we propose a novel matrix-multiplication engine that supports *relaxed* structured sparsity patterns, while still employing a simple and *decoupled* hardware organization. Unlike conventional systolic arrays that co-locate the Multiply-Accumulate (MAC) and storage units within each tile, the proposed <u>De</u>coupled <u>M</u>atrix-<u>M</u>ultiplication (DeMM) engine decouples the two. It essentially re-organizes the (dispersed) memory portion of a traditional systolic array into a regular standard-cell memory structure with multiple read ports. This transformation enables the support of relaxed structured sparsity and maintains the required regularities in the data flow and the physical layout.

Overall, the proposed DeMM engine provides a two-fold benefit. First, it enables support for relaxed structured sparsity patterns that combine hardware simplicity (similar to handling fine-grained sparsity) with additional flexibility during DL model pruning [82,83]. Secondly, through appropriate reconfiguration, the DeMM engine can also support denser sparsity, which allows for the tackling of more common fine-grained structured sparsity patterns [31].

The experimental results demonstrate substantial improvements in overall execution latency over state-of-the-art matrix engines built to support fine-grained [36] and relaxed structured sparsity [38,85] when executing structured-sparse CNN models. It should be noted that, even though said approaches, including the proposed DeMM engine, are effective for the low-sparsity levels of DNNs (i.e., 10%-90%), they are not as efficient in high-sparsity levels of above 95%. At such high-sparsity levels, other accelerator architectures perform better. Examples include architectures following a non-systolic dataflow-like organization [86,87] and ones that optimize memory traffic by operating either near-memory [88,89], or in-memory [90].

### 4.1.1 Simplifying Sparse×Dense Matrix Multiplication

The proposed DeMM engine employs a row-wise approach [91] in computing the matrix product $A \times B$. Matrix $A$ follows a relaxed structured sparsity template and $B$ is dense. The product of the multiplication is produced row-by-row, as follows:

$$C[i,:] = \sum_k A[i,k]B[k,:] \tag{4.1}$$

All the non-zero elements in a single row of matrix $A$ should be multiplied in parallel with the corresponding rows of matrix $B$, where the row index of matrix $B$ is determined by the column index of the

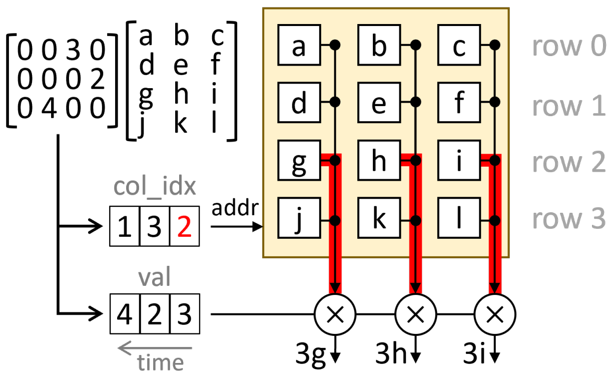


Figure 4.4: Read and multiply operations are sufficient to perform matrix multiplication when the sparse matrix contains at most one non-zero element per row.

non-zero value in matrix $A$.

## The proposed DeMM engine

To achieve the desired parallelism, we *decouple* the *storage* (as used in a systolic array) from the *multiply-add* units and treat each portion separately. Matrix $B$ is assumed to be pre-loaded in the storage area of DeMM. The pre-loading resembles the pre-loading operation of the input- (or weight-) stationary dataflow applied in systolic arrays. In each cycle, another row of $B$ is written into the memory block, using the one available write port. Subsequently, multiplication is computed row-by-row, by feeding the engine with all the non-zero elements of each row of stuctured-sparse matrix $A$.

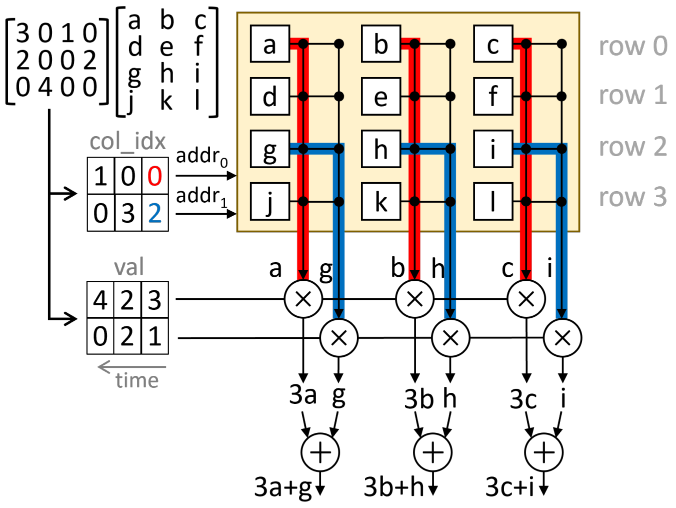


Figure 4.5: Multiplying a sparse matrix with at most two non-zero elements per row requires two separate memory read ports and two rows of multipliers. The products of each port are then independently added in parallel to form the final result of the output row.

Let us initially assume the trivial case where each row of $A$ consists of at most one non-zero element. This element is passed to the engine, together with its column index, one after the other. For each {value, column_index} pair, multiplication is performed in two serial steps, as depicted in Figure 4.4. First, the column index is treated as a row address to the memory that stores matrix $B$. This address allows us to read out all elements of the corresponding row of $B$. In the second step, the read elements are all multiplied in parallel with the value of the non-zero element of $A$. Repeating these two steps for all rows of matrix $A$ would complete the multiplication. The hardware engine required in this trivial case of a single non-zero element per row is just a memory block with 1 read port and a row of multipliers.

To support more than one non-zero element per row, one must simply increase the number of read ports in the memory block and, correspondingly, the number of multipliers per read port. Adders are also needed for the final addition. Figure 4.5 illustrates an example of the operation of the proposed engine when operating with a row sparsity of two non-zero elements per row. In this case, the pairs of non-zero elements of each row of matrix $A$ are sent in parallel to the multiplication engine, one after the other. Each non-zero element is forwarded to its dedicated read port. The column index of each non-zero element selects (reads) the corresponding row of matrix $B$, and the value of each non-zero element is multiplied in parallel with all the elements of the selected row. The two products generated at each read port are added to finalize the result for this output row.

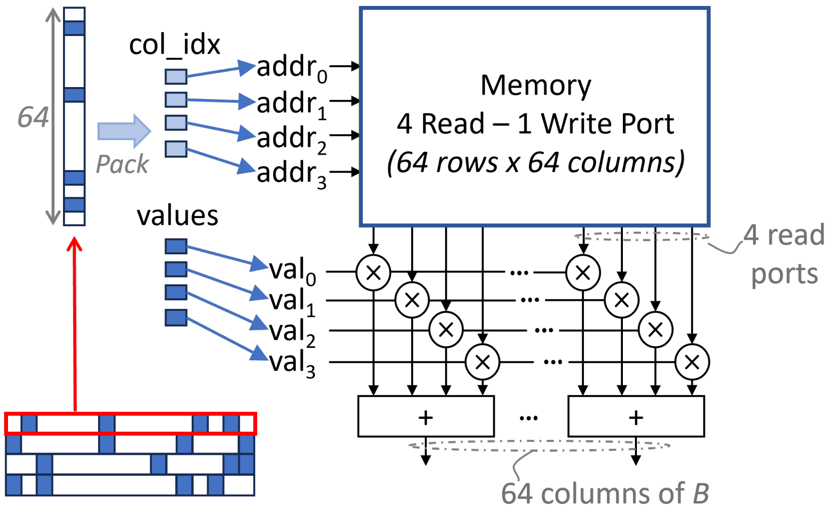


Figure 4.6: The overall organization of a DeMM engine that supports relaxed structured sparsity of 4:64 using a memory block of four read ports and four multipliers and one add-reduction unit per output element. The example assumes that 64 outputs (columns) are computed in parallel.

In the general case, matrix $A$ follows an $N$:$M$ row-sparsity pattern, where $M$ is much larger than $N$, e.g., $N$=8 and $M$=128, or $M$=256. Hence, the proposed DeMM engine consists of a regular memory of $N$ read and 1 write port. Each read port of DeMM's memory block outputs one data item per column, as selected by the column index address ('col_idx') , which points to a row of matrix $B$. Figure 4.6 depicts a complete DeMM engine supporting 4:64 structured sparsity ($N$=4 and $M$=64). Each read port is associated with a multiplier and the products of all read ports are reduced to one sum. Summation at the bottom of Figure 4.6 is implemented as a pipelined multi-operand adder of logarithmic depth.

**Supporting denser structured sparsity**

To support denser structured sparsity, e.g., $kN{:}M$, in a reconfigurable manner, DeMM should be able to read more than $N$ non-zero elements from *the same* $M$ rows of matrix $B$. Since the memory block of each DeMM engine consists of $N$ read ports, it means that reading the $kN$ non-zero elements of the same row of $A$ requires time-sharing of the $N$ read ports for $k\times$ more cycles. To enable this sharing, every read port is associated with a $k$-to-1 multiplexer. Note that, irrespective of the exact structured sparsity pattern supported, the memory of the DeMM engine is pre-loaded with the same $M$ rows of matrix $B$. The value of $k$ just determines how many times this block would be read before completing the computation for a row of $A$.

The overall organization of the DeMM engine is depicted in Figure 4.7. The value chosen for $k$ reflects the reconfiguration properties of the proposed engine. For instance, by selecting $k$=4 for the engine depicted in Figure 4.6, which operates by default on 4:64 relaxed structured sparsity ($N$=4 and $M$=64), it means that all denser (and more fine-grained) structured sparsities can also be supported by the proposed design, e.g., 4:32 (as 8:64), 4:16 (as 16:64). Moving to an even denser scenario, such as 4:8, implemented in DeMM as a 32:64 pattern, would need larger multiplexers at each read port, i.e., $k$=8.

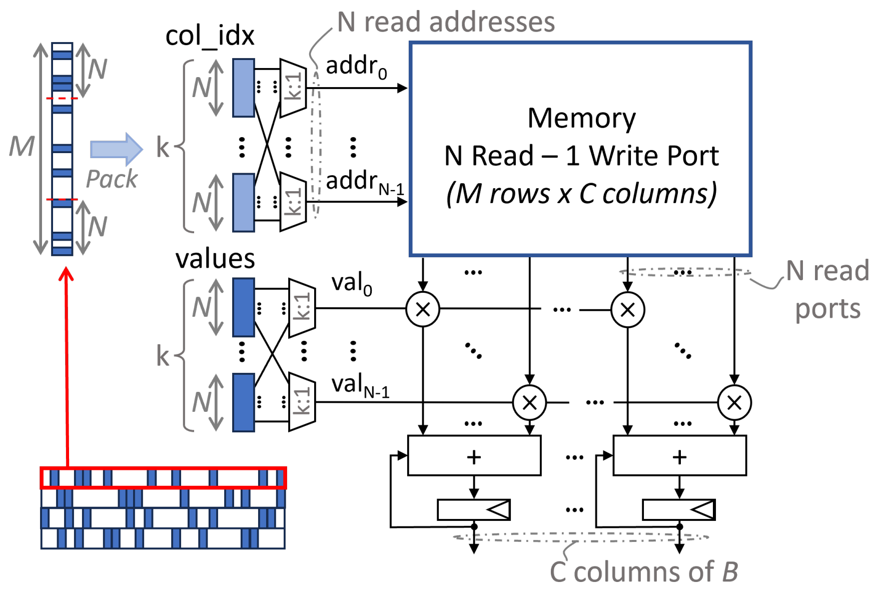


Figure 4.7: The overall architecture of the DeMM engine that supports an $N{:}M$ relaxed structured sparsity and can be reconfigured for all $kN{:}M$ denser variants.

To identify the various design options, we define the DeMM($N, M, C, k$) configuration as the one that operates on a structured-sparse matrix $A$ of row size $M$ (this is also the number of rows in matrix $B$) and a matrix $B$ with $C$ columns. Structured sparsity can be as relaxed as $N{:}M$, or as dense as $kN{:}M$. The corresponding hardware block requires $N \times C$ multipliers, $C$ $N$-to-1 reduction trees, and an $M \times C$ memory block of $N$ read ports.

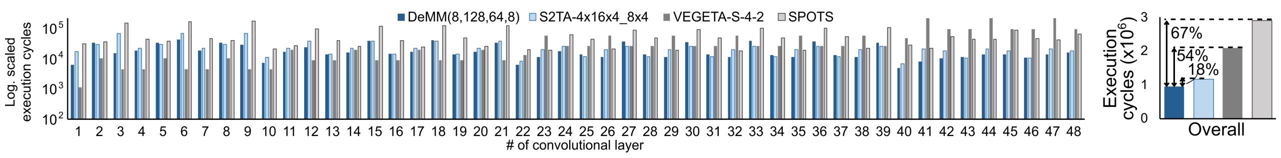


Figure 4.8: The execution latencies of all CNN layers of ResNet50 [2], when using the proposed DeMM(8,128, 64, 8) design, the S2TA-4×16×4_8×4 design [36], the VEGETA-S-4-2 design [38], and the SPOTS design [85]. All three architectures have equal amount of computational resources.

## 4.1.2 Evaluation

The effectiveness of the proposed decoupled matrix engine is evaluated by running inference in state-of-the-art CNN applications [2, 56] with inputs from ImageNet. In the first set of experiments, we consider highly sparse CNN models derived from unstructured pruning. As reported in [82,83], unstructured pruning can achieve higher model compaction with better performance, as compared to structured pruning. In particular, we employ ResNet50 [2], pruned with RigL [33] at 95% sparsity that roughly matches the relaxed sparsity of 8:128 targeted by the proposed DeMM architecture. Any rows exceeding the sparsity of 8:128 are computed in multiple consecutive cycles. For completeness, in the second set of experiments, we also include scenarios with fine-grained structured sparsity of 1:2, 1:4 and 1:8, derived with Tensorflow for ResNet50 [2] and ConvNeXt [56].

**Relaxed structured sparsity**

For relaxed row sparsity of 8:128, we compare DeMM with three state-of-the-art architectures: (a) VEGETA [38], which is able to support such sparsity degrees in a structured form of 1:16 blocks, operating with a weight-stationary dataflow [43]; (b) a version of S2TA [36] configured to support block density 1:16 using output stationarity; and (c) SPOTS [85], which skips groups of weights and input data that consist of only zero elements, following an output-stationary dataflow. Specifically, we compare DeMM(8,128,64,8) following an input-stationary dataflow to VEGETA-S-4-2 [38], to S2TA-4×16×4_8×4 [36], and to SPOTS [85], with *all* designs under evaluation having *the same amount of computational resources of 512 multiply-add units*. The evaluated VEGETA and S2TA designs have array sizes of 32 × 16 PEs, while SPOTS has an array size of 128 × 4 that can be reconfigured as four 32×4 blocks operating in parallel. The configuration selected is the one that offers the best performance depending on the size of the input matrices.

The obtained results are summarized in Figure 4.8, which shows the execution latencies of all CNN layers of ResNet50 [2]. DeMM's performance in the first layers is not the best, but it substantially outperforms the other three designs in the later layers. This behavior is the combined result of DeMM's engine architecture and the size of the stationary matrices in each case [43]. DeMM leads to an *overall* (across all CNN layers) **latency improvement** of **18%**, **54%** and **67%**, as compared to S2TA [36], VEGETA [38], and SPOTS [85], respectively.

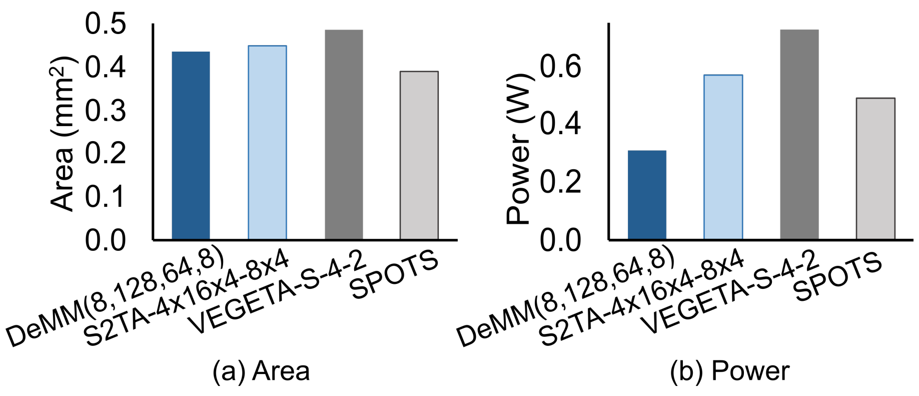


Figure 4.9: Hardware (a) area, and (b) power consumption comparisons between DeMM(8,128,64,8), S2TA-4$\times$16$\times$4_8$\times$4 [36], VEGETA-S-4-2 [38] and SPOTS [85]. DeMM and SPOTS use a block density of 8:128; S2TA and VEGETA use the equivalent 1:16.

### Hardware complexity

All four evaluated designs were implemented in SystemVerilog and synthesized using the Cadence digital implementation flow and a 28 nm standard-cell library. The designs operate on 16-bit integer quantized inputs and weights, while the accumulations are performed with 32 bits. A block density of 8:128 is used, which is DeMM's primary target. However, the instance of DeMM that is evaluated can also support more fine-grained patterns, down to the equivalent of 1:2. The equivalent density for S2TA and VEGETA is 1:16. All designs under evaluation operate at a clock frequency of 500 MHz at 0.88 V.

Figure 4.9(a) compares the hardware area of the four architectures. Compared to S2TA and VEGETA, the DeMM engine requires **2.7%** and **10.4%**, respectively, **lower area**, which is a testament to the simplicity of its organization. DeMM is slightly larger than SPOTS (the overhead is less than 10%), due to the additional multiplexing logic required for supporting reconfigurability and multi-porting. Each additional read port added to the 128$\times$64 standard-cell-based memory block used in this DeMM setup costs 16% more area.

In terms of power consumption, DeMM demonstrates significantly better behavior than the other designs. As shown in Figure 4.9(b), DeMM consumes **36.4%**, **45.8%** and **56.1% lower power** than SPOTS, S2TA, and VEGETA, respectively. This substantial power reduction is mainly attributed to the minimization of data movement in pipeline registers. Both S2TA and VEGETA have an input data demand of a multiple-of-$M$ inputs and $C$ weights, while DeMM has a much lower input data demand of $C$ inputs and $kN$ weights, for a $kN$:$M$ structured sparsity, along with their addresses. The extra hardware cost of VEGETA relative to S2TA stems from its reconfiguration-rich organization that is also offered by the DeMM engine. SPOTS has a low multiplexing overhead that reduces its area requirements. However, its deeply pipelined operation increases its power consumption relative to DeMM.

### Fine-grained structured sparsity

Furthermore, we also compare DeMM to VEGETA and S2TA in use-cases that better fit these two architectures, i.e., in scenarios with fine-grained block densities. The designs are evaluated using ResNet50 [2] and ConvNeXt [56]. The selected workloads are pruned to fine-grained structured block sparsities of 1:2,

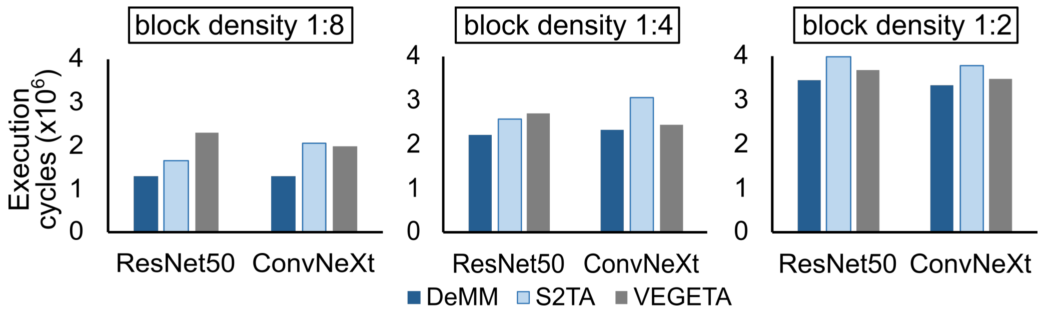


Figure 4.10: Overall execution latencies of ResNet50 [2] and ConvNeXt [56] for configurations with block densities of 1:8, 1:4 and 1:2.

1:4 and 1:8 to ensure optimal conditions for both VEGETA and S2TA, even if this choice is not the best option for DeMM, which inherently supports a wide range of sparsity formats. SPOTS is omitted in this comparison, since, under such fine-grained structured sparsity, it is very difficult to find contiguous groups of zero data, as required by SPOTS. Consequently, SPOTS exhibits significantly higher latencies.

The results are shown in Figure 4.10. DeMM engine *outperforms both* S2TA and VEGETA architectures in terms of overall latency. Specifically, for block sparsity 1:8, DeMM achieves average **latency improvements** of **29%** and **39%**, as compared to S2TA and VEGETA, respectively. For block sparsity 1:4, DeMM's respective improvement in average latency is **19%** and **12%**. Finally, for block sparsity 1:2, DeMM still yields **14%** and **5%** average latency improvements, as compared to S2TA and VEGETA, respectively.

## 4.2 Diagonal Structured Sparse Tensor Arrays

Both dense and sparse SAs require careful input data alignment to maintain synchronized pipelined reductions across processing elements, as illustrated in Figure 2.1. This synchronization is achieved through skewing, typically implemented using shift registers along the western edge of each row. While effective, this approach introduces additional hardware overhead in terms of area and power consumption, necessitating optimization efforts to mitigate these costs while maintaining computational correctness.

In this section, we introduce a Diagonal Structured Sparse Tensor Array (DSSTA) SA architecture, based on [92], that eliminates the need for input skewing, thereby removing the requirement for shift registers. By redesigning the dataflow mechanism, our proposed architecture ensures that input elements arrive at processing elements in a synchronized manner without additional hardware overhead. Furthermore, PE grid and the underlying weight permutations are organized in such a way that minimizes the overall required wire-length of the final layout. This not only reduces area and power consumption but also simplifies circuit design, making it particularly advantageous for energy-efficient ML inference accelerators.

### 4.2.1 DSSTA Architecture

The proposed DSSTA architecture is illustrated in Figure 4.11. Unlike traditional sparse tensor arrays, which employ classic Tensor Processing Elements (TPEs) (Figure 4.2), the DSSTA array is composed of

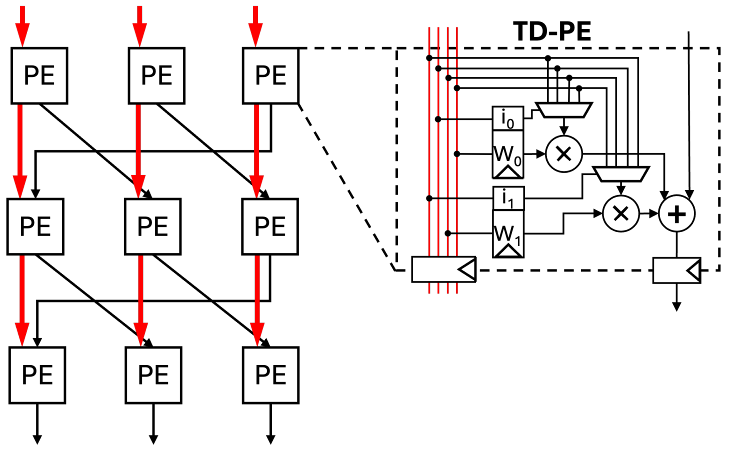


Figure 4.11: Proposed DSSTA architecture that can be configured for 2:4 and 1:4 sparsity patterns

TD-PEs. The primary distinction between these architectures lies in the dataflow direction—rather than being injected from the western edge, as in conventional systolic arrays, input values in DSSTA enter from the northern edge, as indicated by the red paths in Figure 4.11, while the partial sums propagate diagonally through the processing elements (PEs).

Partial sums are accumulated along columns, similar to classical systolic arrays. However, to accommodate the diagonal data movement in DSSTA, the rightmost PE of each row passes its partial sum to the leftmost PE of the row below. This architectural modification preserves the regular accumulation pattern while aligning with the diagonal structure.

Another notable difference is how weights are loaded, rather than utilizing dedicated weight-loading paths, DSSTA repurposes the existing input data pathways, reducing wire length and overall hardware complexity. More specifically, the architecture supports block sparsities of 2:4 and 1:4, meaning each input path carries four values per cycle. During the weight-loading phase, one or two of these values are used to transmit weights, while the remaining one or two values encode the corresponding index information, which typically requires fewer bits than the weight and input values. This approach optimizes data transmission efficiency without introducing additional routing overhead. While to minimize power consumption, multipliers within the PEs are gated off during weight loading, ensuring that unnecessary computations do not take place.

To support the proposed diagonal dataflow, the loaded weight matrix should be permuted offline before loading. The required weight permutation is depicted in Figure 4.12(a). According to this, each column is first permuted and shifted by each column index. Thus, the first column is not permuted, the second column is rotated by one element, the third column rotated by two elements and so on. Hereinafter, each row is permuted by its row index, resulting in the depicted diagonal structure. More specifically, the first row remains as is, the second row is permuted by one element, i.e. the last element becomes first, the third row is permuted by two elements and so forth.

In Figure 4.12 (b)-(i), we present a running example of a $3-by-3$ DSSTA array. Let's assume that we have to compute the matrix product $AW = C$ with:

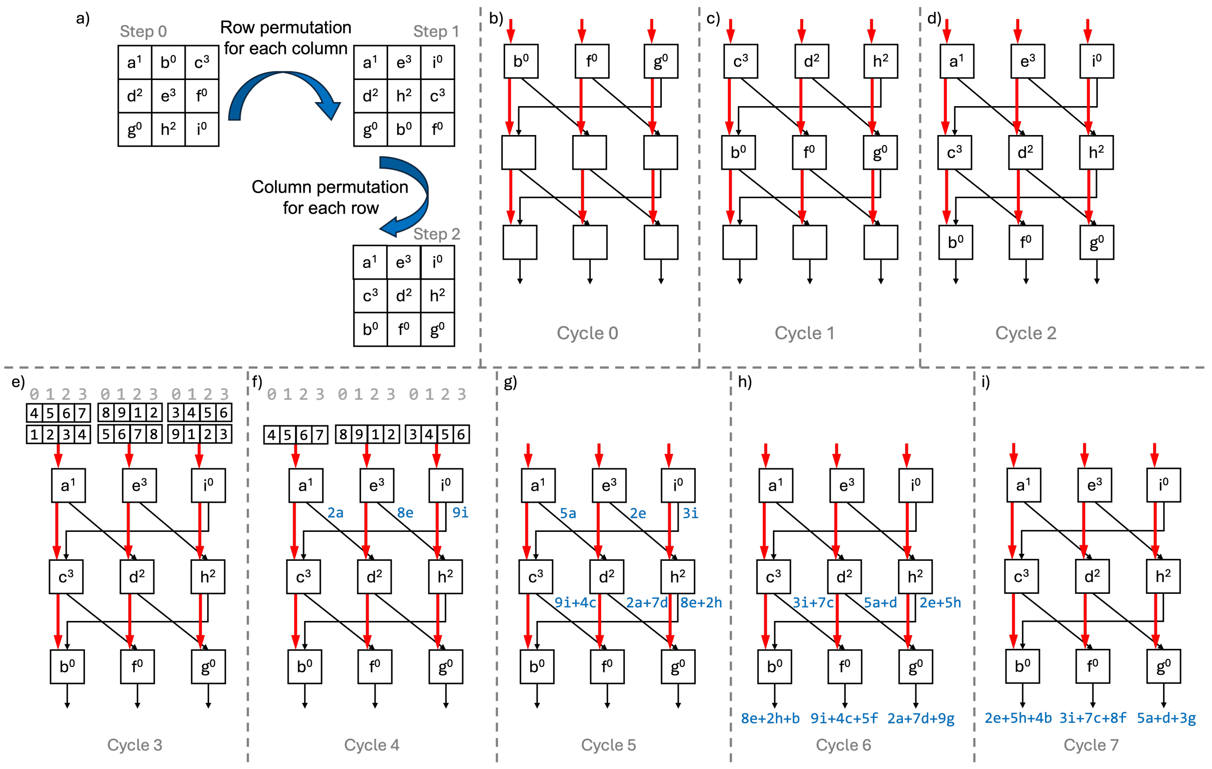


Figure 4.12: (a) The required permutation of the weight matrix. (b)-(i)Running example of the proposed DSSTA architecture.

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 & 8 & 9 & 1 & 2 & 3 & 4 & 5 & 6 \end{bmatrix}$$

$$W = \begin{bmatrix} a^1 & b^0 & c^3 \\ d^2 & e^3 & f^0 \\ g^0 & h^2 & i^0 \end{bmatrix}$$

Where $W$ is the condensed weight matrix with 1:4 block sparsity. The superscripts next to the weight value represents the index of the respective weight. And $A$ is the dense input matrix.

First, Figures 4.12(b)-(d), the weights along with their indexes are loaded to the SA from the northern edge, one row per cycle. After the weights are loaded, the input blocks start to flow through the SA, shown in red in Figure 4.12. In each PE, the stored index selects the appropriate input value to be multiplied with the weight. The computed product added with the partial sum coming daigonaly from the above PE row and the produced partial sum forwarded diagonaly to the appropriate PE of the next row, following the paths depicted in black in Figures 4.12(e)-(i). In detail, the process evolves as follows:

- **Cycle 0:** The last row of the weight matrix $(b, f, g)$ is loaded to the first PE row.
- **Cycle 1:** The last row of the weight matrix $(b, f, g)$ is shifted to the second PE row and a new row $(c, d, h)$ is loaded to the first PE row.
- **Cycle 2:** The last row of the weight matrix $(b, f, g)$ is shifted to the last PE row, the second weight row $(c, d, h)$ is shifted to the second PE row and a new weight row $(a, e, i)$ is loaded to the first PE

row. Thus the weight loading phase is completed.

- **Cycle 3:** Loaded first input matrix row. The stationary weights selects the appropriate input of each block to be multiplied.
- **Cycle 4:** Input blocks from the first PE row forwarded diagonally to the second PE row and a new input matrix row loaded to the first PE row. Partial sums $(2a, 8e, 9i)$ are permuted to $(9i, 2a, 8e)$ and forwarded to the second PE row.
- **Cycle 5:** Input blocks are forwarded from the second to the third PE row, from the first to the second PE row and a new input matrix row loaded to the first PE row. Partial sums $(9i+4c, 2a+7d, 8e+2h)$ are permuted to $(8e+2h, 9i+4c, 2a+7d)$ and forwarded to the third PE row. In the same manner, partial sums from the first PE row are permuted and forwarded to the second PE row.
- **Cycle 6:** There are no more input matrix rows to be fed to the SA, so the first PE row remains idle. The first row of the output matrix $(8e+2h+b, 9i+4c+5f, 2a+7d+9g)$ produced on the south edge of the SA in a permuted manner, i.e. the first column of the original output is last at the south edge of the SA. For the rest of the PE rows are permuted and forwarded to the next PE row.

The latency of the DSSTA architecture can be analyzed by considering a general systolic array of size $R \times C$, where $R$ represents the number of rows and $C$ the number of columns, processing an input matrix with $N$ rows. The computation unfolds in two distinct phases: weight loading and input processing. First, $R$ cycles are required to load the weights into the array. Once the weights are in place, the $N$ input rows are streamed into the array in a pipelined fashion, with each row requiring $R-1$ cycles to propagate through the diagonal data paths. As a result, the total latency is computed as follows:

$$L_{s^3t-DiP} = 2R + N - 1 \tag{4.2}$$

On the contrary, the latency of a classic STA follows the latency of a dense systolic array. As described in [43] this latency for the same grid size is:

$$L_{STA} = 2R + C + N - 2 \tag{4.3}$$

The Equations 4.2 and 4.3 clearly shows that the proposed architecture, not only reduces the hardware overhead by eliminating the need of skewing, but also reduces the latency of the computation. More specifically, DSSTA requires $C - 1$ less clock cycles per tile of the matrix multiplication operation, derived from the exemption of the input skewing.

### 4.2.2 Experimental Results

The experimental evaluation serves a twofold purpose. First, it aims to demonstrate the **reductions in area and power** achieved by the DSSTA compared to conventional sparse systolic tensor arrays. Second, it highlights the significant **minimization of wire-length** enabled by the DSSTA dataflow, showcasing its advantages in hardware efficiency and layout simplicity.

In this scope, we developed in SystemVerilog RTL parametrized models of three array types: (a) a conventional sparse systolic tensor array (STA); (b) a sparse systolic tensor array that follows exactly the dataflow described in [92] for dense SAs (DiP); and (c) the proposed DSSTA architecture. All three arrays

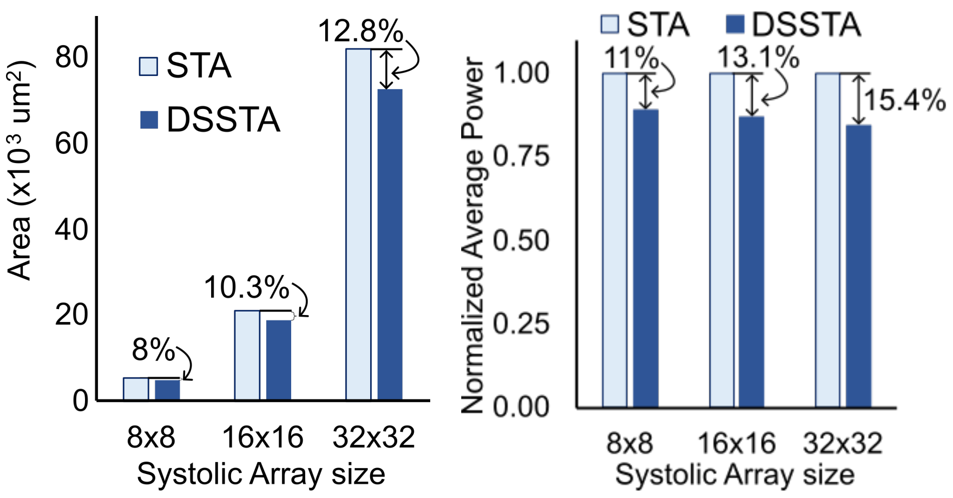


Figure 4.13: Area and Normalized Average Power consumption of the DSSTA against STA for array sizes $8 \times 8$, $16 \times 16$ and $32 \times 32$.

were synthesized down to the physical layout level using the Open-Road physical synthesis flow [93], utilizing the ASAP 7nm standard-cell library with a target clock frequency of 1GHz.

The evaluations done under two well-known Transformer workloads: Swin [94] and DeiT [95]. The data used are $8 - bit$ integer quantized and pruned to $2:4$ structured block as described in [34], while the accumulators are $16 - bit$ wide. All the evaluated designs could be reconfigured to operate on $1:4$ structured block sparsity, by simply disabling one of the multipliers.

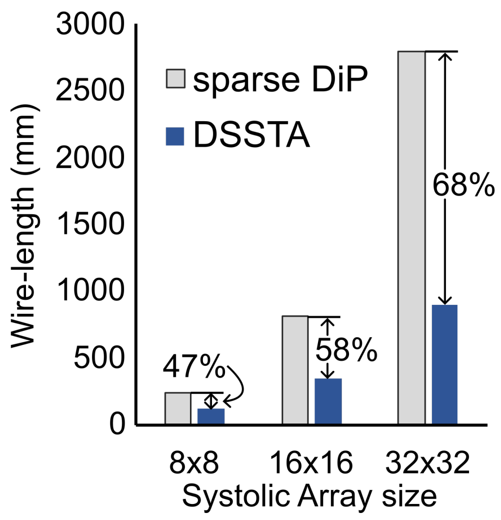
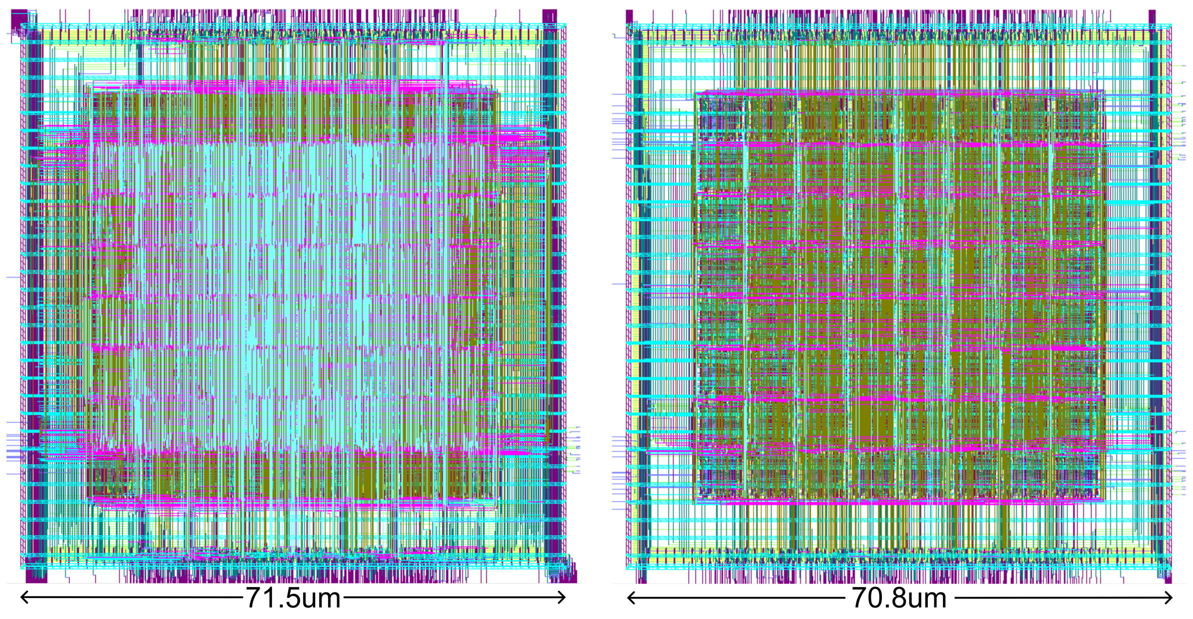


Figure 4.14: Total wire-length required for the DSSTA compared to using the DiP dataflow as described in [92].

Figure 4.13 depicts the area and the normalized average power consupmtion of the proposed DSSTA against a conventional STA architecture. DSSTA, by removing the registers required for input skewing, reduces the overall occupied area by 8% for a $8 \times 8$ array, by 10.3% for a $16 \times 16$ array and by 12.8% for a $32 \times 32$ array. This area reduction translates to savings on the average power consumption that ranges from 11% for the $8 \times 8$ array, to 15.4% for a $32 \times 32$ array.

The proposed weight permutation, used in the DSSTA architecture, allows us to place the TD-PEs in a



(a) DiP (b) DSSTA

Figure 4.15: Layout of an 8x8 sparse systolic tensor array architecture following (a) the dataflow exactly as described in [92] and (b) the proposed DSSTA dataflow.

rectangular grid where the input blocks flows vertically resulting in reduced total wire-length compared to following the dataflow of [92], where the input blocks would flow diagonally. This wire-length reduction is shown in Figure 4.14 and concludes to 47% for a $8 \times 8$ grid, 58% for a $16 \times 16$ grid and 68% for a $32 \times 32$ array. From Figures 4.15a and 4.15b, that show the layouts of the two evaluated designs for an $8 \times 8$ systolic array, it is clear that the proposed DSSTA architecture requires significantly less wiring.

## 4.3 Conclusions

This section has presented two architectural contributions aimed at advancing the efficiency of structured sparse matrix multiplication for modern machine learning accelerators. First, we introduced *DeMM*, a decoupled matrix multiplication engine that natively supports *relaxed* structured sparsity patterns while preserving the simplicity and regularity of systolic designs. By decoupling memory elements from MAC units, DeMM enables flexible support for fine-grained sparsity patterns with minimal reconfiguration. Experimental evaluation demonstrates that DeMM achieves substantial reductions in execution latency for CNN workloads compared to state-of-the-art structured-sparse architectures [36, 38, 85], while also delivering significant power savings.

We further proposed *DSSTA*, a structured sparse systolic tensor array that eliminates input skewing through a diagonal and permuted-weight dataflow. By removing shift registers and leveraging a carefully crafted weight permutation scheme, DSSTA simplifies the hardware design while maintaining synchronization across processing elements. Experimental results highlight its efficiency gains, including up to 12.8% reduction in area, 15.4% lower power consumption relative to conventional sparse systolic arrays, and a 68% reduction in interconnect wirelength compared to the dense-oriented DiP dataflow. Furthermore, the elimination of skew-induced delays translates into improved computational latency.

Taken together, DeMM and DSSTA demonstrate the potential of rethinking both the micro-architectural organization and the underlying dataflow of systolic arrays to better accommodate structured sparsity. While DeMM emphasizes flexibility and performance scalability through disaggregation, DSSTA highlights the importance of lightweight synchronization and interconnect efficiency. These two approaches collectively provide a foundation for next-generation accelerators that deliver higher performance and energy efficiency, paving the way for more sustainable deployment of deep learning workloads in both data centers and edge devices.

Chapter 5

# Online fault detection for Sparse Systolic Tensor Arrays

The use of structured sparsity introduces both opportunities and challenges for accelerator design. On the one hand, skipping operations corresponding to pruned weights reduces the computational burden and power dissipation, making sparse systolic tensor arrays attractive for mobile and embedded inference. On the other hand, structured sparsity complicates the dataflow within tensor processing elements, where multiplexers and local weight registers determine which input activations participate in each multiply-accumulate operation. While this design ensures efficient execution of sparse GEMM operations, it also introduces new reliability considerations, as the additional control logic and indexing mechanisms may be susceptible to hardware faults. With the increasing prevalence of ML accelerators in domains such as automotive [96], aviation [97], and healthcare [6], ensuring correctness under fault conditions is becoming a critical requirement.

Modern semiconductor technologies exacerbate this need due to susceptibility to random faults, including soft errors caused by radiation-induced events and hard errors due to device degradation and process variability. Such faults can silently corrupt intermediate values in neural network computations, leading to incorrect inferences that may compromise safety. For dense systolic arrays, various error detection and fault-tolerance mechanisms have been investigated, ranging from redundant execution to algorithm-based fault tolerance (ABFT) [98–100]. ABFT, in particular, offers a lightweight alternative by embedding checksums into matrix operations [101–104] and verifying results against predicted values. However, directly applying ABFT to sparse tensor arrays is non-trivial, since the structured sparsity alters the mapping of inputs to outputs and requires checksum logic that can adapt to the dynamic input selection within TPEs.

In this section we therefore focused firstly on extending ABFT to sparse systolic tensor arrays. The proposed approach introduces checksum accumulation logic along the periphery of the array, enabling online verification of results a few cycles after computation completes. By reusing the tensor array itself to compute predicted checksums, this method minimizes hardware overhead and avoids duplicating functionality. To handle the increased bitwidths of checksum accumulations, which can exceed those of the quantized inputs and weights typically used in inference, digit-serial arithmetic is employed. This technique serializes wide checksum values into narrower operands that can be processed by the existing hardware, thereby supporting large matrix sizes without modifying the core structure of the array.

Complementary to this, we also introduce a novel periodic online self-testing methodology to detect permanent faults in sparse systolic arrays prior to the commencement of the actual calculations by the running application. Hence, a possible fault is identified before any erroneous (thus, wasteful) work is performed. The proposed technique exploits the presence of already-stored data within the array to minimize the number of required test vectors

Together, these mechanisms establish a foundation for incorporating reliability into sparse tensor array design. They show that efficient error checking can be integrated into high-performance accelerators without sacrificing throughput or introducing significant hardware costs. Moreover, the compatibility of the proposed error checking schemes with multiple sparsity levels (e.g., 1:4 and 2:4) highlights their practical applicability across different pruned models. This is particularly relevant for mobile and edge deployments, where energy constraints must be balanced with fault tolerance.

## 5.1 Algorithm-Based Fault Tolerance on Sparse Systolic Tensor arrays

Checking the correctness of sparse matrix multiplications in the presence of random hardware faults is necessary for safety in critical applications [96]. Recent works [105–107] quantified the robustness and reliability of systolic accelerators when affected by random faults. Also, new fault-tolerant architectures were proposed that bypass or deactivate faulty PEs supported also by appropriate weight pruning and model compression [108–111]. In such cases, the fault detection was done with application-specific test vectors [112].

Managing random hardware faults [113, 114], requires special hardware modules for fault detection [115]. Faults should be detected *online* and possibly within a few cycles of their occurrence, thus simplifying recovery.

Algorithm-Based Fault Tolerance (ABFT) [101,102,116] offers a low-cost mechanism to detect errors in matrix-based computations [103, 104] by comparing the true output checksum with a predicted one. For dense SAs, specialized checksum circuits have been proposed [98–100] that compute checksums and perform error detection, while the systolic array processes input data.

### 5.1.1 Online Error Checking for Sparse Systolic Tensor Arrays

Following the ABFT methodology, to predict the checksum of the multiplication we have to compute the dot product of the per-column checksum vector of $A$ and the per-row checksum vector of $W$. This predicted checksum should be compared to the actual checksum of matrix multiplication.

The actual checksum is computed in the OC modules put on the south edge of the sparse tensor array, as shown in Figure 5.1. The OC modules add in a pipelined manner the results produced at each column of the sparse tensor array. The total sum is then accumulated at the actual-checksum accumulator shown in the south-east corner of Figure 5.1.

In the proposed mechanism, computing the predicted checksum utilizes both the sparse tensor array as well as extra checking logic. First, we need to compute the per-column checksum vector of $A$. Similar to [98, 100], this is computed online using the input accumulator blocks (IC) attached at the west side

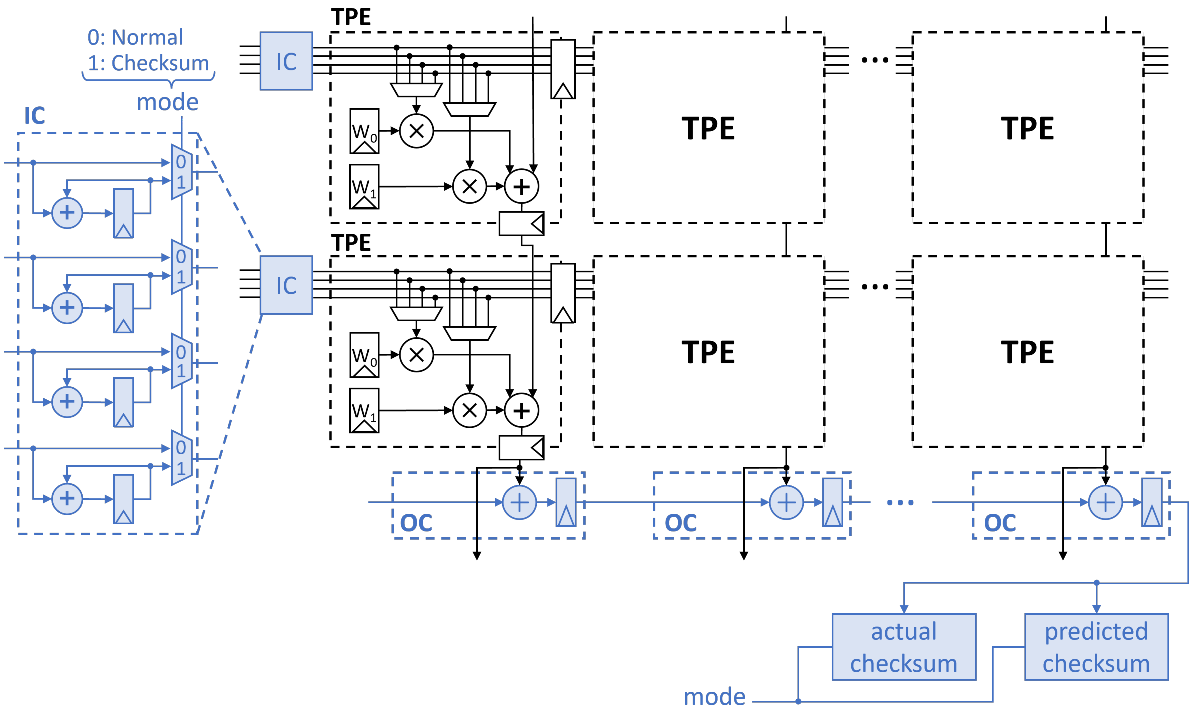


Figure 5.1: A two-row sparse systolic tensor array that can be configured for 2:4 and 1:4 sparsity patterns. ABFT error checking that computes the actual and the predicted checksum is placed at the periphery of the array.

of the sparse tensor array. Each IC block, shown in Figure 5.1, consists of four accumulators. This requirement stems from the fact that we do not know beforehand which inputs will be selected by the weights stored in a row of the TPE. Thus, we need to accumulate all incoming inputs of dense matrix $A$.

Then, we need to compute the dot product of the per-column checksum vector of $A$ and the per-row checksum vector of $W$. To do this, we re-use the TPEs of the sparse tensor array. For instance, after the last row of matrix $A$ enters the sparse tensor array, we gradually stop normal computation and push inside the sparse tensor array the column checksum vector $\sum_i A_{i,j}$ accumulated at the IC blocks using the same skewed data arrival pattern. Effectively, this action extends the computation with an “extra row” of checksum inputs. This column checksum vector of $A$ will propagate inside the sparse tensor array and be multiplied with the corresponding non-zero weights. At the bottom of the $k$th column of the sparse tensor array the sum $\sum_j(\sum_i A_{i,j})W_{j,k}$ will be produced. This sum is equivalent to $(\sum_i A_{i,j})(\sum_j W_{j,k})$ that was originally sought. In this way, computing separately the per-row checksum vector of sparse matrix $W$ (i.e., $\sum_j W_{j,k}$) is redundant and can be skipped. Also, the same non-zero elements of $W$ that affect the result of matrix multiplication contribute to the computed checksum.

Checksum prediction is completed at the south edge of the sparse tensor array. The checksums produced per column after normal computation has finished should be added to complete the predicted checksum. This addition is also done by the OC modules. However, in this case, since each sum computed by an OC module refers to the predicted checksum and not the actual one, it gets accumulated to a predicted-checksum accumulator at the south-east corner.

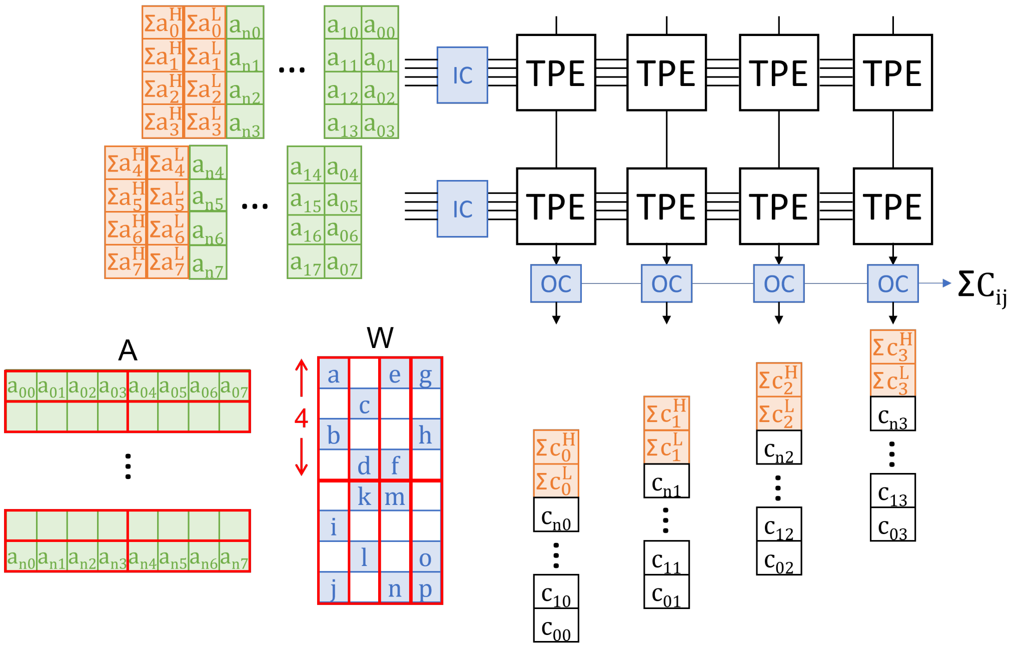


Figure 5.2: Checksum computation follows back-to-back normal computation. Checksums are computed in a digit-serial manner reusing the fixed structured of the sparse tensor array.

**Tackling the increased width of accumulators with digit-serial additions**

To save power and to reduce the memory footprint of ML models, model pruning and quantization is applied during training, while inference is executed using integer arithmetic. Without loss of generality, let's assume that the sparse tensor array operates on 8-bit quantized inputs and weights producing per column 24-bit results.

The proposed ABFT checking logic computes the predicted checksum by reusing the multiply-add logic of the sparse tensor array. In particular, in the last wave of computation, the sparse tensor array does not receive the data of input matrix $A$ but their per-column checksum. However, the bit width of the accumulators is much larger than the input bit width, e.g., accumulating 256 8-bit inputs we would need 16-bit accumulators. Therefore, passing this 16-bit accumulated value through the 8-bit inputs of the sparse tensor array cannot be done in the same cycle.

To address this issue, we perform the *checksum operation* for the first time, to the best of our knowledge, in a *digit-serial manner*. The input checksum is broken into smaller 8-bit wide digits and are fed into the SA one digit per cycle. The 8-bit checksum digits are propagated inside the sparse tensor array and produce new checksum values at the bottom of each column of the array. The derived digit-wise checksums are accumulated using the OC modules and the predicted-checksum accumulator. This accumulator handles also the needed shifting of the arriving data and their sign extension to perform the signed digit-serial addition properly. The back-to-back evolution of normal and checksum computation, where checksum is computed in a digit-serial manner is highlighted in Figure 5.2.

In the general case, assuming that the input accumulators are by construction $M$-bits wide and that the sparse tensor array operates on $N$-bit quantized integers, the maximum number of rows of $A$ that can be safely accumulated is $t = 2^M/2^N$. Thus, every $t$ cycles, we need to interrupt normal operation and compute the respective checksum. Digit-serial checksum computation involves $M/N$ digits that are passed serially one after the other to the sparse tensor array. In this way, we can support large input matrices and large per-column checksum values without altering the design of the sparse tensor array. Digit-serial checksum computation can be equally applied to dense systolic arrays.

### 5.1.2 Experimental results

In the experimental results, we aim to highlight two aspects of the proposed architecture. In the first set of experiments, we measure the hardware overhead induced by the error checking logic in terms of area and power to the sparse systolic tensor array. In the second set of experiments, we explore the fault detection properties of the checker, and the impact of sparsity to its performance.

#### Hardware complexity evaluation

The examined sparse tensor arrays and their associated error checking logic have been been synthesized to Verilog RTL using Catapult HLS. Clock frequency target was set to 500 MHz. The reported timing/area results were derived from the Oasys-RTL logic synthesis tool driven by a 45-nm standard-cell library. Power estimation was done using PowerPro tool by running inference in state-of-the-art CNN applications, such as Resnet50 [2], with inputs from ImageNet. The weights used in the experiments assume structured sparsity of 1:4 and 2:4 derived with Tensorflow for ResNet50 [2].

Each sparse tensor array follows the organization shown in Figure 5.1. It consists of $R$ rows and $C$ columns of TPEs and can be configured for 2:4 or 1:4 structured sparsity. Effectively the sparse tensor array receives $4R$ inputs at each west edge TPE. Inputs and weights are 8-bit quantized integers. The necessary data width of the output of each column of the sparse tensor arrays depends on the number of rows of the array. To have a uniform design for all examined cases, we assume that the output bitwidth equals 24 bits.

The checker consists of $R$ IC modules in the west edge of the sparse tensor array, $C$ OC modules on the south edge and two accumulators for the actual and the predicted checksum on the south-east edge. The input accumulators in IC are 16-bits wide, the adders and pipeline registers in the OC modules are 24 bits, while the actual and the predicted checksum accumulators are 48-bits wide.

The hardware complexity of the examined designs is summarized in Figure 5.3. Figure 5.3(a) presents the area overhead that is introduced by the checker for different sparse tensor array sizes. For an $8 \times 32$ sparse tensor array, the area overhead is small, around 5%. This overhead gets further reduced to 1.9% and 1.1% as we increase the size of the sparse tensor array to $16 \times 64$ and $32 \times 128$, respectively.

Figure 5.3(b) reports the average power per layer of Resnet50 using inputs from ImageNet and weights that have a structured sparsity of 2:4 and 1:4. In the case of 2:4 sparsity, the power overhead of the checker is 7.5% on average for the three examined sparse tensor array sizes. By changing the structured sparsity of the weights to 1:4, the average power consumption is reduced. This is the result of the increased number of non-zero weights that leads to a higher inactivity of the sparse tensor array. On the other hand, since the number of IC and OC modules of the checker are not affected by the sparsity pattern, and is only determined by the size of the sparse tensor array, their power consumption remains almost constant. In this case, the power overhead ranges from 7.4% to 9% for the different sparse tensor array sizes.

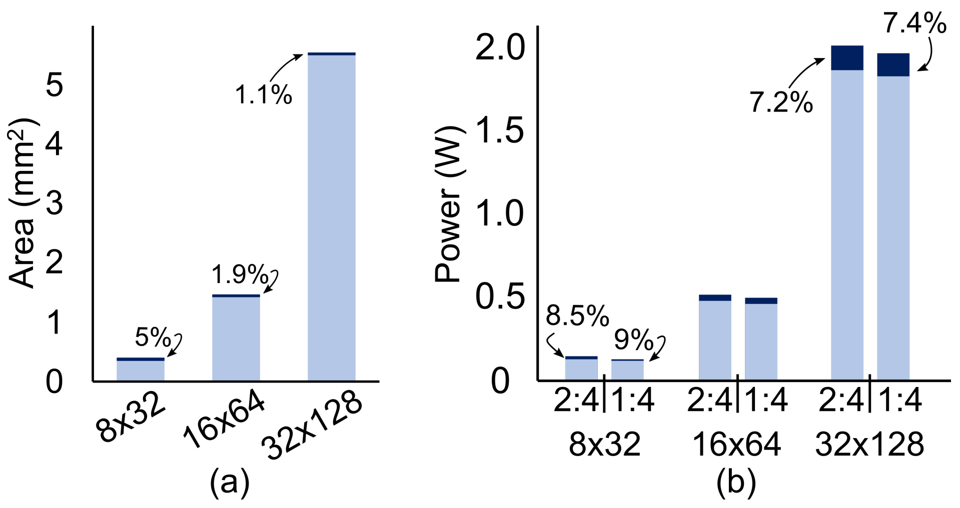


Figure 5.3: The total (a) area and (b) power of sparse tensor arrays for various sizes and for 2:4 and 1:4 sparsity patterns. Both diagrams highlight separately the contribution of the checker's logic.

### Evaluation of fault detection properties

For evaluating the quality of fault-detection of the proposed design we inject bit-flips in random clock cycles. Faults are injected to randomly selected storage elements covering both the registers of the sparse tensor array and the registers of the checking logic. Whether a fault will be injected on the sparse tensor array or the checker depends on the amount of their storage elements. A fault is more probable to hit the sparse tensor array than the checker's logic since it includes significantly more storage elements including the locally-stored weights as well as the pipeline registers placed at the borders of each TPE. The weights used during each fault-injection campaign correspond to *all* pruned CNN layers of Resnet50 used also for power estimation.

At the end of each matrix multiplication that corresponds to a CNN layer, we record the outcome of the fault injection campaign. The observed behavior may fall into one of four categories:

1. Detected: The checker detected an actual fault that occurred in the sparse tensor array.
2. Silent: The checker did not detect an actual fault that occurred in the sparse tensor array. The fault was masked at the checksum level although the checker remained fault free.
3. False positive: The checker flagged a fault detection, but no fault occurred in the sparse tensor array.
4. False negative: A fault occurred in the sparse tensor array and the checker did not detect it. Equivalently, the fault remained silent due an error inside the checker.

Table 5.1 summarizes the outcome of 15000 independent fault-injection campaigns done on all pruned CNN Layers of Resnet50 (i.e., 300 independent fault-injection campaigns per CNN layer). Each fault injection campaign injects a different number of faults to a $8\times32$ sparse tensor array and its checker. The same experiment is performed for 1:4 and 2:4 sparsity patterns.

In the case of a single fault injected, the majority of the faults belong to the detected and silent categories. Silent faults can occur in many scenarios. For sparse tensor arrays the most common case refers to faults affecting pipeline registers in the horizontal dataflow that refer to inputs that are not

Table 5.1: Fault Detection of the proposed error checking logic on a 8×32 sparse tensor array that can be configured for 2:4 and 1:4 sparsity patterns for Resnet50 CNN layers.

| # of faults | 1 fault | | 1–5 faults | |
|---|---|---|---|---|
| Block Sparsity | 2:4 | 1:4 | 2:4 | 1:4 |
| Detected | 81.11% | 74.15% | 95.61% | 93.34% |
| Silent | 12.46% | 20.75% | 2.15% | 4.95% |
| False Positive | 2.36% | 2.58% | 0.58% | 0.69% |
| False Negative | 4.07% | 2.51% | 1.65% | 1.02% |

actually selected by the corresponding weights in the TPE. Hence, it is not possible such errors to affect the final result. Silent faults increase in the case of 1:4 sparsity compared to the case of 2:4 sparsity. This is the result of faults occurring in the weight registers that remain idle in the case of 1:4 sparsity. False checker reactions cover a small overall percentage that reflects that the probability that a fault hits the checker and not the sparse tensor array is small.

As the number of injected faults per fault-injection campaign increases (1–5 faults are randomly injected) the observed results change significantly. Fault detection increases above 93% and the possibility of having a false alarm drops to 1% on average.

## 5.2 Periodic Online Testing for Sparse Systolic Tensor Arrays

This section introduces a novel periodic online self-testing methodology to detect permanent faults in sparse systolic arrays *prior* to the commencement of the actual calculations by the running application. Hence, a possible fault is identified *before* any erroneous (thus, wasteful) work is performed. The proposed technique exploits the presence of already-stored data within the array to *minimize* the number of required test vectors. Faults that occur during the computation can also be detected by Algorithm-Based Fault Tolerance (ABFT) techniques [101–104, 117], which *concurrently* perform checks alongside the application computations. Under ABFT, faults are detected *after* a computation is completed.

The proposed technique is inspired by the work in [112], which explores online self-testing in *dense* systolic array architectures and reuses the stationary weights already loaded into the systolic array for testing purposes. In a similar vein, the technique proposed here also reuses the existing weight values, but it targets *sparse* systolic arrays with their distinctive Tensor Processing Elements (TPE) and attributes that necessitate a more complex testing procedure, as compared to [112].

The contribution of the proposed testing mechanism can be summarized as follows:

- By utilizing the weight values of the running application during the testing phase, the presented technique requires merely ***four* test vectors** to provide very high coverage against permanent faults, which minimizes the time lost for on-line testing before the initiation of computation.
- The proposed approach reuses the systolic array's existing storage elements to minimize the incurred hardware overhead, as compared to other self-testing techniques that use scan chains [111].
- To assess the achieved fault coverage of the proposed technique, we employ random fault injections at the gate level, while executing three established Convolutional Neural Networks (CNN) applications. The obtained results demonstrate very high achieved coverage across all three benchmarks,

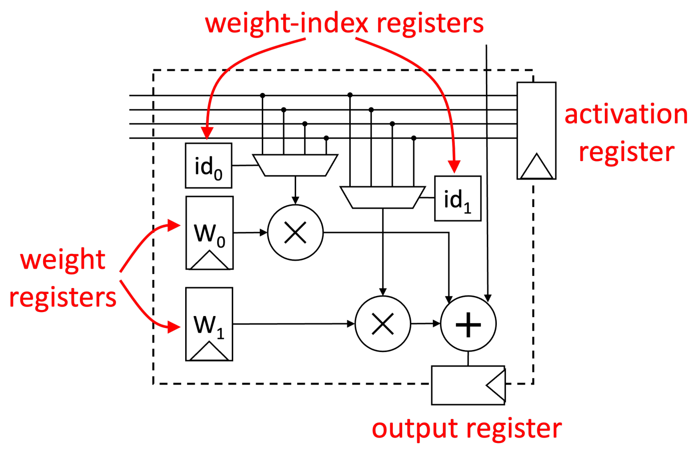


Figure 5.4: The four types of registers in a single TPE of a sparse systolic array.

thereby validating the effectiveness of the new lightweight online checking mechanism. Additionally, the incurred overheads to the system performance and the hardware area are shown to be minimal.

### 5.2.1 Fault Detection in Sparse Tensor Arrays

The first step toward a fault-tolerant sparse tensor array is a fast and light-weight (in terms of both hardware overhead and the impact on application performance) fault-*detection* mechanism that can trigger an appropriate reaction.

We hereby propose an online test technique that periodically checks – every time a new group of weights is loaded – the systolic array TPEs for permanent errors. For simplicity, let us assume that a permanent fault can only occur within the storage elements, i.e., the *registers*, of each TPE, as shown in Figure 5.4. Hence, a permanent fault may afflict one of the following registers:

1. The so called *activation* registers, which store the incoming blocks of consecutive input elements and propagate them along the horizontal (west-to-east) direction in each row of the array.
2. The $N$ *weight* registers that store the stationary weights.
3. The *weight-index* registers that control the multiplexers and select the up to $N$ appropriate input (activation) elements out of the $M$-element input block.
4. The *output* registers that store the accumulated sums and propagate them downwards along the vertical direction (north-to-south) in each column of the array.

Note that the simplifying assumption that permanent faults may only occur in the above-mentioned registers does *not* ignore faults manifesting in the remaining logic of a PE. As can be seen in Figure 5.4, the micro-architecture of the TPE indicates that such faults will yield an erroneous result in either the multiplication and/or the accumulation steps, which will be captured as errors in the output register of each TPE. All paths of the internal TPE logic are “funneled” into the output register at the bottom right of each TPE. Hence, error coverage of the output register will also detect faults occurring within the remaining TPE logic.

Table 5.2: The 3 unique test vectors employed by the proposed online checking methodology

| Test number | Test vector | Top-row TPE sum input |
|---|---|---|
| 1 | $[1, 1, 1, 1]$ | $0$ |
| 2 | $[-1, -1, -1, -1]$ | $-1$ |
| 3,4 | $[1, 2, 3, 4]$ | $0$ |

The proposed online checking mechanism detects and locates permanent (stuck-at) faults through a simple self-testing sequence. The sequence consists of four different tests, executed consecutively, with each test requiring a single test vector. Said test vectors are carefully crafted to reveal permanent faults occurring in any register within any TPE of the systolic array. The four test vectors use the weight values that are already stored (by the running application) in the array, thereby immensely reducing the test-latency overhead. The weights are not affected in any way and there is never a need to reload any data after a test session concludes. The application simply resumes normal operation.

The testing sequence is initiated after the weight-loading phase of each tile[1] is completed. At this point, the pre-defined test vectors are fed as inputs to the systolic array and propagate through the array; i.e., they are multiplied-and-accumulated with the local weight values in each TPE along the way, until the final outputs are produced at the bottom of each column of the array (south edge).

A single testing session is depicted in Figure 5.5 for the case that the inputs to each TPE are fed as groups of $M = 4$ elements. Each test vector is fed into every row of TPEs in the array. For each test vector, there is an accompanying value that must be fed into the *sum* (adder) input of the top-row TPEs of the array. The test vectors and the corresponding values fed into the top-row TPE adders are shown in Table 5.2. The last two tests use the same test vector.

When fed into the array, the test vectors trigger the calculation of a “weighted” sum of all the weights already stored in the TPEs across each column of the array. This weighted sum is output at the bottom of each column and is then compared with the corresponding golden reference. This ‘comparison’ is facilitated by the existing accumulators at the south edge of the SA, highlighted in red in Figure 5.5. In other words, the comparison is simply an addition operation.

Due to the use of the existing, specific weight values, it is not possible to ensure fault coverage for all possible paths in the design, i.e., achieve 100% fault coverage. Nevertheless, if the test response is error free, there is very high certainty that, *for the currently-loaded weight tile*, there will be no error affecting the output of the array. In other words, functional correctness is ensured with very high certainty *for this particular weight tile*. The test must then be repeated when the next tile is loaded, and so forth.

Also, the localization granularity is limited to a single column of the systolic array. In other words, the proposed mechanism can locate the faulty column of the array, but it cannot identify the specific TPE where the fault is located.

[1]Since the input and weight matrices of ML applications are typically larger than the size of the systolic array, the multiplication is performed gradually and progressively by loading *tiles* of these matrices, whereby a tile corresponds to the size of the systolic array accelerator.

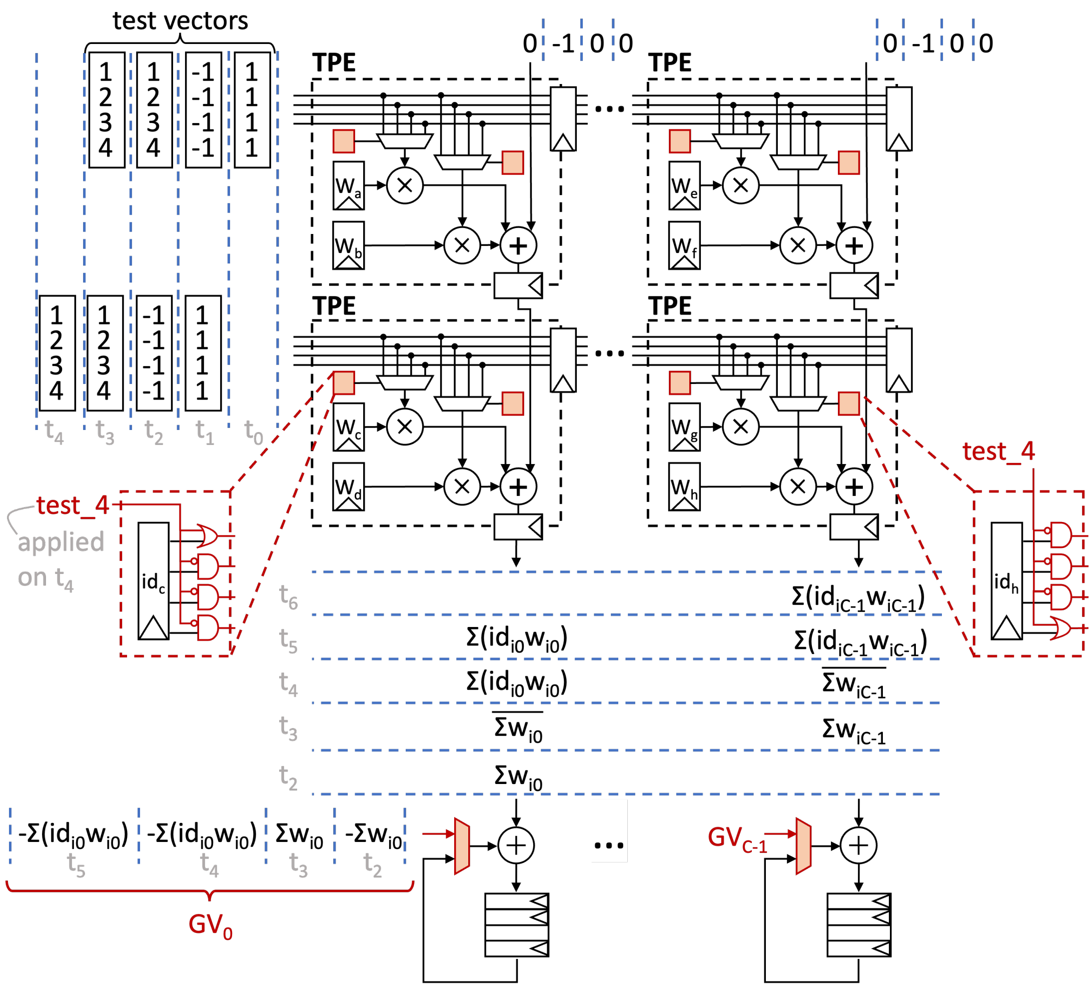

test vectors
TPE
0 -1 0 0
test_4
applied on $t_4$
$t_4$ $t_3$ $t_2$ $t_1$ $t_0$
$t_6$ $t_5$ $t_4$ $t_3$ $t_2$
$\Sigma(id_{i0}w_{i0})$
$\Sigma(id_{iC-1}w_{iC-1})$
$\overline{\Sigma w_{i0}}$
$\Sigma w_{i0}$
$\overline{\Sigma w_{iC-1}}$
$\Sigma w_{iC-1}$
$-\Sigma(id_{i0}w_{i0})$ $-\Sigma(id_{i0}w_{i0})$ $\Sigma w_{i0}$ $-\Sigma w_{i0}$
$GV_0$
$GV_{C-1}$

Figure 5.5: The four test vectors are applied periodically to the sparse systolic tensor array to detect permanent faults within any one register.

### Testing for faults in the weight and output registers

The first test vector computes the sum $V$ of the weight values across all TPEs in each column $j$, i.e., $V_j = \sum_i w_{ij}$. Index $i$ refers to all the weight registers across column $j$. For this first test, the golden reference value equals

$$GV_j = -\sum_i w_{ij}$$

The minus sign is included so that the *addition* of the computed test sum $V_j$ with the golden reference value $GV_j$ should result in a zero value in the absence of a fault within each column $j$.

The second test vector, used in the second test of each test session, computes the *bit-wise complement* of the sum of the weight values across all TPEs in each column $j$, i.e., $\overline{V_j}$. This is because the second test subtracts one (the input value fed into the top-row TPE) from the negative sum (due to the $-1$s in the test vector) of the weights of each column. By the 2's complement definition, the subtraction of 1 from a negative signed operand effectively computes its bit-wise complement:

$$-V_j = \overline{V_j} + 1 \Rightarrow -V_j - 1 = \overline{V_j}$$

Thus, when the bit-wise complement of the sum of weights, $\overline{V_j}$ is added to a golden reference value of

$$GV_j = \sum_i w_{ij} \quad (= \text{error-free } V_j)$$

the result should be $-1$ (i.e., a bit string of all 1s in the output value, since we use 2's complement arithmetic) in the absence of a fault, since $\overline{V_j} + V_j = -1$.

In summary, under fault-free operation, the results of the first test vector at the south edge of the array should be 0 for all the columns, while the results of the second test vector should be $-1$ (all 1s in the output values).

If there is any discrepancy in these first two tests, further checking is needed to locate the fault. As described in [112] for dense systolic arrays, this fault localization is achieved by examining the outputs of Test 1 and Test 2 *prior* and *after* the comparison with the golden reference values, and checking whether they are bit-wise complementary, or not. Based on these bit-wise complementary checks, the fault can be localized using the cases depicted in Table 5.3. Note that the localization identifies the register type that is faulty within a particular column of the array, but it *cannot identify the specific TPE* in the column where the faulty register is located.

Table 5.3: Localization of faults by checking if the systolic array outputs and the comparison (with the golden reference) outputs of Tests 1 and 2 are bit-wise complementary

| | | | |
|---|---|---|---|
| **Test 1 and 2 outputs before comparison with golden reference** | Complementary | Not Complementary | Complementary |
| **Test 1 and 2 results after comparison with golden reference** | Complementary | Not Complementary | Not Complementary |
| **Fault Location** | Weight register | Output Register | Comparison Adder |

**Testing for faults in the weight-index registers**

The third test vector, as used in the third test of each test session, targets potential faults in the weight-index registers of each TPE in the array. The vector computes the "weighted" sum of the weight values, multiplied by their index position in the $M$-element block, as follows:

$$V_j = \sum_i (idx_{ij} \times w_{ij})$$

Similar to the first test, the golden value reference is the negative of this sum, i.e., $GV_j = -V_j$ Consequently, if a fault occurs in any of the weight-index registers within a column of the array, a wrong element of the input test vector $[1, 2, 3, 4]$ (assuming here that $M$=4) will be selected to be multiplied with a stationary weight, thereby resulting in an erroneous sum at the bottom of the column.

**Testing for faults in the input activation registers**

The first three test vectors target faults that affect the *vertical* flow in the systolic array; i.e., faults that only affect the end result at the bottom of a column. The fourth test vector targets faults within the input activation registers of the *horizontal* flow within the array (see Figure 5.4).

This test is more complex in sparse arrays than in dense arrays, since an erroneous activation value, due to a fault within an activation register, may not be selected for calculation until several columns later. Thus, we introduce another, more sophisticated test, as compared to [112], where a fault in any activation register will be 'coerced' to manifest itself as multiple errors in consecutive columns.

This is achieved by utilizing a new control signal (labeled 'test_4' and shown in red in Figure 5.5) and masking gates at the outputs of the weight-index registers (also shown in red in Figure 5.5). When the new control signal is de-asserted, the masking gates are transparent, i.e., they let the output of the weight-index register pass through. The control signal is asserted only in the clock cycle that Test 4 is performed and it 'activates' the masking gates. These force the multiplexers to select one specific value from the $M$-element input block. Specifically, in the first column, the first input element is selected (by the OR masking gate in the top-most position); in the second column, the second input element is selected (the OR gate is one position below), and so forth.

This will force the erroneous value to manifest at the output (south edge) of a column *periodically*, every $M$ columns. That is, errors will be observed in *multiple* columns, all spaced $M$ columns apart. Such behavior indicates that the fault occurred in an activation register in one of the $M$ columns located just before (to the left of) the first error appearance.

### 5.2.2 Experimental Results

The experimental evaluation aims to quantitatively assess: (a) the fault-detection capability of the proposed online checking mechanism; (b) the impact on system performance in a fault-free environment; and (c) the hardware area cost.

An $8\times8$ sparse systolic tensor array was fully implemented in SystemVerilog RTL and augmented with the proposed checking mechanism. This systolic array operates on 16-bit integer quantized inputs and weights executing single-batch inference of CNNs that require matrix multiplications of different sizes. To retain accuracy, the additions in each column of the array are performed at a 32-bit width. The array was synthesized with Cadence's digital implementation flow using a 28 nm standard-cell library. It operates at a clock frequency of 1 GHz.

Three well-known CNNs were used for all the experiments: ResNet50 [2], DenseNet121 [68] and VGG16 [118].

Since the focus of this evaluation is on fault-detection, the experiments were conducted at the synthesized gate-level netlist. To calculate the achieved fault coverage, the Hope sequential fault-simulator [119] was employed, which performs exhaustive fault simulation at the gate-level of the entire systolic array. Specifically, Hope performs a single-fault injection campaign for each point in the given netlist and checks if the application of the applied test-vectors (in our case: the 4 test vectors and all the weight values in the CNN application) produce an output different from the fault-free output.

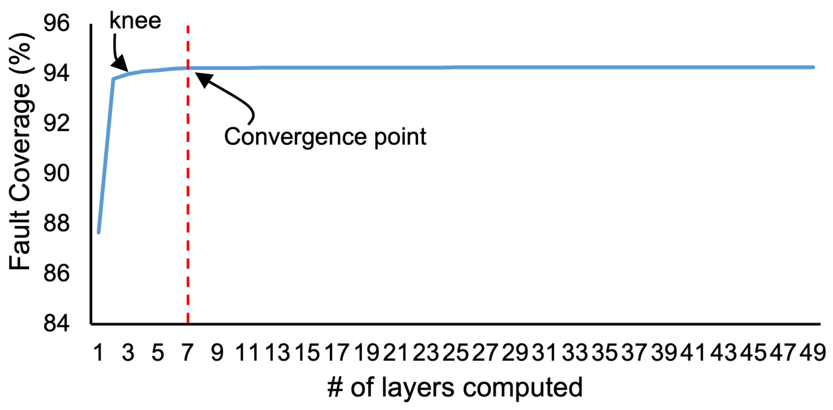


Figure 5.6: The fault coverage achieved after completion of each layer of ResNet50 [2]. The fault coverage converges quite rapidly to a high value.

Table 5.4: Fault coverage achieved by the proposed online testing mechanism for three well-known CNN applications and the layer of convergence for each application.

| **App** | **Total # of layers** | **Fault Coverage** | **Convergence layer** |
|---|---|---|---|
| ResNet50 | 49 | 94.2% | 7 |
| DenseNet121 | 120 | 94.3% | 2 |
| VGG16 | 13 | 94.1% | 5 |

When a new tile of weights is loaded, four distinct tests are performed in consecutive clock cycles using the four proposed test vectors applied as inputs to the array. Consequently, the achieved fault coverage increases progressively as more and more weight tiles are loaded during the execution of the CNN application. Figure 5.6 depicts this increase in the achieved fault coverage as all the layers of ResNet50 [2] pass through the systolic array and a testing session is performed on each new weight tile. After the first CNN layer is completed, the fault coverage is quite low, at 88.7%. However, the fault coverage increases rapidly within the first three layers. At the end of the third layer, there is a 'knee', beyond which the increase in the fault coverage is minimal. A convergence point is reached after completion of the seventh layer, after which the changes in fault coverage are negligible.

Table 5.4 summarizes the achieved fault coverages and convergence points for all three examined CNN applications. As shown, all three CNN applications converge quite rapidly (after only a few layers) to an average fault coverage of 94.2%.

Obviously, since the proposed mechanism uses only 4 test vectors and the existing weights in each loaded weight tile, the achieved gate-level stuck-at fault coverage cannot reach 100%; the weights are not sufficient to exercise all stuck-at faults. Our observations indicate that the vast majority of uncovered faults are paths located inside the multipliers of each TPE, which are not activated due to the constant weight values. However, as previously mentioned, even if the *gate-level* fault coverage is not closer to 100%, an error-free test response indicates with very high certainty *functional* correctness *for the currently-loaded weight tile*; i.e., there will be no error affecting the output of the array during the execution of this particular weight tile.

To evaluate the latency overhead introduced by the proposed periodic online testing mechanism, the

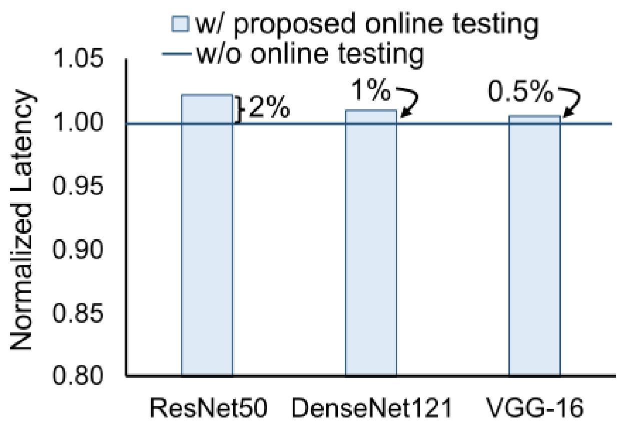


Figure 5.7: The impact on application latency of the proposed periodic online testing methodology. The results are normalized to the latency of the sparse systolic array with no error-checking mechanism.

total runtime of each CNN application *in the presence of online testing* was measured. An online test session was triggered for each newly-loaded weight tile throughout the entire execution of the application. This measured runtime was compared to the runtime achieved *without any online testing*, i.e., the execution latency achieved on a baseline sparse systolic tensor array. The results for all three CNN applications are shown in Figure 5.7. The proposed checking mechanism adds only marginal latency overhead of $0.5\%$ – $2\%$ to the applications' total runtimes. This is due to the use of only 4 test vectors, which merely add a 4-cycle latency overhead for every executed weight tile.

The hardware area overhead to support the proposed mechanism is also minimal. As shown in red in Figure 4.2, the extra hardware added comprises the single 'test_4' control signal, three AND gates and one OR gate per weight-index register in each TPE, and a multiplexer at the input of the bottom accumulators of each column, at the south edge of the SA. The total additional hardware accounts for 3% of the total area of the sparse systolic tensor array.

## 5.3 Related Work

There are numerous efforts aimed at protecting systolic array architectures from faults, which can be categorized into three main approaches: (a) detection-only methods that identify faults and discard faulty results; (b) detection and localization methods that isolate erroneous hardware modules after detecting faults; and (c) detection, localization, and correction methods that also attempt to correct faults for faster system recovery. These fault-tolerant methodologies are applied to dense systolic arrays and vary in complexity and effectiveness.

For fault detection, the work in [120] uses connections between weight and activation registers to form new scan-chains, reducing hardware complexity and power consumption. Another method [98] employs ABFT to detect faults caused by voltage reductions in low-power systolic arrays, while [100] uses extra accumulators to implement ABFT directly on Intel's Tiled Matrix Multiplication (TMUL) units. These methods primarily focus on identifying faults without correcting them, thereby preventing faulty results from impacting system operations.

More advanced methodologies not only detect, but also localize and correct faults. Techniques like those in [121] and [122] combine fault-aware pruning and retraining to minimize accuracy degradation and to bypass critical faults in Tensor Processing Units (TPUs). Additionally, the work in [123] tests near-threshold systolic array architectures without extra hardware, and RunSafer [112] uses specific test vectors to protect against permanent faults with minimal latency overhead. Correction methods like STRAIT [111] use self-test and recovery architectures to address faults, employing weight pruning and row/column swapping to maintain accuracy. Another approach, in [124], focuses on error detection and correction in Transformer networks, mitigating out-of-range neuron outputs through saturation or zeroing, thus ensuring minimal accuracy degradation even under high error rates. Finally, the work in [125] introduces fault-injection frameworks to analyze the impact of various factors on fault propagation in systolic arrays, further enhancing fault-tolerance strategies.

## 5.4 Conclusion

This section addresses the growing demand for fault-tolerant machine learning accelerators, particularly in safety-critical domains such as autonomous driving, medicine, and aviation. To reduce inference time and memory storage, modern ML models are often pruned and quantized, resulting in structured-sparse weight matrices executed on sparse systolic tensor arrays. Ensuring the reliable operation of such accelerators requires methodologies that can detect both transient and permanent faults with minimal overhead. To this end, we have first adapted the widely used Algorithm-Based Fault Tolerance (ABFT) methodology to the unique characteristics of sparse tensor arrays, enabling online error checking during matrix multiplications. By integrating checksum logic along the periphery of the array and reusing its existing computational structure under a digit-serial arithmetic paradigm, the design efficiently computes both actual and predicted checksums. Hardware analysis confirms the low-cost nature of the approach, while fault injection studies demonstrate strong fault-detection capabilities.

Complementing this, we introduced a periodic online self-testing methodology to target permanent faults before application execution begins. Unlike ABFT, which detects errors during execution, this technique identifies faults proactively, thereby preventing wasted computation. Each test session requires only four test vectors, the already stored model weights, and simple comparisons with precomputed reference values. The method not only detects faults but also localizes them at column-level granularity. Experimental evaluation shows high stuck-at fault coverage with minimal impact on performance and negligible hardware overhead. Looking ahead, we plan to extend the current approach by incorporating a small set of strategically selected random vectors into the test regime. These additional vectors will exercise previously unreachable datapaths, further improving overall fault coverage without significantly increasing testing overhead. Taken together, these contributions provide a foundation for robust, low-cost error detection in sparse systolic tensor arrays, advancing their deployment in mission-critical ML applications.

# Chapter 6

# Hardware Accelerators for Graph Convolution Networks

While CNN accelerators have traditionally focused on exploiting the regularity of dense tensor computations, the emergence of Graph Convolutional Networks (GCNs) introduces a fundamentally different set of challenges. Unlike CNNs, where computation is dominated by structured and highly parallelizable convolutional kernels, GCNs operate on graph-structured data characterized by sparsity, irregular memory access, and heterogeneous workloads across nodes and edges. These properties limit the efficiency of conventional systolic or SIMD-style accelerators, motivating the design of specialized architectures that can accommodate sparse matrix–vector multiplications, irregular aggregation patterns, and dynamic data dependencies. As a result, GCN accelerators must carefully balance programmability, dataflow flexibility, and energy efficiency in order to achieve high utilization while preserving scalability to large and diverse graph structures.

In a GCN, each node is associated with a feature vector, and layer-wise updates are carried out by aggregating features from neighboring nodes and combining them with learnable weights. This operation can be expressed as a three-matrix product:

$$H^k = \phi\left(SH^{k-1}W^k\right), \tag{6.1}$$

where $S = D^{-1/2}\tilde{A}D^{-1/2}$ is the normalized adjacency matrix, $\tilde{A} = A + I$ is the adjacency matrix with self-loops, $H^{k-1}$ represents the features from the previous layer, $W^k$ is the trainable weight matrix of the current layer, and $\phi$ is a nonlinear activation function such as ReLU. The first multiplication, $SH^{k-1}$, performs the *aggregation* step, while the second multiplication with $W^k$ carries out the *combination* step. Together, these operations capture both local graph structure and trainable transformations of features.

The computational demands of GCNs have driven a growing interest in hardware acceleration. State-of-the-art GCN accelerators typically decompose the computation of Eq. (6.1) into two phases: a sparse $\times$dense multiplication for aggregation and a dense$\times$dense multiplication for combination [126–128]. These phases exhibit distinct dataflows and sparsity patterns, motivating diverse accelerator designs optimized for either aggregation or combination. However, separating the two steps incurs overheads in data movement and control, limiting overall efficiency. To address this, we propose *FusedGCN*, a novel systolic architecture that fuses the aggregation and combination steps into a single three-matrix product computation. By unrolling or partially unrolling the operation in a tiled fashion, FusedGCN adapts to varying input/output feature sizes and bandwidth constraints. This design enables significant reductions in execution time compared to the best-performing state-of-the-art accelerators [128], while incurring only modest hardware overheads of less than 4% in area and around 8% in power.

Beyond performance, the reliability of GCN accelerators has become a pressing concern. As ML systems are increasingly deployed in safety-critical domains such as autonomous vehicles [96], medicine [6], and aviation [97], ensuring correctness in the presence of hardware faults is crucial [129]. Modern devices are susceptible to both transient soft errors and permanent faults, which can silently corrupt computations. Traditional Algorithm-Based Fault Tolerance (ABFT) [101, 102] provides a lightweight method for error detection in matrix operations by comparing predicted and actual checksums. However, directly applying ABFT to GCNs is inefficient, as it requires separate verification of the aggregation and combination phases, thereby duplicating effort.

To overcome this, we introduce *GCN-ABFT*, a custom-designed ABFT scheme tailored for graph convolution. Instead of validating the two multiplications independently, GCN-ABFT computes a fused checksum that directly corresponds to the entire three-matrix product of Eq. (6.1). This significantly reduces the number of operations required for error checking—by an average of 21% for representative GCN workloads—compared to baseline ABFT. Furthermore, experimental fault injection results reveal that GCN-ABFT achieves slightly higher fault detection accuracy, due to its reduced check state and lower susceptibility to false positives.

Taken together, these contributions address both performance and reliability in GCN acceleration. On the one hand, FusedGCN demonstrates that architectural innovation can significantly reduce execution time while maintaining low hardware overhead. On the other hand, GCN-ABFT illustrates how algorithmic fault-tolerance techniques can be adapted to the unique computational patterns of GCNs to provide robust error checking at low cost. The synergy of these two approaches highlights the importance of co-optimizing efficiency and reliability in next-generation GNN accelerators.

## 6.1 FusedGCN: A Systolic Three-Matrix Multiplication Architecture for Graph Convolutional Networks

In FusedGCN, our goal is to design an efficient systolic architecture that would compute the triple matrix multiplication $S\,H^{k-1}\,W^k$ of Equation (6.1) in a *fused* manner, while also handling the sparseness of matrix $S$. For simplicity, we henceforth remove the layer indices from matrices $H$ and $W$ and denote the targeted three-matrix multiplication as $H^* = S\,H\,W$.

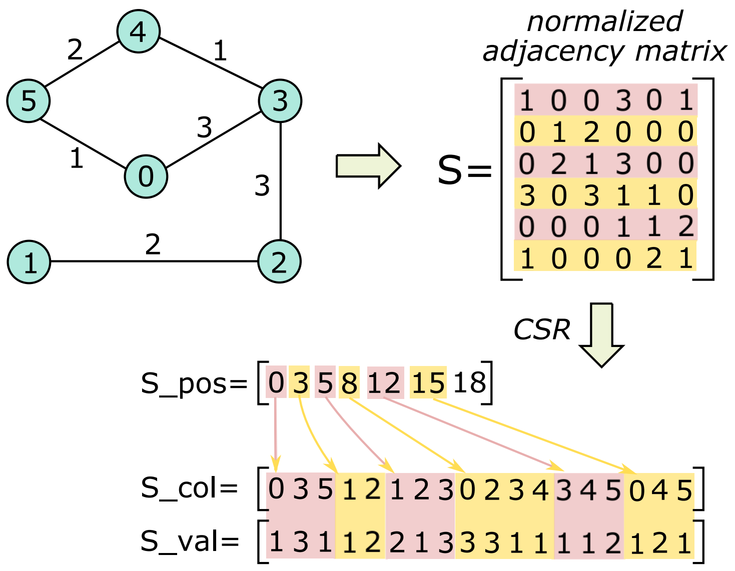
### 6.1.1 Enabling fused three-matrix multiplication

Let us denote the matrix product $H\,W$ as $C$. Then, each element $c_{jq}$ of $C$ can be written as the dot product of the $j$th row of $H$, $row_j(H)$ and the $q$th column of $W$, $col_q(W)$, i.e., $c_{jq} = row_j(H) \cdot col_q(W)$. The final output can be computed using $C$ as $H^* = S\,C$. For every element of $H^*$, it follows that

$h^*_{iq} = \sum_{j=0}^{N-1} s_{ij}c_{jq}$. Substituting the value of $c_{jq}$, we get

$$h^*_{iq} = \sum_{j=0}^{N-1} (s_{ij}\, row_j(H)) \cdot col_q(W) = \sum_{j=0}^{N-1} \sum_{k=0}^{I-1} s_{ij} h_{jk} w_{kq} \tag{6.2}$$
$$i \in [0, N-1] \text{ and } q \in [0, O-1]$$

To compute Equation (6.2) we follow two main principles: (a) we completely compute the $i^{\text{th}}$ row of the output before moving to the next one; (b) each element of the $i^{\text{th}}$ row of $S$ is read only once. To compute all elements $h^*_{iq}$ of the $i^{\text{th}}$ output row, we need to access all elements of the *same row* of $S$, i.e., $s_{ij}$. According to Equation (6.2), for each element, we should select an associated row of $H$. The association is performed using the column index of the corresponding element of $S$, e.g., $s_{24}$ is associated and multiplied with $row_4(H)$. The multiplications between the elements of $S$ and the associated rows of $H$ are the same for all elements of the same output row of $H^*$, and thus, they can be reused. The only difference between the two outputs of the same row of $H^*$ is the selected column of the weight matrix $W$.



normalized adjacency matrix
S=
1 0 0 3 0 1
0 1 2 0 0 0
0 2 1 3 0 0
3 0 3 1 1 0
0 0 0 1 1 2
1 0 0 0 2 1
CSR
S_pos= [0 3 5 8 12 15 18]
S_col= [0 3 5 1 2 1 2 3 0 2 3 4 3 4 5 0 4 5]
S_val= [1 3 1 1 2 2 1 3 3 3 1 1 1 1 2 1 2 1]
0
1
2
3
4
5
2
1
3
1
3
2

Figure 6.1: The CSR format of an example normalized adjacency matrix.

In practice, matrix $S$ is sparse. Therefore, it is stored in a compressed storage format, where only non-zero elements are indexed. Specifically, the Compressed Sparse Row (CSR) format uses three arrays for storing a sparse matrix. Figure 6.1 depicts an example graph, its normalized adjacency matrix $S$, and the CSR representation of said $S$ matrix. Array `S_val` stores the values of the non-zero elements of $S$ in row-wise order. The `S_col` array stores the corresponding column indices of each non-zero value, and `S_pos` contains pointers to the start of each row in `S_col`.

To transform the fused three-matrix multiplication to operate directly on CSR representation, we need to replace the instances of $s_{ij}$ with the corresponding CSR representation of the non-zero elements of $S$. The fused three-matrix multiplication kernel for a CSR-encoded matrix $S$ is shown Figure 6.2. The elements associated for computing one output row of the result are highlighted in Figure 6.3.

```
for (i = 0; i < N; i++) //temporal
 for (p = 0; p < I; p++) //spatial parallel
  for (q = 0; q < O; q++)  //spatial parallel
   for (j = S_pos[i]; j < S_pos[i+1]; j++) //temporal
    h_star[i][q] += (S_val[j] * h[S_col[j]][p]) * w[p][q];
```

Figure 6.2: The fused three-matrix multiplication kernel for a sparse normalized adjacency matrix encoded in CSR format.

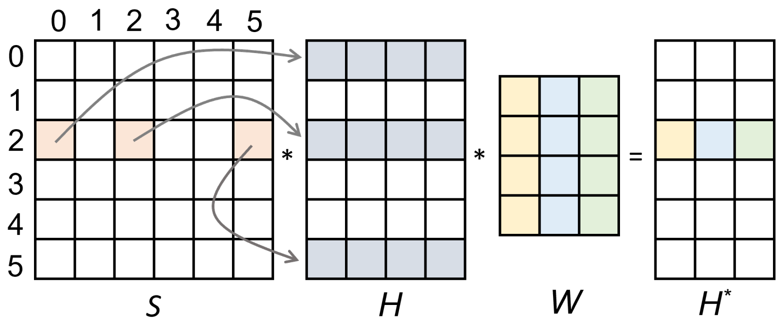


Figure 6.3: The elements involved in the fused three matrix multiplication. Computing the 2nd output row requires all non-zero elements of the 2nd row of $S$, the corresponding columns of $H$, and all columns of $W$.

The outer loop passes through all the rows of matrix $S$ (each node of the graph is visited once). The non-zero elements of a selected row of $S$ (row 2 in the example of Figure 6.3) with value `S_val[j]` are multiplied with all the elements of the row of $H$ that corresponds to the column `S_col[j]`. The product derived for each non-zero element is multiplied with all columns of $W$ to produce a complete row of the output. Since $S$ is sparse, only selected rows of $H$ are fetched: the ones that correspond to the columns of the non-zero elements of $S$. The same steps are performed in parallel for all elements that belong to the same output row, using the systolic array of Figure 6.4.

### 6.1.2 Supporting Arbitrary Input Feature Sizes

The basic FusedGCN systolic architecture of Figure 6.4 assumes that a row of matrix $H$ (consisting of $I$ elements) can be fetched as a single entity. This approach is practical and feasible for small values of $I$, but it does not scale with increasing numbers of input features. More realistically, a row of $H$ would be fetched in multiple segments/partitions, over consecutive clock cycles. The same limitation exists on the output side, i.e., when trying to write in memory a computed row of the output $H^*$. In most practical cases, the computed row would be stored in multiple segments.

The computation engine should match this read/write throughput limits to avoid underutilization. Figure 6.5 depicts a 3×3 systolic array used to compute in *tiles* the fused product of a GCN layer with $I$=9 input features and $O$=6 output features per node, respectively. Even if the computation evolves in more steps to facilitate the limited input-output bandwidth, the basic rule of the operation of FusedGCN does not change: one output row is first fully computed before moving to the next one.

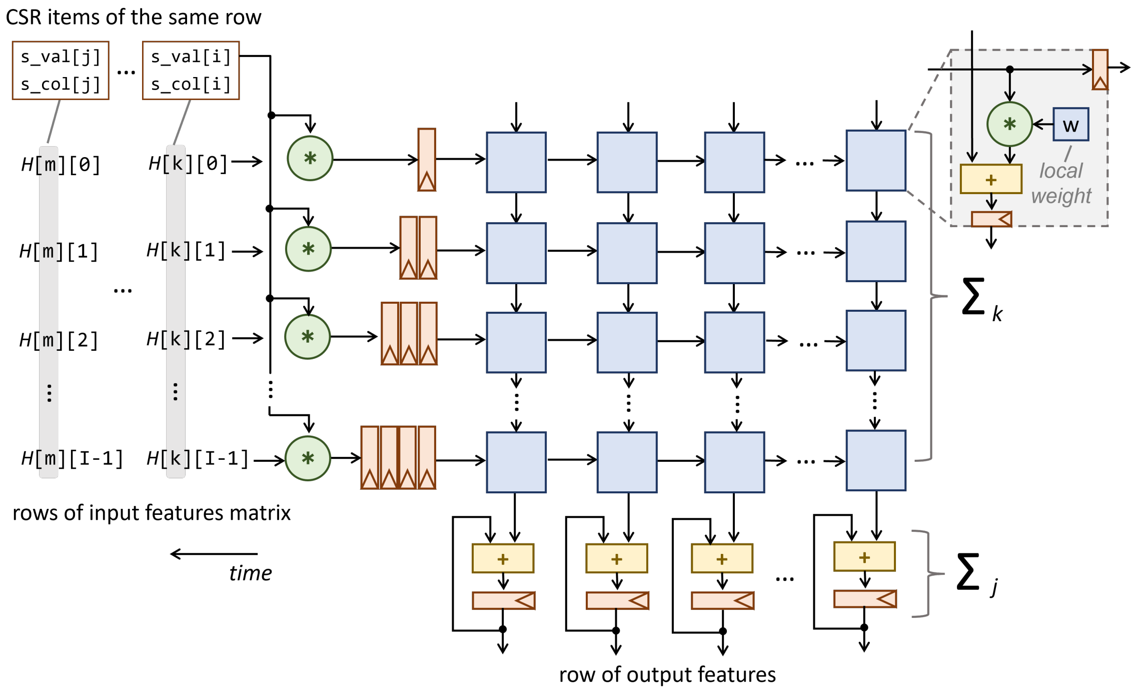


Figure 6.4: The weight-stationary FusedGCN systolic array that facilitates fused three-matrix multiplication. The datapath is reused for many cycles until all rows of the output have been computed. The weights of $W$ are preloaded into the systolic array and reused for all elements of $S$ and $H$.

### Dense Input Features

Each non-zero element of $S$ indexes the associated row of $H$, based on its column index. The selected row arrives in blocks of $K$ words and the computation associated with it evolves in $I/K$ phases. In each phase, we compute a partial result for the same output row using the blocks of the input features $H$ for multiple cycles, and by recycling the tiles of the weight matrix. Figure 6.6 highlights which elements of $row(H)$ and the weight matrix are involved in each computation step, assuming that $row(H)$ is partitioned in three blocks/segments. Following the fused matrix multiplication principle of Equation (2.13) to compute the $j^{\text{th}}$ output row of $H^*$, we need to fetch all non-zero elements of the same row of $S$, i.e., $s_{ij}$, together with their associated $row_j(H)$. Each $s_{ij}$ is fetched once and waits to be multiplied with all blocks of $row_j(H)$. Each element $s_{ij}h_{jk}$ of this intermediate vector should be multiplied with all $w_{kq}$ for all columns of the weight matrix.

In the example shown in Figure 6.6, the intermediate vector produced for ‘Block0’ should be multiplied (inner product) with all elements of the first three rows of $W$. Since the systolic array does not have more than three columns, the weights of the first three rows are partitioned in tiles A and B. The systolic array first uses the weights of tile A, and then the weights of tile B, both for ‘Block0’ of the input features. The partial result computed for the columns of tile A at the output accumulators should be kept and reloaded when computation is performed in weight tiles C and E. To enable this functionality, each column of $W$ has its own dedicated buffered accumulator. When the corresponding column is activated, its buffered accumulator value is loaded into the appropriate accumulators of the systolic array. Once the partial computation is finished, the partial result of the running accumulator is stored back into the

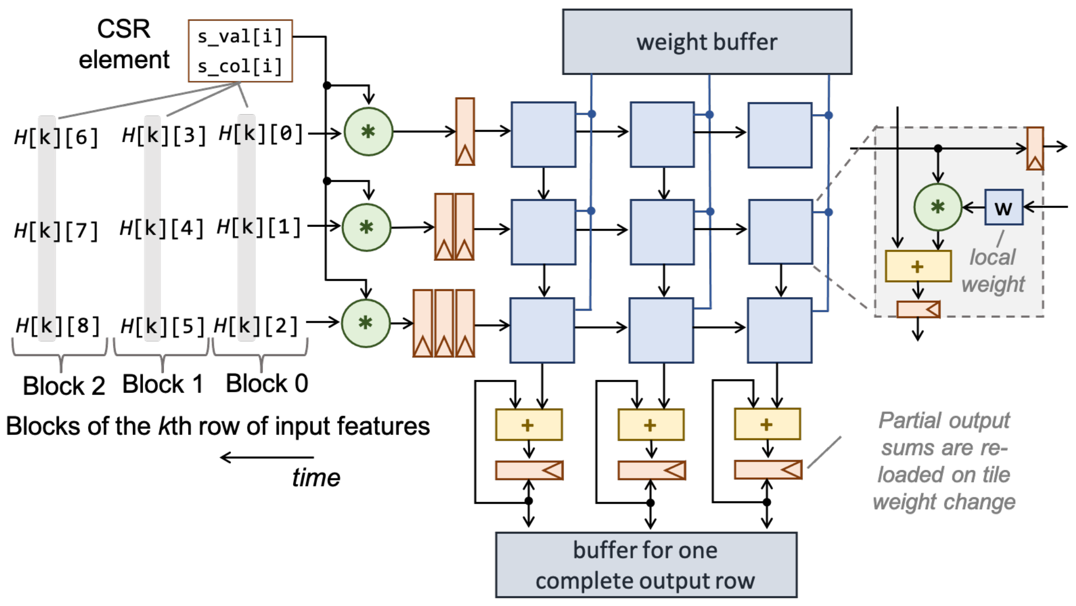


Figure 6.5: A tiled version of FusedGCN using a $3 \times 3$ systolic array for the computation of the tiled fused three-matrix multiplication. The non-zero elements of $S$ arrive serially and index the corresponding row of $H$. The row is fetched in blocks of three words each. The output row is also computed in blocks using an appropriate tile of weights in each cycle (recycled from the weight buffer).

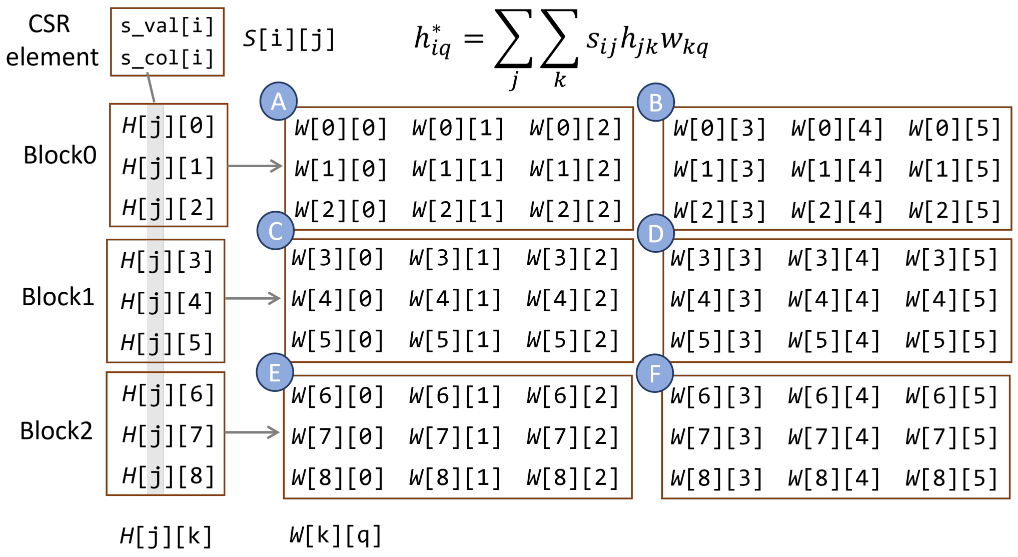


Figure 6.6: The non-zero elements of $S$ and the blocks of its associated row of $H$ are multiplied with the tiles of the weight matrix to produce the corresponding partial output result. The example here assumes 9 input features and 6 output features per node, organized in blocks of three elements.

buffered accumulator. Similar to the rewind of the partial output results, the tiles of weights are reloaded from the local weight buffer.

### Sparse Input Features

In many applications, as the ones used in the experimental results [130], the input features of the first layer of the GCN, $H^0$, are also highly sparse. The remaining layers are mostly dense. To take advantage of this characteristic, and without altering the fundamental operation of FusedGCN, we would like the computation process to skip the blocks of a row of H that consist of only zero elements.

For instance, following the example of Figure 6.6, if we know beforehand that 'Block1' of $row_j(H)$ is an all-zero vector, i.e., `H[j][3]=H[j][4]=H[j][5]=0`, then we can completely skip that block and let the systolic array operate only on'Block0' and 'Block2'. To completely skip an all-zero block, we need to encode the blocks of the input features matrix in a CSR format, as the one shown in Figure 6.7.

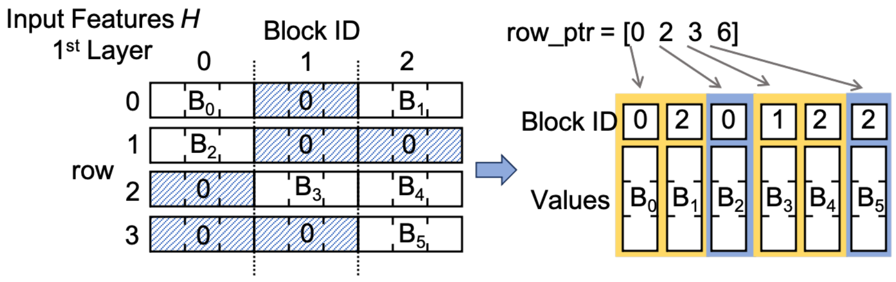


Figure 6.7: Encoding in CSR format the non-zero blocks of the input features matrix $H$ of the first layer.

Since only the non-zero blocks of each row of $H$ would be fetched, each block should use its block ID to determine which tiles of the weight matrix would be activated. For a dense matrix $H$, all the tiles of weights are used and always used in the same order (e.g., $A \rightarrow B \rightarrow \cdots \rightarrow E \rightarrow F$, for the example of Figure 6.6). On the contrary, for a sparse matrix $H$, only the tiles that correspond to non-zero blocks are needed. The block's ID determines which groups of rows of $W$ are used per block.

### 6.1.3 Evaluation

Having presented and analyzed the underlying concepts and the salient micro-architectural details of the FusedGCN design, we now proceed with a thorough evaluation of its performance and hardware cost.

#### Application characteristics

Table 6.1 summarizes the key characteristics of the 2-layer GCN datasets used in our evaluations. The first notable observation is that, in all applications, the number of input and output features between the first and the second (hidden) layer of the GCN are highly asymmetric. The input features of the first layer of the GCN (denoted as '#Inp. Features' in the table) are two, or even three, orders of magnitude larger than the input features of the second layer (i.e., the outputs of the first layer, denoted as '#Output Features').

Secondly, in all applications, the normalized adjacency matrices (i.e., S matrices) are very sparse, with their density ranging roughly from 0.0077% (Nell) to 0.21% (Reddit). Sparseness is also high in the input features of the first layer of all GCNs. For instance, in the Cora and CiteSeer datasets, the number

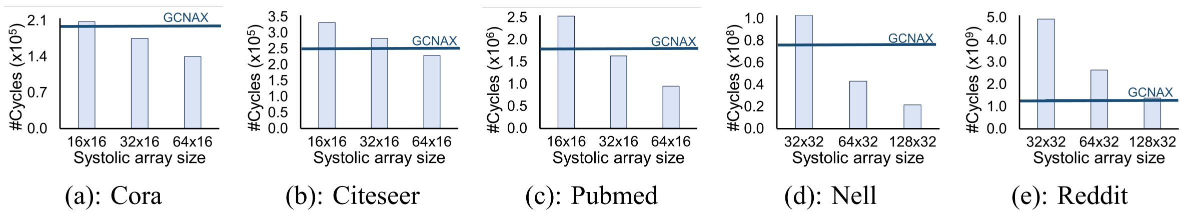


Figure 6.8: The execution time of various 2-layer GCN applications on the proposed FusedGCN design with varying size of systolic arrays, as compared to the performance of the GCNAX state-of-the-art architecture [128].

Table 6.1: The key characteristics of the 2-layer GCN applications used in the evaluations.

| | Cora | Citeseer | PubMed | Nell | Reddit |
|---|---|---|---|---|---|
| #Nodes | 2708 | 3327 | 19717 | 65755 | 232965 |
| nnz | 13264 | 12431 | 108365 | 331899 | 114848857 |
| Density of $S$ Matrix | 0.18% | 0.11% | 0.028% | 0.0077% | 0.21% |
| #Inp. Features $H^0$ | 1433 | 3703 | 500 | 61278 | 602 |
| Density | 1.27% | 0.85% | 10% | 0.11% | 51.6% |
| #Hidden Features $H^1$ | 16 | 16 | 16 | 64 | 64 |
| #Output Features $H^2$ | 7 | 6 | 3 | 186 | 41 |

of input features of the first GCN layer is 1433 and 3703, respectively. From those input features, only 1.27% and 0.85% in each case, respectively, are non zero.

### Execution time comparisons

The proposed FusedGCN architecture is compared against GCNAX [128], which is the most recent state-of-the-art GCN accelerator that exhibits the lowest execution time relative to other highly efficient approaches [131], [132]. The GCNAX design computes aggregation and combination in two separate phases, which take advantage of the sparseness of the normalized adjacency matrix and the possible sparseness of the input features of the first layer of the GCN. GCNAX takes advantage of this attribute in a *fine-grained* manner, whereas FusedGCN employs a *coarser-grained* approach; it only skips *blocks* of the input rows that consist of all zero elements, as previously described in Section 6.1.2.

The FusedGCN architecture was fully implemented in C++ and synthesized to Verilog RTL using Catapult HLS. The resulting RTL Verilog implementation was validated and verified to ensure functional correctness against the C++ testbench. The execution times reported here were derived after cycle-accurate RTL Verilog simulations of the GCN applications under investigation. The execution times for GCNAX are taken verbatim from [128].

Figure 6.8 reports the obtained execution times for the investigated 2-layer GCN applications of Table 6.1. For FusedGCN, execution times are provided for varying sizes of systolic arrays. Only the first dimension of the systolic array is varied, because the performance of FusedGCN is sensitive to the number of blocks with all-zero elements in the *input* features. For Nell and Reddit, we use larger array sizes, because these applications have very large datasets with larger feature sizes in the hidden layer (as shown in Table 6.1). For each application, the execution latency of GCNAX – as reported in [128] – is illustrated as a straight line.

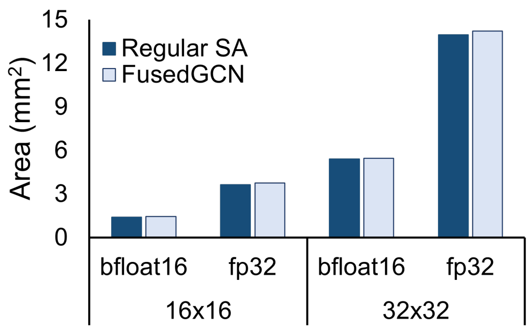
It is evident in Figure 6.8 that GCNAX outperforms the proposed FusedGCN design in systolic arrays with a small-size first dimension (corresponding to the input features). This is due to the above-mentioned fine-grained capitalization of sparseness enjoyed by GCNAX. Instead, the block-based approach of FusedGCN is not as effective in taking advantage of sparseness when the input-feature dimension of the systolic array is small. However, as the size of the first dimension of the systolic array increases, FusedGCN overtakes GCNAX in performance.

**Hardware complexity analysis**

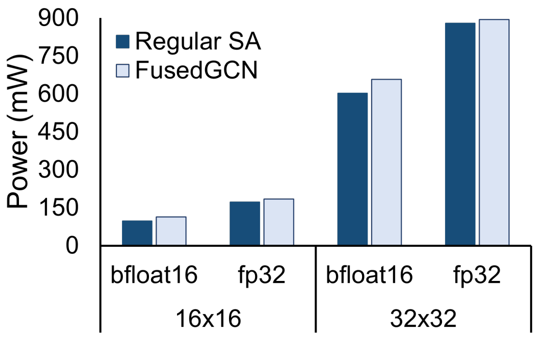
To evaluate the hardware complexity of FusedGCN, we compare it against a regular weight-stationary systolic array [43] that computes a standard matrix multiplication of two dense matrices. Structurally, FusedGCN consists of a regular systolic array and an extra column of multipliers at its west input (the green multipliers on the left side of Figures 6.4 and 6.5). To quantify the area/power overhead of the extra multipliers for various configurations, we mapped the Verilog RTL of both designs to a 45 nm standard-cell library using the Oasys logic synthesis tool, assuming a target clock frequency of 1 GHz.

Power was estimated after logic synthesis using the PowerPro power analysis and optimization tool. Switching activity information was gathered after simulating each design using the same random input matrices $H$ and $W$. To have a fair comparison, we set matrix $S$ equal to the identity matrix, on purpose. Hence, even though FusedGCN computes $SHW$, it effectively produces the two-matrix product $HW$.

For each implementation, we examined two different systolic array sizes (16×16 and 32×32), and two different floating point datatypes: ‘fp32’ for single-precision floating point format and ‘bfloat16’. In both designs under evaluation, Catapult HLS derived efficient pipelined floating-point datapaths.



(a)



(b)

Figure 6.9: The (a) area and (b) power cost of FusedGCN, as compared to an equal-size regular Systolic Array (SA), for two different floating point datatypes and two different array sizes.

The goal is to highlight the logic overhead of the extra multipliers in the proposed FusedGCN architecture. Therefore, for both designs, we assume that each PE stores one weight of the same width as the input data using a local register. The obtained area/power results are depicted in Figure 6.9. In all examined cases, the complexity of the fused systolic array closely tracks the complexity of the regular systolic array. Due to the extra multipliers, FusedGCN introduces a small area overhead, in the range of 1.2% to 3.6%, as compared to a weight-stationary regular systolic array (see Figure 6.9(a)). The average power overhead of FusedGCN, as compared to the regular systolic array, is 8% in all examined cases (Figure 6.9(b)).

### 6.1.4 Related Work

One of the earliest works on GCN acceleration, GraphACT [133] implements a GNN accelerator in a hybrid CPU-FPGA platform. The forward and backward passes of the GNN are computed on the FPGA, while the loss gradients and activation functions are computed on the CPU. Similarly, HyGCN [131], consists of separate aggregation and combination modules. Aggregation utilizes a set of SIMD cores in combination with a sampler and a sparsity eliminator, whereas combination is implemented with a standard systolic array. GNNerator [134] follows a similar approach and optimizes the data flow between the dense matrix engine and the graph engine. Zhang *et al.* in [135] focus on accelerating GCN training on FPGAs. By sampling parts of the adjacency matrix and input features in each training epoch, they show that they can achieve similar quality of results with lower computational complexity.

EnGN [136], inspired by CNN accelerators, treats graph convolution as a concatenated matrix multiplication of feature vectors, adjacency matrices, and weights. Similarly, Auten *et al.* in [137] combine existing accelerator modules designed for DNNs with graph-specific function accelerators.

Other approaches focused more on restructuring the computation involved in GCNs. AWB-GCN [132] removes the workload imbalance introduced by the irregular non-zero distribution in the adjacency matrix, using three workload balancing functions. Also, combination is computed first as a means to reduce the operations needed during the aggregation phase. BoostGCN [138] relies on data reorganization and tiling across all dimensions, i.e., vertices, edges, and input features, to improve memory accessing and increase utilization. Finally, DyGNN [139] dynamically skips vertices and edges that are considered redundant, thus saving computation time without reducing the quality of results.

## 6.2 GCN-ABFT: Low-Cost Online Error Checking for Graph Convolutional Networks

The straight-forward choice to apply error checking on graph convolution of Equation (6.1) (before the application of the activation function), is to apply ABFT separately on the two phases of the computation.

Assuming a combination-first computation, which is the preferred dataflow in recent GCN accelerators [140] and requires the less operations in many applications, we should first compute the $X = H\,W$ and then compute the output feature matrix as $H_{out} = S\,X$.

Adhering to the ABFT methodology [101], for validating the first matrix multiplication $X = H\,W$, it is essential to compare the *actual checksum* of all elements of the output matrix $X$ with a *predicted checksum* derived independently by the elements of $H$ and $W$. The final checksum of $X$ can be computed as $e^T X e$ with $e^T = [1, 1, \dots 1]$. Expanding the checksum equation we get that $e^T X e = e^T H W e = (e^T H)(W e) = h_c w_r$. To compute this checksum, it suffices to enhance $H$ with an extra row that represents the per-column checksum vector of $H$, i.e., $h_c = e^T H$, and matrix $W$ with an extra column that includes the corresponding per-row checksum of $W$, i.e., $w_r = W\,e$. Since the weights are known beforehand

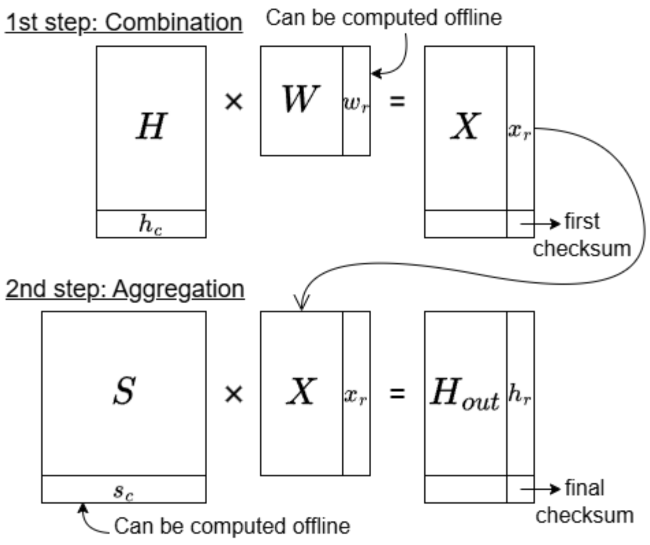


Figure 6.10: ABFT applied separately on the two phases of graph convolution operation.

$w_r$ can be computed offline or during weight loading to the GCN accelerator. On the contrary, the per-column checksum $h_c$ of $H$ can be computed only *online* (except only for the first GCN layer), since the input matrix $H$ of a GCN layer is the output matrix of the previous layer.

Performing matrix multiplication with the enhanced matrices $H$ and $W$ leads to

$$\begin{bmatrix} H \\ h_c \end{bmatrix} \begin{bmatrix} W & w_r \end{bmatrix} = \begin{bmatrix} \mathbf{HW} & Hw_r \\ h_c W & \mathbf{h_c w_r} \end{bmatrix} \tag{6.3}$$

Since in (6.3) matrices $H$ and $W$ are enhanced with their extra check state, the final checksum $h_c w_r$ of $X$ is naturally computed at the lower right end of the resulting matrix. This operation is also graphically depicted in Figure 6.10. To identify any erroneous result, the actual checksum accumulated online should be compared with the predicted checksum $h_c w_r$.

Completing the computation of the output features $H_{out}$ for a GCN layer requires multiplying the normalized adjacency matrix $S$ with the output of the previous step, i.e., $H_{out} = S\,X$. To check this multiplication with ABFT, we should enhance $S$ with the per-column checksum vector $s_c = e^T S$ and $X$ with the per-row checksum $x_r = Xe = HWe = Hw_r$. For static graphs $s_c$ can be computed offline and reused. Also, $x_r = Hw_r$ is already computed online at the upper right part of the output of (6.3). Multiplying the two enhanced matrices is performed as follows (shown also in Figure 6.10):

$$\begin{bmatrix} S \\ s_c \end{bmatrix} \begin{bmatrix} X & x_r \end{bmatrix} = \begin{bmatrix} \mathbf{SX} & Sx_r \\ s_c X & \mathbf{s_c x_r} \end{bmatrix} \tag{6.4}$$

Fault detection for this GCN layer occurs after finding a discrepancy between the predicted checksum $s_c x_r$ and the true checksum of $H_{out}$ computed online.

### 6.2.1 GCN-layer specific ABFT

Graph convolution is computed using two matrix multiplication steps. However, we are not obliged to apply ABFT at each step separately. Instead, we design a new error checker that computes the checksum of the three-matrix multiplication $H_{out} = S\,H\,W$ for a GCN layer in a *fused* manner. This consolidation of checksum computation removes unnecessary computations found in the split checking process, thus reducing significantly the overall number of operations required for applying ABFT in GCNs.

The checksum $H_{out}^C$ of the output of a GCN layer $H_{out}$ is equal to

$$H_{out}^C = e^T H_{out} e = e^T (SHW) e = (e^T S) H (We) = s_c H w_r \tag{6.5}$$

$s_c = (e^T S)$ and $w_r = We$ correspond to the per-column checksum vector of the normalized adjacency matrix $S$ and the per-row checksum vector of the weight matrix $W$, respectively.

In GCN-ABFT, based on the fused checksum (6.5) and assuming the same dataflow as the baseline approach, we perform checksum computation as follows: $H$ is multiplied *as is* and *without any check state* with the enhanced weight matrix $W$ that includes also the per-row checksum vector $w_r$. This operation is performed as follows:

$$H \begin{bmatrix} W & w_r \end{bmatrix} = \begin{bmatrix} \mathbf{H\,W} & \mathbf{H\,w_r} \end{bmatrix} \tag{6.6}$$

The resulting matrix $[HW \quad Hw_r]$ includes both the actual result of $X = H\,W$ and its check vector $H\,w_r$ that corresponds to the right part of (6.5). In fact, the computed check vector $H\,w_r$ corresponds to the per-row checksum vector $x_r$ of the intermediate output $X$ since $x_r = Xe = HWe = H(We) = Hw_r$. This feature is graphically depicted in Figure 6.11.

In the next step, the normalized adjacency matrix $S$ is multiplied with the output of the first step. To enable ABFT, we enhance $S$ with an extra row that corresponds to the per-column checksum of its elements $s_c$. The enhanced version of $[X \quad x_r] = [HW \quad Hw_r]$ is already available by (6.6). The multiplication of the enhanced matrices $S$ and $X$ is described as follows:

$$\begin{bmatrix} S \\ s_c \end{bmatrix} \begin{bmatrix} H\,W & H\,w_r \end{bmatrix} = \begin{bmatrix} \mathbf{S\,H\,W} & S\,H\,w_r \\ s_c\,H\,W & \mathbf{s_c\,H\,w_r} \end{bmatrix} \tag{6.7}$$

This matrix multiplications computes at its lower right part the fused checksum $s_c H w_r$ of Equation (6.5). Therefore, the final checksum is computed without relying on any intermediate per-column checksum for matrix $H$ of each GCN layer. The removal of this check state for $H$ removes redundant computations and makes the checker less vulnerable to faults that may affect its check state. Also, GCN-ABFT relies only on the per-column and per-row checksum vector of statically defined matrices $S$ and $W$. Therefore, it is easier for GCN-ABFT to compute such vectors offline and reuse them for multiple GCN inferences.

Overall, GCN-ABFT is a generic approach that can be applied to aggregation-first dataflows as well, since the computation of the fused checksum given by (6.5) holds independent of the order of computations and the structure or sparsity of the involved matrices. The fused checksum computation of GCN-ABFT may possibly be adapted to other classes of GNNs that their operation can be expressed as a three-matrix product [141].

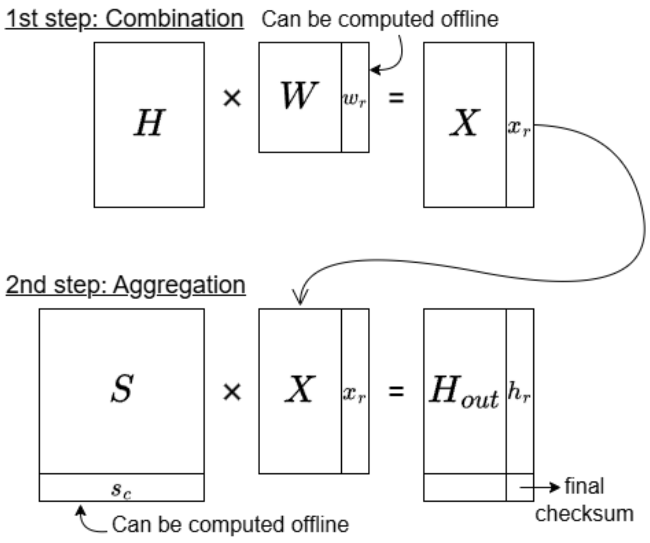


Figure 6.11: GCN-ABFT applied in the two phases of graph convolution. Fusing checksum computation removes the need to enhance matrix $H$ with additional check state.

The adoption of GCN-ABFT introduces two minor trade-offs. First, GCN-ABFT may miss a certain fault case that can be detected by baseline ABFT, which checks each matrix multiplication separately. A fault can go undetected by GCN-ABFT when the normalized adjacency matrix $S$ contains a column filled entirely with zeros. In this scenario, any fault affecting the first multiplication $H\,W$ is nullified at the output $S(H\,W)$, preventing its detection. Baseline ABFT, on the other hand, would have detected the fault in the first matrix multiplication. While this case is theoretically possible, it cannot occur in valid GCN scenarios as no column in $S$ can be entirely filled with zeros.

The second tradeoff refers to the timeline of error detection in GCN-ABFT. If an error occurs in the first matrix multiplication step $X = HW$, the baseline ABFT can report the error directly and avoid performing the second matrix multiplication $H_{out} = SX$. With GCN-ABFT this is not possible and error detection can be reported only *at the end of each GCN layer*, after both matrix multiplications per layer are completed.

The fixed-delay reaction of GCN-ABFT (once per GCN layer) matters only in the rare event of an actual error occurrence. In the vast majority of cases no error happens during GCN execution and the energy spent for validating the output's correctness is redundant and should be minimized as much as possible. Thus, the savings offered by GCN-ABFT are truly important, since energy is saved when it matters most and that is during error-free operation. Additionally, when the error occurs in the second matrix multiplication of graph convolution both baseline ABFT and GCN-ABFT detect the error at the same time and always within the time bounds of a GCN layer.

### 6.2.2 Evaluation

The goal of the experimental results is to compare the performance of GCN-ABFT relative to baseline ABFT both in terms of fault-detection accuracy and the number of operations required to perform checking. This quantitative analysis was performed on four well-known GCN applications of varying complexity used for node classification: Cora, Citeseer, PubMed and Nell [142].

### Experimental Setup

To evaluate the quality of error detection for both approaches, we introduce random single-bit flips [143] into the results of arithmetic operations [103, 144] within matrix multiplication (multiply and add) or checksum accumulation, at randomly selected time points per GCN layer. The affected arithmetic operations for matrix multiplications involve single-precision floats, while checksum accumulation uses double-precision floats. All bits of every arithmetic operation output can be flipped with equal probability.

A fault can occur at any time point during the execution of a GCN layer. Consequently, faults are more likely to occur during the matrix multiplication step that lasts longer (involving larger matrices). Similarly, due to the higher number of multiply-add operations in matrix multiplication compared to checksum accumulation, faults are more likely to affect multiply-add operations.

We assume memory is protected by error detection [145] and thus input data fetched are fault-free. However, arithmetic faults on index increments that track non-zero elements in sparse matrices $S$ and $H$, which are stored in CSR format [146], can indirectly corrupt memory accesses. This can lead to incorrect data fetching that ABFT cannot detect. As a result, we did not inject such faults. In any case, address generation logic is relatively small compared to wide floating-point datapaths, making bit flips in this area infrequent.

After fault injection the observed behavior, at the end of a GCN layer, may fall into one of three categories:

1. Detected: A faulty output was computed and ABFT detected it.
2. False positive: A fault injected into the checksum accumulation led ABFT to incorrectly identify a correct output result as erroneous.
3. Silent: Error remains undetected by ABFT. This case is a result of floating point rounding errors that does not allow to distinguish the effect of an injected fault from a correct result.

To prevent silent faults during our fault-injection campaigns, we need to establish a threshold [147] that differentiates between silent faults and other fault categories. This threshold, determined experimentally for specific GCN applications and datasets, considers error bounds ranging from $10^{-4}$ to $10^{-7}$. We found that setting the threshold above $10^{-7}$ eliminates silent faults in all GCNs.

False negative faults require a fault injected to matrix multiplication and checksum accumulation to cancel each other thus causing ABFT to fail to identify an actual fault in the output. Such cases were not observed in our experiments.

### Error detection quality

Table 6.2 summarizes the categorization of the injected faults derived from 5000 independent fault-injection campaigns. Using more fault-injection campaigns do not change the observed behavior. In this setup, a single fault was injected in each application. Which GCN layer actually experiences the

Table 6.2: Fault detection accuracy for representative GCN applications for a single injected fault using several error bounds

| GCN | Critical Faults | Avg. Nodes Affected | | $10^{-4}$ | | $10^{-5}$ | | $10^{-6}$ | | $10^{-7}$ | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | Split | GCN-ABFT | Split | GCN-ABFT | Split | GCN-ABFT | Split | GCN-ABFT |
| Cora | 96.92% | 68.61% | **Detected** | 95.46% | 96.42% | 95.52% | 96.06% | 95.80% | 96.66% | 95.80% | 96.66% |
| | | | **False Pos** | 3.84% | 3.14% | 4.14% | 3.18% | 4.20% | 3.34% | 4.20% | 3.34% |
| | | | **Silent** | 0.70% | 0.80% | 0.34% | 0.40% | 0.00% | 0.00% | 0.00% | 0.00% |
| Citeseer | 92.60% | 33.70% | **Detected** | 93.64% | 94.42% | 95.16% | 96.72% | 95.44% | 97.06% | 95.44% | 97.06% |
| | | | **False Pos** | 3.62% | 2.42% | 4.48% | 2.86% | 4.56% | 2.94% | 4.56% | 2.94% |
| | | | **Silent** | 2.74% | 3.16% | 0.36% | 0.42% | 0.00% | 0.00% | 0.00% | 0.00% |
| PubMed | 96.10% | 28.32% | **Detected** | 94.32% | 94.96% | 94.72% | 95.86% | 96.14% | 97.02% | 96.38% | 97.42% |
| | | | **False Pos** | 3.06% | 2.54% | 3.08% | 2.56% | 3.54% | 2.56% | 3.62% | 2.58% |
| | | | **Silent** | 2.62% | 2.50% | 2.20% | 1.58% | 0.32% | 0.42% | 0.00% | 0.00% |
| Nell | 95.32% | 61.06% | **Detected** | 94.88% | 95.38% | 95.40% | 96.00% | 96.84% | 97.62% | 96.90% | 97.82% |
| | | | **False Pos** | 2.58% | 1.78% | 2.64% | 1.92% | 3.02% | 2.14% | 3.10% | 2.18% |
| | | | **Silent** | 2.54% | 2.84% | 1.96% | 2.08% | 0.14% | 0.24% | 0.00% | 0.00% |

Table 6.3: Millions of arithmetic operations needed for executing and validating representative GCN applications

| | True | Split | | GCN-ABFT | | Savings | |
|---|---|---|---|---|---|---|---|
| GCN | Out | Check | Total | Check | Total | Check | Total |
| Cora | 2.8 | 0.55 | 3.35 | 0.44 | 3.24 | 20.0% | 3.3% |
| Citeseer | 4.6 | 0.80 | 5.40 | 0.60 | 5.20 | 25.0% | 3.7% |
| Pubmed | 37.6 | 4.60 | 42.20 | 4.04 | 41.64 | 12.2% | 1.3% |
| Neil | 1745.9 | 84.30 | 1830.20 | 59.9 | 1805.8 | 28.9% | 1.3% |

fault is proportional to its execution time. Fault injection results in 71.1% of all possible bit flips in multiply-add outcomes and 55.8% in the checksum accumulator, on average.

Columns 2–3 quantify how much critical were the injected faults for each GCN applications and Columns 5–12 report the fault detection accuracy of both ABFT variants. A fault is critical if it changes the final classification of at least one node of the graph. For instance, as shown in Column 2 of Table 6.2, in Cora, 96.92% of the injected faults triggered the misclasification of at least one node of the graph. Additionally, in Column 3 of Table 6.2, we report the average number of nodes of each graph that are critically affected by each injected fault. In this way, we quantify the spread of the effect of the fault in all nodes of the graph. For instance, in Citeseer, each fault causes the misclassification of 33.7% of the nodes, on average.

Fault criticality at the application level does not change the fault detection properties of ABFT. For instance, a fault may be non-critical at the application level and still be considered as detected since it was identified as ruining the result of matrix multiplication at the checksum level.

Both techniques achieve high fault detection accuracy, exceeding 93% in all cases. Silent faults disappear in all applications, when assuming that a fault is detected if the absolute difference between the predicted and output checksum exceeds $10^{-7}$. In all cases, GCN-ABFT exhibits fewer false positives because it calculates the checksum using Equation (6.5) and doesn't require a check state for matrix $H$, unlike the baseline ABFT approach (Eq. (6.3)).

We performed additional experiments by injecting more than one single-bit arithmetic faults per GCN application in randomly selected time points. In such cases, fault detection for baseline ABFT and GCN-ABFT reaches 100% offering an almost indistinguishable error detection quality.

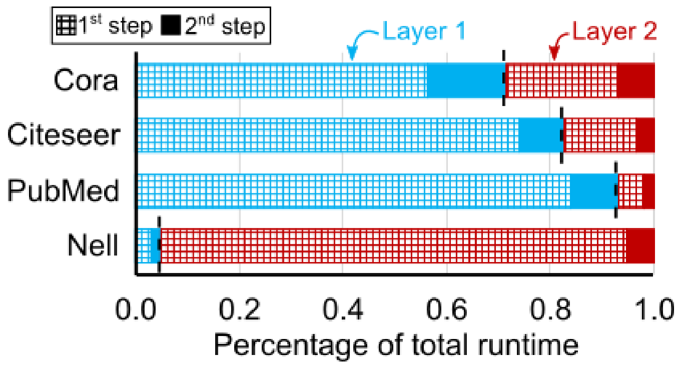


Figure 6.12: How the execution time is split across the first and the second matrix multiplication step of each GCN layer for both layers of the examined GCN applications.

### Computation cost

Computing and validating a GCN layer using the GCN-ABFT's checksum computed via Equations (6.6) and (6.7) requires *less operations in overall* than the split-checksum approach implemented using Equations (6.3) and (6.4).

To quantify the efficiency gains of GCN-ABFT compared to baseline ABFT (which checks each matrix multiplication separately), we measured the total number of operations required in each case including: (a) Operations essential for computing the actual GCN layer output. (b) Operations needed to compute the actual checksum (sum of all output elements) and the predicted checksum at each check step. Table 6.3 details the number of operations per category (Multiplications and additions are counted equally). GCN-ABFT demonstrates savings in checksum computation that range between 12% and 28% and translate to 1.3%–3.7% savings in overall computation cost. This efficiency improvement stems from the fused-approach used by GCN-ABFT that computes one check per GCN layer: (a) actual checksum computations are halved because GCN-ABFT only verifies the final GCN layer output, requiring the sum of elements only from the second matrix multiplication; (b) the reduced check state needed by Equation (6.6) compared to Equation (6.3) translates to fewer operations for calculating the predicted checksum.

### Error-detection latency gap between GCN-ABFT and baseline ABFT

GCN-ABFT reliably detects errors at the end of each GCN layer, regardless of the error's timing within the two multiplication steps. While baseline ABFT could potentially detect first-step errors earlier, the domination of the runtime of first multiplication phase (combination) in each GCN layer, as shown in Figure 6.12, makes this advantage negligible.

Figure 6.12 shows the time spent at each multiplication step of a GCN layer normalized to the total runtime of the specific layer for all examined applications. The applications examined are two-layer GCNs. The textured regions represent the runtime of the first multiplication step in both layers of the examined GCNs. The uniformly colored regions represent the second multiplication step in each case. In all cases, the first multiplication step of both layers dominates the overall runtime. For instance, for PubMed, the first multiplication step of both layers are responsible for more than the 90% of the runtime, while, for Nell the first multiplication step of both layers is 95% of the overall execution time. Therefore, while baseline ABFT can signal errors after the first multiplication, GCN-ABFT's delay in waiting for the second multiplication as well, is negligible per layer, making its fixed-delay response a minor issue.

## 6.3 Conclusion

In this section we introduced, to the best of our knowledge, the first systolic array architecture capable of performing three-matrix multiplication in a fused manner, thereby accelerating graph convolutions. The proposed *FusedGCN* architecture efficiently exploits the inherent sparsity of the adjacency matrix as well as the potential sparsity in the first-layer input features. Its systolic structure allows dataflow to be fully or partially unrolled, enabling adaptability to diverse input and output bandwidth constraints. Although specifically designed with GCN workloads in mind, FusedGCN is inherently a generic three-matrix multiplier. By organizing data within the matrices $S$, $H$, and $W$, or by setting any of these matrices to the identity, FusedGCN can seamlessly implement arbitrary two-matrix multiplications, spanning both sparse$\times$dense and dense$\times$dense cases.

Complementing this architectural advance, we proposed *GCN-ABFT*, a reliability mechanism tailored to the computational patterns of GCNs. Unlike conventional approaches that verify each multiplication step independently, GCN-ABFT directly computes a fused checksum corresponding to the final output of the three-matrix product. This approach significantly reduces the number of operations required for online verification while preserving strong fault detection capabilities. Experimental evaluation across diverse GCN applications confirms the effectiveness of this method, showing that the fixed-delay response of GCN-ABFT introduces negligible overhead in practice. Since the first multiplication step in a GCN layer dominates execution time, the additional response delay relative to baseline two-step ABFT is effectively masked.

Together, FusedGCN and GCN-ABFT address both performance and reliability challenges in GCN acceleration. The fused architecture delivers high efficiency and flexibility for sparse and dense workloads, while the integrated ABFT mechanism ensures correctness at low cost. These complementary contributions illustrate the value of co-designing computation and reliability mechanisms, paving the way for next-generation accelerators that are not only fast and efficient but also robust enough for deployment in safety-critical machine learning applications.

# Chapter 7

# SIMD Softmax: Reusing Softmax for Function Evaluation

Systolic accelerators have emerged as the dominant architectural template for deep learning workloads, as they excel at executing the dense matrix multiplications that dominate the arithmetic intensity of neural networks. Yet, these accelerators are not self-sufficient: matrix multiplications alone are not enough to implement modern models. At the end of every linear block, the results must pass through *activation functions* such as ReLU, sigmoid, tanh, softmax, or GELU, which inject non-linearity and statistical robustness into the network. Thus, systolic architectures are commonly paired with dedicated *function evaluation modules* that act as a computational back-end, transforming raw matrix products into valid activations before they are consumed by subsequent layers.

This layered organization can be viewed hierarchically, as shown in Figure 7.1. At the top of the accelerator, large systolic arrays stream and process tiles of data in parallel, while at the bottom, a function evaluation unit is invoked repeatedly to execute diverse non-linear transformations. From a hardware standpoint, the systolic array and the function evaluation module form a producer–consumer pair: the array produces partial sums at high throughput, and the function evaluation engine consumes them to compute the corresponding activations. Efficiency depends not only on the performance of the systolic array itself, but equally on the ability of the function unit to sustain throughput and avoid becoming a bottleneck.

The importance of this organization is amplified in Transformer networks [21], which have become the de facto standard for natural language processing and, increasingly, computer vision. In transformers, activation functions are more than auxiliary operations: the softmax function defines the attention mechanism itself, and GELU controls the non-linearity in the feed-forward block. Figure 7.2 shows the structure of an encoder-only transformer. After the systolic array projects the inputs to the Query ($Q$), Key ($K$), and Value ($V$) matrices and computes their products, the results are passed to a softmax unit to normalize attention scores, and later to a GELU operator in the feed-forward network. Both functions appear repeatedly across all layers of the network and contribute significantly to execution time [148].

Current transformer accelerators mitigate this overhead by integrating specialized softmax hardware units [148–156]. Despite differences in their micro-architectures, these designs all exploit the inherently parallel nature of softmax: given a vector of inputs, they simultaneously produce a vector of normalized

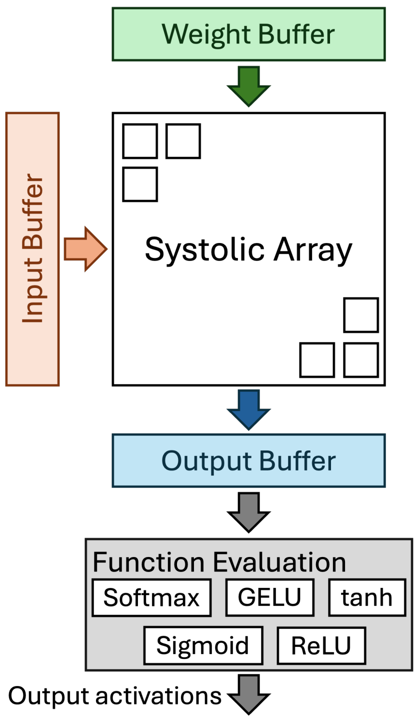


Figure 7.1: Overall organization of systolic accelerators: the systolic array executes matrix multiplications, while the function-evaluation module at the bottom computes non-linear activations.

outputs. This SIMD-style datapath closely resembles the systolic array itself, suggesting a broader design principle: instead of constructing independent hardware units for every activation function, one can *reuse the softmax operator* as a general-purpose function-evaluation engine for functions with an exponential-based formulation. GELU, for instance, can be expressed in terms of a two-element softmax, allowing multiple GELUs to be evaluated in parallel using the same SIMD structure.

In this chapter, our objective is to *leverage* state-of-the-art softmax architectures and their inherent parallelism to enable the computation of additional activation functions. To this end, we first reformulate the evaluation of these functions in terms of the softmax operator. We then demonstrate how a conventional softmax hardware unit can be incrementally extended into a dual-mode unit capable of computing both GELU and softmax, which are the most widely used activation functions in Transformer architectures [4,5,21]. Crucially, this approach ensures that GELU evaluation benefits from the intrinsic vectorized (parallel) operation of softmax, thereby allowing multiple GELU computations to be carried out in parallel.

To quantify the accuracy of computing GELU via softmax, we computed inference on various BERT-based applications [157–161] using three GELU variants. The proposed one, a fully accurate one that uses floating point arithmetic and a state-of-the-art integer GELU [25]. In all cases, the proposed approach achieves the same accuracy as compared to the rest models.

To assess hardware complexity we implemented the proposed combined design and compared it to the integer GELU [25] assuming the same degree of available parallelism. The combined GELU-softmax hardware unit leads to more area and power efficient designs than the ones that compute softmax and GELU separately. The savings achieved are 3.8%–8.4% for area and 10.7%–13.2% for power.

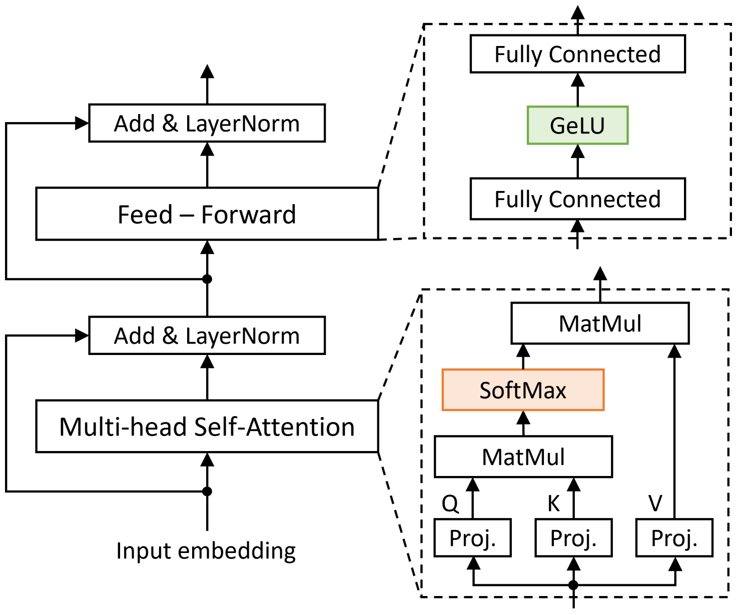


Figure 7.2: An encoder-only transformer layer. For instance, BERT-base [4] consists of twelve such layers.

## 7.1 Activation function computation via a softmax operator

Softmax transforms a vector of $N$ values $\vec{x} = [x_1, x_2, \ldots, x_n]$ to a vector of probabilities. The $i$-th element of the resulting output vector $\vec{y} = \text{softmax}^N(\vec{x})$ equals

$$y_i = \text{softmax}_i^N(\vec{x}) = \frac{e^{x_i}}{\sum_{j=1}^{N} e^{x_j}} \tag{7.1}$$

In its simplest form, the first element of the output of softmax when applied on a two-element vector $[x_1, x_2]$ (i.e., $N = 2$), equals

$$\text{softmax}_1^2([x_1, x_2]) = \frac{e^{x_1}}{e^{x_1} + e^{x_2}} \tag{7.2}$$

In order to evaluate different activation functions within the framework of a softmax operator, each function is expressed as a fraction, with the numerator given by an exponential term and the denominator corresponding to the sum of exponentials.

### 7.1.1 Sigmoid

The sigmoid activation could be mathematically expressed as

$$\text{sigmoid}(x) = \frac{1}{1 + e^{-x}} \tag{7.3}$$

Which could be reformulated to a fraction of exponentials as

$$\text{sigmoid}(x) = \frac{e^0}{e^0 + e^{-x}} \tag{7.4}$$

The fraction of exponentials in ( 7.4) corresponds exactly to the output of the softmax function applied to the two-element vector $[0, -x]$. Therefore, the sigmoid function can be equivalently computed by means of a two-element softmax operator:

$$\text{sigmoid}(x) = \text{softmax}_1^2([0, -x]) \tag{7.5}$$

### 7.1.2 Hyperbolic Tangent ($tanh$)

The $tanh$ function is written as a fraction of exponents, by definition

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} = \frac{e^x}{e^x + e^{-x}} - \frac{e^{-x}}{e^x + e^{-x}} \tag{7.6}$$

The expression in ( 7.6) exhibits the same functional form as the difference of the two components of a softmax operation over the two-element vector $[x, -x]$. Hence, the $tanh$ function may equivalently be realized through a two-element softmax operator, as shown below:

$$\tanh(x) = \text{softmax}_1^2([x, -x]) - \text{softmax}_2^2([x, -x]) \tag{7.7}$$

### 7.1.3 swish

Swish activation function, by definition, expressed through sigmoid, as

$$\text{swish}(x) = x \cdot \text{sigmoid}(\beta \cdot x) \tag{7.8}$$

Combining the equations ( 7.5) and ( 7.8), swish activation function could be computed via a softmax operator, as follows:

$$\text{swish}(x) = x \cdot \text{softmax}_1^2([0, -\beta \cdot x]) \tag{7.9}$$

### 7.1.4 GELU

GELU follows a different form and is computed via the error function as follows [162]:

$$\text{GELU}(z) = \frac{1}{2} z \left(1 + \text{erf}\left(\frac{z}{\sqrt{2}}\right)\right) \tag{7.10}$$

In [162] it was shown that GELU can be approximated via the tanh function as follows:

$$\text{GELU}(z) = \frac{1}{2} z \left(1 + \tanh\left[\sqrt{\frac{2}{\pi}} \left(z + 0.044715 z^3\right)\right]\right) \tag{7.11}$$

Setting $k = \sqrt{\frac{2}{\pi}} \left(z + 0.044715 z^3\right)$, we get that

$$\text{GELU}(z) = \frac{1}{2} z \left(1 + \tanh(k)\right) \tag{7.12}$$

Expressing tanh function in exponential form, we can write the term in the parenthesis of (7.12) as follows:

$$1 + \tanh(k) = 1 + \frac{e^k - e^{-k}}{e^k + e^{-k}} = \frac{2e^k}{e^k + e^{-k}} \tag{7.13}$$

By replacing (7.13) in (7.12), we effectively replace tanh from the approximate computation of GELU (7.11) with a fraction of exponentials:

$$\text{GELU}(z) = z \left( \frac{e^k}{e^k + e^{-k}} \right) \tag{7.14}$$

The fraction of exponentials in (7.14) has exactly the same form as the first output element of softmax when applied on a two-element vector $[k, -k]$ (see Eq. (7.2)). Following this observation, we can compute GELU via a two-element softmax operator as follows:

$$\text{GELU}(z) = z \,\text{softmax}_1^2\big([k, -k]\big) \tag{7.15}$$

To minimize cost, we aim at computing the two-element softmax (i.e., softmax$^2$) needed in (7.15) by *reusing a generic softmax hardware unit* that operates on $N$ input values $[x_1, x_2, \ldots, x_n]$. This goal can be satisfied in two ways:

- Set $x_1 = k$, $x_2 = -k$, and the rest $x_i$, with $i > 2$ equal to the minimum possible value. Keep only the first output element $y_1$ of softmax.
- Modify the $N$-element softmax hardware operator to operate independently on $N/2$ two-element subvectors.

In this chapter, we follow the second approach and show how a representative softmax hardware operator can be transformed to $N/2$ independent softmax operators that each one operates on two-element subvectors. As shown in the next Section 7.2 this choice also simplifies the parallelization of GELU computations.

## 7.2 Combined GELU-Softmax Hardware Unit

The reuse of a softmax hardware unit for computing GELU according to (7.15) requires two hardware modifications. The first one refers to the extra hardware needed “around” softmax for computing $k$ for each input and for finalizing the GELU result after multiplying the input with the result of the two-element softmax. The second one involves the modifications needed in softmax itself to operate in parallel and independently across multiple two-element vectors of the form $[k, -k]$.

### 7.2.1 Softmax with configurable vector width

Softmax (7.1) receives a vector of $N$ input values and returns a vector of $N$ probabilities. To have a stable result, the maximum of all elements of $\vec{x}$ is identified first and subtracted from the rest values of $\vec{x}$:

$$y_i = \text{softmax}_i^N(\vec{x}) = \frac{e^{x_i - \max \vec{x}}}{\sum_{j=1}^{N} e^{x_j - \max \vec{x}}} \tag{7.16}$$

In this way, the powers seen at each exponent $e^{x_i - \max \vec{x}}$ refer to zero or negative values. This simplifies the approximation of the exponents that can be performed either through table lookup, polynomial approximations or piece-wise linear approaches.

The next step after approximating input exponents $e^{x_i - \max \vec{x}}$, is the addition of the exponents of all input elements $\sum_{i=1}^{N} e^{x_i - \max \vec{x}}$. This can be done either using a tree of adders [149–151] or following an online approach [148, 152, 163] that computes this sum in parallel to the identification of the maximum value of $\vec{x}$ needed in the first step.

The last step for computing softmax involves the normalization of the input exponents, where each input exponent is divided by the sum of all other exponents. This last step is either performed using a divider unit per vector element [148][1], or performing computation in the logarithm domain where division is turned into a subtraction [149].

In this section, following the architecture of [149–151], we perform division in the logarithm domain. Thus, we consider for softmax the following implementation:

$$y_i = \text{softmax}_i^N(\vec{x}) = e^{x_i - \max \vec{x} - \log\left(\sum_j e^{x_j - \max \vec{x}}\right)} \tag{7.17}$$

For each exponentiation step we adopt the piecewise linear approximations (PWL) similar to [165]. To do this, each $e^{x - \max \vec{x}}$ is written as $2^{(x - \max \vec{x}) \log_2 e} = 2^u \cdot 2^v$, where $u$ is the integer part of the number and $v$ is the fraction [154, 165]. Integer powers of 2 such as $2^u$ are implemented as shifts, while $2^v$ is computed as an eight-piece PWL approximation on the range $[0, 1)$. The coefficients are derived using [166] on the said range. For computing the logarithm of the sum of exponents, we use exactly the same architecture as the PWL forward logarithm converter of [167].

In this section, we need a configurable softmax module that operates in two different modes. When in normal mode softmax operates in all $N$ input elements at once. On the contrary, when in GELU mode, it operates independently on $N/2$ two-element vectors thus maximizing parallelism. To achieve this property we need to apply three incremental changes to a softmax module that computes (7.17):

1) When in normal mode, the module that identifies the maximum of all input elements should return one output. In GELU mode, the same module should return one maximum element for every pair of elements ($N/2$ in total). This feature adds only a minimum multiplexing cost, since computing the global maximum of a vector already involves comparing locally all pairs of vector elements. Then, according to the chosen mode, the appropriate maximum value is subtracted from the input elements.

2) The adder tree that computes the sum of all exponents should return both the final sum (when in normal mode), as well as, the $N/2$ sums of consecutive pairs (when in GELU mode). This functionality is readily available in the design, since any adder tree computes the sum of consecutive pairs at the first level of the tree and the full-length sum at the last level of addition.

3) The computed sums are driven to separate logarithm unit. In normal mode, only one logarithm unit is needed for the whole softmax. On the contrary, when in GELU mode, $N/2$ logarithms are computed independently. Besides necessary multiplexing logic, these logarithm units are the only extra hardware needed for designing this dual-mode softmax unit.

To complete computation of (7.17) the output of each logarithm unit is subtracted from the corre-

[1]The number of dividers can be less than the output vector size depending on the output throughput requirements [164].

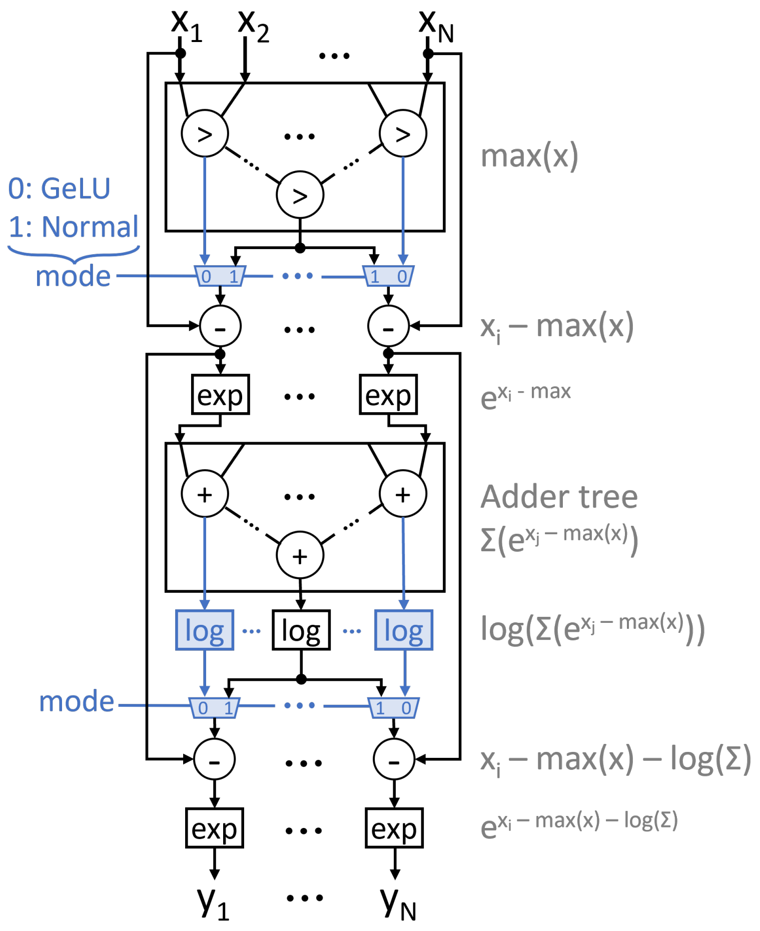


Figure 7.3: The dual-mode softmax hardware unit. Computation follows Equation (7.17) that implements division in the logarithm domain. The extra logic required for supporting the dual-mode of operation is highlighted in blue.

sponding difference $x_i - \max \vec{x}$ that is already available in both operating modes. At the end, to return from the logarithm domain, the result of each subtraction is passed to a separate exponent unit.

Figure 7.3 highlights the overall architecture of the dual-mode softmax unit that implements the required changes (steps 1–3). The experimental results presented in the next Section 7.3 show that the transformation of a state-of-the-art single-mode softmax unit [151] to a dual-mode one, incurs only marginal costs in area and power.

### 7.2.2 Computing multiple GELUs using dual-mode softmax unit

The dual-mode softmax unit allows us to compute independently the softmax of $N/2$ two-element vectors. We leverage this property *to compute $N/2$ GELU outcomes in parallel*. The organization of this combined GELU-softmax unit is depicted in Figure 7.4.

The $N/2$ inputs $z_i$, with an odd index $i$, are passed to a datapath of multipliers and adders that computes the corresponding values $k_i$ and $-k_i$, respectively. Then, the $N/2$ two-element vectors $[k_i, -k_i]$ are given to the softmax unit that is configured to operate in GELU mode. Following (7.15), to finish the

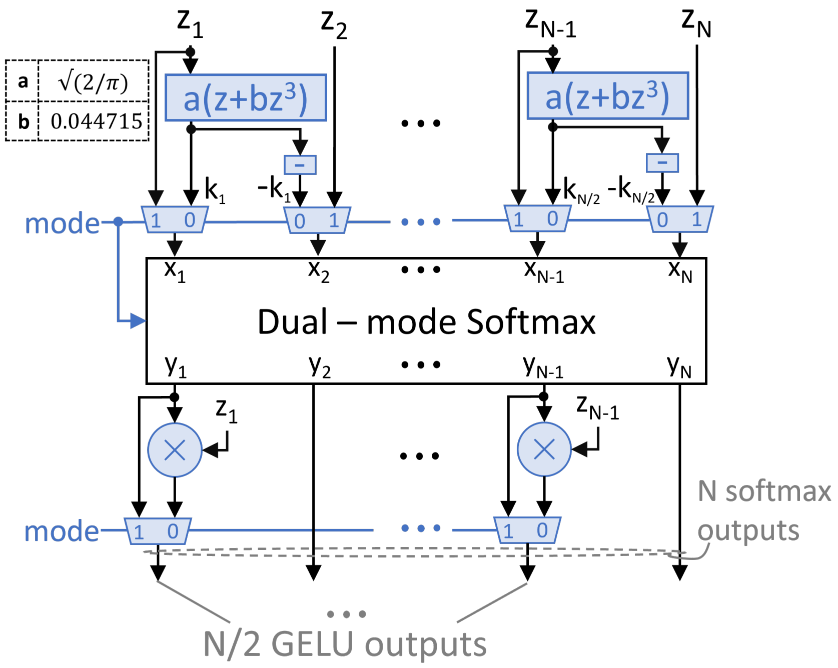


Figure 7.4: The combined GELU/softmax hardware unit. When in normal mode, softmax is driven by $N$ inputs $z_i$ and produces $N$ output probabilities. When in GELU mode, half of the inputs and outputs are used to compute $N/2$ GELU outcomes in parallel.

Table 7.1: Performance of the approximate GELU functions on BERT model for 8 tasks of the GLUE dataset [168].

| | | STS2 | MNLI-m | MNLI-mm | QQP | QNLI | CoLA | MRPC | RTE |
|---|---|---|---|---|---|---|---|---|---|
| Accuracy (%) | FP32 | 92.7 | 79.1 | 84.2 | 90.8 | 94.1 | 72.4 | 85.8 | 73.8 |
| | i-GELU | 92.4 | 79.0 | 84.0 | 90.7 | 94.1 | 72.1 | 86.6 | 73.7 |
| | Proposed | 92.5 | 79.1 | 84.1 | 90.7 | 94.0 | 72.3 | 85.7 | 73.8 |
| Mean Absolute Error | i-GELU | 0.054 | 0.0824 | 0.094 | 0.098 | 0.064 | 0.076 | 0.1204 | 0.179 |
| | Proposed | 0.0039 | 0.0057 | 0.0061 | 0.0083 | 0.0045 | 0.0067 | 0.0109 | 0.0154 |

computation of each GELU$(z_i)$, we need to multiply each $z_i$ with the first output produced by softmax for each two-element subvector. In this way, we are able to compute multiple GELU outcomes in parallel using extra multipliers and adders around *a shared dual-mode softmax unit.*

## 7.3 Experimental results

The experimental evaluation is twofold: First, we explore the accuracy of computing GELU using a dual-mode softmax unit in representative BERT-based applications [157–161]. Second, we quantify the savings achieved by using the combined GELU-softmax hardware unit shown in Figure 7.4 relative to state-of-the-art implementations that employ independent designs for GELU and a single-mode softmax unit. For all designs under comparison, we assume 16-bit fixed-point inputs with five integer bits and 32-bit integer arithmetic for all internal operations similar to the one employed for i-GELU in [25].

### 7.3.1 Accuracy of GELU function in BERT applications

We evaluate the **accuracy of the proposed GELU computation** by executing inference on the BERT model [4] using eight tasks of the GLUE dataset [168]. For comparisons, we considered three transformer implementations: (a) Model 'FP32' computes **all operations** including matrix multiplications and function evaluation using **single-precision** floating point arithmetic; (b) Model 'i-GELU' employs single-precision floating point arithmetic for all parts of the transformer **except for the GELU function**, where computations are done in integer arithmetic according to [25]; (c) The 'Proposed' model that employs floating point arithmetic for all parts of the transformer **except for GELU and softmax** functions that are computed using the proposed combined integer GELU-softmax unit.

The results gathered are depicted in Table 7.1. In terms of inference accuracy the performance of all three methods is indistinguishable. To have a more clear insight on the arithmetic performance of the proposed design and 'i-GELU' relative to 'FP32', we measured the mean absolute error of the outputs of the model in those two cases and compared it to that of 'FP32'. In all cases, the proposed design that follows the original tanh approximation that is computed via softmax, exhibits a smaller error that ranges between $10^{-2}$ – $10^{-3}$ for the various applications.

### 7.3.2 Hardware Complexity Evaluation

The hardware implementation of the proposed module has two goals. First, we aim at quantifying the cost of transforming a state-of-the-art single-mode softmax hardware unit [151] to a dual-mode one that could operate on two different vector widths. Second, we need to assess how much hardware area and power is saved when using the combined GELU-softmax unit relative to state-of-the-art designs that use separate hardware modules for GELU and softmax.

All designs that are open-sourced in Github [169] were implemented in C++ and synthesized to Verilog RTL using Catapult HLS for a target clock frequency of 500 MHz driven by a 45-nm standard-cell library. The final timing/area results were derived from the Oasys logic synthesis tool. The power consumption was estimated after synthesis using the PowerPro power analysis and optimization tool. The input data used for power estimation come from traces derived by the applications examined in Section 7.3.1.

Table 7.2: The area and power of a single-mode and a dual-mode softmax unit for two vector widths.

| N | Area ($\mu m^2$) | | | Power ($mW$) | | |
|---|---|---|---|---|---|---|
| | single mode | dual mode | Diff. | single mode | dual mode | Diff. |
| 8 | 92309 | 100675 | 9.1% | 13.51 | 13.95 | 3.3% |
| 32 | 363703 | 403025 | 10.8% | 57.74 | 58.84 | 1.9% |

Table 7.2 depicts the area and power results of a single-mode and a dual-mode softmax unit for two input sizes, e.g., $N = 8$ and $N = 32$. The results show that the **hardware cost for enhancing the functionality of a state-of-the-art single-mode softmax** unit [149–151] is marginal and scales well for both small and large input vector sizes. The average area and power overhead of adding an extra mode of operation to softmax for the two examined vector sizes is 9.9% and 2.6%, respectively.

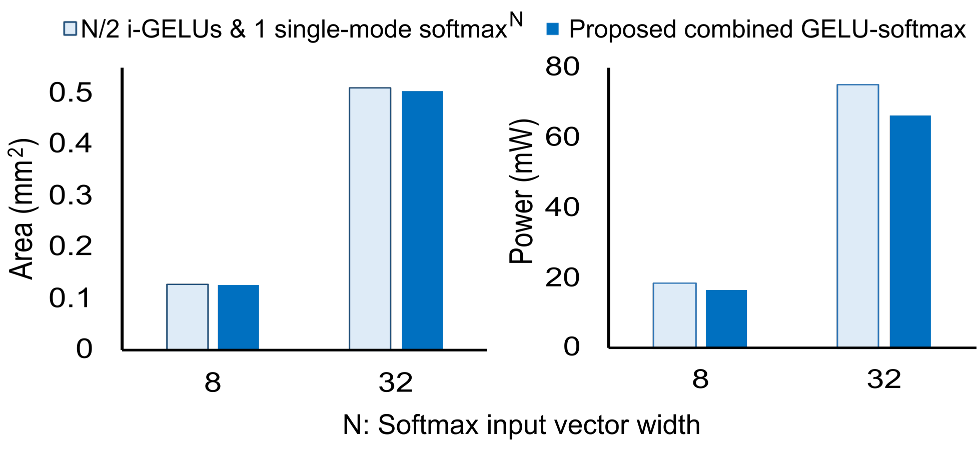


Figure 7.5: The area and power of the combined GELU-softmax unit introduced in this section and the design that employs $N/2$ i-GELU [25] units and a state-of-the-art single-mode softmax unit [151]. The dual-mode softmax unit used in the proposed design is an enhanced derivative of the same single-mode softmax unit.

To quantify the **area-power savings earned when following the combined GELU-softmax** architecture, we performed the following comparison. We designed **a combined GELU-softmax unit** for $N = 8$ and $N = 32$ following the organization shown in Figure 7.4 in each case. This unit can either produce $N$ output elements of a softmax$^N$ operation, or it can produce in parallel $N/2$ GELU outcomes. To have a fair comparison, we compared this architecture with a design that uses $N/2$ **i-GELU units** [25] that compute multiple GELUs in parallel and **one single-mode softmax unit** [149–151]. The single-mode softmax unit, used in comparisons, is the same design from which the dual-mode softmax unit, used in the proposed design, was derived. By using $N/2$ i-GELU units, this alternative design matches the computation throughput of the proposed design.

Figure 7.5 depicts the area and power of both designs under comparison. For $N = 8$, the proposed unit requires 8.4% less area and 10.7% less power. While for $N = 32$ elements, we report a 3.8% decrease in area and 13.2% decrease in power. In both cases, the majority of the power and area is due to the softmax units used in each case. The power reductions observed are a result of the smaller dynamic power used by the datapath used 'around' the dual-mode softmax relative to the polynomial evaluations employed in i-GELU [25].

## 7.4 Conclusions

GELU and softmax are two critical components of modern transformer architectures and their efficient acceleration in hardware is of paramount importance. In this section, we leverage the benefits of state-of-the-art softmax architectures and their inherent vector-parallel operation to design efficient vector-parallel combined GELU-softmax units. This combined operation made possible by (a) a new mathematical transformation that computes GELU using a two-element softmax operator and additional multiplication and additions, as well as (b) the low-cost design of a softmax operator with configurable vector width. The proposed combined GELU-softmax hardware unit not only reduces area and power consumption but also offers the same accuracy for representative NLP applications.

# Chapter 8

# Conclusions and Future Work

## 8.1 Summary

The increasing computational demands of modern machine learning (ML) workloads have placed domain-specific hardware accelerators at the center of research and industrial deployment. Among these, systolic arrays stand out for their ability to exploit data reuse and parallelism in dense linear algebra operations. However, their efficiency is challenged by emerging trends such as structured sparsity, graph-based learning models, and the growing importance of reliability in safety-critical systems. This thesis addressed these challenges through a comprehensive set of architectural, physical, and algorithmic innovations, targeting both performance and robustness across diverse application domains.

The first set of contributions explored optimizations for dense systolic arrays. We proposed ArrayFlex, a systolic array architecture with configurable transparent pipelining, which dynamically adapts pipeline depth to workload requirements. This flexibility reduces execution time and energy consumption by balancing the tradeoff between cycle count and clock period. Complementing this, an *asymmetric floorplanning methodology* was introduced to account for the inherent imbalance between horizontal and vertical interconnect demands. By optimizing processing element (PE) aspect ratios, we achieved measurable reductions in wirelength and interconnect power, highlighting the importance of layout-aware design in energy-efficient accelerators.

Beyond dense designs, the thesis made significant advances in sparse systolic tensor arrays. We proposed *DeMM*, a disaggregated matrix multiplication engine that supports relaxed structured sparsity patterns while maintaining regularity in hardware organization. By decoupling memory from compute units, DeMM enables fine-grained reconfiguration with minimal overhead, delivering substantial improvements in execution latency and power efficiency. We also introduced *DSSTA*, which eliminates input skewing through a diagonal and permuted-weight dataflow. This approach simplifies synchronization across PEs, reduces interconnect length, and lowers both area and power, demonstrating that dataflow innovation can substantially enhance sparse computation efficiency.

Since reliability is a prerequisite for deploying ML accelerators in mission-critical applications, the thesis also addressed fault tolerance in sparse systolic tensor arrays. We adapted Algorithm-Based Fault Tolerance (ABFT) to structured sparsity by integrating checksum logic along array peripheries and employing digit-serial arithmetic for efficient accumulation. This enables low-cost online error detection during inference. To complement ABFT, we proposed a periodic online self-test mechanism that proactively detects permanent faults before execution, requiring only a few test vectors and reusing stored weights. Together, these schemes deliver high fault coverage with modest area and power overheads, ensuring robust operation in energy-constrained environments.

We then extended these innovations to irregular workloads, particularly Graph Convolutional Networks (GCNs). The thesis introduced *FusedGCN*, the first systolic architecture capable of executing the entire three-matrix product of a GCN layer in a fused manner. By unifying aggregation and combination into a single dataflow, FusedGCN reduces inference latency relative to state-of-the-art designs while maintaining support for sparse adjacency matrices and input features. On the reliability side, we developed *GCN-ABFT*, which computes a fused checksum for the three-matrix product. This approach reduces verification overhead compared to conventional ABFT and improves fault detection accuracy, demonstrating that co-designing architecture and reliability mechanisms can benefit irregular workloads.

We introduce a SIMD softmax hardware unit capable of handling both full-vector and segmented softmax operations. The unit can execute a complete softmax over an input vector or several smaller ones over its partitions, with the intermediate results efficiently repurposed to implement various activation functions by expressing them through softmax mappings. This unified design allows a single SIMD module to support multiple nonlinear activations, lowering hardware complexity and redundancy while enhancing efficiency in large-scale Transformer workloads.

In summary, this thesis has advanced the state of systolic and tensor array accelerators by integrating layout-aware design, sparsity exploitation, and fault-tolerance mechanisms, while extending their applicability from CNNs to GCNs and Transformers.

## 8.2 Future Work

Future work can extend this research toward energy-efficient data-parallel architectures that dynamically adapt computation and data movement based on workload characteristics. Techniques such as adaptive voltage and frequency scaling (DVFS), approximate computing, and workload-aware power gating can be explored to minimize energy consumption without compromising accuracy. Furthermore, integrating in-memory and near-memory computing concepts within systolic or tensor-array structures could significantly reduce data transfer energy. These directions will be increasingly important as models grow in size and complexity, emphasizing the need for architectures that balance compute throughput with sustainable energy budgets.

From a performance perspective, future systolic-array-based accelerators can explore heterogeneous dataflow architectures that adaptively switch between systolic, SIMD, and sparse modes to match workload structure. Co-optimizing scheduling, dataflow mapping, and hardware reconfiguration can unlock new levels of utilization and throughput. Furthermore, the integration of 3D-stacked memory, chiplet-based modular designs, and high-bandwidth interconnects offers promising paths to overcome current bandwidth and scalability bottlenecks. These enhancements would enable accelerators to efficiently execute the increasingly diverse and irregular workloads characteristic of modern ML applications.

Reliability will continue to be a crucial research dimension, particularly as accelerators are deployed in safety-critical or edge environments where environmental and process variations increase fault susceptibility. Future research can investigate combining online self-test frameworks with error-predictive machine learning models could enable proactive reliability management with negligible overhead.

Finally, emerging device and packaging technologies offer exciting opportunities to rethink the boundaries of energy efficiency and performance. Non-volatile memories, photonic interconnects, and neuromorphic or analog in-memory computing could be explored as complementary technologies for future ML accelerators. Hybrid architectures that combine digital precision control with analog computation efficiency may yield significant gains in both power and performance density. Integrating these technologies within the context of systolic and tensor-based architectures will be a key step toward realizing next-generation machine learning accelerators that are energy-aware, fault-tolerant, and scalable to the needs of future AI workloads.

# Bibliography

[1] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You Only Look Once: Unified, Real-Time Object Detection," in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 779–788.

[2] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *IEEE Conference on Computer Vision and Pattern Recognition(CVPR)*, Jun 2016.

[3] P. Hansen, A. Vilkin, Y. Krustalev, J. Imber, D. Talagala, D. Hanwell, M. Mattina, and P. N. Whatmough, "ISP4ML: The Role of Image Signal Processing in Efficient Deep Learning Vision Systems," in *25th International Conference on Pattern Recognition (ICPR)*, 2021, pp. 2438–2445.

[4] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *North American Chapter of the Association for Computational Linguistics*, 2019.

[5] Z. Lan, M. Chen, S. Goodman, K. Gimpel, P. Sharma, and R. Soricut, "ALBERT: A lite bert for self-supervised learning of language representations," in *International Conference on Learning Representations (ICLR)*, 2020.

[6] D. Sarvamangala and R. V. Kulkarni, "Convolutional neural networks in medical image understanding: a survey," *Evolutionary intelligence*, vol. 15, no. 1, pp. 1–22, 2022.

[7] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[8] A. Kurani, P. Doshi, A. Vakharia, and M. Shah, "A comprehensive comparative study of artificial neural network (ann) and support vector machines (svm) on stock forecasting," *Annals of Data Science*, vol. 10, no. 1, pp. 183–208, 2023.

[9] F. Amato, A. López, E. M. Peña-Méndez, P. Vaňhara, A. Hampl, and J. Havel, "Artificial neural networks in medical diagnosis," pp. 47–58, 2013.

[10] S.-i. Amari, "Backpropagation and stochastic gradient descent method," *Neurocomputing*, vol. 5, no. 4-5, pp. 185–196, 1993.

[11] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[12] M. C. Mukkamala and M. Hein, "Variants of rmsprop and adagrad with logarithmic regret bounds," in *International conference on machine learning*. PMLR, 2017, pp. 2545–2553.

[13] Y.-H. Chen, J. Emer, and V. 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)*, 2016, pp. 367–379.

[14] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 2002.

[15] A. Vasudevan, A. Anderson, and D. Gregg, "Parallel multi channel convolution using general matrix multiplication," in *IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, 2017, pp. 19–24.

[16] B. Perozzi, R. Al-Rfou, and S. Skiena, "Deepwalk: Online learning of social representations," in *ACM International Conference on Knowledge Discovery and Data Mining (KDD)*, 2014, p. 701–710.

[17] Y. Wang, Z. Li, and A. Barati Farimani, "Graph neural networks for molecules," in *Machine learning in molecular sciences*. Springer, 2023, pp. 21–66.

[18] W. L. Hamilton, *Graph Representation Learning*. Morgan and Claypool, Synthesis Lectures on Artificial Intelligence and Machine Learning, 2020.

[19] R. Ying, R. He, K. Chen, P. Eksombatchai, W. L. Hamilton, and J. Leskovec, "Graph convolutional neural networks for web-scale recommender systems," in *ACM International Conference on Knowledge Discovery and Data Mining (KDD)*, 2018, p. 974–983.

[20] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," in *International Conference on Learning Representations (ICLR)*, 2017.

[21] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[22] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat *et al.*, "Gpt-4 technical report," *arXiv preprint arXiv:2303.08774*, 2023.

[23] A. Chowdhery, S. Narang, J. Devlin, M. Bosma, G. Mishra, A. Roberts, P. Barham, H. W. Chung, C. Sutton, S. Gehrmann *et al.*, "Palm: Scaling language modeling with pathways," *Journal of Machine Learning Research*, vol. 24, no. 240, pp. 1–113, 2023.

[24] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar *et al.*, "Llama: Open and efficient foundation language models," *arXiv preprint arXiv:2302.13971*, 2023.

[25] S. Kim, A. Gholami, Z. Yao, M. W. Mahoney, and K. Keutzer, "I-BERT: Integer-only bert quantization," in *Proceedings of Machine Learning Research (PMLR)*, 18–24 Jul 2021, pp. 5506–5518.

[26] T. Dao, D. Fu, S. Ermon, A. Rudra, and C. Ré, "Flashattention: Fast and memory-efficient exact attention with io-awareness," *Advances in neural information processing systems*, vol. 35, pp. 16 344–16 359, 2022.

[27] T. Dao, "Flashattention-2: Faster attention with better parallelism and work partitioning," *arXiv preprint arXiv:2307.08691*, 2023.

[28] Google. (2022) The bfloat16 numerical format. [Online]. Available: https://cloud.google.com/tpu/docs/bfloat16

[29] S. Markidis, S. W. Der Chien, E. Laure, I. B. Peng, and J. S. Vetter, "Nvidia tensor core programmability, performance & precision," in *International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2018, pp. 522–531.

[30] N. P. Jouppi, D. H. Yoon, M. Ashcraft, M. Gottscho, T. B. Jablin, G. Kurian, J. Laudon, S. Li, P. Ma, X. Ma, T. Norrie, N. Patil, S. Prasad, C. Young, Z. Zhou, and D. Patterso, "Ten lessons from three generations shaped google's tpuv4i: Industrial product," in *International Symposium on Computer Architecture (ISCA)*. IEEE, Jun 2021, pp. 1–14.

[31] A. Mishra, J. A. Latorre, J. Pool, D. Stosic, D. Stosic, G. Venkatesh, C. Yu, and P. Micikevicius, "Accelerating sparse deep neural networks," *arXiv preprint arXiv:2104.08378*, 2021.

[32] A. Zhou, Y. Ma, J. Zhu, J. Liu, Z. Zhang, K. Yuan, W. Sun, and H. Li, "Learning N:M fine-grained structured sparse neural networks from scratch," in *International Conference on Learning Representations (ICLR)*, May 2021.

[33] U. Evci, T. Gale, J. Menick, P. S. Castro, and E. Elsen, "Rigging the lottery: Making all tickets winners," in *International Conference on Machine Learning*, Jul. 2020, pp. 2943–2952.

[34] C. Yu, T. Chen, Z. Gan, and J. Fan, "Boost vision transformer with gpu-friendly sparsity and quantization," in *IEEE Conference on Computer Vision and Pattern Recognition*, 2023, pp. 22 658–22 668.

[35] D. Haziza, T. Chou, D. Choudhary, L. Wehrstedt, F. Massa, J. Yu, G. Jeong, S. Rao, P. Labatut, and J. Cai, "Accelerating transformer inference and training with 2:4 activation sparsity," *arXiv preprint arXiv:2503.16672*, 2025.

[36] Z.-G. Liu, P. N. Whatmough, Y. Zhu, and M. Mattina, "S2TA: Exploiting structured sparsity for energy-efficient mobile CNN acceleration," in *IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, Apr. 2022, pp. 573–586.

[37] Z.-G. Liu, P. N. Whatmough, and M. Mattina, "Systolic tensor array: An efficient structured-sparse gemm accelerator for mobile CNN inference," *IEEE Computer Architecture Letters*, vol. 19, no. 1, pp. 34–37, 2020.

[38] G. Jeong, S. Damani, A. R. Bambhaniya, E. Qin, C. J. Hughes, S. Subramoney, H. Kim, and T. Krishna, "VEGETA: Vertically-integrated extensions for sparse/dense gemm tile acceleration on cpus," in *IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, Feb. 2023, pp. 259–272.

[39] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally, "Scnn: An accelerator for compressed-sparse convolutional neural networks," *ACM SIGARCH computer architecture news*, vol. 45, no. 2, pp. 27–40, 2017.

[40] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *International Symposium on Computer Architecture (ISCA)*, 2017, p. 1–12.

[41] T. Moreau, T. Chen, and L. Ceze, "Leveraging the vta-tvm hardware-software stack for fpga acceleration of 8-bit resnet-18 inference," in *Proceedings of the 1st on Reproducible Quality-Efficient Systems Tournament on Co-Designing Pareto-Efficient Deep Learning*. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3229762.3229766

[42] H. T. Kung, "Why systolic architectures?" *Computer*, vol. 15, no. 1, pp. 37–46, 1982.

[43] A. Samajdar, J. M. Joseph, Y. Zhu, P. Whatmough, M. Mattina, and T. Krishna, "A systematic methodology for characterizing scalability of DNN accelerators using scale-sim," in *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2020, pp. 58–68.

[44] X. Wei, C. H. Yu, P. Zhang, Y. Chen, Y. Wang, H. Hu, Y. Liang, and J. Cong, "Automated systolic array architecture synthesis for high throughput CNN inference on FPGAs," in *Design Automation Conference (DAC)*, 2017.

[45] B. Asgari, R. Hadidi, and H. Kim, "Meissa: Multiplying matrices efficiently in a scalable systolic architecture," in *IEEE International Conference on Computer Design (ICCD)*, 2020, pp. 130–137.

[46] I. Ullah, K. Inayat, J.-S. Yang, and J. Chung, "Factored radix-8 systolic array for tensor processing," in *Design Automation Conference (DAC)*, Jul 2020.

[47] R. Xu, S. Ma, Y. Wang, X. Chen, and Y. Guo, "Configurable multi-directional systolic array architecture for cnns," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 18, no. 4, July 2021.

[48] J. Lee, J. Choi, J. Kim, J. Lee, and Y. Kim, "Dataflow mirroring: Architectural support for highly efficient fine-grained spatial multitasking on systolic-array npus," in *Design Automation Conference (DAC)*, 2021, pp. 247–252.

[49] C. Peltekis, D. Filippas, G. Dimitrakopoulos, C. Nicopoulos, and D. Pnevmatikatos, "ArrayFlex: A Systolic Array Architecture with Configurable Transparent Pipelining," in *Design Automation and Test in Europe (DATE)*, Apr 2023.

[50] R. Hadidi, J. Cao, M. S. Ryoo, and H. Kim, “Toward collaborative inferencing of deep neural networks on internet-of-things devices,” *IEEE Internet of Things Journal*, vol. 7, no. 6, pp. 4950–4960, 2020.

[51] S. I. Venieris, I. Panopoulos, I. Leontiadis, and I. S. Venieris, “How to reach real-time ai on consumer devices? solutions for programmable and custom architectures,” in *IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, July 2021, pp. 93–100.

[52] F. Silfa, J. M. Arnau, and A. González, “E-BATCH: Energy-efficient and high-throughput RNN batching,” *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 19, no. 1, 2022.

[53] H. Shimada, H. Ando, and T. Shimada, “Pipeline stage unification: a low-energy consumption technique for future mobile processors,” in *International Symposium on Low power electr. and design (ISLPED)*, 2003, pp. 326–329.

[54] J. H. Choi, B. G. Kim, A. Dasgupta, and K. Roy, “Improved clock-gating control scheme for transparent pipeline,” in *Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2010, pp. 401–406.

[55] T. Risset, “A method to synthesize modular systolic arrays with local broadcast facility,” in *IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, 1992, pp. 415–428.

[56] Z. Liu, H. Mao, C.-Y. Wu, C. Feichtenhofer, T. Darrell, and S. Xie, “A ConvNet for the 2020s,” in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2022, pp. 11 976–11 986.

[57] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, “Mobilenets: Efficient convolutional neural networks for mobile vision applications,” *arXiv:1704.04861*, 2017.

[58] S. Galal and M. Horowitz, “Latency sensitive fma design,” in *IEEE Symposium on Computer Arithmetic (ARITH)*, 2011, pp. 129–138.

[59] D. Filippas, C. Peltekis, G. Dimitrakopoulos, and C. Nicopoulos, “Reduced-Precision Floating-Point Arithmetic in Systolic Arrays with Skewed Pipelines,” in *IEEE International Conference on Artificial Intelligence Circuits and Systems (AICAS)*, Jun 2023.

[60] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, “ImageNet Large Scale Visual Recognition Challenge,” *International Journal of Computer Vision (IJCV)*, vol. 115, no. 3, pp. 211–252, 2015.

[61] C. Peltekis, D. Filippas, G. Dimitrakopoulos, and C. Nicopoulos, “Low-power data streaming in systolic arrays with bus-invert coding and zero-value clock gating,” in *International Conference on Modern Circuits and Systems Technologies (MOCAST)*, 2023.

[62] M. Andersch *et al.* (2022) NVIDIA Hopper architecture. [Online]. Available: https://developer.nvidia.com/blog/nvidia-hopper-architecture-in-depth

[63] S. Wimer and I. Koren, “Design flow for flip-flop grouping in data-driven clock gating,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 22, no. 4, pp. 771–778, 2014.

[64] M. R. Stan and W. P. Burleson, “Bus-invert coding for low-power i/o,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 3, no. 1, pp. 49–58, 1995.

[65] Y. Shin, S.-I. Chae, and K. Choi, “Partial bus-invert coding for power optimization of application-specific systems,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 9, no. 2, pp. 377–383, 2001.

[66] L. Benini, A. Macii, E. Macii, M. Poncino, and R. Scarsi, “Synthesis of low-overhead interfaces for power-efficient communication over wide buses,” in *Design Automation Conference (DAC)*, 1999, pp. 128–133.

[67] A. Acquaviva and R. Scarsi, “A spatially-adaptive bus interface for low-switching communication,” in *International Symposium on Low Power Electronics and Design (ISLPED)*, 2000, pp. 238–240.

[68] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger, “Densely connected convolutional networks,” in *IEEE Conference on Computer Vision and Pattern Recognition(CVPR)*, 2017, pp. 4700–4708.

[69] E. Musoll and J. Cortadella, “High-level synthesis techniques for reducing the activity of functional units,” in *International Symp on Low Power Design(ISLPED)*, 1995, pp. 99–104.

[70] K. Masselos, S. Theoharis, P. Merakos, T. Stouraitis, and C. E. Goutis, “Low power synthesis of sum-of-products computation,” in *International Symp on Low Power Electronics and Design(ISLPED)*, 2000, pp. 234–237.

[71] R. Henning and C. Chakrabarti, “An approach to switching activity consideration during high-level, low-power design space exploration,” *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, vol. 49, no. 5, pp. 339–351, 2002.

[72] B. Feinberg, B. C. Heyman, D. Mikhailenko, R. Wong, A. C. Ho, and E. Ipek, “Commutative data reordering: a new technique to reduce data movement energy on sparse inference workloads,” in *ACM/IEEE International Symposium on Computer Architecture (ISCA)*, May 2020, pp. 1076–1088.

[73] S. Mittal and S. Nag, “A survey of encoding techniques for reducing data-movement energy,” *Journal of Systems Architecture*, vol. 97, pp. 373–396, 2019.

[74] W. Fornaciari, M. Polentarutti, D. Sciuto, and C. Silvano, “Power optimization of system-level address buses based on software profiling,” in *International workshop on Hardware/software codesign*, May 2000, pp. 29–33.

[75] D. Lee, M. O'Connor, and N. Chatterjee, "Reducing data transfer energy by exploiting similarity within a data transaction," in *International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2018, pp. 40–51.

[76] L. Benini, G. De Micheli, E. Macii, D. Sciuto, and C. Silvano, "Asymptotic zero-transition activity encoding for address busses in low-power microprocessor-based systems," in *Proceedings Great Lakes Symposium on VLSI*, 1997, pp. 77–82.

[77] ——, "Address bus encoding techniques for system-level power optimization," in *Design, Automation and Test in Europe (DATE)*, 1998, pp. 861–866.

[78] J. Yang, R. Gupta, and C. Zhang, "Frequent value encoding for low power data buses," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 9, no. 3, pp. 354–384, 2004.

[79] C. K. Jha, S. Singh, R. Thakker, M. Awasthi, and J. Mekie, "Zero aware configurable data encoding by skipping transfer for error resilient applications," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 68, no. 8, pp. 3337–3350, 2021.

[80] R. Xu, S. Ma, Y. Wang, Y. Guo, D. Li, and Y. Qiao, "Heterogeneous systolic array architecture for compact cnns hardware accelerators," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 11, pp. 2860–2871, 2021.

[81] H. Genc, S. Kim, A. Amid, A. Haj-Ali, V. Iyer, P. Prakash, J. Zhao, D. Grubb, H. Liew, H. Mao *et al.*, "Gemmini: Enabling systematic deep-learning architecture evaluation via full-stack integration," in *Design Automation Conference (DAC)*, 2021, pp. 769–774.

[82] T. Hoefler, D. Alistarh, T. Ben-Nun, N. Dryden, and A. Peste, "Sparsity in deep learning: Pruning and growth for efficient inference and training in neural networks," *The Journal of Machine Learning Research*, vol. 22, no. 1, pp. 10 882–11 005, 2021.

[83] M. Zhu, T. Zhang, Z. Gu, and Y. Xie, "Sparse tensor core: Algorithm and hardware co-design for vector-wise sparse neural networks on modern gpus," in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Oct. 2019, pp. 359–371.

[84] S. Muralidharan, "Uniform sparsity in deep neural networks," *Proceedings of Machine Learning and Systems*, vol. 5, 2023.

[85] M. Soltaniyeh, R. P. Martin, and S. Nagarakatte, "An accelerator for sparse convolutional neural networks leveraging systolic general matrix-matrix multiplication," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 19, no. 3, pp. 1–26, 2022.

[86] O. Hsu, M. Strange, R. Sharma, J. Won, K. Olukotun, J. S. Emer, M. A. Horowitz, and F. Kjølstad, "The sparse abstract machine," in *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2023, pp. 710–726.

[87] G. Zhang, N. Attaluri, J. S. Emer, and D. Sanchez, “Gamma: Leveraging gustavson’s algorithm to accelerate sparse matrix multiplication,” in *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2021, pp. 687–701.

[88] G. Gerogiannis, S. Yesil, D. Lenadora, D. Cao, C. Mendis, and J. Torrellas, “Spade: A flexible and scalable accelerator for spmm and sddmm,” in *International Symposium on Computer Architecture (ISCA)*, 2023.

[89] S. Feng, X. He, K.-Y. Chen, L. Ke, X. Zhang, D. Blaauw, T. Mudge, and R. Dreslinski, “MeNDA: a near-memory multi-way merge solution for sparse transposition and dataflows,” in *International Symposium on Computer Architecture (ISCA)*, 2022, pp. 245–258.

[90] Z. Li, Z. Wang, L. Xu, Q. Dong, B. Liu, C.-I. Su, W.-T. Chu, G. Tsou, Y.-D. Chih, T.-Y. J. Chang *et al.*, “RRAM-DNN: An RRAM and model-compression empowered all-weights-on-chip DNN accelerator,” *IEEE Journal of Solid-State Circuits*, vol. 56, no. 4, pp. 1105–1115, 2020.

[91] N. Srivastava, H. Jin, J. Liu, D. Albonesi, and Z. Zhang, “Matraptor: A sparse-sparse matrix multiplication accelerator based on row-wise product,” in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Oct. 2020, pp. 766–780.

[92] A. J. Abdelmaksoud, S. Agwa, and T. Prodromakis, “Dip: A scalable, energy-efficient systolic array for matrix multiplication acceleration,” *IEEE Transactions on Circuits and Systems I: Regular Papers*, 2025.

[93] T. Ajayi, V. A. Chhabria, M. Fogaça, S. Hashemi, A. Hosny, A. B. Kahng, M. Kim, J. Lee, U. Mallappa, M. Neseem *et al.*, “Toward an open-source digital flow: First learnings from the openroad project,” in *Design Automation Conference (DAC)*, 2019, pp. 1–4.

[94] Z. Liu, Y. Lin, Y. Cao, H. Hu, Y. Wei, Z. Zhang, S. Lin, and B. Guo, “Swin transformer: Hierarchical vision transformer using shifted windows,” in *IEEE International Conference on Computer Vision*, 2021, pp. 10 012–10 022.

[95] H. Touvron, M. Cord, M. Douze, F. Massa, A. Sablayrolles, and H. Jégou, “Training data-efficient image transformers & distillation through attention,” in *International Conference on Machine Learning*, 2021, pp. 10 347–10 357.

[96] R. Salay, R. Queiroz, and K. Czarnecki, “An analysis of ISO 26262: Using machine learning safely in automotive software,” 2017.

[97] L. Ma and S. Tian, “A hybrid cnn-lstm model for aircraft 4d trajectory prediction,” *IEEE access*, vol. 8, pp. 134 668–134 680, 2020.

[98] M. Safarpour, R. Inanlou, and O. Silvén, “Algorithm level error detection in low voltage systolic array,” *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 69, no. 2, pp. 569–573, 2021.

[99] F. Libano, P. Rech, and J. Brunhaver, "Efficient error detection for matrix multiplication with systolic arrays on fpgas," *IEEE Transactions on Computers*, 2023.

[100] S. Bal, C. S. Mummidi, V. Da Cruz Ferreira, S. Srinivasan, and S. Kundu, "A novel fault-tolerant architecture for tiled matrix multiplication," in *Design, Automation & Test in Europe (DATE)*, 2023.

[101] K.-H. Huang and J. A. Abraham, "Algorithm-based fault tolerance for matrix operations," *IEEE Transactions on Computers*, vol. C-33, no. 6, pp. 518–528, 1984.

[102] J. A. Abraham, P. Banerjee, C.-Y. Chen, W. K. Fuchs, S.-Y. Kuo, and A. N. Reddy, "Fault tolerance techniques for systolic arrays," *Computer*, vol. 20, no. 7, pp. 65–75, 1987.

[103] P. Wu, Q. Guan, N. DeBardeleben, S. Blanchard, D. Tao, X. Liang, J. Chen, and Z. Chen, "Towards practical algorithm based fault tolerance in dense linear algebra," in *Proceedings of the ACM International Symposiumon High-Performance Parallel and Distributed Computing*, 2016, p. 31–42.

[104] D. Filippas, N. Margomenos, N. Mitianoudis, C. Nicopoulos, and G. Dimitrakopoulos, "Low-cost online convolution checksum checker," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 30, no. 2, pp. 201–212, 2022.

[105] R. L. R. Junior and P. Rech, "Reliability of google's tensor processing units for convolutional neural networks," in *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2022, pp. 25–27.

[106] S. Kundu, S. Banerjee, A. Raha, S. Natarajan, and K. Basu, "Toward functional safety of systolic array-based deep learning hardware accelerators," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 29, no. 3, pp. 485–498, 2021.

[107] K. T. Chitty-Venkata and A. Somani, "Impact of structural faults on neural network performance," in *IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, 2019.

[108] Y. Zhao, K. Wang, and A. Louri, "FSA: An efficient fault-tolerant systolic array-based dnn accelerator architecture," in *IEEE International Conference on Computer Design (ICCD)*, 2022, pp. 545–552.

[109] K. T. Chitty-Venkata and A. K. Somani, "Model compression on faulty array-based neural network accelerator," in *IEEE Pacific Rim International Symposium on Dependable Computing (PRDC)*, 2020, pp. 90–99.

[110] J. J. Zhang, K. Basu, and S. Garg, "Fault-tolerant systolic array based accelerators for deep neural network execution," *IEEE Design & Test*, vol. 36, no. 5, pp. 44–53, 2019.

[111] H. Lee, J. Kim, J. Park, and S. Kang, "STRAIT: Self-test and self-recovery for AI accelerator," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2023.

[112] E. Vacca, G. Ajmone, and L. Sterpone, “RunSAFER: A novel runtime fault detection approach for systolic array accelerators,” in *IEEE International Conference on Computer Design (ICCD)*, 2023.

[113] R. Baumann, “Radiation-induced soft errors in advanced semiconductor technologies,” *IEEE Transactions on Device and Materials Reliability*, vol. 5, no. 3, pp. 305–316, 2005.

[114] S. Borkar, “Designing reliable systems from unreliable components: the challenges of transistor variability and degradation,” *IEEE Micro*, vol. 25, no. 6, pp. 10–16, 2005.

[115] I. Koren and C. Krishna, *Fault-Tolerant Systems*. Morgan Kaufmann, 2020.

[116] S.-J. Wang and N. Jha, “Algorithm-based fault tolerance for FFT networks,” *IEEE Transactions on Computers*, vol. 43, no. 7, pp. 849–854, 1994.

[117] C. Peltekis, D. Filippas, and G. Dimitrakopoulos, “Error checking for sparse systolic tensor arrays,” in *IEEE International Conference on Artificial Intelligence Circuits and Systems (AICAS)*, 2024.

[118] K. Simonyan and A. Zisserman, “Very deep convolutional networks for large-scale image recognition,” in *International Conference on Learning Representations*, 2015.

[119] H. K. Lee and D. S. Ha, “HOPE: An efficient parallel fault simulator for synchronous sequential circuits,” *IEEE Transactions on Computer-Aided Design of Integr. Circ.s and Syst.*, vol. 15, no. 9, pp. 1048–1058, 1996.

[120] U. S. Solangi, M. Ibtesam, M. A. Ansari, J. Kim, and S. Park, “Test architecture for systolic array of edge-based ai accelerator,” *IEEE Access*, vol. 9, pp. 96 700–96 710, 2021.

[121] J. J. Zhang, T. Gu, K. Basu, and S. Garg, “Analyzing and mitigating the impact of permanent faults on a systolic array based neural network accelerator,” in *IEEE VLSI Test Symposium (VTS)*, 2018, pp. 1–6.

[122] M. Sadi and U. Guin, “Test and yield loss reduction of ai and deep learning accelerators,” *IEEE Transactions on Computer-Aided Design of Integr. Circ. and Syst.*, vol. 41, no. 1, pp. 104–115, 2021.

[123] S. Lee, J. Park, S. Park, H. Kim, and S. Kang, “A new zero-overhead test method for low-power ai accelerators,” *IEEE Transactions on Circ. and Syst. II*, pp. 1–5, 2023.

[124] K. Ma, C. Amarnath, and A. Chatterjee, “Error resilient transformers: A novel soft error vulnerability guided approach to error checking and suppression,” in *IEEE European Test Symposium (ETS)*, 2023, pp. 1–6.

[125] U. K. Agarwal, A. Chan, A. Asgari, and K. Pattabiraman, “Towards reliability assessment of systolic arrays against stuck-at faults,” in *IEEE/IFIP International Conference on Dependable Syst. and Networks (DSN-S)*, 2023, pp. 230–236.

[126] R. Garg, E. Qin, F. Munoz Martinez, R. Guirado, A. Jain, S. Abadal, J. L. Abellán, M. Acacio, E. Alarcón, S. Rajamanickam, and T. Krishna, "A taxonomy for classification and comparison of dataflows for gnn accelerators," in *IEEE International Parallel and Distributed Processing Symposium*, May 2022.

[127] S. Abadal, A. Jain, R. Guirado, J. López-Alonso, and E. Alarcón, "Computing graph neural networks: A survey from algorithms to accelerators," *ACM Comput. Surv.*, vol. 54, no. 9, oct 2021.

[128] J. Li, A. Louri, A. Karanth, and R. Bunescu, "GCNAX: A flexible and energy-efficient accelerator for graph convolutional neural networks," in *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2021, pp. 775–788.

[129] F. Su, C. Liu, and H.-G. Stratigopoulos, "Testability and dependability of ai hardware: Survey, trends, challenges, and perspectives," *IEEE Design & Test*, vol. 40, no. 2, pp. 8–58, 2023.

[130] P. Sen, G. Namata, M. Bilgic, L. Getoor, B. Galligher, and T. Eliassi-Rad, "Collective classification in network data," *AI Magazine*, vol. 29, no. 3, Sep. 2008.

[131] M. Yan, L. Deng, X. Hu, L. Liang, Y. Feng, X. Ye, Z. Zhang, D. Fan, and Y. Xie, "HyGCN: A GCN accelerator with hybrid architecture," in *IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, Feb. 2020, pp. 15–29.

[132] T. Geng, A. Li, R. Shi, C. Wu, T. Wang, Y. Li, P. Haghi, A. Tumeo, S. Che, S. Reinhardt, and M. C. Herbordt, "AWB-GCN: A graph convolutional network accelerator with runtime workload rebalancing," in *IEEE International Symposium on Microarchitecture*, 2020, pp. 922–936.

[133] H. Zeng and V. K. Prasanna, "Graphact: Accelerating GCN training on CPU-FPGA heterogeneous platforms," *CoRR*, vol. abs/2001.02498, 2020. [Online]. Available: http://arxiv.org/abs/2001.02498

[134] J. R. Stevens, D. Das, S. Avancha, B. Kaul, and A. Raghunathan, "GNNerator: A hardware/software framework for accelerating graph neural networks," in *Design Automation Conference (DAC)*, 2021, pp. 955–960.

[135] B. Zhang, H. Zeng, and V. Prasanna, "Hardware acceleration of large scale GCN inference," in *IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, 2020, pp. 61–68.

[136] S. Liang, Y. Wang, C. Liu, L. He, H. Li, D. Xu, and X. Li, "EnGN: A high-throughput and energy-efficient accelerator for large graph neural networks," *IEEE Transactions on Computer*, vol. 70, no. 9, pp. 1511–1525, 2021.

[137] A. Auten, M. Tomei, and R. Kumar, "Hardware acceleration of graph neural networks," *Design Automation Conference (DAC)*, 2020.

[138] B. Zhang, R. Kannan, and V. Prasanna, "BoostGCN: A framework for optimizing GCN inference on FPGA," in *IEEE International Symposium on Field-Programmable Custom Computer Machines*, 2021, pp. 29–39.

[139] C. Chen, K. Li, X. Zou, and Y. Li, "Dygnn: Algorithm and architecture support of dynamic pruning for graph neural networks," in *Design Automation Conference (DAC)*, 2021, pp. 1201–1206.

[140] Z. Shao, C. Xie, Z. Ning, Q. Wu, L. Chang, Y. Du, and L. Du, "An efficient GCN accelerator based on workload reorganization and feature reduction," *IEEE Transactions on Circuits and Systems I*, vol. 71, no. 2, pp. 646–659, 2024.

[141] M. Besta and T. Hoefler, "Parallel and Distributed Graph Neural Networks: An In-Depth Concurrency Analysis," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 46, no. 5, pp. 2584–2606, 2024.

[142] J. Calder, "Graphlearning python package," 2022. [Online]. Available: https://jwcalder.github.io/GraphLearning/

[143] B. Sangchoolie, K. Pattabiraman, and J. Karlsson, "One bit is (not) enough: An empirical study of the impact of single and multiple bit-flip errors," in *IEEE International Conference on Depend. Syst. and Networks (DSN)*, 2017.

[144] J. Kosaian and K. Rashmi, "Arithmetic-intensity-guided fault tolerance for neural network inference on gpus," in *International Conference for High Performance Computing, Networking, Storage and Analysis*, 2021.

[145] M. Horiguchi and K. Itoh, *Nanoscale Memory Repair*. Springer, 2011.

[146] C. Peltekis, D. Filippas, C. Nicopoulos, and G. Dimitrakopoulos, "FusedGCN: A systolic three-matrix multiplication architecture for graph convolutional networks," in *IEEE International Conference on Application-specific Systems, Architecture and Processors (ASAP)*, 2022.

[147] C. Braun, S. Halder, and H. J. Wunderlich, "A-ABFT: Autonomous algorithm-based fault tolerance for matrix multiplications on graphics processing units," in *IEEE International Conference on Depend. Syst. and Networks (DSN)*, 2014.

[148] J. R. Stevens, R. Venkatesan, S. Dai, B. Khailany, and A. Raghunathan, "Softermax: Hardware/software co-design of an efficient softmax for transformers," in *Design Automation Conference (DAC)*, 2021, pp. 469–474.

[149] B. Yuan, "Efficient hardware architecture of softmax layer in deep neural network," in *IEEE International System-on-Chip Conference (SOCC)*, 2016.

[150] Z. Wei, A. Arora, P. Patel, and L. John, "Design space exploration for softmax implementations," in *IEEE, International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, 2020.

[151] T. Tambe, C. Hooper, L. Pentecost, T. Jia, E.-Y. Yang, M. Donato, V. Sanh, P. Whatmough, A. M. Rush, D. Brooks, and G.-Y. Wei, "EdgeBERT: Sentence-level energy optimizations for latency-aware multi-task nlp inference," in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2021, p. 830–844.

[152] Y. Zhang, Y. Zhang, L. Peng, L. Quan, S. Zheng, Z. Lu, and H. Chen, "Base-2 softmax function: Suitability for training and efficient hardware implementation," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 69, no. 9, pp. 3605–3618, 2022.

[153] I. Kouretas and V. Paliouras, "Hardware implementation of a softmax-like function for deep learning," *Technologies*, vol. 8, p. 46, 2020.

[154] N. A. Koca, A. T. Do, and C.-H. Chang, "Hardware-efficient softmax approximation for self-attention networks," in *IEEE International Symposium on Circuits and Systems (ISCAS)*, 2023.

[155] H. C. Prashanth and M. Rao, "Somalib: Library of exact and approximate activation functions for hardware-efficient neural network accelerators," in *IEEE International Conference on Computer Design (ICCD)*, 2022, pp. 746–753.

[156] J. Kim, J. Lee, J. Choi, J. Han, and S. Lee, "Range-invariant approximation of non-linear operations for efficient bert fine-tuning," in *Design Automation Conference (DAC)*, 2023.

[157] A. Williams, N. Nangia, and S. R. Bowman, "A broad-coverage challenge corpus for sentence understanding through inference," in *International Conference of the North American Chapter of the Association for Computational Linguistics*, 2018.

[158] A. Warstadt, A. Singh, and S. R. Bowman, "Neural network acceptability judgments," *Transactions of the Association for Computational Linguistics*, vol. 7, pp. 625–641, 2019.

[159] R. Socher, A. Perelygin, J. Wu, J. Chuang, C. D. Manning, A. Y. Ng, and C. Potts, "Recursive deep models for semantic compositionality over a sentiment treebank," in *International Conference on Empirical Methods in Natural Language Processing*, 2013, pp. 1631–1642.

[160] B. Dolan and C. Brockett, "Automatically constructing a corpus of sentential paraphrases," in *International Workshop on Paraphrasing (IWP2005)*, 2005.

[161] L. Bentivogli, P. Clark, I. Dagan, and D. Giampiccolo, "The fifth pascal recognizing textual entailment challenge." in *Text Analysis Conference (TAC)*, 2009.

[162] D. Hendrycks and K. Gimpel, "Gaussian error linear units (gelus)," *arXiv preprint arXiv:1606.08415*, 2016.

[163] M. Milakov and N. Gimelshein, "Online normalizer calculation for softmax," *arXiv preprint arXiv:1805.02867*, 2018.

[164] G. Islamoglu, M. Scherer, G. Paulin, T. Fischer, V. J. Jung, A. Garofalo, and L. Benini, "Ita: An energy-efficient attention and softmax accelerator for quantized transformers," in *IEEE International Symposium on Low Power Electronics and Design (ISLPED)*, 2023.

[165] D. Zhu, S. Lu, M. Wang, J. Lin, and Z. Wang, “Efficient precision-adjustable architecture for softmax function in deep learning,” *Transactions on Circuits and Systems II: Express Briefs*, vol. 67, pp. 3382–3386, 2020.

[166] C. F. Jekel and G. Venter. (2019) PWLF: a python library for fitting 1d continuous piecewise linear functions. [Online]. Available: https://github.com/cjekel/piecewise_linear_fit_py

[167] H. Kim, B.-G. Nam, J.-H. Sohn, J.-H. Woo, and H.-J. Yoo, “A 231-mhz, 2.18-mw 32-bit logarithmic arithmetic unit for fixed-point 3-d graphics system,” *IEEE journal of solid-state circuits*, vol. 41, no. 11, pp. 2373–2381, 2006.

[168] A. Wang, A. Singh, J. Michael, F. Hill, O. Levy, and S. R. Bowman, “GLUE: A multi-task benchmark and analysis platform for natural language understanding,” in *International Conference on Learning Representations (ICLR)*, 2019.

[169] C. Peltekis, K. Alexandridis, and G. Dimitrakopoulos. (2024) SIMD-Softmax. [Online]. Available: https://github.com/ic-lab-duth/SIMD-Softmax.git