ΔΗΜΟΚΡΙΤΕΙΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΡΑΚΗΣ
ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ
ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Αυτοματοποιημένη Σχεδίαση Επιταχυντών Υλικού για Εφαρμογές Μάθησης Μηχανής

ΔΙΔΑΚΤΟΡΙΚΗ ΔΙΑΤΡΙΒΗ

Διονύσιος Φίλιππας, 70394:

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

Αυτοματοποιημένη Σχεδίαση Επιταχυντών Υλικού για Εφαρμογές Μάθησης Μηχανής

ΔΙΔΑΚΤΟΡΙΚΗ ΔΙΑΤΡΙΒΗ

Διονύσιος Φίλιππας, 70394:

ΕΞΕΤΑΣΤΙΚΗ ΕΠΙΤΡΟΠΗ

Μέλη της Συμβουλευτικής Επιτροπής:

Επιβλέπων Καθηγητής: Γεώργιος, Δημητρακόπουλος, Καθηγητής, Τμήμα Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών, Δημοκρίτειο Πανεπιστήμιο Θράκης

Μέλος 2: Γεώργιος, Συρακούλης, Καθηγητής, Τμήμα Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών, Δημοκρίτειο Πανεπιστήμιο Θράκης

Μέλος 3: Χρυσόστομος, Νικόπουλος, Καθηγητής, Τμήμα Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών, Πανεπιστημίου Κύπρου

Μέλη της Εξεταστικής Επιτροπής:

Μέλος 4: Ιωάννης, Καραφυλλίδης, Καθηγητής, Τμήμα Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών, Δημοκρίτειο Πανεπιστήμιο Θράκης

Μέλος 5: Διονύσιος, Πνευματικάτος, Καθηγητής, Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών, Εθνικό Μετσόβιο Πολυτεχνείο

Μέλος 6: Ιωάννης, Βούρκας, Αν. Καθηγητής, Τμήμα Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών, Δημοκρίτειο Πανεπιστήμιο Θράκης

Μέλος 7: Λάζαρος, Παπαδόπουλος, Επ. Καθηγητής, Τμήμα Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών, Δημοκρίτειο Πανεπιστήμιο Θράκης

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

Automated Design of Machine Learning Hardware Accelerators

DOCTORAL THESIS

Dionysios Filippas, 70394:

COMMITTEE OF EXAMINERS

Members of the Advisory Committee:

Supervisor: Giorgos, Dimitrakopoulos, Professor, Department of Electrical and Computer Engineering, Democritus University of Thrace

Member 2: Georgios, Sirakoulis, Professor, Department of Electrical and Computer Engineering, Democritus University of Thrace

Member 3: Chrysostomos, Nicopoulos, Assoc. Professor, Department of Electrical and Computer Engineering, University of Cyprus

Members of the Committee of Examiners:

Member 4: Ioannis, Karafyllidis, Professor, Department of Electrical and Computer Engineering, Democritus University of Thrace

Member 5: Dionisios, Pnevmatikatos, Professor, School of Electrical and Computer Engineering, National Technical University of Athens

Member 6: Ioannis, Vourkas, Assoc. Professor, Department of Electrical and Computer Engineering, Democritus University of Thrace

Member 7: Lazaros, Papadopoulos, Assist. Professort, Department of Electrical and Computer Engineering, Democritus University of Thrace

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

ΑΝΑΦΟΡΑ ΣΤΗΝ ΤΗΡΗΣΗ ΤΩΝ ΑΚΑΔΗΜΑΪΚΩΝ ΑΡΧΩΝ ΔΕΟΝΤΟΛΟΓΙΑΣ

Η παρούσα εργασία με τίτλο «Αυτοματοποιημένη Σχεδίαση Επιταχυντών Υλικού για Εφαρμογές Μάθησης Μηχανής» είναι πρωτότυπη και πραγματοποιήθηκε από τον φοιτητή του Τμήματος Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών με Αρ. Μητρώου ΑΕΜ 70394 στο Εργαστήριο Ολοκληρωμένων Κυκλωμάτων του τομέα Ηλεκτρονικής και Τεχνολογίας Συστημάτων Πληροφορικής, υπό την επίβλεψη του καθηγητή Γεώργιου Δημητρακόπουλου. Η συγγραφή της εργασίας/διατριβής πραγματοποιήθηκε εξολοκλήρου από τον φοιτητή Διονύσιο Φίλιππα, υπό την καθοδήγηση και τις υποδείξεις του επιβλέποντά του.

Βεβαιώνεται ότι, κατά την εκπόνηση και τη συγγραφή της εργασίας/διατριβής του, ο φοιτητής τήρησε τα προβλεπόμενα από τον νόμο και τον αντίστοιχο εσωτερικό κανονισμό του Τμήματος, σεβάστηκε πλήρως τις Αρχές της Ακαδημαϊκής Ηθικής και του Κώδικα Δεοντολογίας, οι οποίες απαγορεύουν την παραποίηση των ερευνητικών/πειραματικών αποτελεσμάτων, την αναφορά ψευδών στοιχείων, την κατάχρηση της διανοητικής ιδιοκτησίας τρίτων και τη λογοκλοπή και ότι έγινε με σεβασμό στις αρχές.

Περίληψη

Η παρούσα διατριβή παρουσιάζει ένα ολοκληρωμένο πλαίσιο για τον αυτοματοποιημένο σχεδιασμό επιταχυντών υλικού για Συνελικτικά Νευρωνικά Δίκτυα (Convolutional Neural Networks - CNNs), αξιοποιώντας τη Σύνθεση Υψηλού Επιπέδου (High-Level Synthesis) και περιγραφές μοντέλων υψηλού επιπέδου σε Python (μέσω TensorFlow ή PyTorch). Η προτεινόμενη μεθοδολογία απλοποιεί την ανάπτυξη επιταχυντών για CNNs, επιτρέποντας την άμεση παραγωγή υλικού από περιγραφές μοντέλων σε γλώσσα προγραμματισμού υψηλού επιπέδου και μειώνοντας σημαντικά την πολυπλοκότητα του σχεδιασμού και το χρόνο ανάπτυξης.

Το πλαίσιο εισάγει ένα πλούσιο σύνολο παραμέτρων βελτιστοποίησης, επιτρέποντας την εξερεύνηση διαφορετικών αρχιτεκτονικών σε σχέση με την απόδοση, την ενεργειακή αποδοτικότητα και τη χρήση πόρων. Πέρα από τις συμβατικές παραμέτρους σχεδίασης, η παρούσα εργασία ενσωματώνει, για πρώτη φορά, χαρακτηριστικά λειτουργικής ασφάλειας σε επίπεδο υλικού, προηγμένες αρχιτεκτονικές ενδιάμεσης αποθήκευσης, καθώς επίσης υποστηρίζει διάφορες αριθμητικές αναπαραστάσεις δεδομένων, πλήρους και μειωμένης ακρίβειας (reduced-precision) κινητής και σταθερής υποδιαστολής.

Παρουσιάζονται δύο εξειδικευμένες μονάδες επιτάχυνσης συνελίξεων για αρχιτεκτονικές συνεχούς ροής δεδομένων (streaming convolution engines): η μονάδα LazyDCstream, βελτιστοποιημένη για διασταλμένες συνελίξεις (dilated convolutions), και η μονάδα LeapConv, βελτιστοποιημένη για συνελίξεις με μη μοναδιαίο βήμα (strided convolutions). Η LazyDCstream ελαχιστοποιεί την κατανάλωση ισχύος μέσω της αποδοτικής επαναχρησιμοποίησης των ενδιάμεσων δομών αποθήκευσης και της «τεμπέλικης» μετακίνησης δεδομένων που παραμένει ανεξάρτητη του βαθμού διαστολής. Η LeapConv αντιμετωπίζει τις αναποτελεσματικότητες των συνελίξεων με μη μοναδιαίο βήμα αποσυνθέτοντάς τις σε κανάλια με μοναδιαίο βήμα και συγχωνεύοντάς τα σε μια ενιαία μονάδα υλικού, μειώνοντας έτσι την περιττή πρόσβαση στη μνήμη με ελάχιστο επιπλέον κόστος χώρου.

Για την υποστήριξη της λειτουργικής ασφάλειας, η εργασία εισάγει έναν μηχανισμό ελέγχου ανθεκτικό σε σφάλματα (Algorithm-Based Fault Tolerant) με χαμηλό κόστος, ειδικά σχεδιασμένο για CNNs. Ο ελεγκτής ορθότητας των συνελίξεων προβλέπει τα αθροίσματα ελέγχου των εξόδων χρησιμοποιώντας μόνο τα συνοριακά εικονοστοιχεία της εισόδου, αξιοποιώντας μια νέα συνθήκη αναλλοίωτης συνελίξεως. Αυτό μειώνει δραστικά τις απαιτήσεις σε ενδιάμεση αποθήκευση και επιπλέον λογική, επιτρέποντας τον αποδοτικό εντοπισμό σφαλμάτων σε πραγματικό χρόνο.

Το πλαίσιο επικυρώνεται μέσω της αυτόματης σύνθεσης πολλαπλών επιταχυντών για CNNs σχεδιασμένα για πραγματικές εφαρμογές. Τα πειραματικά αποτελέσματα δείχνουν σημαντικές βελτιώσεις στην ενεργειακή αποδοτικότητα, την δυνατότητα παραμετροποίησης και τη χρήση του υλικού σε σύγκριση με τα πιο σύγχρονα εργαλεία. Συνολικά, η παρούσα εργασία προσφέρει μια κλιμακούμενη και πρακτική λύση για την επιτάχυνση των CNNs σε επίπεδο υλικού, καλύπτοντας τις κρίσιμες απαιτήσεις απόδοσης, ενεργειακής κατανάλωσης και αξιοπιστίας των σύγχρονων συστημάτων μηχανικής μάθησης.

Λέξεις Κλειδιά: Επιταχυντές υλικού, Μηχανική μάθηση, Συνελικτικά νευρωνικά δίκτυα, Σύνθεση υψηλού επιπέδου, Αρχιτεκτονικές χαμηλής ισχύος, Ασφαλή συστήματα, Αριθμητική κινητής υποδιαστολής, Εργαλεία αυτοματοποιημένης σχεδίασης υλικού

Abstract

This thesis presents a comprehensive framework for the automated design of hardware accelerators targeting Convolutional Neural Networks (CNNs), leveraging High-Level Synthesis (HLS) and high-level model descriptions in Python (via TensorFlow or PyTorch). The proposed methodology streamlines the development of CNN dataflow accelerators by enabling direct hardware generation from software models, significantly reducing design complexity and development time.

The framework introduces a rich set of configurable optimization parameters, allowing exploration of different architectural designs with respect to performance, energy efficiency, and resource utilization. Beyond conventional design knobs, this work integrates, for the first time, hardware-level functional safety features, advanced buffering architectures, and support for diverse data representations including reduced-precision floating-point and fixed-point formats.

Two specialized streaming convolution engines are introduced: LazyDCstream, optimized for dilated convolutions, and LeapConv, optimized for strided convolutions. LazyDCstream minimizes power consumption through efficient buffer reuse and "lazy" data movement that remains independent of dilation rate. LeapConv addresses inefficiencies in strided convolutions by decomposing them into unity-stride channels and merging them into a single hardware unit, thereby reducing redundant memory access with minimal area overhead.

To support functional safety, the thesis introduces a low-overhead Algorithm-Based Fault Tolerant (ABFT) checker tailored for CNNs. The convolution checker predicts output checksums using only border pixels of the input, leveraging a novel convolutional invariance condition. This drastically reduces buffer and logic requirements, enabling efficient online fault detection.

The framework is validated through the automatic synthesis of multiple CNN accelerators from real-world models. Experimental results demonstrate significant improvements in energy efficiency, configurability, and hardware utilization compared to state-of-the-art tools. Overall, this thesis delivers a scalable and practical solution for CNN hardware acceleration, addressing critical demands in performance, energy, and reliability for modern machine learning systems.

Keywords: Hardware accelerators, Machine learning, Convolutional neural networks, High-level synthesis, Low-power architecture, Fault-tolerant systems, Floating-point arithmetic, Electronic design automation

Acknowledgements

First and foremost, I would like to express my deepest gratitude to my advisor, Giorgos Dimitrakopoulos, for his continuous guidance and unwavering support, not only throughout the years of my Ph.D. studies, but also prior to that. His strong work ethic and depth of knowledge have been instrumental in helping me grow both as a person and as an engineer. I am truly grateful for the opportunity to work on meaningful research under his supervision, and for the privilege of learning from his insightful advice and ideas.

I would also like to thank the members of my advisory and defensive committee, Georgios Sirakoulis, Chrysostomos Nicopoulos, Ioannis Karafyllidis, Dionisios Pnevmatikatos, Ioannis Vourkas and Lazaros Papadopoulos for evaluating and reviewing my work, and for providing insightful comments that helped to improve this thesis. Especially, I would like to thank Chrysostomos Nicopoulos for our productive collaboration and his beneficial contribution to my published work.

Many thanks also go to my labmates, Christodoulos, Vasilis, and Kosmas, with whom I shared many insightful discussions, both related to our work and beyond. I am especially grateful to Christodoulos for our excellent collaborations and the publications we produced together. In addition, I would like to thank my fellow Ph.D. friends, Kostas, Aggelos, and Manolis, whose company during quick coffee breaks or long beer evenings helped me unwind and ease the pressures of research.

Lastly, I would like to thank my parents, Epameinondas and Athanasia, as well as my sister, Popi, for their continuous support and encouragement throughout all these years. Without them I would not have been able to achieve this degree.

Contents

1 Introduction

Until the early 21st century, computing systems primarily evolved to improve the performance and efficiency of general-purpose processors. Their widespread adoption and ongoing refinement led to a vast software ecosystem, making them accessible and versatile for a broad range of applications. However, the slowing of Moore's Law, combined with growing computational demands and the need for greater energy efficiency, has driven a shift toward domain-specific hardware accelerators.

The transition to domain-specific computing has roots extending back to the late 20th century, with early implementations in areas like signal processing and graphics rendering. In fact, the idea of tailoring hardware for specific applications predates this period, with early computers often designed for dedicated tasks.

A significant milestone in this evolution was the introduction of application-specific integrated circuits (ASICs) in the 1980s. These chips are custom-designed to perform particular functions, offering substantial performance and efficiency gains over general-purpose CPUs in targeted domains. ASICs quickly found adoption in telecommunications, automotive systems, and consumer electronics.

The 1990s saw the emergence of field-programmable gate arrays (FPGAs), which introduced a new level of flexibility in custom hardware design. FPGAs enabled designers to implement and iterate on custom logic and algorithms, making them ideal for domains such as digital signal processing, networking, and embedded systems.

In the 21st century, the development and widespread use of graphics processing units (GPUs) marked another major leap. Originally created for rendering images in graphics applications, GPUs demonstrated high suitability for general-purpose parallel computing. Their architectural strengths, i.e., massive parallelism and high throughput, opened the door to new applications in scientific computing, machine learning, and cryptocurrency mining.

More recently, the rise of artificial intelligence, deep learning, and edge computing has intensified interest in highly specialized hardware. Domain-specific accelerators tailored for neural network inference and training—such as tensor processing units (TPUs) and custom ASICs, have become essential for workloads like image recognition, natural language processing, and autonomous systems. These accelerators deliver orders-of-magnitude improvements in performance and energy efficiency compared to traditional CPU-based approaches.

Overall, the evolution from general-purpose computing to domain-specific architectures reflects a broader trend: the pursuit of optimized hardware-software co-design to meet the performance, efficiency, and scalability demands of modern applications. This shift is particularly evident in machine learning, where specialized accelerators now play a pivotal role in enabling next-generation intelligent systems.

1.1 Machine Learning Models

The term "Machine Learning" (ML) refers to a subcategory of Artificial Intelligence (AI) and Computer Science that focuses on using data and algorithms to emulate the way humans learn. Currently, "deep learning" is a popular term used to describe various ML applications. Essentially, a deep learning model is an ML model characterized by its many layers, which increase its depth [43].

Although we often view ML as a modern and thriving field, it has actually been evolving over a long period. In the late 1950s, Frank Rosenblatt introduced the Perceptron [101], the first form of neural network, used in pattern recognition and classification tasks. However, it was not until the development of backpropagation [102], that ML began to attract the attention of researchers. Backpropagation enabled the training of models with more layers, leading to the development of MultiLayer Perceptrons (MLPs), which could tackle more complex problems.

Since the early 2000s, advancements in hardware that facilitated better training, combined with the development of Convolutional Neural Networks (CNNs), have revolutionized computer vision tasks. These improvements significantly enhanced model accuracy, leading to the widespread adoption of CNNs in applications such as facial recognition, object detection, and autonomous driving. In addition, the introduction of Recurrent Neural Networks (RNNs) and Long Short-Term Memory networks (LSTMs) has shown remarkable performance in Natural Language Processing (NLP) tasks, including language translation, sentiment analysis, and speech recognition.

However, the biggest breakthrough in ML came with the development of Large Language Models (LLMs), which revolutionized the field of NLP. LLMs are transformer-based models, which leverage massive amounts of data and computational power to achieve human-level performance across a wide range of natural language understanding tasks. Their introduction has enabled machines to generate coherent and contextually relevant text, perform complex language-understanding tasks, and even engage in conversations with humans.

1.1.1 Multi-Layer Perceptrons

MLPs are considered as the first form of AI models and they still serve as fundamental blocks in modern deep neural networks. Their architecture is based on neurons that actually follow the architecture of the perceptron. The structure of each perceptron neuron is summarized in Figure 1.1. Each neuron accepts a number of inputs that are multiplied by some weights before being accumulated. The weighted sum of the inputs passes through an activation function before the output is computed.

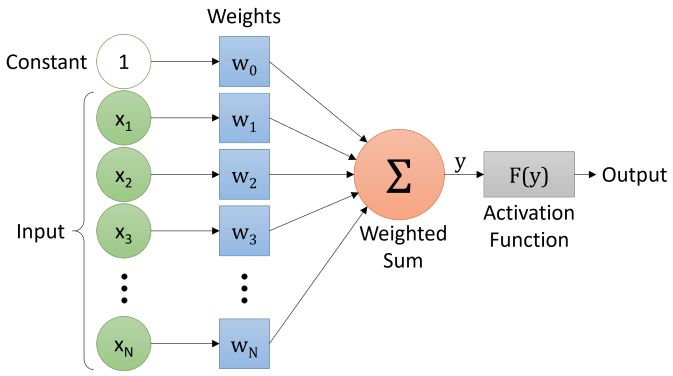


Figure 1.1: The basic structure of the perceptron model.

MLPs are composed of multiple layers built using neurons as fundamental units, as illustrated in Figure 1.2. Their architecture includes three distinct types of layers: the input layer, one or more hidden layers, and the output layer. This structure is referred to as fully connected because each node in a layer is connected to every node in both the preceding and following layers.

The first layer, the input layer, passes the input features to the initial hidden layer. Its size corresponds to the number of input features. In practice, the input layer simply links each input to every

neuron in the next layer without performing any computation. The core computational work of the MLP takes place in the hidden layers. In each neuron of a hidden layer, a weighted sum of inputs is calculated and then passed to all neurons in the subsequent layer. The number of hidden layers and the number of neurons per layer significantly influence the network's performance.

The final layer, the output layer, produces the network's result. The number of neurons in this layer depends on the specific task. For instance, in [26], the output layer contains 10 neurons to classify inputs into one of the 10 digits from 0 to 9.

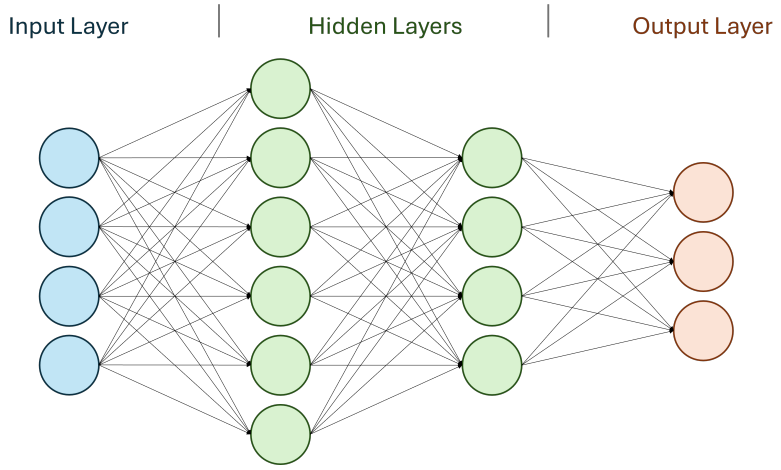


Figure 1.2: The architecture of an MLP model consists of one input layer, multiple hidden layers, and one output layer.

The classification capabilities of a Multi-Layer Perceptron (MLP) arise from both its network structure and the values of its weights. Once the MLP's architecture is defined, by choosing the number of hidden layers and the number of neurons in each layer, the weights are optimized through training.

1.1.2 Convolution Neural Networks

Convolution is the fundamental building block of CNNs, a class of deep learning architectures widely used in image recognition [48], video analysis [17], and natural language processing [141]. In essence, the convolution operation enables CNNs to automatically and efficiently extract hierarchical spatial features from input data, usually called input feature maps (IFMs). Unlike fully-connected layers that treat input features equally, convolutional layers exploit the local structure of the data by applying small filters (kernels) that slide over the input, detecting patterns such as edges, textures, or complex shapes.

A two-dimensional (2D) convolution operation between an input feature map and a filter (or kernel) can be defined as in equation 1.1, where I is the IFM matrix, K is the kernel matrix and O is the output matrix which is usually called output feature map (OFM).

$$O_{i,j} = \sum_{m=0}^{K-1} \sum_{n=0}^{L-1} I_{i+m,j+n} \cdot K_{m,n} \tag{1.1}$$

This operation performs element-wise multiplication between the overlapping regions of the IFM and kernel, followed by a summation. Each $O_{i,j}$ element of the OFM is the result of a dot product

operation. When the kernel slides over the input spatially (row-wise and column-wise), it generates an entire 2D feature map, capturing specific features depending on the learned weights in the kernel. These overlapping regions of the IFM and the kernel as well as the sliding of the kernel over the input are depicted in Figure 1.3. In this example, the 3×3 kernel overlaps with a region of the input and the dot product operation generates an output. Then the kernel slides towards the right in the same row. When the kernel reaches the final column of the row (step 3 of Figure 1.3) it moves in the row below (step 4).

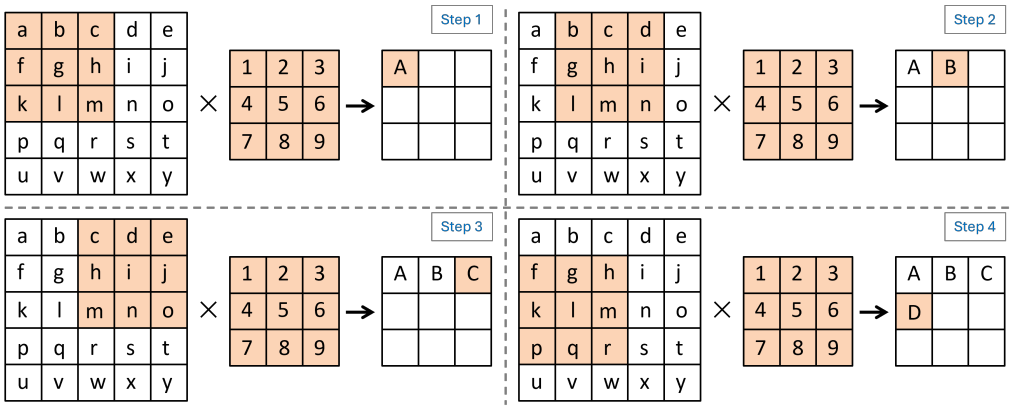


Figure 1.3: A convolution example between a 5×5 IFM and a 3×3 kernel.

Convolutions can also take non-standard forms derived by the way the kernel is structured and moves on the input.

In *strided* convolutions the kernel moves in steps larger than one, enabling spatial downsampling of feature maps. This reduces output dimensionality while retaining key features, functioning similarly to pooling but with learnable parameters. Strided convolutions also improve computational efficiency, making them valuable in deep or resource-constrained CNN architectures.

Dilated convolutions expand the receptive field of a kernel by inserting gaps between its elements, allowing it to capture broader context without increasing parameters or reducing resolution. This technique is especially useful in tasks like semantic segmentation, where modeling long-range dependencies while preserving fine spatial detail is crucial. By offering a wider view with minimal added computational cost, dilated convolutions are an efficient alternative to pooling or deeper convolutional layers.

Transposed convolutions, also called deconvolutions, are used to upsample feature maps by reversing the effects of standard convolutions through learned, spatially adaptive operations. They play a key role in generative models and decoder architectures by enabling the reconstruction of high-resolution outputs from low-resolution inputs. Unlike fixed interpolation, transposed convolutions can learn complex upsampling patterns, though they require careful design to avoid artifacts like checkerboard effects.

Apart from convolutional layers, a CNN also comprises other types of layers such as pooling layers, activation layers and fully-connected layers.

Pooling layers reduce the spatial dimensions of feature maps while retaining the most important information. Max pooling selects the maximum value in a local region, helping with translation invariance and reducing computation. Another popular version of pooling layer is the average pooling one.

Fully Connected These layers are used near the end of the network to aggregate features and

perform classification. Every neuron is connected to all outputs from the previous layer, enabling high-level reasoning. These layers have the same architecture as the MLPs that were presented earlier.

Batch Normalization This layer normalizes the inputs to a layer across a mini-batch, stabilizing and accelerating training. It helps reduce internal covariate shift and often allows the use of higher learning rates.

Dropout Dropout is a regularization technique where a fraction of neurons is randomly deactivated during training. This prevents overfitting by ensuring that the network does not become overly reliant on specific neurons.

Flatten This layer converts multidimensional feature maps into a 1D vector before passing it to fully connected layers. It's often used to transition from convolutional layers to dense layers.

Such layers are separated by appropriate activation function layers that introduce non-linearity into the system, enabling the network to learn complex patterns and approximate arbitrary functions. Without activation functions, a neural network composed of only linear operations would be equivalent to a single-layer linear model, regardless of its depth, severely limiting its representational power. Common activation functions include ReLU (Rectified Linear Unit), which is computationally efficient and helps mitigate the vanishing gradient problem; sigmoid and tanh, which are historically significant but prone to saturation; and more recent variants like Leaky ReLU, ELU, and GELU that aim to address specific training challenges. The choice of activation function significantly affects convergence speed, model performance, and gradient flow, making it a critical design decision in the architecture of deep learning systems.

1.2 Evolution of Neural Network Architectures Beyond Convolution

Following the wide adoption of CNNs in image recognition and related computer vision tasks, the machine learning landscape has undergone significant transformations, particularly with the emergence of models that prioritize dynamic context modeling over local spatial hierarchies. While CNNs excel at capturing spatial correlations using convolutional filters and pooling, they operate with fixed receptive fields and locality assumptions, which can become limiting in scenarios requiring long-range dependencies or adaptive feature interactions.

A major breakthrough came with the introduction of attention mechanisms, initially in the domain of natural language processing. The attention paradigm allows models to weigh input features differently depending on their contextual relevance, dynamically recalibrating the focus during inference. This culminated in the development of the Transformer architecture, first popularized by the *"Attention is All You Need"* paper in 2017 [125]. Transformers abandoned recurrence and convolution entirely, instead relying on multi-head self-attention to process inputs in parallel while capturing global dependencies. This design proved remarkably effective in tasks like translation, text summarization, and later, large-scale vision tasks.

Vision Transformers (ViTs) extended this success to the image domain by treating image patches as tokens, akin to words in a sentence. Despite the apparent lack of inductive biases like locality and translation invariance inherent in CNNs, ViTs demonstrated competitive performance, particularly when trained on large datasets with sufficient regularization. Subsequent refinements, such as the Swin Transformer [78] and ConvNeXt [134], incorporated hierarchical representations and hybrid architectures to combine the strengths of attention and convolution, achieving state-of-the-art results across multiple benchmarks.

Other evolutionary strands in post-CNN architectures include Graph Neural Networks (GNNs), which generalize convolution to non-Euclidean domains, and MLP-Mixers, which simplify the mod-

eling pipeline by replacing attention with multi-layer perceptrons operating over spatial and channel dimensions. These architectures reflect a broader trend in machine learning: designing models that are more flexible, data-adaptive, and conducive to massive pretraining regimes. However, these gains often come at the cost of increased computational and memory demands, especially in dense attention layers with quadratic complexity.

Despite the rise of attention-based models, CNNs remain highly relevant in practice, particularly in edge computing, embedded systems, and scenarios with limited training data. Their structured sparsity, weight sharing, and spatial inductive biases make them not only computationally efficient but also well-suited for hardware optimization. Furthermore, many vision backbones in real-world applications still employ CNNs either directly or as part of hybrid architectures. As such, CNN-specific hardware accelerators continue to be vital for enabling low-latency, energy-efficient inference in a wide array of applications, from autonomous vehicles to mobile devices and IoT platforms.

1.2.1 Using Machine Learning Models

Today, ML models have achieved remarkable performance across a broad range of tasks, leading to widespread adoption by developers, engineers, and researchers in various industries. This growing adoption has been significantly facilitated by the emergence of high-level ML libraries such as TensorFlow [1], PyTorch [92], and Open Neural Network Exchange (ONNX) [90]. These frameworks offer streamlined application programming interfaces (APIs), comprehensive documentation, and robust toolchains that abstract the complexities of model construction, training, and deployment. They empower users not only to build custom neural network architectures from scratch but also to leverage an extensive ecosystem of pre-trained models.

Using libraries like PyTorch or TensorFlow, developers can effortlessly load state-of-the-art models such as YOLOv5 [122] for real-time object detection, BERT [66] for natural language understanding, or ResNet [48] for image classification tasks. These models can be fine-tuned on domain-specific datasets, enabling rapid adaptation to new applications, such as industrial defect detection, medical imaging, or financial forecasting. Furthermore, modular architectures allow components of these models—like feature extractors or attention mechanisms—to be repurposed as building blocks in novel architectures. The ONNX format further enhances this flexibility by allowing seamless model interoperability across frameworks and platforms, facilitating cross-framework deployment on diverse hardware backends. This democratization of ML development through user-friendly libraries and reusable models has accelerated innovation and lowered the barrier to entry for real-world AI solutions.

1.3 Hardware Acceleration of Machine Learning Models

The rapid evolution of ML and AI has driven the need for specialized hardware capable of handling the growing complexity and scale of AI workloads. AI chips—dedicated processors designed specifically for accelerating AI computations—offer several key advantages over conventional, general-purpose processors, particularly in terms of energy efficiency, computational accuracy, and adaptability.

One of the most critical benefits of AI chips is their energy efficiency. Unlike traditional CPUs or GPUs, AI chips often incorporate low-precision arithmetic (such as 8-bit or mixed-precision operations), which allows them to execute computations using fewer transistors. This architectural efficiency directly translates to lower energy consumption. Furthermore, AI chips are built with a focus on parallelism, enabling them to process multiple operations simultaneously and thereby optimize workload distribution. These features are especially important in large-scale data centers,

where energy consumption and heat dissipation are major operational concerns. As the AI industry continues to grow, energy-efficient hardware could play a significant role in reducing its environmental impact, particularly its carbon footprint. On the edge computing front, AI chips enable mobile and embedded devices—such as smartphones or IoT sensors—to perform on-device inference without relying on constant communication with the cloud. This not only improves privacy and latency but also extends battery life, making such devices more sustainable and responsive.

In addition to efficiency, AI chips are designed to deliver higher computational accuracy for AI-specific tasks. These processors are optimized for operations central to deep learning, such as matrix multiplications, tensor manipulations, and non-linear activation functions. As a result, AI chips often outperform general-purpose chips in executing complex algorithms associated with image recognition, natural language processing, and real-time decision-making. Their architectural precision reduces numerical errors and enhances the reliability of results, making them well-suited for safety-critical applications like medical diagnostics, industrial automation, and autonomous driving, where high inference accuracy and low latency are essential.

Another advantage of AI hardware acceleration lies in the availability of specialized processing units designed to meet the diverse computational demands of machine learning workloads. Devices such as GPUs, TPUs, FPGAs, and ASICs each offer distinct trade-offs in terms of flexibility, performance, and energy efficiency. Among these, FPGAs and ASICs stand out for their customizability, allowing hardware to be tailored to specific AI models or applications. This customization can involve tuning internal parameters, optimizing data paths, or reconfiguring logic blocks to better support unique algorithmic requirements, thereby maximizing performance for tasks ranging from real-time video analytics to large-scale language modeling. Further performance gains are achieved through dataflow-oriented architectures like systolic arrays and spatial pipeline accelerators, which exploit parallelism and data locality to boost throughput. At the edge, lightweight hardware and model compression techniques enable efficient inference on resource-constrained devices. Together, these hardware advancements and architectural strategies form a robust and adaptable ecosystem for accelerating both training and inference across the full spectrum of modern AI applications.

The proliferation of AI chips has paralleled the explosive growth in machine learning applications across industries. From edge devices to cloud-scale AI systems, these chips have become central to enabling efficient, scalable, and reliable deployment of modern AI technologies. Their continued evolution will be vital in addressing future challenges in compute performance, energy sustainability, and application-specific optimization.

1.4 High-Level Synthesis of Machine Learning Accelerators

High-Level Synthesis (HLS) is a transformative design methodology that enables the creation of hardware accelerators using high-level programming languages such as C, C++, or SystemC. This shift from traditional hardware description languages (HDLs) like Verilog or VHDL allows developers to design and test hardware at a much higher level of abstraction, significantly reducing development time and effort. HLS tools automatically translate behavioral code into register-transfer level (RTL) representations, facilitating synthesis onto platforms like FPGAs and ASICs. This abstraction is particularly valuable in AI and machine learning domains, where rapid prototyping, algorithmic innovation, and hardware adaptability are critical.

The evolution of customized hardware accelerators for ML applications has substantially increased the efficiency of computing large workloads tailored to specific tasks. However, the traditional design flow for developing a custom accelerator for each individual application is prohibitively expensive in terms of time, engineering effort, and financial resources. Designing accelerators at the RTL level leads to long time-to-market cycles and high verification overhead. Most critically, exist-

ing RTL implementations are often difficult to repurpose for different architectures or applications, due to their low flexibility and tight coupling to specific hardware configurations. To overcome these limitations, the industry increasingly turns to High-Level Synthesis.

HLS provides a pathway to highly parameterizable and reusable accelerator designs. Unlike traditional RTL implementations, HLS accelerates design time and drastically reduces verification and debugging overhead. Instead of writing low-level, clock-cycle-accurate behavior by hand, designers describe functionality in a high-level behavioral language such as C++, while the synthesis tool—such as Siemens' Catapult HLS [111] or AMD/Xilinx's Vitis HLS [4]—automatically generates the corresponding RTL. The resulting architecture depends on user-defined constraints, allowing the designer to explore different implementations by adjusting parameters such as target clock frequency, memory configuration, latency goals, and loop transformation strategies. This not only enables faster development but also supports robust design-space exploration, essential in an era where AI models evolve rapidly and hardware must adapt in real time.

One of the most powerful aspects of HLS is its support for scheduling, which plays a central role in performance optimization. Scheduling refers to the allocation of operations to specific clock cycles, affecting overall latency, resource usage, and throughput. HLS tools allow designers to guide scheduling decisions using techniques like loop pipelining, loop unrolling, and operation chaining. These techniques help exploit parallelism and data locality—critical factors in the performance of AI accelerators. Through iterative exploration and constraint tuning, designers can efficiently evaluate multiple architectural options without rewriting the design from scratch, which is a major advantage over fixed RTL implementations.

Transitioning to an HLS design flow requires a new mindset. Although HLS increases abstraction and productivity, it still demands a deep understanding of hardware architecture. Designers must write synthesizable C++ code with awareness of how the HLS tool maps this behavior into hardware structures. For instance, a seemingly simple operation like an adder module must be properly structured with defined interfaces, control logic, and synchronization. While SystemVerilog allows designers to clearly distinguish between input and output ports and define precise timing behavior, HLS shifts focus toward functional correctness and architectural constraints. In summary, HLS introduces a powerful methodology for designing AI accelerators—enabling faster development, broader reuse, and efficient optimization—while requiring designers to bridge the abstraction gap between software and hardware with care and expertise.

1.5 Thesis Contribution

This thesis contributes a novel and highly configurable design framework for the automated generation of hardware accelerators for CNNs, leveraging HLS [36]. Motivated by the ever-increasing demand for high-throughput, low-latency, and energy-efficient machine learning inference, especially in embedded and edge systems, the proposed framework bridges the gap between high-level model specification in Python (via TensorFlow or PyTorch) and low-level hardware realization. By automating the transformation from software models to synthesizable hardware, it significantly reduces design complexity and time-to-market for CNN accelerators.

The core of the proposed framework is a CNN accelerator generator that supports the automatic exploration of architectural design alternatives through a rich set of optimization knobs. These include parallelism levels, buffer sizing, tiling strategies, and dataflow scheduling choices. Beyond traditional optimizations, the framework uniquely integrates support for functional safety and advanced buffering schemes. It also introduces customizable options for data representation and arithmetic precision, enabling efficient hardware deployment across a wide range of applications.

More specifically for the case of non-traditional spatial convolution architectures, we introduce a

streaming convolution engine called LazyDCstream [34], optimized specifically for *dilated convolution*. LazyDCstream is based on a sliding-window architecture that exploits the structured reuse inherent in dilated convolution. It decomposes the dilated convolution operation to maximize sharing of window buffers and minimize redundant data movement. A key innovation is its "lazy" data movement strategy, which ensures that the number of input data transfers per clock cycle remains minimal and independent of the dilation rate. This approach leads to significant reductions in dynamic power consumption without compromising throughput or incurring area penalties compared to traditional efficient convolution engines.

Complementing LazyDCstream, the framework also introduces LeapConv [32], a specialized engine for the efficient execution of strided convolutions. Strided convolutions typically exhibit irregular computation patterns, which challenge conventional streaming architectures. LeapConv addresses this by decomposing strided convolutions into multiple parallel unity-stride operations, which are then merged into a single hardware structure. This strategy improves data reuse and eliminates unnecessary memory access, resulting in lower energy consumption that scales favorably with stride size. Despite the added multiplexing complexity required for reconfigurability, the area overhead remains marginal, as confirmed by experimental synthesis results.

An essential innovation presented in this thesis is the integration of online functional safety mechanisms through ConvGuard [31], an Algorithm-Based Fault Tolerant (ABFT) error-checking unit tailored for CNN convolution layers. ConvGuard enables runtime detection of hardware faults by comparing output checksums against predicted values computed from the borders of the input image, leveraging a newly discovered convolutional invariance condition. This novel technique eliminates the need for extensive intermediate checksum storage, making ConvGuard significantly more area- and power-efficient than existing solutions. The checker is fully parametrizable and supports a range of convolution configurations, including various kernel sizes and strides, without compromising on real-time fault detection capabilities.

To support deployment across diverse hardware platforms and performance requirements, the framework includes comprehensive support for *data representation flexibility*. Users can select from traditional single- and reduced-precision floating-point formats as well as integer and fixed-point representations. Each format is paired with optimized arithmetic units [33, 35] to maintain high computational throughput while minimizing area and energy costs. This adaptability ensures compatibility with a wide spectrum of application-specific design targets, from low-cost embedded systems to high-performance computing accelerators.

The proposed framework has been validated through the automatic generation and synthesis of multiple CNN accelerators directly from high-level Python models. Experimental results demonstrate that the generated designs offer competitive, and in many cases superior, performance and energy efficiency compared to state-of-the-art CNN accelerator generators. In particular, the integration of advanced buffering schemes, functional safety mechanisms, and configurable data arithmetic options provides a feature-rich and robust hardware design environment. Comparative studies also confirm the superior resource utilization and configurability of the proposed framework across different CNN topologies and FPGA platforms.

In summary, this research delivers a practical and innovative solution for automated CNN accelerator generation. By combining high-level design automation with low-level hardware optimization, it provides an effective path forward for deploying machine learning models in power- and latency-sensitive environments, while also addressing critical challenges related to hardware safety, flexibility, and performance.

1.6 Thesis Organization

The remainder of this thesis is organized as follows:

Chapter 2 provides an overview of two fundamental classes of machine learning accelerators: Systolic Arrays and Spatial Pipeline Dataflow architectures. It explores the design principles and trade-offs associated with these architectures, particularly in the context of CNNs. Special attention is given to the spatial pipeline dataflow approach, including how data moves through the system and interacts with computational resources. The chapter also compares different architectural configurations, analyzing their impact on performance, scalability, and energy efficiency.

Chapter 3 introduces two specialized CNN accelerator architectures: LeapConv and lazyDCstream. Streaming architectures typically suffer from redundant data movement when handling non-unity stride and dilated convolutions, leading to inefficiencies in power and resource utilization. LeapConv addresses this challenge by reorganizing on-chip buffering to minimize unnecessary data transfers, thereby improving overall hardware efficiency. Similarly, lazyDCstream presents a novel buffering strategy that reduces redundancy in dilated convolution computations. Together, these architectures demonstrate how careful memory organization can significantly enhance performance in streaming CNN accelerators.

Chapter 4 focuses on fault tolerance in convolutional operations, emphasizing the use of ABFT techniques. A novel method is introduced to compute convolution checksums implicitly, reducing the overhead typically associated with explicit fault detection. The proposed mathematical formulation simplifies the checksum computation, significantly lowering the number of operations required. This technique is supported by a custom hardware architecture that can efficiently implement the implicit checksum generation. The chapter highlights the potential of ABFT to enhance reliability without sacrificing computational efficiency.

Chapter 5 presents FastFloat4HLS, a custom C++ floating-point library designed specifically for HLS. The library includes a fused dot-product operator, which is crucial for optimizing the multiply-and-accumulate patterns found in convolutional layers. By focusing on floating-point precision and performance, FastFloat4HLS enables the generation of efficient hardware implementations for machine learning workloads. The library aims to balance numerical accuracy with hardware resource constraints in HLS-based design environments.

Chapter 6 addresses the efficient hardware acceleration of deep learning kernels, particularly matrix multiplications executed on systolic arrays. The chapter emphasizes the importance of reduced-precision floating-point arithmetic for lowering energy and area costs while maintaining acceptable model accuracy. A new skewed pipeline design is proposed, aimed at optimizing the chained multiply-add operations inherent in systolic structures. This design introduces novel exponent forwarding paths, enabling overlapping of pipeline stages across adjacent processing elements.

Chapter 7 outlines a high-level framework for the rapid generation of CNN accelerators directly from Python-based models. At its core is a custom-developed C++ library tailored for HLS, allowing seamless translation from high-level descriptions to efficient hardware implementations. The framework emphasizes automation and modularity, significantly reducing the design time and complexity for deploying custom accelerators. This chapter encapsulates the practical contributions of the thesis by bridging algorithm design and hardware realization.

Chapter 8 concludes the thesis by summarizing its core contributions and outlining future research directions. It also identifies open challenges and outlines potential directions for future work, including the design of dataflow accelerators that can efficiently support irregular and dynamic workload, the co-optimization of data movement, memory hierarchy, and interconnects as well as the optimization of accelerators for ultra-low-power systems and end-to-end system-level safety.

2 Hardware Accelerators for Convolutional Neural Networks

CNN acceleration is typically achieved through GPUs, FPGAs, or ASIC chips. GPUs leverage efficient parallelization of operations, seamlessly utilizing the computational resources available. Conversely, FPGA and ASIC implementations rely on innovative architectural designs and specialized dataflow models to accelerate the computation-intensive operations inherent in CNNs.

These hardware architectures for CNN accelerators fall into two primary categories: systolic array (SA) and spatial pipeline-dataflow (SPD) accelerators. While each architecture operates on different principles, both must be efficiently designed to meet the requirements of the applications they serve. SA architectures provide a uniform computational flow, requiring efficient mapping of each application to the available hardware. SPD accelerators are typically tailored to specific applications, utilizing architectures designed to meet the requirements of individual layers within a CNN. This chapter delves into the architectures of these two types of accelerators, with a particular focus on SPD accelerators, which have been predominantly utilized in the research conducted in this work.

2.1 Systolic-Array Accelerators

Systolic computation traces its roots back to the 1960s with the emergence of linear array processors tailored for signal processing tasks. However, it was not until the late 1970s that the work of H.T. Kung and C.E. Leiserson [71] introduced the first SA architecture. This innovative design showcased a regular structure of processing elements interconnected in a synchronized manner, optimizing computational efficiency by facilitating the seamless flow of data between the processing elements. Subsequently, SA architectures found wide-ranging applications in digital signal processing and matrix computations, where they proved instrumental in accelerating operations such as matrix multiplication, convolution, and Fourier transforms.

The efficiency of SAs prompted researchers to explore their potential in a wider variety of areas, such as image processing, pattern recognition and artificial intelligence. Concurrently, efforts were made to optimize the designs to suit specific application requirements, enhancing their performance and versatility. Moreover, the advancements in semiconductors technology in addition to the increased demand for high-performance computing resulted in novel SA architectures, capable of handling complex computations with remarkable efficiency. Given the increasing demand for hardware accelerators across various domains like edge computing, autonomous vehicles and healthcare, SAs have found integration into specialized hardware platforms. These platforms are designed to accelerate computational intensive operations like CNNs, thus aligning with the continuous growth of deep learning.

In recent years, SA accelerators have been adopted by leading technology companies to meet the growing computational demands of AI and deep learning workloads. These architectures, known for their regular, data-parallel structure and high throughput, are particularly well-suited for accelerating the matrix operations that dominate neural network computations.

One of the most prominent implementations of SAa is found in Google's TPUs. Custom-designed specifically for machine learning tasks, TPUs are optimized for executing large-scale matrix multi-

plications—an essential operation in both neural network inference and training. Deployed extensively across Google's data centers, TPUs play a central role in powering a wide range of AI-driven services, including Google Search, Gmail, Google Photos, and Google Translate. Their architecture exemplifies the practical advantages of SA-based designs in achieving low-latency and energy-efficient computation at scale.

Another notable example is NVIDIA's Tensor Cores, which are integrated into its high-performance GPUs. Although not pure systolic arrays in the traditional sense, Tensor Cores adopt a systolic-like execution pattern to accelerate tensor operations such as convolutions and matrix multiplications. These cores are integral to enabling real-time performance in deep learning applications across diverse domains, including natural language processing, computer vision, and autonomous systems. By combining massive parallelism with architectural enhancements for tensor operations, NVIDIA has made Tensor Cores a foundational component of modern deep learning infrastructure.

Additionally, Intel's Habana Gaudi AI Processor, developed by Habana Labs (a subsidiary of Intel), employs a systolic array architecture to deliver high-throughput training and inference performance. Habana Gaudi processors feature a large array of processing elements designed for efficient parallel execution of deep learning workloads. They support advanced interconnects and memory hierarchies to sustain the dataflow demands of modern neural networks. These processors are used in enterprise-level data centers to accelerate AI tasks in sectors such as healthcare, finance, and automotive, highlighting the versatility and scalability of systolic array-based designs in industrial applications.

Collectively, these implementations underscore the critical role of SA accelerators in today's AI ecosystem. Their ability to provide high performance, energy efficiency, and scalability makes them indispensable in both cloud-based and edge computing environments, reinforcing their importance in the ongoing evolution of hardware for deep learning.

2.1.1 Architecture of a Systolic Array

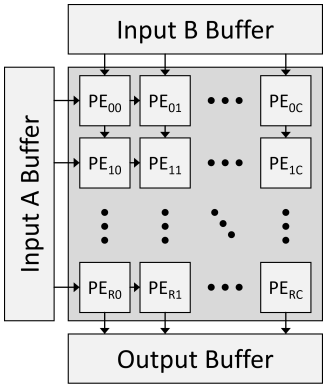


Figure 2.1: The overall architecture of a systolic array.

Systolic arrays are designed to efficiently compute GEneral Matrix Multiplication operations (GEMM). They are composed as a grid of Processing Elements (PEs) interconnected in a pipeline fashion. Each PE consists of a multiplier, an adder, and necessary registers to appropriately pipeline the streaming operation. 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 edge. Figure 2.1 illustrates the comprehensive architecture of a SA accelerator, featuring corresponding local buffers utilized throughout

its operation.

Each PE of the SA computes a Multiply-and-Accumulate (MAC) operation, between the data that enter from its west and north edges. These data are also propagated to the following PEs, exiting from the east and south edges. The exact structure of the PE as well as the data movement inside the SA depend on the data flow scheme that is utilized by the architecture.

Employing SA accelerators for CNNs necessitates efficient mapping of the convolution operation, a critical aspect of computation, to the available hardware while ensuring proper dataflow. Given that SA accelerators are optimized for accelerating GEMM operations, the convolution operation must be transformed accordingly. This transformation is typically performed using a method known as im2col (image-to-column) [124], which flattens overlapping patches of the iIFM into column vectors. Each patch corresponds to the receptive field of the convolution kernel, and by sliding this field over the input spatially, a matrix is formed where each column represents one such patch. Simultaneously, the convolutional kernels are reshaped into a 2D matrix where each row corresponds to a flattened filter.

$$Y_{col} = X_{col} \times W_{row} \tag{2.1}$$

Once the input and kernel matrices are constructed, the convolution operation is converted into a standard matrix multiplication between the im2col matrix and the kernel matrix, as shown in equation 2.1, where Y_{col} is the matrix of output features in column format, X_{col} is the im2col matrix of the input and W_{row} is a reshaped version of the kernel. This results in an output matrix where each row corresponds to the output of applying all filters to a specific spatial location in the input. Finally, this matrix is reshaped back into the original output shape, matching the expected dimensions of the CNN layer.

2.1.2 Data Flow in Systolic Array Architectures

The *dataflow* scheme employed by the SA determines the internal structure of the PEs and how the matrix multiplication $X \times W$, is executed. For instance, in *weight-stationary* (WS) dataflow [106], matrix W (the 'weights') is pre-loaded in the SA, while matrix X (the 'input') is transposed and fed into the SA from the West side, as shown in Figure 2.2(c). The WS approach is generally preferred over other dataflows, since it exploits the high spatio-temporal reuse of the weights [57]. After the top row is filled, it takes multiple cycles to *reduce* the results of all the PEs in the same column. The number of cycles required for the reduction depends on the FP multiply-add units within each PE; i.e., the result of each PE moves downwards to the next PE in the same column. The SA becomes empty when the reduction is finished in the right-most column, for all incoming columns of matrix X.

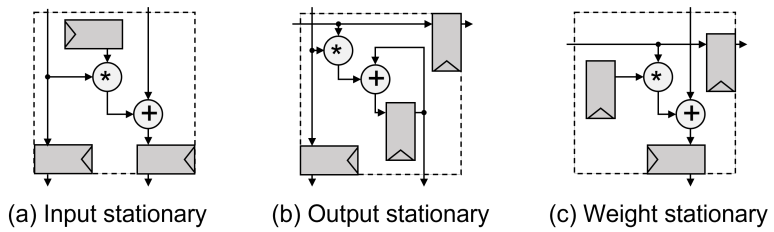


Figure 2.2: The structure of the processing element in a systolic array for each dataflow.

Other popular data flow schemes are the Input Stationary (IS) and the Output Stationary (OS). The

IS data flow is a data movement pattern in which the input data remain fixed in the PEs, while weights and partial sums are passed through the array. This approach minimizes the need to reload input values multiple times, thereby reducing memory bandwidth usage and improving energy efficiency. Each PE holds a portion of the input data and performs computations as new weights arrive from adjacent PEs or enter from the north edge of the SA. The resulting partial sums are propagated through the array toward the output. The structure of the PE to support this functionality is depicted in Figure 2.2(a).

The OS data flow is a pattern in which each PE is responsible for computing and storing a specific output value, keeping the partial and final results stationary within the PE throughout the computation. Input data and weights are streamed through the SA, allowing each PE to accumulate results over time. This approach minimizes the movement of intermediate results, which can be costly in terms of energy and bandwidth. Figure 2.2(b) shows how the PE is structure to operate under this dataflow scheme.

2.2 Spatial Pipeline-Dataflow Accelerators

Unlike SAs, the SPD accelerators emerged very recently because of the need to directly accelerate the computation of CNNs. Their progression has played a pivotal role in the development of hardware architectures for deep learning. Since the early 2000s, these accelerators have evolved from conceptual research models into highly optimized, commercially viable systems that form the computational backbone of modern AI infrastructure.

In the early 2000s, the concept of spatial pipeline architectures began to gain attention as a potential means to accelerate CNNs and other deep learning workloads. Initial research efforts were exploratory, aiming to assess the feasibility of mapping convolutional computations onto spatially distributed processing elements. These studies laid the groundwork for a new class of accelerators characterized by regular, parallel data movement patterns and localized computation.

By the mid-2000s, interest in SPDs intensified as researchers began to focus on architectural optimization. Efforts during this period centered around enhancing computational throughput and energy efficiency, with innovations in the design of data paths, memory hierarchies, and interconnects. Architectural configurations such as one-dimensional and two-dimensional pipelines were extensively studied, and optimization techniques targeting layer-specific characteristics of CNNs—such as weight reuse and input buffering—began to emerge.

In the late 2000s, SPD accelerators gained prominence as a major research direction in the field of hardware acceleration for deep learning. The scope of research expanded beyond convolutional layers to include operations like pooling and activation functions, which are integral components of CNN architectures. Novel approaches to pipeline scheduling, data reuse, and resource allocation were introduced to address the computational bottlenecks of increasingly complex neural networks.

The 2010s marked a transformative decade for SPD accelerators. With the explosion of big data and the widespread adoption of deep learning across industries, there was a pressing demand for high-performance, energy-efficient computing. Researchers continued to refine spatial pipeline architectures, drawing insights from both academic advancements and industrial requirements. These refinements led to better scalability, improved parallelism, and tighter integration with memory subsystems.

By the mid-2010s, SPD accelerators had begun transitioning from academic prototypes to practical, deployable systems. This period saw the integration of spatial dataflow architectures into commercial hardware platforms such as FPGAs and ASICs. Companies recognized the advantages of these architectures in supporting low-latency and high-throughput inference workloads, particularly in resource-constrained environments like edge devices.

From the late 2010s to the present, SPD accelerators have become a staple in the landscape of deep learning hardware. A wide array of both commercial and open-source implementations has become available, reflecting the maturity and flexibility of the spatial pipeline paradigm. These accelerators are now widely employed in cloud services, embedded AI systems, and edge computing applications, offering scalable performance, reduced energy consumption, and the ability to handle diverse neural network models efficiently.

In summary, the evolution of SPD accelerators mirrors the trajectory of deep learning itself—from experimental beginnings to industrial-scale deployment. Their ability to deliver high-performance computation with predictable and efficient data movement has made them an essential architectural solution for modern AI workloads.

2.2.1 Organization of a Spatial Pipeline-Dataflow Accelerator

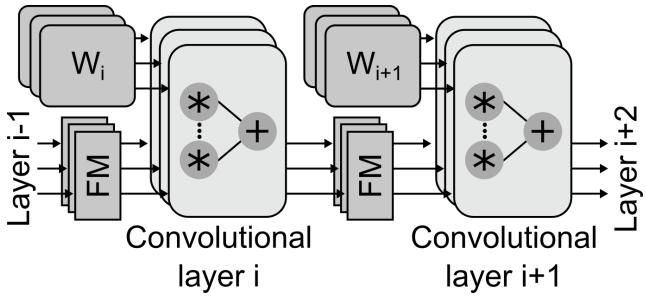


Figure 2.3: The organization of a spatial pipeline-dataflow CNN accelerator.

A SPD accelerator for CNNs is designed as a collection of convolution engines connected in series, creating a pipelined architecture as shown in Figure 2.3. On each stage of the pipeline, a CNN layer is computed by an engine designed according to the characteristics of the layer [11, 25]. In addition, the parameters of each CNN layer, i.e. the weights and biases, are stored in local buffers, dedicated to the convolution engine that computes the specific layer.

The input features are streamed inside the accelerator and their efficient handling depends on the architecture of the convolution engine. A streaming convolution engine manages to re-organize the stream of data so that the computation can be performed.

Apart from convolution engines, other smaller engines may exist, such as specialized architectures for pooling layers or fully connected layers that are usually found in CNN architectures. Although these other blocks are equally important for accelerator's performance, the one that dominates in terms of complexity is the convolution engine itself.

2.2.2 Streaming Convolution Engine

The streaming convolution engines utilize a sliding-window memory architecture [94, 131]. The organization of a streaming convolution engine can be broken down into three main components, which can be seen in Figure 2.4. First, a set of line buffers is used to store the input features that arrive at the engine. The features remain inside the line buffers until they are no longer required. The size of each line buffer equals the size of a row in the input feature map, while the amount of line buffers depends on the size of the kernel K and equals to $K-1$. Second, to align the correct features from the line buffers with the currently arriving input, a window buffer is utilized that stores the features participating in the current output computation. Last, the actual computation is performed

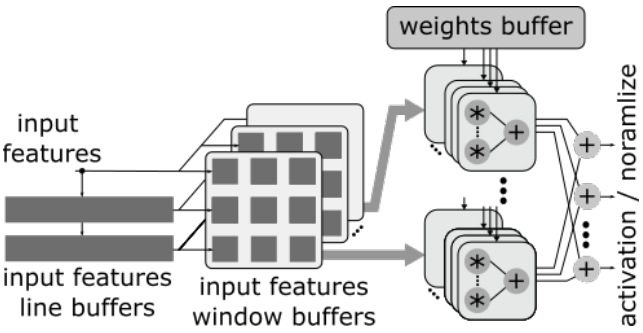


Figure 2.4: The organization of a streaming convolution engine that implements a convolution layer.

by a dot product unit where the input features are multiplied with the corresponding weights and the individual products are added to produce the output feature. Then the activation and/or normalization is performed on the outputs before being transferred to the next layer.

In each clock cycle, a streaming convolution engine – such as the one shown in Figure 2.4 – accepts one input feature and computes one output feature [55]. To do so, all features within the corresponding input window must be available. For a $K \times K$ filter, the features from the last K rows of the input are required. This requires line buffers with the ability to store $K-1$ rows, plus a window buffer that holds the currently active input features. This sliding-window memory architecture allows for data reuse and fine-grained parallelism. Window buffers are normally implemented with registers, while larger line buffers are mapped either to standard-cell-based memories [84], or SRAM blocks [130].

Algorithm 1 Algorithmic description of the data movement involved in the window and line buffers of a streaming convolution engine.

```
1  foreach input feature (i,j)
2  // read line buffers and shift window buffer
3  for (m = 0; m < K; m++) {
4    tmp = (m < K-1) ? lb[m][j] : input;
5    for (n = 0; n < K; n++)
6      window[m][n] = (n < K-1)? window[m][n+1] : tmp;
7  }
8  // move data downwards in the line buffers
9  for (m=0; m < K-1; m++)
10   lb[m][j] = (m < K-1)? lb[m+1][j] : input;
```

A set of parallel multipliers and an addition tree perform the actual computation by applying the weights of the kernel to the features stored in the window buffer. The engine manages to keep the input features properly aligned to the filter's coefficient by shifting the contents of the window buffer. In each clock cycle, the input feature is pushed in the top left corner of the window buffer and in the top row of the line buffers. In parallel, the window buffer is filled with features that come from the line buffers and shifts its contents to the right to simulate the rightward sliding of the filter over the input. The features that belong to the same column with the incoming feature are also moved downwards in the line buffers, to simulate the downward sliding of the filter. An algorithmic view of these operations, assuming a $K \times K$ window buffer (`window`) and $K-1$ line buffers (`lb`), is listed in Algorithm 1.

Repeating the same operation for each IFM and adding the individual results, a complete output feature is produced. Although this description refers to a 2D convolution, by repeating the same operation for each IFM and OFM, the convolution engine can compute a complete CNN layer.

2.3 Customizing the Architecture of a Spatial Pipeline-Dataflow Accelerator

The architecture of a SPD accelerator depends on two main factors, the characteristics of the CNN and the available resources. In the case of ASIC implementations the resources constraint can be either a power or an area constraint. On the other hand, when designing for FPGAs, the available resources i.e. Look-Up Tables (LUTs), Block RAMs (BRAMs), Digital Signal Processing cells (DSPs), constrain the architectural choices.

CNN applications are in most cases implemented by multiple convolutional layers in series, interleaved by other layers like pooling, activation and normalization. As the convolutional layer is the most complex component of the CNN, and the most demanding in terms of resources, its structure has been studied thoroughly in order to explore the different architectural choices when designing an accelerator for a CNN layer [23, 82, 88, 100, 142].

The code of Algorithm 2 describes the functionality of a convolutional layer, which performs a convolution between N IFMs and M filters to produce M OFMs. The size of each IFM is $R \times C$, while the size of the kernel that convolves with each IFM is $K \times L$. The N IFMs, see "Inputs" array in Algorithm 2, convolve with the N kernels, see "Weights" array in Algorithm 2, to produce one OFM of size $(R-K+1) \times (C-L+1)$, see "Out" array in Algorithm 2. Each feature of an OFM is, in fact, the sum of the dot-products across all IFMs increased by the value of a bias. The same operation is repeated for each of the M $K \times L \times N$ filters to produce all the OFMs.

Algorithm 2 Algorithmic implementation of a CNN layer.

```
1  INPUTS: Inputs[R][C][N], Weights[K][L][N][M], Biases[M]
2  OUTPUT: Out[R-K+1][C-L+1][M]
3  LA: for r = 0; r < R-K+1; r++
4    for c = 0; c < C-L+1; c++
5      Initialize Out[r][c][M] to Biases[M]
6      LB: for n = 0; n < N; n++
7        LC: for m = 0; m < M; m++
8          kernel = 0
9          LD: for k = 0; k < K; k++
10           for l = 0; l < L; l++
11             kernel += Weights[k][l][n][m] * Inputs[r+k][c+l][n]
12         Out[r][c][m] += kernel
```

Looking at Algorithm 2, the operation of a convolutional layer is described by four main loop groups. The *LB* and *LC* loops iterate over the input and output feature maps respectively, while *LA* refers to two nested loops, that iterate over the rows and columns of each IFM and *LD* iterates over the two dimensions of the kernel. The structure of these loops defines the architecture of the convolution engine that accelerates a CNN layer. The different architectures can be studied by just applying different loop optimization techniques, like loop unrolling and loop tiling and loop interchange.

2.3.1 Loop Unrolling

When designing a CNN accelerator, especially for FPGA implementations, the latency of the computation, the amount of buffering and the data transfers between the convolution engine and the external memory depend on the structure of the loops. Keeping the loops fully rolled, would result in a very simple hardware with one addition and one multiplication unit, however, the architecture would suffer from increased latency and data transfers from the memory, as it would support only one input read per $N \times M \times K \times L$ cycles. On the other hand, supporting a fully parallel architecture, by fully unrolling the loops, can be very demanding in terms of resources, especially in cases of CNNs where both the number of layers and their sizes are very big. In reality, the structures of modern CNNs indicate that in most cases these loops has to be partially unrolled to design an efficient hardware accelerator.

As it was studied in [82], unrolling each one of these loops affects the way that the unrolled operators will be utilized. To understand the differences we will see the impact of unrolling each of these four loops separately. Starting from the most inner loops of Algorithm 2, *LD* computes the dot-product of the weights and the features currently in the window buffer. This operation can be either performed in multiple cycles, using only one multiplier and one adder, or completely in parallel, using $K \times L$ multipliers and the appropriate number of adders to construct the addition tree for the reduction of the products. Having the features of the IFM that participate in a single output already available in the window buffer, a fully parallel implementation of the dot-product unit could efficiently improve the latency of the computation.

Second, based on the equation of the convolution (2.2), the features in the window buffer need to be multiplied with weights from different filters, as each IFM participates in the computation of all M OFMs,

$$O_{r,c,m} = B_m + \sum_{n=0}^{N} \sum_{k=0}^{K} \sum_{l=0}^{L} I_{r+k,c+l,n} W_{k,l,n,m}, m \in [0, M) \tag{2.2}$$

where I is the array of IFMs, W the weights of the filter and O the OFMs. Unrolling the *LC* loop by a factor of T_m results in the instantiation of T_m parallel and completely independent dot-product operators, as illustrated in Figure 2.5. Each one of these operators reads the same window buffer and the corresponding weights from the weight buffer and computes a partial output that is stored in an output buffer. The size of the output buffer is equal to the size of an OFM, and the partial outputs that are stored in each position of the buffer get accumulated with the corresponding partial outputs from different IFMs. When all IFMs have been read, the final accumulated values are the output of the layer.

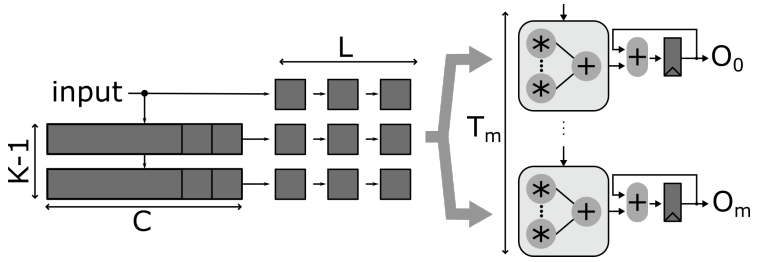


Figure 2.5: The organization of a dedicated CNN layer accelerator that is unrolled in relation to the OFMs.

Instead of unrolling *LC* to design a parallel hardware in respect to the OFMs, another solution would be to unroll the *LB* loop. Unrolling this loop by a factor T_n means that features from T_n

different IFMs arrive in parallel. These T_n input features are treated as a single input, with a bigger word length, by the convolution engine. This can be seen in Figure 2.6, where each register of the window buffer as well as each word in the line buffers is $T_n \times DW$ wide, where DW is the width of the data. Having data from T_n feature maps available, requires the use of T_n dot product units where each one computes the dot-product in the respective channel. If T_n is equal to N, the individual result are accumulated and produce an output feature, else the partial sum is stored and waits to be accumulated with the next set of T_n inputs.

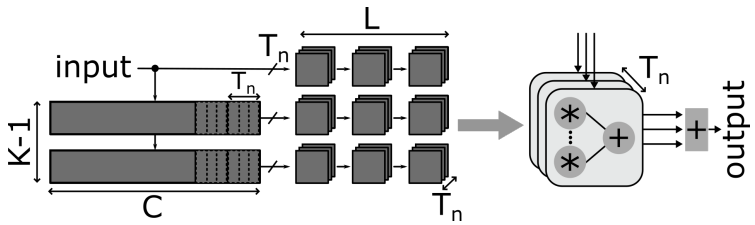


Figure 2.6: The organization of a dedicated CNN layer accelerator that is unrolled in relation to the IFMs.

Finally, the most outer loops, *LA*, of Algorithm 2 iterate across every feature in a feature map. Unrolling these loops by T_r and T_c means that $T_r \times T_c$ features from the same IFM arrive simultaneously at the convolution engine. If these features are far apart, for example if the IFM is tiled into $T_r \times T_c$ blocks, then as Figure 2.7 illustrates, $T_r \times T_c$ window buffers and smaller line buffer sets are utilized to re-organize the input features. In addition $T_r \times T_c$ dot-product units, that share the same weights, are used to compute the partial outputs. The solution of Figure 2.7 assumes that the input is tiled, however, if the inputs refer to neighboring features, then different architectures can be applied that allow the sharing of the resources [116].

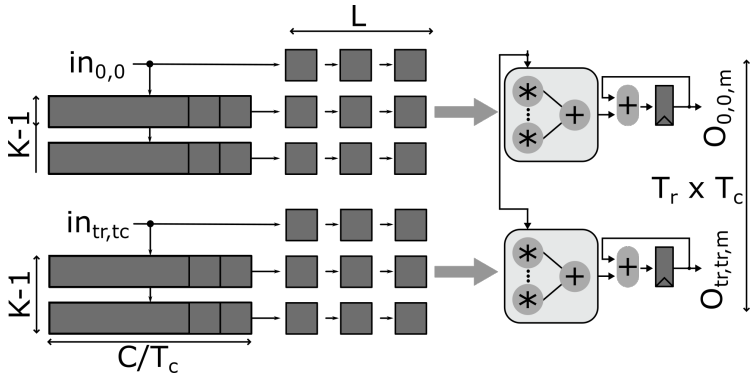


Figure 2.7: The organization of a dedicated CNN layer accelerator that is unrolled in relation to the features inside a feature map.

2.3.2 Loop Tiling

Loop unrolling is an optimization that results in the instantiation of parallel hardware, however, the increase of the parallel computation units leads to an equal increase in the need for local buffering to support the resources requirement. This demand for local buffering stresses the on-chip storage

resources that often are not enough, making the use of the external memory to store the data essential. In these cases, instead of unrolling every loop of the algorithm, we can perform a tiling optimization, that allows us to divide the data into multiple blocks. This blocks are small enough to be stored in the on-chip memory, enabling the efficient reuse of the data in the local buffers [23, 82].

In general, the factors T_r, T_c, T_m and T_n can be used to either unroll the corresponding loops or perform a tiling optimization. Algorithm 3 presents an implementation of a convolutional layer with tiled loops. Looking at this example, by fully unrolling the *LD* loops in lines 14-15 and the tiles of the *LB* and *LC* loops in lines 11 and 12 respectively, then $T_n \times T_m$ dot-product units are utilized. The rest of the tiled loops in lines 9-10 correspond to the *LA* loops and are used to break each input feature map in blocks that can be stored in the local buffers.

Algorithm 3 A tiled version of the convolutional layer.

```
1  INPUTS: Input[R][C][N], Weight[K][L][N][M], Biases[M]
2  OUTPUT: Out[R-K+1][C-L+1][M]
3  LA: for r = 0; r < R-K+1; r+=Tr
4   for c = 0; c < C-L+1; c+=Tc
5    Initialize Out[r][c][M] to Biases[M]
6    LB: for n = 0; n < N; n+=Tn
7     LC: for m = 0; m < M; m+=Tm
8      for tr = 0; tr < Tr; tr++
9       for tc = 0; rc < Tc; tc++
10       for tn = 0; tn < Tn; tn++
11        for tm = 0; tm < Tm; tm++
12         kernel = 0
13         LD: for k = 0; k < K; k++
14          for l = 0; l < L; l++
15           kernel+=Weight[k][l][n+tn][m+tm]*Input[r+tr+k][c+tc+l][n+tn]
16         Out[r+tr][c+tc][m+tm] += kernel
```

Following this architectural example, at first $T_r \times T_c \times T_n$ features will be read from the external memory. These data will remain there and be reused for each of the M filters that are being read from the buffer in groups of $T_m \times T_n \times K \times L$ weights. When all the partial outputs have been computed, these data will not be needed anymore and the next tile of $T_r \times T_c \times T_n$ features will be stored in the local buffers. The operation will continue until all the features have been used and the OFMs have been completely calculated. From the example, it is clear that depending on the available on-chip memory, the loops should be carefully tiled.

Finding the optimal unroll-tiling factors T_n, T_m, T_r and T_c is very crucial to achieve an efficient hardware architecture with reduced latency. The selection of these factors, mainly for FPGA implementations, derive from a roofline model [133] that takes into account the available resources and the bandwidth of the memory. Using this model, we can select the unrolling of the hardware and the tiles of the data that will be stored in the local buffers in order to design a fully utilized and efficient accelerator.

2.3.3 Loop Interchange

Finding the optimal tiling options is important, however, the sequence of the loops is equally important as in High-Level Synthesis implementations, the hardware architecture depends in the structure of the code. If we apply a loop interchange optimization, the access patterns of the data is changed and the optimal unrolling of the hardware may need to be reconsidered [100].

Following the sequence of loops in Algorithm 3, at first, the features in the first $T_r \times T_c$ block of all OFMs will be computed before moving to the next tiles of the OFMs. Having the featured maps tiled in tiles of $T_r \times T_c$, when the features stored in the on-chip memory have already been used and are no longer required, then they can be overwritten. In this example, the input features remain stationary in the buffers while the weight kernels are being read in groups of T_m. This means that the same group of $T_n \times T_m \times K \times L$ weights in the local buffer will be fetched multiple times across the iterations of loop *LB*.

A different sequence for the loops can result in a different access pattern in the memory. For example, if the *LB* and *LC* loops switch positions, then the weights will remain stationary, while the $T_r \times T_c \times T_n$ input features will be loaded M/T_m times, until they are no longer needed. Depending on the number of the IFMs and the filters, this change in the sequence would result in a different number of data transfers between the external and the on-chip memory.

3 Energy Efficient Buffering for Non-Traditional Spatial Convolutions

Spatial convolutions are fundamental operations in CNNs that enable the extraction of spatial hierarchies from input data, such as images or feature maps. Beyond standard convolutions, several variations have been introduced to control the resolution, receptive field, and computational behavior of convolutional layers [28]. Strided convolutions downsample feature maps by skipping spatial positions, effectively reducing their dimensions while preserving important patterns. Dilated convolutions expand the receptive field without increasing the number of parameters or computation by inserting gaps between kernel elements, making them ideal for tasks requiring broader contextual understanding, such as semantic segmentation. Transpose convolutions perform the inverse of a standard convolution, enabling the upsampling of feature maps—crucial for image generation and reconstruction tasks. Together, these spatial convolution variants provide flexible mechanisms for spatial resolution control and feature extraction across a wide range of deep learning applications.

This chapter describes two architectures for strided and dilated convolutions that aim to improve the efficiency of SPD accelerators when computing such spatial variations of convolutions.

3.1 Optimized Buffering for Strided Convolutions

Ideally, the scalability of streaming convolution engines should also be maintained when computing *strided* convolutions, and, especially, in environments where the stride can be dynamically reconfigured at runtime. The convolution stride controls the number of pixels by which the kernel's window moves after each operation. This feature facilitates the reduction of the output's resolution. Figure 3.1 graphically depicts the application of strided convolution with stride length 2 of a 3×3 kernel on a 7×7 input image, resulting in a 3×3 output image.

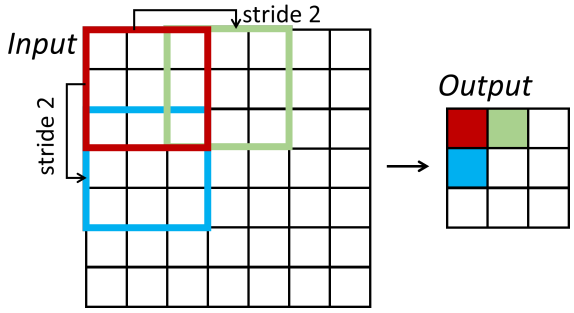
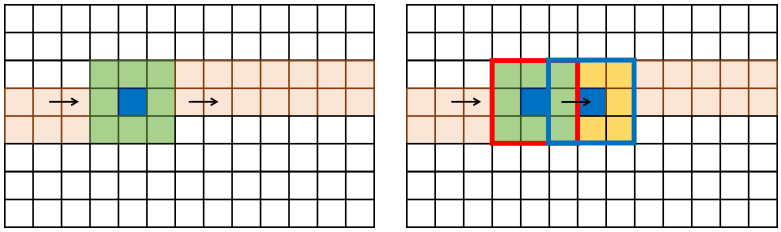


Figure 3.1: The application of convolution with stride length 2 on a 7×7 input image using a 3×3 kernel.

As discussed in previous sections, in standard convolutions the filter slides by 1 position each time (Figure 3.2(a)). In strided convolutions, however, the filter slides over the image with a step S, in

both dimensions. Figure 3.2(b) illustrates this strided slide in the 'x' dimension. As the filter moves S pixels away from its previous location in the same row, the engine needs to receive S new pixels in a row-wise manner to align the input pixels with the newly shifted location of the filter. Only then a new output can be computed. The same alignment should be performed when the filter moves downwards with a step S. This means that after the engine has produced the last output of a row, it must wait for $S-1$ complete rows to be read before it can restart to produce a valid output of the next active row.



(a) (b)

Figure 3.2: The slide of the filter over the input depending on the stride length. (a) Standard slide with step 1. (b) Strided slide with step 2.

In fact, strided convolution inherently involves long periods of inactivity interrupted by periods of actual computation. With that in mind, strided convolution can be easily computed using a standard streaming convolution engine by producing valid outputs only when the output pixels align with the strided slide of the kernel. Although conceptually simple, this approach requires data movements that are equivalent to unity-stride convolutions, which are highly redundant when computing strided convolutions.

Using a standard streaming convolution engine, like the one described in Section 2.2, means that the engine would have an *active* and an *inactive* period. During the *active period*, where, for an incoming pixel (i, j), it holds that $i \mod S = (K-1) \mod S$, the engine produces one pixel every S clock cycles, where $K \times K$ is the size of the filter. On the contrary, the output remains idle during *the inactive period*, i.e., when $i \mod S \neq (K-1) \mod S$. During this time, the engine waits to read the next S rows, before a new period of activity may resume. As the value of S increases, the *inactive* period becomes significantly larger than the "active" one.

By enhancing the functionality of a standard streaming convolution engine, it can compute strided convolutions by computing an output only when needed. However, to keep the pixels aligned with the kernel, the data should always be shifted inside the window buffer, and the line buffers must still follow the data movement of standard unity-stride convolutions, even during the large periods of inactivity. This redundant data movement is effectively removed by LeapConv, thereby leading to significant power consumption benefits.

Other approaches that do not rely on streaming convolution engines try to reduce the complexity of strided convolutions by utilizing Winograd's algorithm [138, 140]. The architecture of LeapConv, that is described in this section, is a streaming convolution engine architecture that *efficiently* supports arbitrary and reconfigurable strides and exhibits the following key attributes:

- It decomposes strided convolutions into multiple independent unity-stride convolution channels to avoid redundancy in computation and data movements. Even though the computations of each channel are performed in parallel, the overall hardware implementation is merged in one unified structure to maximize efficiency and resource utilization.

- The parallel channels are mapped to the same window buffer, while the input is forwarded only to the registers of the active channel. This approach improves the clock-gating efficiency by limiting the data switching activity during the engine's operation to the absolute minimum required for the correct implementation of the algorithm.
- The aforementioned architectural features reduce the power consumption from 10% to 32%, for different ASIC configurations implemented using a 45-nm standard cell library, without introducing any throughput penalty. The area is slightly increased, due to the multiplexing logic added to support the reconfigurability of the stride length.

3.1.1 LeapConv: Architectural Overview

The LeapConv architecture is based on the decomposition of strided convolutions into a set of parallel non-strided convolution channels [64]. To avoid redundancy in computation and data movement, instead of assigning the computation directly to the parallel channels, the hardware implementation of the latter is merged in one unified structure. Furthermore, the parallel channels are mapped to the same window buffer, while the input is forwarded only to the registers of the active channel, thus effectively reducing the data switching activity.

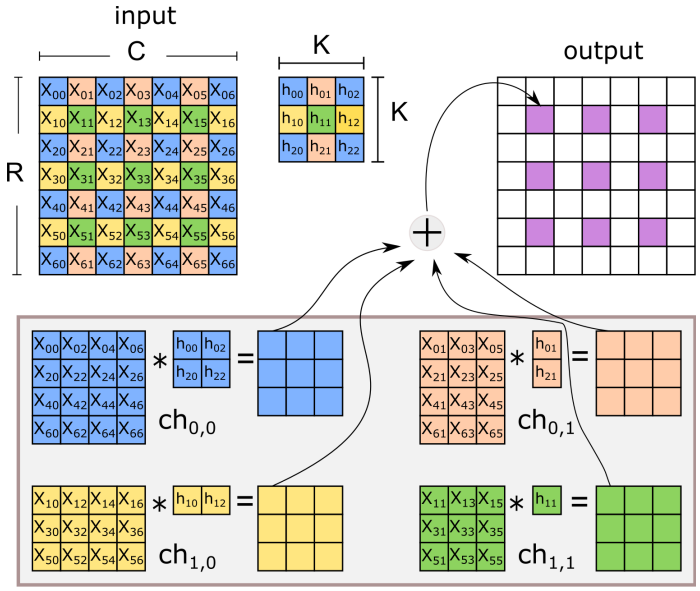


Figure 3.3: The decomposition of a 2-stride convolution into 4 parallel unity-stride convolution channels.

Decomposition

Any strided convolution can be decomposed into the sum of multiple unity-stride convolutions [91]. Since the filter is applied with a stride S, each coefficient of the filter will be multiplied only with a subset of the input elements. Therefore, by grouping together the input pixels that each coefficient "touches," we can derive independent *channels* of sub-images and sub-filters. Figure 3.3 depicts the transformation of a strided convolution with $S = 2$ into four independent channels. In the general case, the number of channels created is equal to S^2. A pixel (i, j) belongs to the channel $Ch_{k,l}$ with

$k = i \mod S$ and $l = j \mod S$. Similarly, the coefficient $h_{m,n}$ belongs to channel $Ch_{k,l}$ when $k = m \mod S$ and $l = n \mod S$.

The operation within each channel is, effectively, a unity-stride convolution, since every pixel of each sub-image is multiplied with every coefficient of the corresponding sub-filter. After computing the output of each channel, we can reconstruct the output of the original strided convolution by simply performing a pixel-wise addition between the individual outputs of each channel.

LeapConv utilizes this decomposition but time-shares the operation of each channel, thereby saving considerable amount of redundant data switching activity and offering a highly efficient overall hardware implementation.

Merged hardware architecture

In a direct implementation of the decomposition transformation, the strided convolution can be computed using S^2 independent unity-stride convolution engines. The outputs of all engines are added to produce a valid result. Therefore, to produce the correct output, the operation of the engines should be aligned.

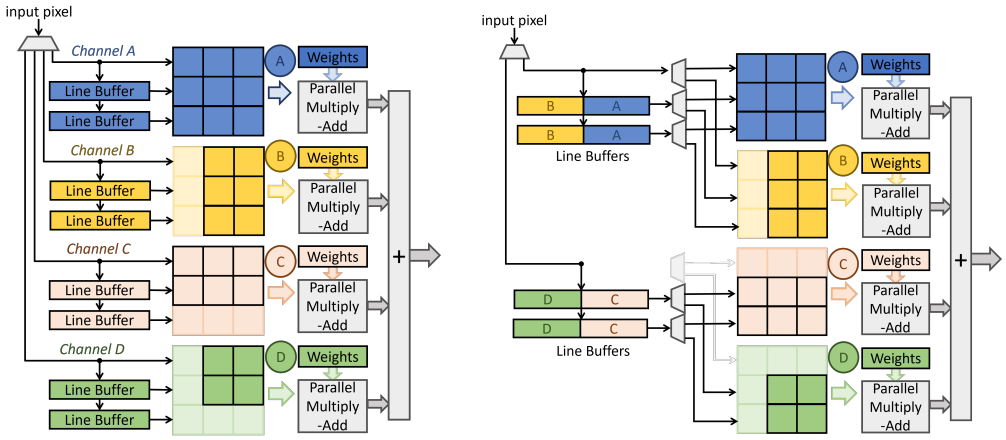


Figure 3.4: The (a) multi-channel architecture that allows the computation of a 2-stride convolution, and (b) the optimized architecture with shared line buffers.

The hardware implementation of the decomposed strided convolution is shown in Figure 3.4(a) for a 5×5 kernel applied with a stride of $S = 2$. The engine of each channel utilizes a smaller window buffer and requires fewer and shorter line buffers, as the input image of each channel is a subset of the original image.

The fragmentation of the line buffers can be avoided by taking advantage of the fact that each channel uses a different part of the input. For instance, channels A and B refer to pixels that belong to even-indexed rows, while channels C and D refer to odd-indexed rows. Therefore, the line buffers of channels A and B can be merged, resulting in larger line buffers that are equal to the size of the ones used in a unity-stride engine. Adding some de-multiplexing, as shown in Figure 3.4(b), the line buffers can now push data either to channel A or B, depending on the column index of the current input pixel. Pixels from even columns will get pushed to channel A and the rest to channel B. In the same way, we can merge the line buffers for channels C and D. For uniform treatment, channels C and D are also equipped with two line buffers.

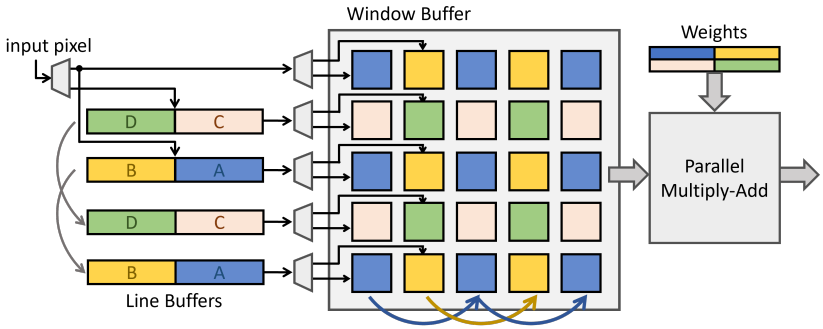


Figure 3.5: Mapping in place the window buffers of the 4 channels. Note that no more registers are used here than in the unity-stride streaming architecture.

The window buffer of each channel can be viewed as a subset of the original window buffer. By rearranging the window buffers of the four channels, we can reconstruct the window buffer of the unity-stride engine, as illustrated in Figure 3.5. The only difference lies in the connectivity between the registers; each register connects only to the registers of the same channel.

To keep the operation of all channels aligned, data movement inside the window buffer occurs only during the active periods. During said periods, output computation occurs only when the window buffer has every pixel of the window of the input image that overlaps with the filter. On the contrary, during the inactive periods that arise naturally due to the strided movement of the window, the window buffer in LeapConv is completely inactive.

To understand the connectivity between registers and the involved data transfers, Figure 3.6 highlights the movement of the data inside a 5×5 window buffer for a strided convolution with $S = 2$, during an active period. Assume that in cycle t_0, when pixels $\{A_3, C_2, A_4, C_3, A_5\}$ are being pushed in the first column (column 0) of the window buffer, channels A and C are activated. At that time, the first 3 columns of the window buffer are filled with data, while the last 2 have not received any input yet. This means that channel A and C have data in two of their columns, while channels B and D have data only in their first column.

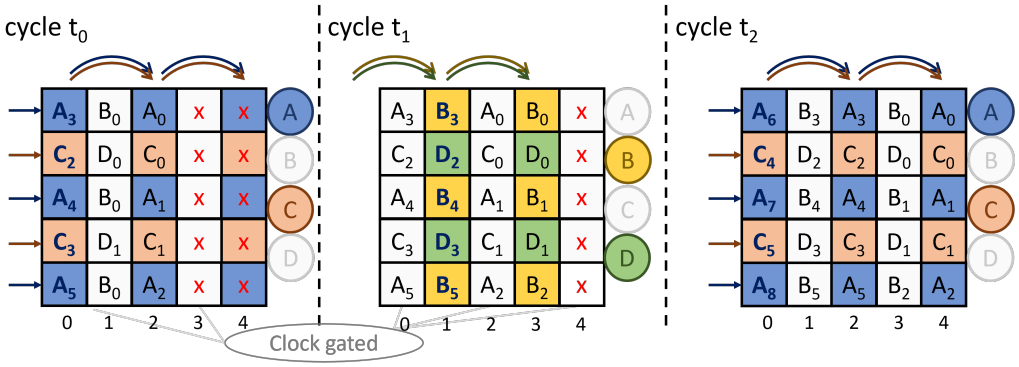


Figure 3.6: The movement of data in the window buffer. Data moves between the registers of each channel, thus, effectively, mimicking the strided kernel movement.

In cycle t_1, pixel B_3 is pushed into the window buffer from the input. The rest of the second column of the window buffer is filled with pixels D_2, B_4, D_3 and B_5 that come from the line buffers.

Pixels B_3, B_4 and B_5 belong to channel B, while D_2 and D_3 belong to channel D. These two channels are active and they should shift the corresponding columns 1 and 3 of the merged window buffer. The remaining registers of the window buffer are unaffected. In total, only two columns out of the five columns have experienced any data switching.

In the next cycle, t_2, three columns are being updated. Column 4 receives the data of column 2, and column 2 receives the data of column 0. The newly arrived pixels $\{A_6, C_4, A_7, C_5, A_8\}$ from the input and the line buffers are pushed into column 0. In all cases, data move two columns forward (to the right), following the stride length ($S = 2$) of the convolution. In this cycle, only three columns experience data switching activity.

Similarly to the reduction in data movement inside the window buffer, LeapConv achieves an equal reduction to the movement of the data between the line buffers. In the case of the unity-stride convolution, that was presented in Section 2.2.2, as listed in Algorithm 1 in order to emulate the downward shifting of the filter over the image, the data of a line buffer is pushed to the next one, i.e., `lb[m][j]` receives data from `lb[m+1][j]`. In LeapConv, the data of the line buffers must move to the next active line buffer of the same channel. Since input pixels are streamed in a row-wise manner, until a complete row is read, only the $(K-1)/S$ line buffers that belong to the active channels are being used. As a result, instead of shifting all line buffers downwards in each clock cycle, only the line buffers of the active channels are being updated.

Algorithm 4 Algorithmic description of the data movement involved in the window and line buffers of the proposed LeapConv architecture.

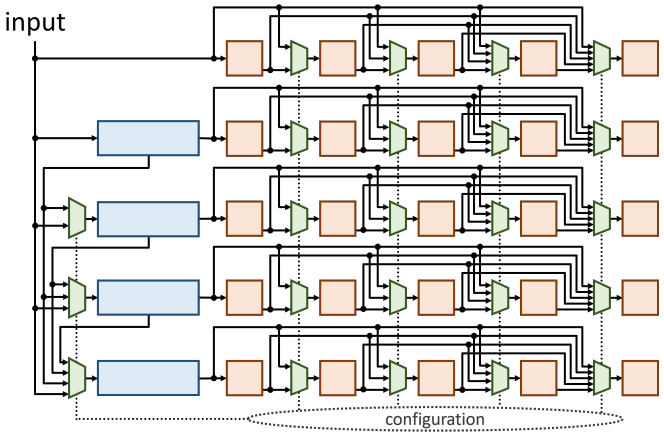
```
1  foreach input pixel (i,j)
2  // read line buffers and shift window buffer
3  if (active_row) // when on an active period
4    for (m = 0; m < K; m++) {
5      tmp = (m < K-1)? lb[m][j] : input;
6      // shift only active columns
7      for (n = j%S; n < K; n += S)
8        window[m][n] = (n < K-S)? window[m][n+S] : tmp;
9    }
10 // move data downwards in line buffers of active rows
11 for (m = i%S; m < K-1; m += S)
12   lb[m][j] = (m < K-1-S)? lb[m+S][j] : input;
```

The update of the window buffer and the corresponding line buffers in LeapConv is detailed in the code segment of Algorithm 4. For each input pixel, the windows buffers are shifted only in the active rows. The shifting does not involve all columns, but only the ones placed S columns apart, i.e., `window[m][n]` is connected to `window[m][n+S]`. The same connectivity pattern is involved across line buffers, i.e., `lb[m][j]` receives a pixel from `lb[m+S][j]`.

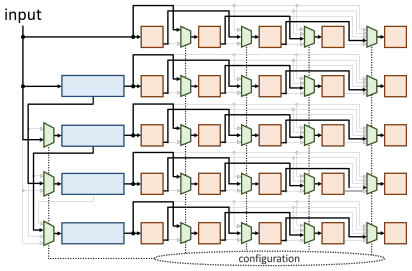
Support for reconfigurability

The purpose of LeapConv is to allow the computation of *any* strided convolution for a specified filter size. By generalizing the architecture of Figure 3.5, we are able to design a streaming convolution engine with *reconfigurable* stride.

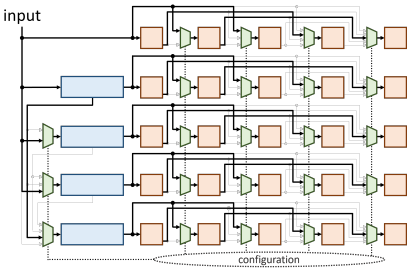
To enable this feature, each register of the window buffer is accompanied by a multiplexer, as shown in Figure 3.7(a). These multiplexers enable the shifting of data from `window[m][n+S]` to `window[m][n]` for arbitrary values of S, assuming that the stride length S is smaller than or equal to the window size K. By appropriately configuring the select signals of the multiplexers, a different



(a) LeapConv with Reconfigurable Stride



(b) Configuration for $S = 2$



(c) Configuration for $S = 3$

Figure 3.7: The (a) overall architecture of LeapConv and the reconfigured design that computes strided convolutions with (b) $S = 2$ and (c) $S = 3$.

stride length may be chosen. The connectivity for a specific stride length should be configured before the start of the computation and, for correctness, it should not change until the output data is computed.

To support an arbitrary stride, each column of the window buffer is connected to all previous columns. Therefore, the cost of multiplexing increases progressively from left to right. Multiplexers are also added to the write port of the line buffers to ensure that `lb[m][j]` receives an input from `lb[m+S][j]` (as listed in Algorithm 4), for all possible values of S. The read port of each line buffer is connected directly to the window buffer. The column of the window buffer that receives the output of the line buffers – only during the active period – is also determined by S. Example configurations for $S = 2$ and $S = 3$ are shown in Figures 3.7(b) and 3.7(c), respectively.

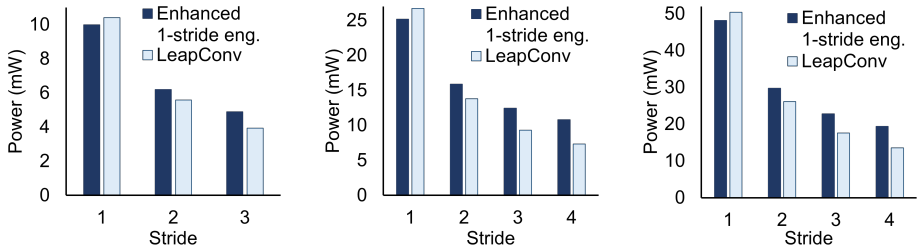
3.1.2 Evaluation

The goal of the evaluation process is to highlight the effectiveness of LeapConv, as compared to current state-of-the-art approaches. To the best of our knowledge, LeapConv is the *first* streaming convolution engine that is also optimized for strided convolutions. Thus far, strided convolutions have been optimized only for large-scale *systolic arrays* using Winograd's algorithm to reduce the cost of multiplications in convolutions [91, 138, 140]. In these approaches, the strided convolution is

computed via smaller convolution kernels mapped to Winograd-specific units. Since each Winograd unit supports a specified kernel size, the decomposed filters are zero-padded to be aligned with the predefined kernel [138]. Moreover, the systolic way of computing the final output does not allow for buffer sharing and regular data movements, as facilitated by streaming convolution engines. Thus, a comparison between a streaming convolution engine, such as LeapConv, and any systolic-array-based design would not provide meaningful insight, or be fair, since the two architectural approaches are very different in both concept and implementation.

Instead, the enhanced version of the unity-stride streaming architecture presented in Section 2.2.2 for computing the strided convolution remains cost efficient (even if it requires larger data transfers than LeapConv) and it does not suffer from irregularity in data accesses and lack of local buffer sharing, which afflict the systolic-architecture-based approaches [138, 140]. Hence, in the presented evaluation, we will compare LeapConv to the above-mentioned enhanced 1-stride streaming convolution engine, which serves as the benchmark for the simplest possible architectural alternative.

Both architectures were fully implemented in C++ and synthesized to Verilog RTL using Catapult HLS. The two architectures were designed for 16-bit input images and filters and are reconfigurable with respect to stride length. For each presented example, the size of the input images is assumed to be equal to 256×256. The designs follow the behavioral models listed in Algorithms 1 and 4. However, the C++ models used were optimized for HLS using coding templates that favor efficient unrolling and reduced dependencies for efficient pipelining. The Verilog RTL for each case was synthesized with the Oasys RTL logic synthesis using a 45 nm standard-cell library and targeting a clock frequency of 500 MHz. The reported power is obtained from the PowerPro power analysis and optimization tool.



(a) 3×3 (b) 5×5 (c) 7×7

Figure 3.8: The power consumption of the LeapConv and the enhanced unity-stride streaming engine implementations for different filter sizes and stride lengths.

The power consumption of the two architectures is illustrated in Figure 3.8 for different kernel sizes. For standard convolutions, where a unity stride is assumed, LeapConv incurs higher power consumption than the enhanced version of the unity-stride streaming architecture. This power overhead is a direct consequence of the reconfigurability provided by LeapConv and the extra multiplexing logic added to support it. Since the amount of the multiplexing is proportional to the size of the window buffer, the difference in power consumption between the two architectures increases as the size of the window buffer increases. This power overhead ranges from 4% to 6%, depending on the filter size.

However, LeapConv achieves a substantial reduction in power consumption when the engine computes convolutions with stride lengths greater than one, as shown in Figure 3.8. The high inactivity of the window buffer and the efficient on-time activation of the line buffers allow LeapConv to significantly reduce the data switching activity. This translates to dynamic power savings that increase with increasing stride length. This result stems from the fact that the periods of inactivity of the

window buffer and the number of active line buffers in each clock cycle are determined by the stride length. For the 3×3 kernel implementation, the reduction for a convolution with stride length 2 is around 10%, while it can reach up to 20% for longer strides. For the 5×5 and 7×7 kernel implementations, the power savings range from 13% to 32%. These savings are significant if we take into account the overhead of the multiplexing logic to support the desired reconfigurability.

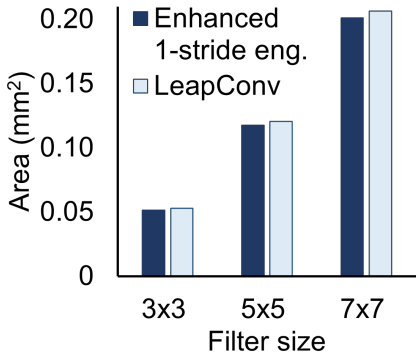


Figure 3.9: The area consumed by LeapConv and the enhanced unity-stride streaming architectures.

Figure 3.9 depicts the area of both designs under comparison for various kernel sizes. LeapConv is only marginally larger than the standard architecture of Section 2.2.2, which – as previously mentioned – is the most efficient approach in implementing streaming convolutions, since it maximizes buffer sharing and relies on simple data access patterns. The source of this area overhead is the added multiplexing logic and wiring required to support the extra feature of stride reconfigurability. The area overhead increases slightly with the window buffer size, and ranges between 2.3% and 2.8%, for the different implementations.

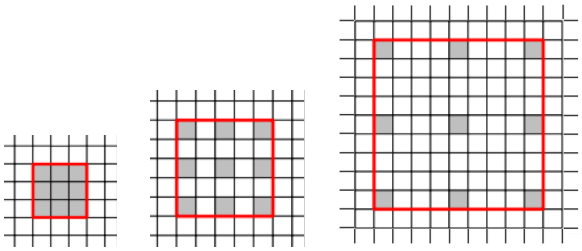
Finally, it should be noted that the area cost of multiplexing does *not* translate to a delay overhead, since the multiplexers drive the input pins of the window's registers. The critical path in all cases starts from the output pins of the same registers and moves to the multiplication and addition logic that implements the arithmetic part of the convolution engine. In other words, LeapConv can achieve the *same* maximum possible operating frequency as the baseline design.

3.2 Optimized Buffering for Dilated Convolutions

Standard 2D convolution assumes that each filter slides across the pixels of an input to produce a filtered output. In an effort to increase the receptive field [16, 19], the filter's coefficients are spaced out R elements apart in dilated convolution. Two examples of an inflated kernel are shown in Figure 3.10. For $R = 1$, dilated convolution is the same as standard convolution.

This section focuses on the design of a power-efficient dilated convolution engine that optimizes the data movement to reduce power consumption without obstructing the uninterrupted pipelined flow of data across consecutive engines in an SPD accelerator, and without relying on any data reorganization, or any irregular data fetching from any DMA engine. The discussed design, called LazyDCstream, does not alter the architecture of the streaming engines themselves. Instead, it optimizes their buffer usage when performing dilated convolutions, without restricting any other orthogonal optimizations that may be employed (e.g., datapath unrolling).

As discussed in Section 2.2, an SPD accelerator is composed of a series of streaming convolution engines, where each engine utilizes the line buffers, a window buffer, and an arithmetic unit that



(a) $R = 1$ (b) $R = 2$ (c) $R = 4$

Figure 3.10: The kernel in a dilated convolution for varying dilation rates.

performs the MAC operations. This architecture can be customized to various performance requirements [82, 142]. Unrolled architectures using wide enough window and line buffers and multiple MAC operators can store data from multiple input feature maps and compute multiple output features per clock cycle. Other area-efficient architectures allow buffer sharing by producing only a part of the output features.

Computing dilated convolution for a reference streaming convolution engine would require more line buffers and a larger window buffer. In dilated convolution, the filter's size is artificially increased according to the dilation rate. For a $K \times K$ kernel and a dilation rate of R, the streaming convolution engine would need $(K-1)R$ line buffers and larger window buffers for all input features. Input data would be continuously shifted in and out of the window buffers following the same access pattern. However, in some clock cycles, those pixels would be multiplied with zero coefficients without affecting the output. The rows of the window buffers that contain just the holes of the inflated kernel can be completely removed, thus simplifying the reference architecture.

LazyDCstream utilizes the decomposition of dilated convolution to multiple non-dilated convolutions [129] to:

- Minimize the amount of buffering needed to support dilated convolutions by appropriate time-sharing of buffers;
- Improve clock gating efficiency by limiting data switching activity during the engine's operation to the absolute minimum needed for the algorithm's correctness. In fact, data movements per cycle are equal to the number of kernel coefficients and are *independent of the dilation rate.*

To highlight the effectiveness of LazyDCstream in reducing power consumption, its structure was embedded in an SPD accelerator that executes inference on a variant of VGG-16 [96, 114] that includes dilation in all CNN layers. The experimental results demonstrate that LazyDCStream can reduce the power consumption of various CNN layers by 15–39% for ASIC implementations and 2–15% for FPGA implementations, without incurring any throughput or area penalty.

3.2.1 LazyDCstream: Architectural Overview

Dilated convolution can be decomposed to multiple standard convolutions applied on different parts of the input [129]. This decomposition is highlighted in Figure 3.11 for a 3×3 kernel applied using a dilation rate of $R = 2$. Each one of the four smaller convolutions (channels) receives a subset of the input and the same non-dilated 3×3 kernel. For the example shown in Figure 3.11 the input is broken down into four channels: A, B, C and D. A and B consist of data that are found in the even

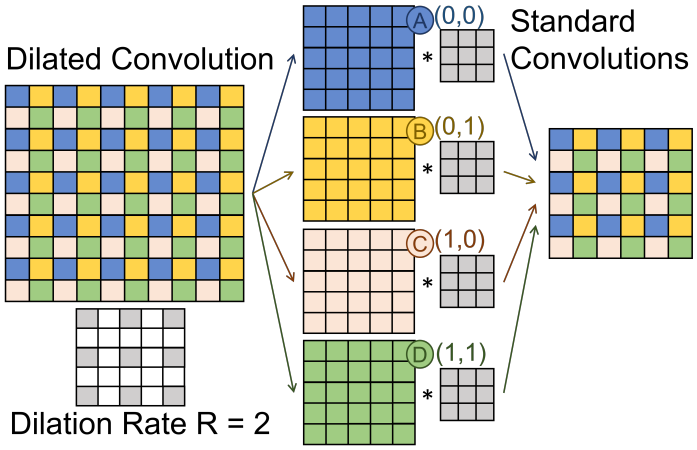


Figure 3.11: The decomposition of a dilated convolution with $R = 2$ to multiple independent non-dilated convolutions as proposed in [129]. Each channel works on the non-dilated version of the kernel applied on parts of the original input.

rows of the input, while C and D consist of data from the odd rows. Equivalently, A and C contain data from the even columns of the input, while B and D contain data from the odd columns.

In the general case, a dilated convolution of rate R is split to R^2 channels. Input (i, j) of the input belongs to channel (k,l) where $k = i \mod R$ and $l = j \mod R$. In the example of Figure 3.11, channels A, B, C, and D represent channels $(0,0)$, $(0,1)$, $(1,0)$, and $(1,1)$, respectively.

Decomposed dilated computation on time-shared hardware

Adopting "as is" the decomposition of a dilated convolution [129] means that it can be computed using multiple standard streaming convolution engines each one operating on the original "small" kernel and a subset of the input.

Although such an approach leads to a functionally-correct design, it has a lot of redundancy. The multipliers and adders that compute the convolution operation can be shared across channels. Each arriving data item belongs to one of the four channels of Figure 3.11. Therefore, in only one of the channels data will need to move (get shifted in the window buffer and move across line buffers). As a result, in this cycle, only one channel needs to compute a new output. To share the multipliers and adders, we insert multiplexing logic at the input of the multipliers to pick the correct data depending on which channel is active. This organization is shown in Figure 3.12 for a 3×3 kernel and a dilation rate of $R = 2$. Each channel consists of a 3×3 window buffer and two smaller line buffers, since each channel performs a convolution on a subset of the input.

In our example, the line buffers of channel A contain data that come from the even rows and even columns of the input. In the case of channel B, line buffers store data from the even rows and odd columns. Since data arrive in a row-wise manner, the line buffers of channel A and B will contain data from the same rows but from different columns. This observation allows us to merge the line buffers from the two channels into a bigger double-size line buffer (the size of the merged line buffer is no more than the one used for reference non-dilated convolution). Data are stored in the line buffers in an interleaved manner; data from even columns are stored in even addresses and those from the odd columns are stored in the odd addresses. The same merging can be done for channels C and D as shown in Figure 3.13. The demultiplexers are used to align the data transfer from the

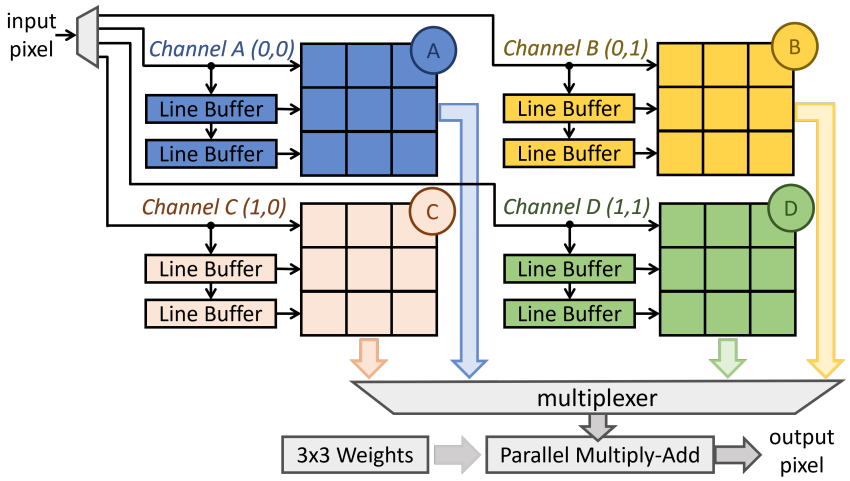


Figure 3.12: The decomposition of dilated convolution allows its computation using multiple smaller standard convolution engines that can share a common parallel multiply-add unit.

line buffers to the window buffers based on the channel that the incoming data item belongs.

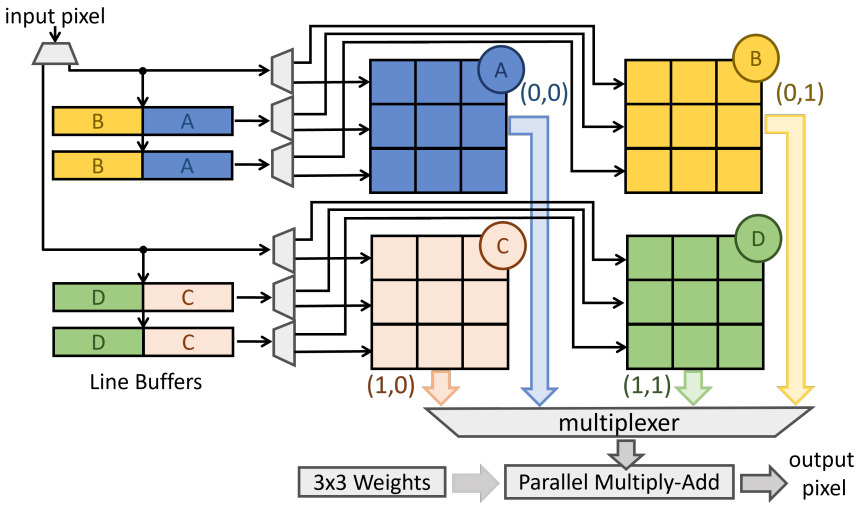


Figure 3.13: Line buffers that correspond to channels of the same group of rows can be merged to avoid memory fragmentation. The size of the merged line buffers is equal to the size of a standard convolution engine.

The streaming mode of data arrival allows our architecture to be further optimized. In the running example for $R = 2$, when reading a specific input row, only two of the four channels will be active. For the even-numbered rows channels A and B will be active and C and D will be inactive. The opposite holds for the odd-numbered rows. This means that when a data item arrives from a new row, the channels that correspond to that row will start pushing data into their window and line buffers, while the rest will stay inactive. This means that *only two window buffers in case of* $R = 2$

are necessary to compute dilated convolution. Those two window buffers will be used either by channels A and B or channels C and D depending on the index of the active row.

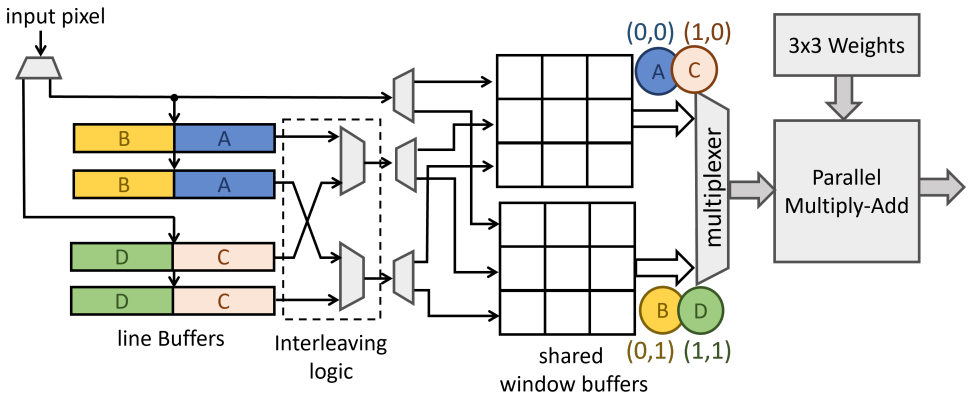


Figure 3.14: The organization of LazyDCstream for a 3×3 kernel and a dilation rate $R = 2$ that consists of merged line buffers, time-shared window buffers and a single multiply-add datapath.

To achieve the sharing of the window buffers among all channels we add extra multiplexing logic at the output of line buffers that enables the sharing of the window buffers between the different channels. This is shown in Figure 3.14 where the four line buffers are interleaved depending on the row index of the current input. Line buffer to window buffer connectivity has not changed when moving from Figure 3.13 to 3.14. In both cases, the upper line buffers of all channels feed the middle row of the corresponding window buffers, while the lower line buffers feed the last row of the window buffers. Multiplexing logic just selects which line buffers would feed the the corresponding rows of the shared window buffers.

In the general case, we need $(K-1)R$ line buffers that are active in groups depending on the row index of the input. We need, as well, R time-shared window buffers, each one holding the original small kernel of $K \times K$ input data.

Lazy data movement

The channels of the decomposed dilated convolution operate in groups in a mutually exclusive way based on the the row and column indexes of the incoming input. R window buffers are time shared across the R active channels. In each cycle, however, only one window buffer will be active shifting data and computing output data. The other window buffers can be clock gated. At most $K \times K$ data movements will occur in each cycle that correspond to the size of the original (non-inflated) kernel. This amount of data transfers per clock cycle is constant and, most importantly, independent of dilation rate.

Figure 3.15 illustrates an example on how data move inside the two time-shared window buffers in the case of 2-dilated convolution for a 3×3 kernel. In cycle t_n, channel A is activated as the new inputs A_7, A_4 and A_1 arrive from the input channel and the line buffers. The window buffer shifts its content to the right. During this time, the third column of the upper window buffer as well as the second and third columns of the lower window buffer have not yet received any input. In fact, the lower window buffer does not receive any inputs in this cycle and thus remains clock gated. On the contrary, in cycle t_{n+1}, channel B is activated and receives input $\{B_7, B_4, B_1\}$ from the corresponding

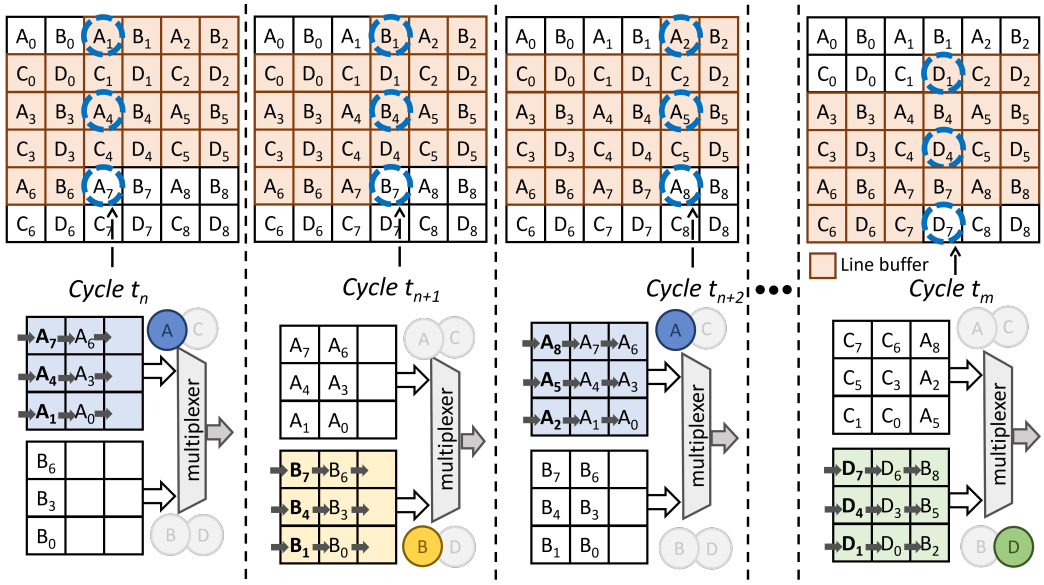


Figure 3.15: A snapshot of four cycles (three consecutive cycles and one at a later time) of the operation of LazyDCstream that highlights the reduced data movement and the clock gating of the registers of the inactive windows.

line buffers and the input to its first column. The rest of its columns are shifted right, following the operation of standard convolution. In this cycle, the upper window buffer is clock gated. In the next cycle t_{n+2}, channel A is activated again while channel B gets deactivated. The same operation is repeated until the end: the two window buffers are activated in turns leading to nine data elements moving per cycle.

When a new row is being read from the input, as in cycle t_m, the two window buffers are used to accommodate the input from channels C and D. During this time, the outputs have no dependency to input data from channels A and B and therefore the data inside the window buffers are overwritten.

Apart from the window buffer, the amount of data transfers per clock cycle between the line buffers is also constant. On each new input, only one channel is activated by writing also data across line buffers (like shifting data downwards). As shown also Figure 3.14, each channel consists of $K-1$ line buffers. Thus, $K-1$ writes will occur in each clock cycle again independent of the dilation rate.

Although LazyDCstream is optimized for dilated convolutions, it can also compute standard convolutions in parallel. For instance, a design that supports a maximum dilation rate of R can utilize the R window buffers available for computing R standard convolutions in parallel.

3.2.2 Evaluation

To highlight the benefits of LazyDCStream, we designed two spatial dataflow accelerators that both execute inference on a variant of VGG-16 [96] that employs dilation in all CNN layers. The first accelerator utilizes LazyDCStream as its streaming convolution engine, while the second one is a state-of-the-art architecture, as employed in [131, 142]. As suggested in [142], even if each CNN layer can employ a separate datapath unroll factor, it is safe to use a uniform unroll factor for all layers. For both designs under comparison, we employed a (4,16) unroll factor for the input and

output features, respectively. Therefore, both designs exhibit the same amount of parallelism per layer and, thus, have the same total execution time. They only differ in their *buffering architecture* for dilated convolution layers.

All designs have been implemented in C++ and synthesized to Verilog RTL using Catapult HLS. The buffers in the reference state-of-the-art design – as used in [55, 131] – operate directly on the inflated kernel and utilize the same number of line buffers as LazyDCstream. Using a carefully-designed C++ model for the reference state-of-the-art, hereafter referred to as "Reference," allowed Catapult HLS to keep the same multiply-add units needed for a non-inflated kernel. In this way, the area complexity of both the Reference and LazyDCstream designs is almost the same.

ASIC implementation results

Figure 3.16 reports the average power consumption of LazyDCStream and the Reference design for each CNN layer of the modified VGG-16 [96]. The modified VGG, similar to the original one, is split in blocks of two or three CNN layers of the same dilation rate. The dilation rate increases exponentially across blocks. Using dilation in all CNN layers (except the first two) enables us to better quantify the expected savings when using LazyDCStream. The numbers for each CNN layer also include the power consumed in padding and in the activation stages that accompany each convolution layer. Pooling and fully-connected layers are not shown, since they consume only a small part of the total power.

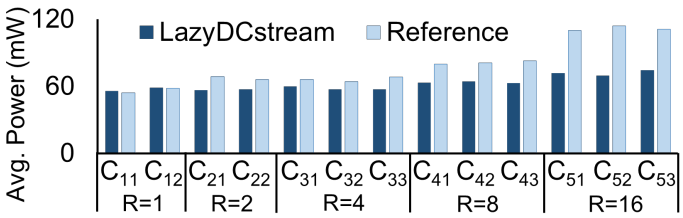


Figure 3.16: The average power consumption of the Reference and LazyDCstream architectures for the CNN layers of a modified VGG-16 [96].

Power consumption – both dynamic and leakage – was estimated after logic synthesis using the PowerPro power analysis and optimization tool for a 45 nm standard-cell library. Both designs operate at 500 MHz with 16-bit fixed-point inputs and weights. Switching activity information was gathered after feeding the modified VGG-16 with sample images from ImageNet. Verilog RTL code was derived from C++ using Catapult HLS and synthesized using the Oasys logic synthesis tool. The line buffers are mapped to SRAM macro blocks to minimize the area of the convolution engines.

As can be seen in Figure 3.16, LazyDCstream is more power efficient in all layers that involve dilation, mainly due to its reduced data movement and its time-shared window buffers. Power savings range between 15%, for layers with R=2, up to 39%, for the last block with R=16. Since all CNN layers utilize the same datapath unroll factor, they exhibit almost the same power for the same dilation rate. The differences are only due to data switching activity. For the Reference design, the power consumption increases with increasing dilation rate. On the contrary, as expected, LazyDCStream is only marginally affected by the increase in dilation rate.

Even though LazyDCstream requires additional multiplexing logic to implement the time sharing of the window buffers, the area overhead is negligible. The area is dominated by the line buffers and the parallel multiply-and-add units that are the same for both the LazyDCstream and Reference designs.

FPGA implementation results

Similar conclusions are drawn when considering the FPGA implementations of the same Reference and LazyDCstream architectures. We implemented each CNN layer separately on the Virtex Ultrascale VCU108 Evaluation Board, targeting a clock frequency of 150 MHz. The results were obtained after mapping the Verilog RTL code produced by Catapult HLS to the FPGA using Xilinx Vivado 2021.1.

The average power consumption numbers – for each layer of the modified VGG-16 – in the two compared designs are shown in Figure 3.17. The obtained results clearly indicate that LazyDCstream computes dilated convolution with less power. Depending on the dilation rate of each layer, the power savings range between 2% (for R=2) to 15% (for R=16), while, in both cases, static power consumption contributes around 900 mW to the overall consumption. The key takeaway point is that, unlike the Reference design, LazyDCstream's power consumption is independent of the dilation rate.

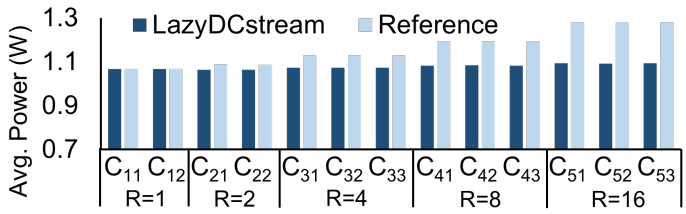


Figure 3.17: The power consumption of the Reference and LazyDCstream architectures when implementing the modified VGG-16 layers on the Virtex Ultrascale VCU108 Evaluation board at 150 MHz.

Regarding the area consumption, Table 3.1 reports the utilization of FPGA resources for two layers of the modified VGG-16 [96] for the two architectures. Both designs utilize the same number of DSP and BRAM blocks, since they employ the same number of multipliers and adders and the same number of line buffers. In each case, 576 DSP blocks are used to implement the parallel MAC operations of 4×16 unrolled 3×3 filters, while the number of BRAM blocks differs per layer, as the dilation affects the number of line buffers in the design. The time-shared operation in LazyDCstream has two contradictory effects with respect to FPGA utilization: it increases the number of LUTs required to implement the multiplexing logic, while it reduces the amount of registers needed to implement the window buffers.

Table 3.1: The utilization of FPGA resources for Reference and LazyDCstream for two layers of the modified VGG-16.

Layer	R	Architecture	LUTs	FFs	DSP	BRAM
C21	2	LazyDCstream	14688	5862	576	96
		Reference	14085	6505	576	96
C41	8	LazyDCstream	15384	7033	576	144
		Reference	14256	8160	576	144

3.3 Conclusions

This section was focused on the exploration of architectures that can improve the efficiency of the SPD accelerators, when computing spatial convolution variants like strided and dilated ones.

On one hand, strided convolution can be inherently decomposed into a sum of multiple channels of unity-stride convolutions. The presented LeapConv architecture takes advantage of this decomposed form of computing convolutions of arbitrary stride length to improve the power consumption of the streaming convolution engine. In LeapConv, the result of each channel is computed separately, albeit by using the same merged hardware unit. Both the window and line buffers of a baseline convolution engine built for unity-stride convolutions can hold the input of each channel. Using appropriately selected data movements, the data of each channel is properly aligned in the window buffer, to allow for direct computation of the desired result. The active and inactive periods of computation enable reduced data switching activity and increased clock-gating efficiency for the registers of the window buffer. Finally, with the addition of multiplexing logic, LeapConv can also support reconfigurable stride lengths.

On the other hand, dilated convolution spreads the kernel's coefficient to a larger window that slides on the input, similar to traditional convolution. In this way, the receptive field of the applied filter is enlarged in a computationally efficient manner. The presented architecture of LazyDC-Stream, takes advantage of the "holes" inside the inflated filter of a dilated convolution to perform computation in time-shared streams. These streams operate in groups in a mutually exclusive way, thus requiring only one set of multiply-add units and R window buffers of size equal to the original non-inflated kernel. Most importantly, the data switching activity remains constant per clock cycle and independent of the dilation rate.

4 CNN specific Algorithm-based Fault Tolerance

Besides performance and energy-efficiency requirements, the increasing prevalence of CNNs in safety-critical systems also increases the need for building resilient CNNs as an essential piece in guaranteeing the correctness of inference applications [76, 97]. This combined need for high-performance computation and functional safety is prevalent in various application domains, such as automotive systems. Guaranteeing correct computation in the presence of random hardware faults is necessary for safety and possible standards compliance [104]. For instance, ISO 26262 functional safety compliance requires that systems must function correctly, with potentially unsafe faults detected and controlled to prevent a hazard [51]. Thus, compliant systems must have very high fault-detection capabilities.

Managing random hardware faults, such as soft [9] and hard errors [12], requires special hardware modules for fault detection [65] that allow faults to be detected on-line and rapidly, possibly within a few cycles of their occurrence, thus simplifying recovery. The importance of on-line error detection is further increased, if one considers the additional reliability constraints imposed by modern implementation technologies, including process variations, device wear-out, and aging [12]. The problem is accentuated in ultra-low-power applications that execute CNN models at the edge in low-voltage setups to enable always-on intelligence on mobile and Internet-of-Things (IoT) devices [75, 132].

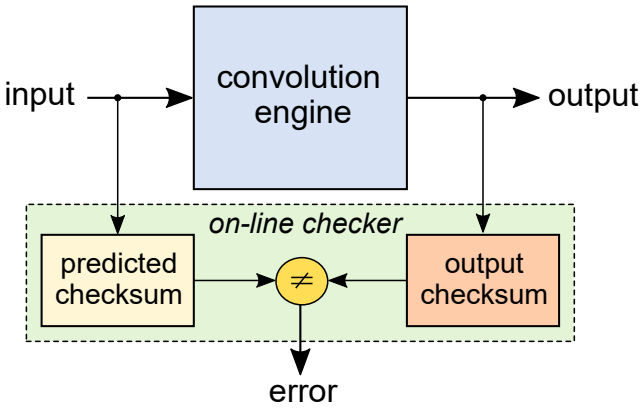


Figure 4.1: The online checksum checker operates in parallel to the convolution engine and compares the true and the predicted checksums of convolution.

Algorithm-Based Fault Tolerance techniques [53, 128] offer a low-cost mechanism to detect abnormal behavior in matrix-based computations [136] by comparing the true output checksum with a predicted one. Checksum computation and checking can be done either in software [18, 145], or in hardware [83]. In this work, we focus on *convolution-specific ABFT hardware checkers*.

In the case of a hardware online checker, as shown in Figure 4.1, the checker is attached to the input and the output of the convolution engine and computes the true and the predicted checksums that characterize the result of convolution. When the two checksums differ, an error flag is asserted.

The checker does not interfere with, or interrupts, the operation of the convolution engine, but simply provides online fault detection at the checksum level. Checksum checking cannot distinguish the correctness of every output pixel produced by the convolution engine. Instead, it checks if the sum of a group of output pixels matches the expected sum. In a similar vein, checking the result of convolution can be done using residue checking architectures [65, 95] that fall behind checksum-based hardware checkers, as shown in [83].

The prediction of the convolution checksum should be done in a cost efficient manner. In state-of-the-art approaches, such as [47, 83, 145], the predicted output checksum is computed explicitly using the same input pixels used for the actual convolution. Significant amount of computation is saved [83] by reusing efficiently the already computed checksums at the cost of additional buffering to store those reusable results.

On the contrary, ConvGuard, that is presented in this section, follows a different approach, enabled by a new fundamental property of convolution checksums. Instead of accumulating the input pixels used for the actual convolution, ConvGuard *predicts the output checksum of convolution implicitly* by accumulating only the peripheral pixels at the border of the input image that are dropped, or not computed, at the output. In this way, ConvGuard significantly reduces the power required for accumulating the input pixels, without requiring large buffers to store intermediate checksum results. Overall, the key contributions of ConvGuard can be summarized as follows:

- ConvGuard introduced a novel invariant condition for 2D convolutions and utilized it to predict implicitly the checksum of the convolution output. This alternative checksum computation can be computed rapidly with a low-cost hardware module that can easily track the performance (clock frequency and throughput) of the monitored convolution engine.
- The proposed checksum convolution checker can be configured to various convolution structures, including output size and stride. Especially in the case of non-unity stride convolutions, only useful input pixels are accumulated and no redundant computation is involved.
- The experimental results, using detailed hardware analysis of synthesized designs, highlight that ConvGuard utilizes only a small percentage of the area/power of an efficient convolution engine, while being significantly smaller than a state-of-the-art checksum checker [83]. The results scale well for increased image and filter sizes. Also, the minimum buffering requirements of ConvGuard reduce its susceptibility to bit-flip errors.

4.1 Prediction of Convolution Checksum

The convolution of an $R \times C$ image x with a filter h of size $K \times K$ is calculated as follows [42]:

$$y_{mn} = \sum_{i=0}^{K-1} \sum_{j=0}^{K-1} h_{ij} x_{m-i,n-j}, \tag{4.1}$$
$$m \in [0, P-1], \qquad n \in [0, Q-1]$$

Formally, the size of the output y is $P \times Q$, with $P = K + R - 1$ and $Q = K + C - 1$, and is larger than the input image. However, in practice, the output pixels on the border of the image may not be computed. In such cases, the output image is either of equal size to the input image, or, most often, smaller. Figure 4.2 depicts two possible convolution outputs (the pixels in blue) for a 3×3 input image and a 2×2 filter.

$$x = \begin{bmatrix} x_{00} & x_{01} & x_{02} \\ x_{10} & x_{11} & x_{12} \\ x_{20} & x_{21} & x_{22} \end{bmatrix} \quad h = \begin{bmatrix} h_{00} & h_{01} \\ h_{10} & h_{11} \end{bmatrix}$$

$$y_B = \begin{bmatrix} y_{00} & y_{01} & y_{02} & y_{03} \\ y_{10} & y_{11} & y_{12} & y_{13} \\ y_{20} & y_{21} & y_{22} & y_{23} \\ y_{30} & y_{31} & y_{32} & y_{33} \end{bmatrix} \qquad y_D = \begin{bmatrix} y_{00} & y_{01} & y_{02} & y_{03} \\ y_{10} & y_{11} & y_{12} & y_{13} \\ y_{20} & y_{21} & y_{22} & y_{23} \\ y_{30} & y_{31} & y_{32} & y_{33} \end{bmatrix}$$

Figure 4.2: Examples of output images as a result of the convolution of a 3×3 input image and a 2×2 filter. The useful output pixels are highlighted in blue. The rest are the extra outputs that should have been dropped, or not calculated.

In the case of y_D, convolution is performed only on the pixels of the input image without requiring any border padding [55]. Hence, the output image is smaller than the input.

Convolution can be equivalently expressed as a matrix-vector multiplication [117]:

$$\mathbf{y} = A\mathbf{h}_{vec} \tag{4.2}$$

Vector $\mathbf{h}_{vec}$ contains all the $\hat{K} = K \times K$ coefficients of the filter arranged one after the other in a row-wise fashion in one column. Output vector $\mathbf{y}$ contains all elements of convolution ($P \times Q$ in total) again in a row-wise fashion. For the multiplication to be valid, matrix A contains one row for each application of the filter to the input image, i.e., for each possible position of the sliding window, including the outer border. Therefore, since the $\hat{K}$ filter coefficients will be multiplied with an equal number of input pixels, matrix A consists of $\hat{K}$ columns.

For the convolution of a 3×3 input with a 2×2 filter, A consists of all elements of the input from where the filter h would slide over, assuming a zero-padded border:

$$A = \begin{bmatrix} 0 & 0 & 0 & x_{00} \\ 0 & 0 & x_{00} & x_{01} \\ 0 & 0 & x_{01} & x_{02} \\ 0 & 0 & x_{02} & 0 \\ 0 & x_{00} & 0 & x_{10} \\ x_{00} & x_{01} & x_{10} & x_{11} \\ x_{01} & x_{02} & x_{11} & x_{12} \\ x_{02} & 0 & x_{12} & 0 \\ 0 & x_{10} & 0 & x_{20} \\ x_{10} & x_{11} & x_{20} & x_{21} \\ x_{11} & x_{12} & x_{21} & x_{22} \\ x_{12} & 0 & x_{22} & 0 \\ 0 & x_{20} & 0 & 0 \\ x_{20} & x_{21} & 0 & 0 \\ x_{21} & x_{22} & 0 & 0 \\ x_{22} & 0 & 0 & 0 \end{bmatrix} \tag{4.3}$$

4.1.1 An Invariant Condition for Convolution Checksum

Since $\mathbf{y} = A\mathbf{h}_{vec}$, and $A = [a_{ij}]$, every element y_i of $\mathbf{y}$ is computed as

$$y_i = \sum_{j=0}^{\hat{K}-1} a_{ij} h_j^{vec} \tag{4.4}$$

Summing all y_i's yields:

$$\sum_{i=0}^{PQ-1} y_i = \sum_{i=0}^{PQ-1} \sum_{j=0}^{\hat{K}-1} a_{ij} h_j^{vec} \tag{4.5}$$

By rearranging the order of the two sums in (4.5), we get:

$$\sum_{i=0}^{PQ-1} y_i = \sum_{j=0}^{\hat{K}-1} \sum_{i=0}^{PQ-1} a_{ij} h_j^{vec} = \sum_{j=0}^{\hat{K}-1} \left(\sum_{i=0}^{PQ-1} a_{ij} \right) h_j^{vec} \tag{4.6}$$

The sum inside the parentheses of (4.6) corresponds to the sum of the elements of the jth column of A. It can easily be observed in (4.3), and proven in the general case, that the sum of the input pixels of each column of A is the same for all columns and equal to the sum of all pixels of the input. Therefore, we can replace $\sum_{i=0}^{PQ-1} a_{ij}$ with $\sum_{k=0}^{R-1} \sum_{l=0}^{C-1} x_{kl}$. Based on this observation, we can write:

$$\begin{aligned} \sum_{i=0}^{PQ-1} y_i &= \sum_{j=0}^{\hat{K}-1} \left(\sum_{k=0}^{R-1} \sum_{l=0}^{C-1} x_{kl} \right) h_j^{vec} \\ &= \left(\sum_{k=0}^{R-1} \sum_{l=0}^{C-1} x_{kl} \right) \left(\sum_{j=0}^{\hat{K}-1} h_j^{vec} \right) \end{aligned} \tag{4.7}$$

In other words, in (4.7) we have shown that, *the sum of all output pixels of the convolution is equal to the product of the sum of all input data elements x_{kl} with the sum of all the filter's coefficients*.

Let S_y denote the set of indices that support image y and S_h, S_x the indices that support h and x, respectively. The invariance condition (4.7) becomes

$$\sum_{i \in S_y} y_i = \left(\sum_{k \in S_x} x_k \right) \left(\sum_{j \in S_h} h_j \right) \tag{4.8}$$

The set S_y can be divided into two sets S_y^{xtr} and S_y^{crp} that denote the pixel indices that are cropped from the original image, and the pixel indices that remain in the cropped image, respectively. The set of S_y^{xtr} is not fixed and it represents all pixels that are left off, depending on the choice of the useful output and the structure of the convolution. It is straightforward to see that $S_y = S_y^{\text{crp}} + S_y^{\text{xtr}}$. Thus, (4.8) becomes:

$$\sum_{i \in S_y^{\text{crp}}} y_i + \sum_{i \in S_y^{\text{xtr}}} y_i = \left(\sum_{k \in S_x} x_k \right) \left(\sum_{j \in S_h} h_j \right) \tag{4.9}$$

Let us see an arithmetic example of this newly introduced invariance condition for the convolution of a 3×3 input image x with a 2×2 filter h:

$$x = \begin{bmatrix} 1 & 1 & 2 \\ 1 & 1 & 2 \\ 1 & 1 & 2 \end{bmatrix} \quad h = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

According to (4.1) the complete output consists of 4×4 pixels, as follows:

$$y = \begin{bmatrix} 1 & 3 & 4 & 4 \\ 4 & 10 & 14 & 12 \\ 4 & 10 & 14 & 12 \\ 3 & 7 & 10 & 8 \end{bmatrix}$$

Depending on which output pixels are considered useful, invariant condition (4.9) would take a different form. For the case of unity-stride convolutions, and assuming that convolution is performed only on the pixels of the input image without padding (like case y_D in Figure 4.2), the useful pixels that will actually be computed by the convolution engine are the ones highlighted in blue. Inevitably, the remaining pixels at the periphery of the output image are the extra pixels that *will not be* computed by the convolution engine. The sum of the highlighted outputs is equal to $\sum y_{\text{crp}} = 48$, while the sum of the unused outputs $\sum y_{\text{xtr}} = 72$. In all cases, according to (4.9), the sum of the two disjoint sets of pixels ($48+72=120$) is equal to the product of sums $(\sum x_k)(\sum h_j) = 12 \times 10$.

4.1.2 Explicit and Implicit Prediction of the Output Checksum

An online checksum checker, similar to the one shown in Figure 4.1, would accumulate all useful output pixels coming out of the convolution engine and compare the derived checksum with the predicted one. Predicting the output checksum either explicitly, or implicitly, means to re-compute $\sum_{i \in S_y^{\text{crp}}} y_i$ directly from the input without interfering at any point with the convolution engine.

The useful output pixels y_{crp} and the extra ones y_{xtr} can both be computed according to (4.2), as follows:

$$y_{\text{crp}} = A^{\text{crp}} \mathbf{h}_{vec} \qquad y_{\text{xtr}} = A^{\text{xtr}} \mathbf{h}_{vec} \tag{4.10}$$

Matrices A^{crp} and A^{xtr} contain only the rows of A that correspond to each disjoint set of outputs. For our running example (case y_D in Figure 4.2),

$$A^{\text{crp}} = \begin{bmatrix} x_{00} & x_{01} & x_{10} & x_{11} \\ x_{01} & x_{02} & x_{11} & x_{12} \\ x_{10} & x_{11} & x_{20} & x_{21} \\ x_{11} & x_{12} & x_{21} & x_{22} \end{bmatrix} A^{\text{xtr}} = \begin{bmatrix} 0 & 0 & 0 & x_{00} \\ 0 & 0 & x_{00} & x_{01} \\ 0 & 0 & x_{01} & x_{02} \\ 0 & 0 & x_{02} & 0 \\ 0 & x_{00} & 0 & x_{10} \\ x_{02} & 0 & x_{12} & 0 \\ 0 & x_{10} & 0 & x_{20} \\ x_{12} & 0 & x_{22} & 0 \\ 0 & x_{20} & 0 & 0 \\ x_{20} & x_{21} & 0 & 0 \\ x_{21} & x_{22} & 0 & 0 \\ x_{22} & 0 & 0 & 0 \end{bmatrix}$$

From (4.10), we can compute the elements of y_{crp} and y_{xtr}, as follows:

$$y^i_{\text{crp}} = \sum_{j=0}^{\hat{K}-1} a^{\text{crp}}_{ij} h_j \qquad y^i_{\text{xtr}} = \sum_{j=0}^{\hat{K}-1} a^{\text{xtr}}_{ij} h_j \tag{4.11}$$

To compute the checksum of the useful output pixels, we need to sum all elements y^i_{crp}. Let us assume that the number of useful output pixels is equal to M:

$$\begin{aligned} \sum_{i=0}^{M-1} y^i_{\text{crp}} &= \sum_{i=0}^{M-1} \sum_{j=1}^{\hat{K}-1} a^{\text{crp}}_{ij} h_j = \sum_{j=0}^{\hat{K}-1} \sum_{i=0}^{M-1} a^{\text{crp}}_{ij} h_j \\ &= \sum_{j=0}^{\hat{K}-1} \left(\sum_{i=0}^{M-1} a^{\text{crp}}_{ij} \right) h_j \end{aligned} \tag{4.12}$$

The derived equation (4.12) tells us how to *explicitly predict the output checksum* using only the pixels that participate in the convolution (i.e., the ones in the center of the input image that are

grouped in A^{crp}). To do so, we need to compute the sum of each column of A^{crp}, i.e., $\sum_{i=0}^{M-1} a_{ij}^{\text{crop}}$ for column j, and multiply the result with the corresponding filter coefficient. Then, we should reduce the derived partial products to one final predicted checksum.

Alternatively, with ConvGuard, we can compute *implicitly* the same sum of output pixels using the new invariance condition (4.9), which can be re-written as:

$$\sum_{i=0}^{K-1} y_{\text{crp}}^{i} = \left(\sum_{k\in S_x} x_k\right)\left(\sum_{j\in S_h} h_j\right) - \sum_{i=0}^{\hat{M}-1} y_{\text{xtr}}^{i} \tag{4.13}$$

Since the number of useful pixels is assumed to be equal to M, the number of extra pixels (zero and non-zero) is equal to $\hat{M} = PQ - M$. The sum of extra output pixels can be expressed similarly to (4.12), as follows:

$$\sum_{i=1}^{\hat{M}-1} y_{\text{xtr}}^{i} = \sum_{j=0}^{\hat{K}-1}\left(\sum_{i=0}^{\hat{M}-1} a_{ij}^{\text{xtr}}\right) h_j \tag{4.14}$$

Also, the product of sums $(\sum x_k)(\sum h_j)$ can be restructured as:

$$\left(\sum_{k\in S_x} x_k\right)\left(\sum_{j\in S_h} h_j\right) = \sum_{j=0}^{\hat{K}-1}\left(\sum_{k\in S_x} x_k\right) h_j \tag{4.15}$$

By replacing (4.14) and (4.15) in (4.13), we get

$$\begin{aligned}\sum_{i=0}^{M-1} y_{\text{crp}}^{i} &= \sum_{j=0}^{\hat{K}-1}\left(\sum_{k\in S_x} x_k\right) h_j - \sum_{j=0}^{\hat{K}-1}\left(\sum_{i=0}^{\hat{M}-1} a_{ij}^{\text{xtr}}\right) h_j \\ &= \sum_{j=0}^{\hat{K}-1}\left(\sum_{k\in S_x} x_k - \sum_{i=1}^{\hat{M}-1} a_{ij}^{\text{xtr}}\right) h_j\end{aligned} \tag{4.16}$$

Eq. (4.16) corresponds to the *implicit prediction of the output checksum*. Instead of directly using the central pixels of the input image, we accumulate the columns of A^{xtr} that consist only of peripheral pixels, i.e., $\sum_{i=1}^{\hat{M}-1} a_{ij}^{\text{xtr}}$ for each column j. Each one of those accumulated sums (one for each filter coefficient) is subtracted from a common sum that corresponds to the sum of all input pixels, irrespective of their position. Then, the derived differences are multiplied with their corresponding filter's coefficients and reduced to a final sum.

In the case of multiple filters, the same approach holds for each separate filter. In addition, the approach can be applied on the case of 3D convolution. Since 3D convolutions are commonly decomposed to depth-wise convolutions or pseudo-3D convolutions [139], it is straightforward to apply the above approach to each separable filter and check the result of the corresponding convolution.

For realistic image sizes and unity-stride convolutions, the number of extra pixels is much smaller than the useful ones. Therefore, choosing to accumulate the peripheral input pixels is expected to lead to overall fewer additions. This choice *is unique to ConvGuard* and a direct consequence of the invariance condition (4.9) introduced in this work.

4.2 On-line Checker Architecture

The architecture of ConvGuard is depicted in Figure 4.3. The checker module operates in parallel to the convolution engine, without interfering with its operation. ConvGuard monitors the input x and the output y of the engine and predicts implicitly the output checksum by computing online Eq. (4.16).

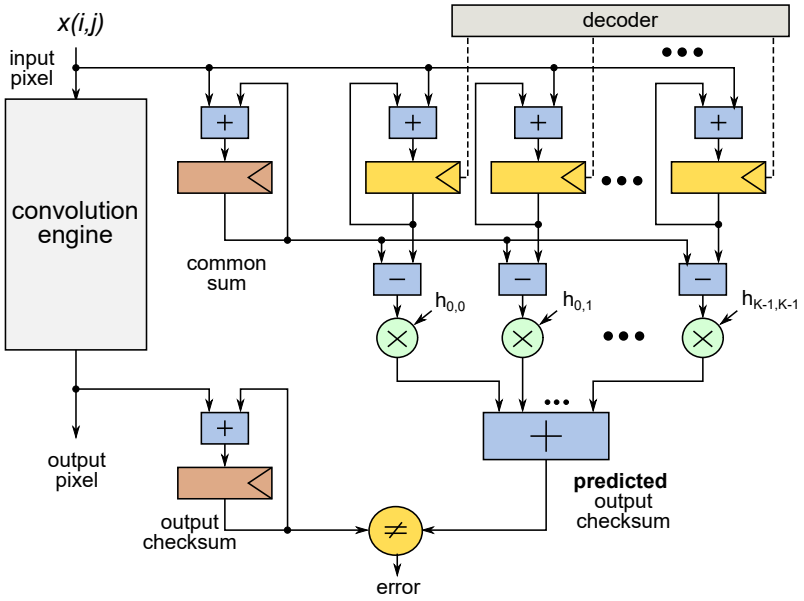


Figure 4.3: The organization of ConvGuard. It runs in parallel to the convolution engine and it receives the same input and the engine's output pixels. Convguard accumulates the sum of the output pixels and compares it to its predicted checksum value. The predicted output checksum utilizes a set of accumulators – one for each filter coefficient – and a common sum accumulator that computes the sum of all input pixels.

4.2.1 Checker Organization

In each cycle, ConvGuard performs two tasks. On the output side, it accumulates the valid output samples produced by the convolution engine. Recall that, without loss of generality, we assume that the convolution engine computes only the useful output pixels and does not produce any invalid output. If it does, it just needs to mark the pixels as invalid, so that ConvGuard can skip them. On the input side, to check the correctness of convolution, ConvGuard computes one sum for each column of A^{xtr} and a common sum of all input pixels. To do this, it employs one accumulator for each column of A^{xtr} ($\hat{K}$ in total) and one accumulator for the common sum.

Initially, all accumulators are reset to zero. Then, as each input pixel arrives, (one per cycle; more pixels can arrive per cycle after marginal design changes), it is added to the appropriate accumulator, while all of the incoming pixels are added to the common-sum accumulator. Depending on the arriving input pixels, multiple accumulators may be enabled in the same cycle. For instance, in our running example, when input pixel x_{00} arrives, it contributes to the running sum of accumulators that correspond to the filter's coefficients h_{01}, h_{10}, and h_{11}. On the contrary, when input pixel x_{20} arrives, only the accumulator of h_{11} is activated. It should be noted that central pixels – like x_{11} – that do not appear in A^{xtr} are skipped and not added to any accumulator besides the common one.

The decision to which accumulator each input pixel contributes is done in the decoder, which is also shown in Figure 4.3. The decoder decides two things: (a) which peripheral input pixels contribute to the computation of the extra output pixels, and (b) to which accumulator they should be added.

As shown in Figure 4.4, the extra output pixels consist of the $(K-1)/2$ rows and columns on the border of the output. To derive those outputs, a larger border of $K-1$ rows and columns is

actually used from the input image. The decoder would allow only those input pixels to be added to the appropriate accumulators. On the contrary, all input pixels are added to the common sum accumulator. Stated formally, an input pixel x_{ij} is added to the accumulator that corresponds to the filter's coefficient h_{mn} when at least one of the following inequalities is satisfied:

$$m > i > R - K + m \qquad n > j > C - K + n \tag{4.17}$$

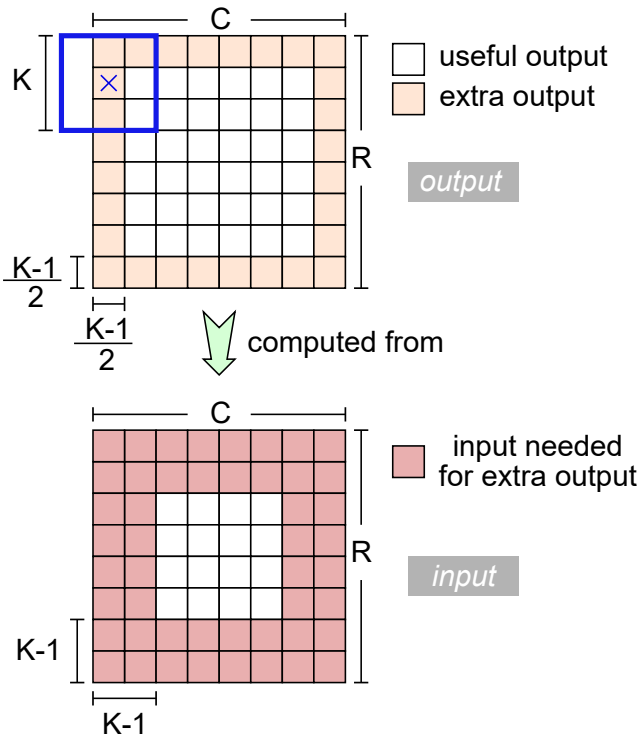


Figure 4.4: The peripheral pixels that should be added for the computation of the sum of extra output pixels. Each highlighted pixel may contribute to many accumulators in the same cycle, as dictated by the decoder function.

When all input pixels have passed through the convolution engine, the checker's accumulators have accumulated all needed sums: one common sum, and one sum for each column of A_{xtr}. At this point, the sum that corresponds to each coefficient is subtracted from the common sum, in order to correctly compute the term in the parentheses of (4.16). Then, each resulting term is multiplied with the corresponding filter's coefficient and the products are added to produce the final value, which corresponds to the predicted output checksum of the convolution.

For fixed-point implementations, which is the focus of this work, all registers and arithmetic units are sized appropriately so as to avoid any overflow conditions that would ruin the output checksum prediction. For checking a floating-point-based convolution engine, we cannot guarantee that the predicted output checksum would match the true output checksum, even under error-free operation. In these cases, the equality comparison should be transformed to a bounds check. If the predicted and the true output checksums differ by a certain small error bound, the convolution would still be considered fault free [10, 65].

Finally, it should be stressed that the prediction of the output checksum is computed gradually *without requiring any buffering of intermediate results*. This lack of buffering is critical in reducing

the cost of the checksum checker. It is expected that an on-line checker should consume only a small percentage of the area of the convolution engine and leave only an incremental energy footprint, as compared to the energy consumed in computing the actual convolution.

4.2.2 When does Implicit Prediction of the Output Checksum make Sense?

Predicting the output checksum implicitly using Eq. (4.16) is useful only when it can be computed with fewer additions, as compared to an explicit prediction of the checksum. To understand when the two approaches break even, we need to count the number of additions needed in each case. Equivalently, we need to count the number of non-zero elements of matrices A^{crp} and A^{xtr}, respectively.

In the case of explicit checksum prediction, A^{crp} consists of only non-zero elements. It has as many rows as the number of useful output pixels. According to Figure 4.4, the number of useful output pixels is $M = (R-K+1)(C-K+1)$ for odd values of K. The number of columns of matrix A^{crp} is always equal to the number of the filter's coefficients $\hat{K}$ (equal to K^2). Therefore, by multiplying the two, the number of additions required for the explicit computation of the checksum is:

$$\#\text{explicit adds} = \hat{K}M = K^2(R-K+1)(C-K+1) \tag{4.18}$$

On the contrary, the implicit checksum prediction has to do with all the remaining output pixels. Recall from Eq. (4.3) that each column of A contains all input pixels and some zero elements. Therefore, the non-zero elements of every column of A^{xtr} that should be added are equal to the number of all input pixels RC minus the elements of the same column of A^{crp}, i.e., $RC-(R-K+1)(C-K+1)$. Since there are $\hat{K}$ columns in A^{xtr}, the number of additions required is equal to

$$\hat{K}\,(RC-(R-K+1)(C-K+1))$$

By replacing Eq. (4.18) in the derived formula, we conclude that, to compute the sum of the extra output pixels, we need

$$\hat{K}RC - \#\text{explicit adds}$$

additions. The checker computes also a common sum that involves the sum of all input pixels. Therefore, ConvGuard requires RC more additions. Overall,

$$\#\text{implicit adds} = (1+\hat{K})RC - \#\text{explicit adds} \tag{4.19}$$

Using Eqs. (4.19) and (4.18), we can compare the efficiency of these two approaches for arbitrary image and filter sizes. The implicit approach proposed by ConvGuard requires fewer additions than the explicit approach when

$$\#\text{explicit adds} > \left(\frac{\hat{K}+1}{2}\right)RC \tag{4.20}$$

Figure 4.5 depicts the number of additions required in each case ("Explicit" and "Implicit") for a 3×3 and a 5×5 filter for various square input dimensions. For really small input images, it is more efficient – in terms of number of additions – to predict the output checksum explicitly. When the input image is larger, implicit prediction is more efficient that explicit prediction. For instance, for a 3×3 input filter, implicit prediction is more efficient for any input image larger than 8×8 pixels. The minimum input image size required to make implicit prediction more cost-efficient for various filter sizes and for stride $S=1$ is presented in the first column of Table 4.1. The presented sizes are encountered in many existing applications. For instance, VGG-16 [114] has an input image of size 224×224, which is convolved with a filter of size 3×3. Furthermore, the 2^{nd} to 5^{th} convolutional

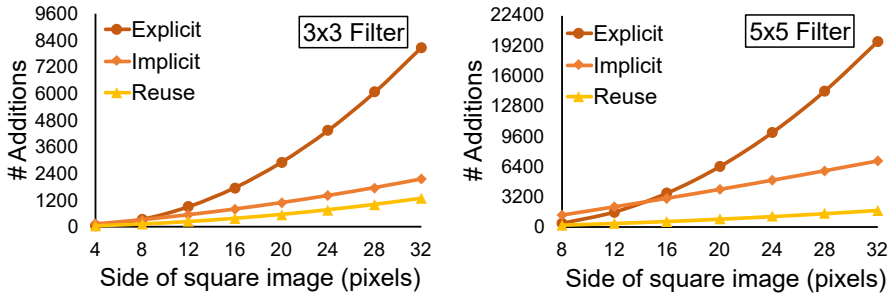


Figure 4.5: The number of additions needed for predicting the output checksum explicitly, implicitly and with maximum reuse of already computed results [83], for two filter cases and various square input image sizes.

layers of AlexNet [69] perform convolutions on images with sizes of 27×27 and 13×13, using 5×5 or 3×3 filters.

Fig 4.5 also shows the number of additions required by the state-of-the-art "Reuse" checksum checker, as presented in [83]. This approach relies on explicit prediction of the output checksum and reduces the total number of additions by reusing many of the already computed sums. However, the reduced number of additions comes at the cost of extra buffering to store the required intermediate results. As clearly shown in the evaluation in Section 4.4.2, this extra buffering significantly increases the total area and power of this checker, relative to ConvGuard. Moreover, the extra buffers make the checker more susceptible to random bit-flips that would lead to false detection alarms, as analyzed in Section 4.4.3.

4.3 Checking Non-Unity Stride Convolutions

In the non-unity stride convolutions found in many practical applications, the useful output pixels are even fewer. Furthermore, in these cases, extra pixels are present not only at the periphery of the image, but in the center as well. In such cases, predicting the output checksum implicitly would always require more additions than the explicit prediction. To enable the applicability of ConvGuard to non-unity-strided convolutions, we utilize a recently-proposed transformation [64, 91] that allows the computation of any convolution with stride $S > 1$ using multiple channels of unity-stride convolutions. By applying the implicit checksum prediction on each independent unity-stride channel, we can still design a low-cost checksum checker.

4.3.1 Checking Independently per Channel

In a unity-stride convolution, the filter is applied to every pixel of the input. On the contrary, in the case of a non-unity stride convolution, the filter moves on the input with a step of S. In this case, each input pixel will not be multiplied with every filter coefficient, but with a subset of them. Figure 4.6 groups the input pixels based on which filter's coefficient "touches" them. The blue input pixels will be multiplied only with the blue filter coefficients, while the green ones will be multiplied only with the green filter coefficient. Based on this observation, the work in [64, 91] proposed to compute any non-unity stride convolution by summing the result of S^2 smaller and independent unity-stride convolutions. The unity-stride convolutions are applied on selected sub-image and sub-filter pairs, as also shown in Figure 4.6.

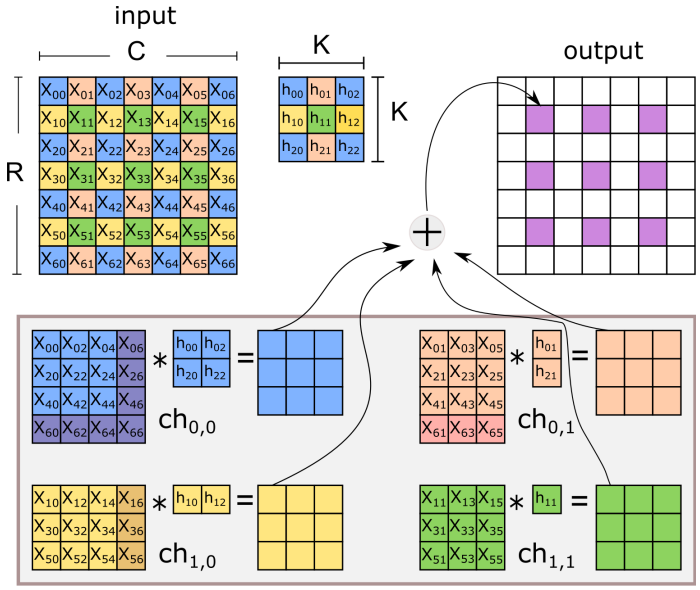


Figure 4.6: Transformation of a strided convolution with $S = 2$ to a 4-channel unity-stride convolutions, depicted with symbol $*$.

Being able to transform a non-unity stride convolution into multiple unity-stride ones allows us to apply ConvGuard efficiently to arbitrary strides. More precisely, ConvGuard predicts the output checksum implicitly using Eq. (4.16) separately, per channel. Since – according to [91] – the result of the each sub-convolution is added to form the final convolution result, then the final prediction of the output checksum is the sum of all intermediate implicit predictions.

4.3.2 Generalized Checker

The organization of generalized ConvGuard is illustrated in Figure 4.7. To compute Eq. (4.16) for each channel we need more accumulators. Since the number of the filter's coefficients does not change, the number of accumulators that sum the input pixels per coefficient remains the same as in the baseline case ($S = 1$). However, we need more than one common-sum accumulators. Since we compute a common sum for the sub-image of each channel, we need S^2 common-sum accumulators in total (one per channel). Thus, overall, for supporting convolutions with stride S, we need $S^2 + K^2$ accumulators.

For convolutions of arbitrary stride, decoding is a two-step procedure. The first step decides to which channel each pixel belongs, and the second step decides if it is a peripheral pixel of the channel's sub-image, or not. The first check determines to which channel's common sum accumulator the pixel should be added, and the second check (also using the result of the first check) decides to which filter coefficients the incoming pixel refers.

For the first check, the common sum accumulator of channel (k,l) is increased when, for the input pixel (i, j), the following hold: $k = i \mod S$ and $l = j \mod S$.

For the second check, we actually need to check if at least one of the inequalities in (4.17) holds after mapping the indices of the input pixels (i, j) and the filter's coefficients (m,n) to the "smaller" co-ordinates of each channel. The considered sizes for the sub-images and sub-filter should be scaled too. To achieve this, we merely need to integer-divide each variable of the inequalities with

the selected stride S.

Once the common sums per channels have been accumulated and the coefficient accumulators get their final values, the appropriate common sums are subtracted from the appropriate accumulators, as shown in Figure 4.7. The mask logic of Figure 4.7 decides the assignment by identifying the common sums and the filter coefficients that belong to the same channel.

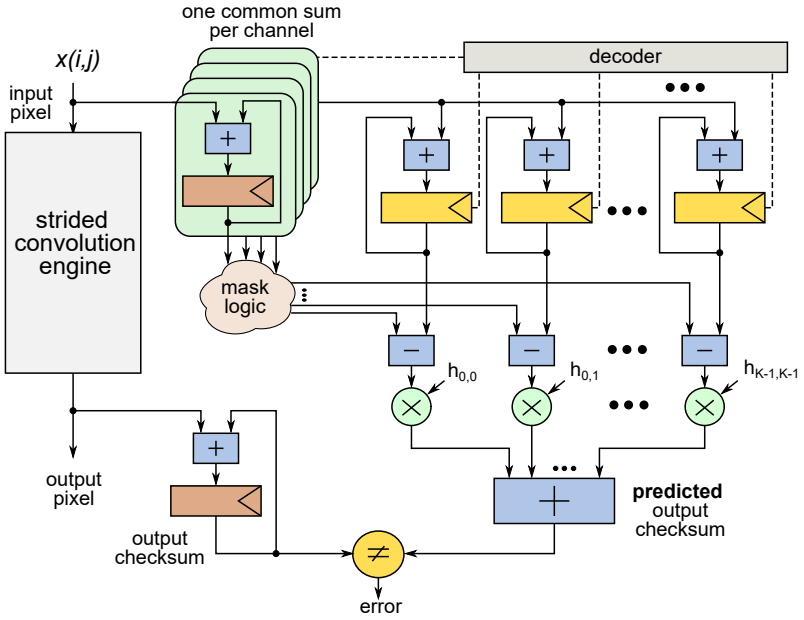


Figure 4.7: The generalized ConvGuard architecture that support arbitrary strided convolutions.

The number of additions required for the implicit prediction of the output checksum depends on the size of the input image, as well as the size of the filter. Additionally, in the case of a non-unity stride convolutions, the efficiency of ConvGuard also depends on the sizes of all sub-images and sub-filters that emerge after the transformation to multi-channel unity stride convolutions. Thus, the number of additions depends on the selected stride as well.

Table 4.1: The minimum size of the side of a square image that favors implicit over explicit prediction of the output checksum.

Filter	Stride - S			
	1	2	4	6
3×3	8	14	-	-
5×5	15	17	43	-
7×7	23	26	34	64
11×11	35	38	46	54

Table 4.1 shows the minimum number of pixels that an input image should have to make the implicit prediction of the output checksum more efficient than its explicit counterpart. For instance, for stride $S = 2$ and a filter or size 5×5 the input image should be at least 17×17 pixels, while for a larger 11×11 filter, the minimum image size increases to a 38×38 pixels. When the stride is larger than the filter, the multi-channel decomposition of the original strided convolution is degenerated.

Table 4.2: The area and power complexity of an application-specific convolution engine and ConvGuard operating at 1 GHz.

Filter	Image	Area (μm^2)		Power (mW)@ 1GHz	
		Engine	Checker	Engine	Checker
3×3	14×14	26908	9860	3.99	1.31
	28×28	32947	11278	4.90	1.53
	56×56	36678	12164	5.26	1.84
5×5	28×28	74104	18651	10.36	2.09
	56×56	100900	20184	11.16	2.35
	112×112	115380	22451	15.56	2.78
11×11	56×56	508873	63536	37.88	4.65
	112×112	544887	74463	54.95	5.37
	224×224	616364	68110	65.39	5.46

In this case, each channel may contain only one filter coefficient or none. Hence, in such cases, the differentiation between implicit and explicit prediction no longer makes sense.

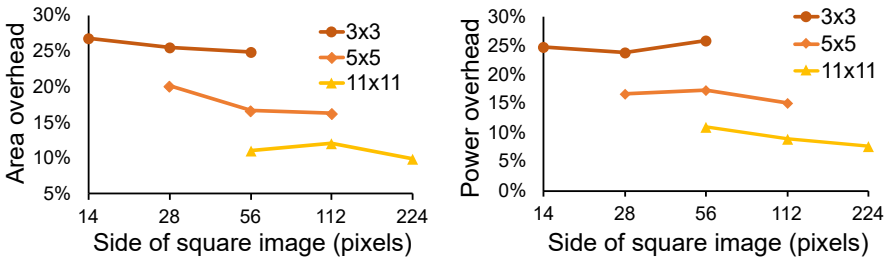
4.4 Evaluation

For the evaluation, we aim to highlight three aspects of ConvGuard. In the first set of experiments, our plan is to measure the hardware overhead of ConvGuard, relative to a customized convolution engine. Then, ConvGuard is compared, in terms of hardware complexity, with a state-of-the-art checker that minimizes the number of required additions to explicitly predict the output checksum. Finally, in the third set of experiments, we explore the fault detection properties of both checkers.

4.4.1 Hardware Overhead added to check an Optimized Convolution Engine

Convolution engines can employ various architectures. Choosing a high-throughput, but area-efficient, sliding-window based architecture – similar to the one used in [55] and [110] – would reveal the worst-case overhead expected from ConvGuard. In such sliding-window-based convolution engines, the incoming pixels are streamed in the engine and stored in an active window buffer of the same size as the filter, and in row buffers that keep the $K-1$ recently fetched lines of the input image [55], as it was also described in detail in Section 2.2.2. Row buffers can be built either using registers, or SRAM blocks. The filtering function is an unrolled and possibly pipelined arithmetic datapath. The baseline input-output throughput of 1 pixel/cycle of these architectures can easily be increased to facilitate parallelism by accepting and producing more pixels/cycle [116].

The sliding-window-based convolution engine and the ConvGuard checker that operates in parallel have been designed in C++ and synthesized to Verilog RTL using Catapult HLS and driven by a commercial-grade 45 nm standard-cell library. Final timing/area results are derived from the Oasys logic synthesis tool. Line buffer memories are mapped to SRAM macro blocks to further minimize the area of the convolution engine. Keeping line buffers in registers would have increased the area of the convolution engine significantly and would unrealistically minimize the overhead of the checker. Power was estimated after synthesis using the PowerPro power analysis and optimization tool. Switching activity information was gathered after simulating the convolution engine and the checker using actual images and filters.



(a) (b)

Figure 4.8: The (a) area and (b) power cost of the ConvGuard checker as a percentage of the total area and power of the protected convolution engine.

Both the convolution engine and ConvGuard have been synthesized for various image and filter sizes assuming 16-bit wide input pixels. In all cases, we assumed a target clock frequency of 1 GHz. Table 4.2 shows the area and power of each constituent part of a protected convolution engine. Additionally, Figure 4.8 highlights the area and power percentage of ConvGuard, relative to the total area and power of the protected convolution engine, for each one of the examined cases.

ConvGuard provides checking capability to the convolution engine by incurring only a small additional area and power overhead. The overhead added is below 10% for 11×11 filters and increases for smaller filters and smaller image widths. The cost of ConvGuard is mostly determined by the size of the filter and is only slightly affected by the size of the image. Image size determines only logarithmically the bit-width of the checker's accumulators. Further, when increasing the bit-width of the input pixels, the cost of the convolution engine that buffers actual pixels increases faster than the cost of ConvGuard, which only stores their sum. For instance, for 32-bit inputs (instead of 16-bit), the highest overhead shown in Figure 4.8(a) for the case of a small 3×3 filter and a small 14×14 input image drops from 25% – for 16-bit inputs – to 19% for 32-bit inputs (not shown in the Figure).

Figure 4.9 illustrates the area and power scaling of the ConvGuard architecture for increasing stride. The synthesized designs assume an 11×11 filter, where using non-unity strides makes more sense. From the reported results, we can see that increasing the stride only marginally increases the total area of the checker. Roughly, for every step of increasing stride, the area and power increases by 6% and 11%, respectively. This result can easily be explained, since the majority of the area of ConvGuard is occupied by the area of the accumulators per filter coefficient and their associated datapath logic, and less by the area of the common sum accumulators used in each channel (see Figure 4.7). Moreover, part of the area/power increase observed when increasing the stride is the result of the complexity of the mask logic shown in Figure 4.7. The mask logic introduces additional multiplexing to forward the result of the multiple common-sum accumulators (one per channel) to the appropriate subtraction units.

4.4.2 Hardware Complexity Comparison with a State-Of-The-Art Checker

Having quantified the overhead of adding ConvGuard to a customized convolution engine, we now aim to highlight its efficiency relative to a recent state-of-the-art checker architecture [83]. In [83], the prediction of the output checksum is done explicitly and the already computed sums of pixels are kept and reused when forming larger sums. On one hand, this approach significantly reduces the number of additions, as shown in Figure 4.5, but, on the other hand, it increases the number of buffers added to store the intermediate results. The HLS-ready C implementation of this "Reuse"

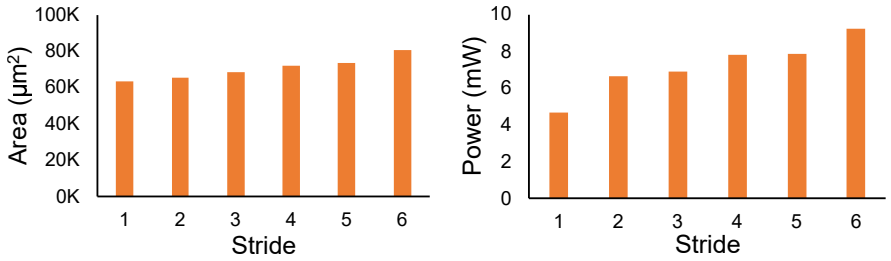


Figure 4.9: The (a) area and (b) power scaling of ConvGuard with increasing stride for an 11×11 filter and a 56×56 example image.

architecture is publicly available in Git and used in this work after easily modifying the Vivado-specific synthesis constraints to Catapult-HLS-specific constraints. Although the design of [83] was targeting an FPGA implementation for testing overclocking possibilities, it was easily ported to an ASIC implementation with marginal modifications that kept the original organization of the checker. To enable a comparison against ConvGuard of the hardware cost, the C model was successfully synthesized to 1 GHz using a 45 nm technology library.

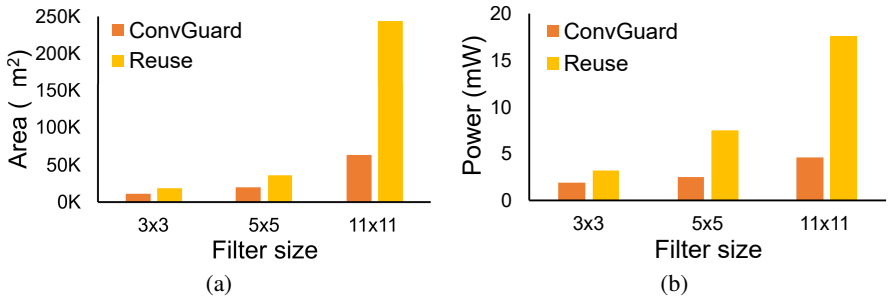


Figure 4.10: The (a) area and (b) power overhead cost of the ConvGuard checker, as compared to the Reuse architecture [83] for an input image size of 56×56.

The area/power results obtained after synthesizing both designs for various filter sizes, and assuming an input image of 56×56 pixels, are shown in Fig 4.10. The trend for other image sizes is the same. The cost of both checkers is mostly affected by the size of the filter, while the size of the input image only determines the bit-width of the accumulators.

In all cases, it is evident that ConvGuard requires significantly less area and power. This attribute of ConvGuard is attributed to the complete lack of buffering resources that fits well to its low-cost profile. Besides its accumulators, ConvGuard does not store any incoming pixels, nor any previous intermediate checksum result. On the contrary, the "Reuse" architecture requires a set of accumulators that handle final additions (equal in number to ConvGuard), and an additional set of accumulators for storing intermediate results. This extra sequential storage is inherent to the organization of the "Reuse" architecture.

Similar conclusions are drawn when considering FPGA implementations of ConvGuard and the "Reuse" architecture. We implemented two versions of the protected convolution engine – the engine and the checker – on a Xilinx Artix-7 chip (XC7A100T) targeting a clock frequency of 100 MHz. The first version includes ConvGuard as the checker, while the second uses the "Reuse" ar-

chitecture [83]. To study the impact of the filter size and the input image size on the final design, we report the implementation results for input image sizes of 56×56 and 112×112, and for filter sizes of 5×5 and 11×11.

Table 4.3: FPGA resource utilization of an unprotected convolution engine and two protected engines using the ConvGuard and Reuse [83] checkers.

Image		56×56		112×112	
Filter		5×5	11×11	5×5	11×11
Unprotected Engine	SLICE	511	2587	524	2707
	BRAM	2	5	2	5
	DSP	25	121	25	121
ConvGuard Protected Engine	SLICE	1268	5406	1274	5513
	BRAM	2	5	2	5
	DSP	26	122	26	122
Reuse [83] Protected Engine	SLICE	2201	13861	2317	14418
	BRAM	2.5	5.5	2.5	5.5
	DSP	26	123	26	123

The results obtained after mapping the Catapult-derived Verilog RTL to the FPGA using Xilinx Vivado 2021.1 are depicted in Table 4.3. The results include the resource utilization of an unprotected engine and the two versions of protected engines. The convolution engine utilizes as many DSP blocks as the square of the filter size to enable a fully unrolled implementation of the datapath. On the contrary, the number of BRAM blocks that implement the line buffers of the convolution engine [55] are determined linearly, both by the size of the input image and the size of the filter. The addition of the checker in parallel to the convolution engine increases the resource utilization for the two protected convolution engines, as shown in Table 4.3. Both checkers need additional DSP blocks and LUT slices to accommodate their arithmetic datapaths. "Reuse" also needs an extra half BRAM to implement its buffers. In all examined cases, it is evident that the ConvGuard checker leads to implementations with lower cost than "Reuse" that scale favorably with increasing image and filter sizes.

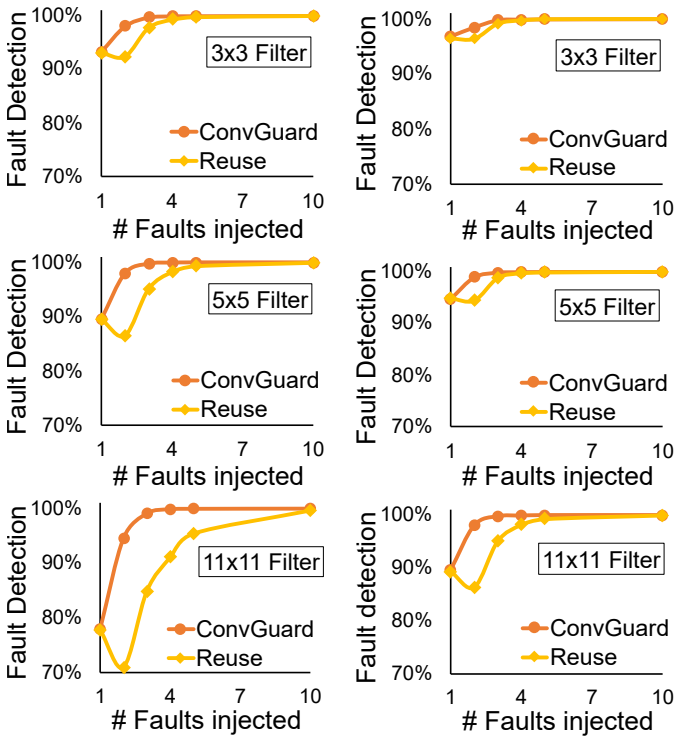
4.4.3 Fault Detection Comparison with a State-Of-The-Art Checker

In the last set of experiments, the goal is to quantify the fault detection properties of ConvGuard and compare them to the state-of-the-art checker analyzed in Section 4.4.2. In this way, we highlight the additional benefit offered by the reduced buffering requirements, as compared to reducing the number of additions. The smaller the number of buffers a checker needs, the smaller the probability to experience a fault inside the checker itself. Checker faults may lead to false alarms and/or missed fault detections.

Our experiments are based on injecting bit-flips in random clock cycles during the time interval needed to complete a convolution. Faults are injected to random storage elements in both the convolution engine and the checker. The number of faults injected in each run is a user parameter. The probability to experience a bit-flip is proportional to the area of the corresponding storage elements. For instance, the SRAM-based row buffers of the convolution engine are expected to experience a bit-flip more often than the accumulators of the checker. The input pixels and the filter coefficients used in each run are the ones used for power estimation. At the end of each convolution, we record the outcome of the fault injection campaign. The observed behavior may fall into one of four

categories:

- **Detected:** A fault occurred in the convolution engine and the checker detected it.
- **Silent:** A fault occurred in the convolution engine and the checker did not detect it. In this case, we must be certain that the checker did not experience any faults. The effect of the fault was masked at the checksum level.
- **False Positive:** The checker flagged a fault detection but no fault occurred in the convolution engine.
- **False Negative:** A fault occurred in the convolution engine and the checker did not detect it. In this case, we must be certain that the checker experienced a fault that caused its malfunction.



(a) 56x56 (b) 112x112

Figure 4.11: Fault detection efficiency of the ConvGuard and "Reuse" [83] architectures after injecting a varying number of faults to the same 10 K convolutions of (a) 56×56, and (b) 112×112 input images using 3×3, 5×5 and 11×11 filters. In the case of 1 injected fault, it is assumed that this is injected only into the convolution engine in a random clock cycle.

Figure 4.11 shows the percentage of faults detected by ConvGuard and the "Reuse" architecture [83] after executing the same 10 K convolutions of 56×56 and 112×112 input images and using 3×3, 5×5 and 11×11 filters. In each case, an increasing number of faults were injected per convolution. In the case of injecting only a single fault, we assume that the fault is always injected

in the convolution engine and the checker remains error-free. This is the reason why the fault detection performances of both checkers match (their performance depends solely on the fault detection properties of checksum-based checking). In the following cases, the faults are injected randomly to both the convolution engine and the checker. The increased number of buffers in the "Reuse" architecture, relative to ConvGuard, reduces its fault detection efficiency. This difference is mostly the result of False Positive outcomes. Even if the convolution engine is error-free, the checker signals an error. When the number of faults injected is increased, both approaches converge to a high fault detection rate. The multiple faults occurring almost certainly cause a difference between the true and the predicted checksum in both cases.

Increasing the filter size increases the number of accumulators needed in both checkers, thus decreasing their fault detection capabilities. However, the effect is more pronounced in the 'Reuse' architecture, due to the buffers needed by construction to store the intermediate results. On the other hand, increasing the input image size affects marginally the detection capabilities of both checkers, since it only affects logarithmically the bit-width of the checksum accumulators.

Table 4.4: The observed behavior after injecting either 2 or 4 random faults in convolutions of increasing input size and a 3×3 filter.

Faults Injected	Image	Fault Categories			
		Detected	Silent	False Pos	False Neg
2	14×14	91.36%	3.98%	4.62%	0.01%
	28×28	95.95%	2.03%	2.01%	0.01%
	56×56	97.96%	1.45%	0.59%	0.00%
	112×112	98.58%	1.17%	0.25%	0.00%
4	14×14	99.54%	0.23%	0.23%	0.00%
	28×28	99.85%	0.12%	0.02%	0.00%
	56×56	99.93%	0.07%	0.00%	0.00%
	112×112	99.94%	0.06%	0.00%	0.00%

To quantify the fault detection efficiency of ConvGuard when increasing the image size, we injected 2 and 4 random faults in 10 K convolutions using a 3×3 filter. The same number of convolutions was repeated for different input image sizes. The results are presented in Table 4.4. With small images, the probability of injecting a fault into the checker is higher, which leads to a measurable amount of false alarms (False Positive and False Negative cases). Instead, when the input image increases, the area of the line buffers increases, as compared to the rest of the sequential storage. Thus, the line buffers inevitably experience the majority of the faults. Since the checker is less likely to experience a fault, it can detect the errors of the convolution engine more often. As the number of injected errors increases, the possibility of having a false alarm drops to almost zero. Overall, due to its low cost and high fault detection efficiency, ConvGuard can act complementary to other protection mechanisms, such as parity checking added to memory blocks [65].

4.5 Conclusions

Algorithm-based fault tolerance is a generic approach for detecting random hardware failures by identifying when there is a difference between the actual and the expected outcome at the checksum level. Such approaches can be used even for post-silicon design validation. In this work, we focus on convolution-specific ABFT implemented directly in hardware.

Our proposal identifies a generic invariance checking condition for convolution and uses it to design simpler online checksum checkers. To avoid any performance degradation, the prediction is computed in the same time frame that the convolution engine produces the true output. The proposed ConvGuard architecture does not re-compute any output pixel; it only quickly predicts their sum. The simple mathematical formulation that guides the design of ConvGuard allows it to adapt to any convolution structure, including arbitrary stride parameters. Its algorithmic nature simplifies the design process and allows its easy adoption in both ASIC and FPGA chips.

In addition to reducing the number of additions by predicting the output checksum implicitly, ConvGuard operates using minimum buffering. Consequently, it saves considerable amount of area relative to a current state-of-the-art checker architecture [83], and it is less susceptible to false negative or false positive alarms for precisely the same reason.

5 Customized Floating-Point Operators for ML Accelerators

Floating-point (FP) arithmetic is a fundamental component in the development and execution of modern AI systems, particularly in deep learning algorithms that dominate areas such as computer vision [79,99], natural language processing [141], and robotics [119]. It provides a powerful method for representing real numbers using a finite number of bits by dividing each number into a significand (or mantissa), an exponent, and a sign bit—allowing for the representation of a vast dynamic range. This flexibility is essential in neural networks, where operations such as weight updates, activations, and gradient computations span a wide range of magnitudes. Training these models requires high numerical precision to ensure stable convergence, avoid issues like vanishing or exploding gradients, and maintain algorithmic fidelity. As such, IEEE-754 single-precision FP (FP32) has long been the standard in training deep learning models due to its balance between numerical accuracy and computational cost.

Despite its strengths, FP arithmetic comes with challenges, primarily due to its limited precision. Not all real numbers can be exactly represented, leading to rounding errors and potential cumulative inaccuracies over successive operations. The IEEE 754 standard [59] provides a structured framework for handling these limitations, defining rules for rounding, special values like not-a-number (NaN), infinity, and denormals, and ensuring consistency across platforms. This standard has enabled the consistent implementation of FP arithmetic in hardware accelerators and general-purpose processors alike. Still, as deep learning models continue to evolve in scale and complexity, the careful selection and implementation of FP formats—tailored to the application's needs—will remain a critical aspect of AI system design.

5.1 Floating-Point Representations

Every FP number consists of three fields: the sign bit (s), the exponent (e), and the mantissa (m). The value of the FP number is given as $(-1)^s \times 1.\text{m} \times 2^{e-bias}$. The bias is a fixed value that depends on the bit-width of the exponent. The mantissa, combined with a hidden bit, forms the normalized fraction of the FP number that is equal to 1.m. Corner cases, such as NaN, infinity, or denormal are also appropriately encoded in every representation. In most deep-learning hardware operators, denormals are flushed to zero for maximum efficiency [2, 127].

The IEEE-754 single- and double-precision formats are widely used in general purpose computations. To achieve lower-cost implementations when employing FP arithmetic, the designers need to achieve a balance between numerical performance and cost. For deep-learning applications, it suffices to use reduced-precision FP arithmetic that may use 16 or fewer bits in total, in an effort to balance numerical performance and cost [2, 5, 127]. For instance, the 16-bit Bfloat16 format [127] provides the same dynamic range as the IEEE-754 single-precision FP format, but with a smaller precision. The 8-bit FP format [87] that was introduced recently by NVIDIA, Intel, and ARM exhibits adequate performance for the training and inference tasks of mainstream CNN models. Figure 5.1 depicts the formats of standard 32-bit floats, 16-bit Bfloats, and the two most dominant 8-bit wide variants. Reduced precision representations lose some of the accuracy of single-precision floats, but

they approach the hardware cost of integer implementations [58, 118].

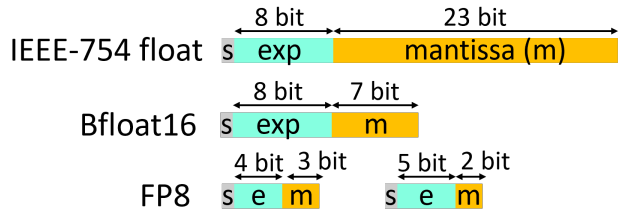


Figure 5.1: The structure of the most commonly used and recently proposed FP formats.

5.2 Basic Floating-Point Operations

Unlike integer arithmetic, FP operations are significantly more complex due to the structure of FP numbers and the precision management required by the IEEE 754 standard. As the FP number consists of three fields, each operation—i.e., addition or multiplication—must manage these components in a carefully coordinated manner to preserve numerical correctness and handle special cases like underflow, overflow, NaNs, infinities and denormals. The hardware implementation of these operations reflects their complexity. Floating-point units (FPUs) typically occupy more silicon area and consume more power compared to integer arithmetic units. For instance, the alignment, normalization, rounding, and exception-handling logic of an FP adder may require several stages in a pipeline, increasing the operation's latency.

5.2.1 Addition

Floating-point addition is inherently more complex than integer addition because it requires alignment of exponents before the significands can be added or subtracted. When two numbers with different exponents are added, the significand of the number with the smaller exponent must be right-shifted to match the exponent of the larger number. This step introduces rounding errors and potential loss of significant bits (a phenomenon known as "catastrophic cancellation" when two nearly equal numbers are subtracted). After mantissas' alignment and addition, the result must be adjusted so the most significant bit of the significand is non-zero (normalization), and then potentially rounded to fit within the allowed number of bits. Each of these steps adds latency and control logic to the hardware, making FP addition considerably more resource-intensive than its integer counterpart. Figure 5.2 outlines the complex organization of a FP addition unit.

5.2.2 Multiplication

Floating-point multiplication, while generally more straightforward than addition in terms of algorithmic flow, still involves considerable complexity. Multiplication proceeds by separately multiplying the significands and adding the exponents, while the sign bit is determined by an XOR of the operand signs. After the raw multiplication, the product must be normalized, as it may produce extra bits requiring adjustment. Like addition, rounding must be applied, and special cases such as zero, infinity, and NaN must be handled explicitly according to the IEEE 754 rules. Although exponent addition and significand multiplication are simpler operations than alignment in addition, they still

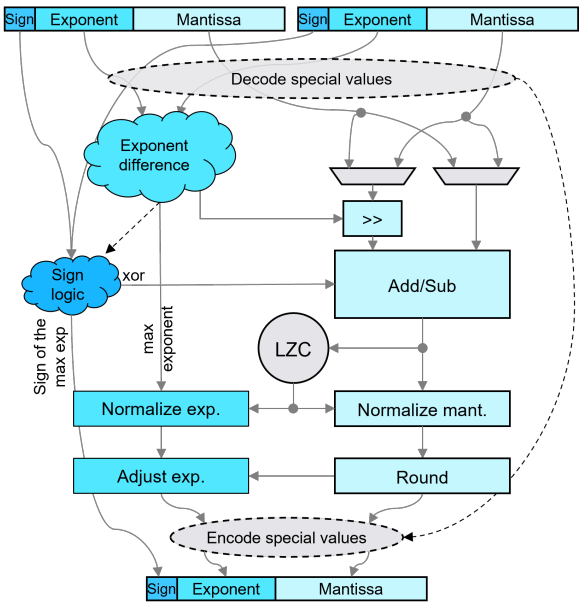


Figure 5.2: The organization of a standard floating-point addition unit.

require wide multipliers and barrel shifters in hardware, particularly in high-precision formats like FP32 or FP64. The organization of a FP multiplier can be observed in Figure 5.3.

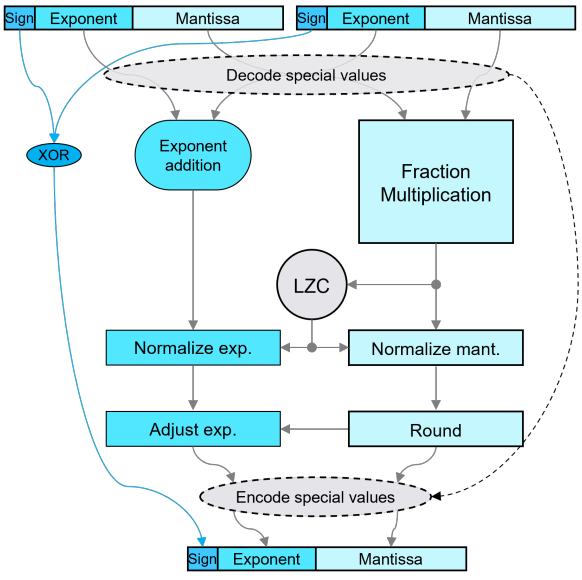


Figure 5.3: The organization of a standard floating-point multiplication unit.

5.3 Fused Dot-Product operators for Dataflow Accelerators

To reduce the inherent overhead of FP arithmetic when implementing vector-wide operations, such as dot products, designers have turned to fusing individual FP operations to more complex ones that implement the needed computation at once [50, 62, 63, 105, 115]. In this way, alignment, normalization, and rounding steps can be shared across independent operations, thereby leading to more efficient hardware architectures. State-of-the-art fused vector FP dot product architectures rely on fixed pipeline organizations designed directly in RTL. Any parameterization within these designs solely facilitates the resizing of certain blocks, based on the structure of the selected FP format.

Another approach to simplifying the implementation of FP operators is to allow wider precision for the output result than the precision of the inputs [13, 86], i.e., operating on two FP8 operands and presenting the result as a 16-bit FP. In this way, rounding may be redundant and the overall hardware cost is reduced.

In this work, our goal is twofold. On one hand, we leverage the state-of-the-art fused dot-product architectures, such as [50, 62, 105], by modeling them in C++ and synthesizing then in RTL using HLS. Thus, the parameterized design allows for compile-time selection of exponent and mantissa widths, as well as vector sizes. On the other hand, we go beyond those designs' fixed-pipeline structure and let HLS tools optimize their internal pipeline structure on a per-application basis by appropriately altering the examined architectural constraints.

Even though all commercial and academic HLS FP libraries [113, 120, 121, 137] can synthesize a vector dot product using efficient primitive FP multiply and add operators, none of them – to the best of our knowledge – supports *fused* dot product computation, thus paying the price of alignment, normalization, and rounding per FP operation.

Overall, the contributions of this work can be summarized as follows:

- A templatized fused vector FP dot product C++ model is presented, which brings the efficiency of fused FP architectures to High Level Synthesis *for the first time*, allowing the design of efficient and customized architectures.
- The design is open-sourced as part of the FastFloat4HLS library [54] that allows the definition of templatized FP datatypes and primitive operators built on top of the publicly available `ac_int` library for integer arithmetic [113].
- Experimental results demonstrate that the proposed designs lead to area and latency savings at the same clock frequency target. For 32-bit standard floats, this benefit comes with a power increase, while for reduced precision 16-bit bfloats [127], power is, in fact, reduced with the proposed architecture.

The proposed fused dot product operator is applied on arbitrary-precision floating point datatypes defined in the FastFloat4HLS C++ library developed in house [54]. The corresponding datatypes are defined as `fast_float<M, E>`, where constants `M` and `E` refer to the size of the mantissa and the exponent fields, respectively. Single-precision floats correspond to `fast_float<23, 8>`, while bfloat16 is equivalent to `fast_float<7, 8>`. FastFloat4HLS contains type-cast functions that allow the conversion from standard C++ floating point datatypes to `fast_float`. Also, similar to all other FP libraries available for HLS [113, 120, 137], FastFloat4HLS implements primitive arithmetic operators, allowing the designer to implement any algorithm in hardware using typical C++ behavior modeling.

5.3.1 Using the Dot Product in C++

The differentiating characteristic of FastFloat4HLS relative to other public FP libraries is the implementation of an efficient fused dot product operator through the templatized function `dot` that is defined as follows:

```
template<int N> // size of vectors
void dot(fast_float<M,E> a[N], fast_float<M,E> b[N])
```

The function accepts two vectors a and b of N elements each and computes in a *fully parallel* approach

$$a_0b_0 + a_1b_1 + \cdots + a_{N-1}b_{N-1}.$$

The only constraint is that the elements of the vectors a and b should follow the same `fast_float` configuration.

Synthesizing the dot product from primitive FP multipliers and adders would limit the expected gains due to the high latency, area, and energy cost for alignment, normalization, and rounding in each step [24, 120]. On the contrary, a fused architecture mitigates these overheads to a single alignment, normalization, and rounding step [50].

```
typedef fast_float<23,8> fp;
template<int M=16, int N=4>
void MatVecMult(fp A[M][N], fp V[N], fp out[M]) {
  for (int i=0; i<M; i++)
    out[i].dot<N>(A[i],V);
}
```

Figure 5.4: A matrix-vector multiplication using the dot product of FastFloat4HLS.

The proposed dot product operator can be used in the context of HLS to implement more complex operations. For instance, Figure 5.4 depicts the implementation of a matrix$\times$vector multiplication, where the matrix A is multiplied with vector V using the proposed vector dot product unit.

5.3.2 Architecture of the Fused FP Dot Product

The fused computation of the dot product follows a tree structure that is split in four consecutive steps. This computation pattern may imply that the hardware organization follows the same four-level fixed-pipeline structure. However, the actual pipeline is determined automatically by HLS, based on the designer's constraints. For, instance, we can get area-efficient single-cycle designs, or high-speed pipelined implementations, without altering the C++ hardware model. Implementing the dot-product with a chain of multiply-add units is not preferred as it leads to inferior designs according to [49].

The N pairs of FP inputs are first unpacked, creating the fraction of each operand by inserting the hidden bit to the mantissa, before being forwarded to the multiplication units. Since we target deep-learning accelerators de-normals at the input are flushed to zero [2].

Multiplication of Fractions Each multiplication unit computes the product of the two fractions 1 m_A and 1 m_B and adds their exponents. The product of the two fractions is computed using an integer unsigned multiplier. The product is positive when the two inputs have equal signs, and negative otherwise.

The sum of the two exponents is computed in parallel to the multiplication. Since the exponent of each operand also contains its bias, the sum of two exponents would result in adding the bias twice, i.e., $e_A + bias + e_B + bias = e_A + e_B + 2bias$. For the result to follow the correct representation,

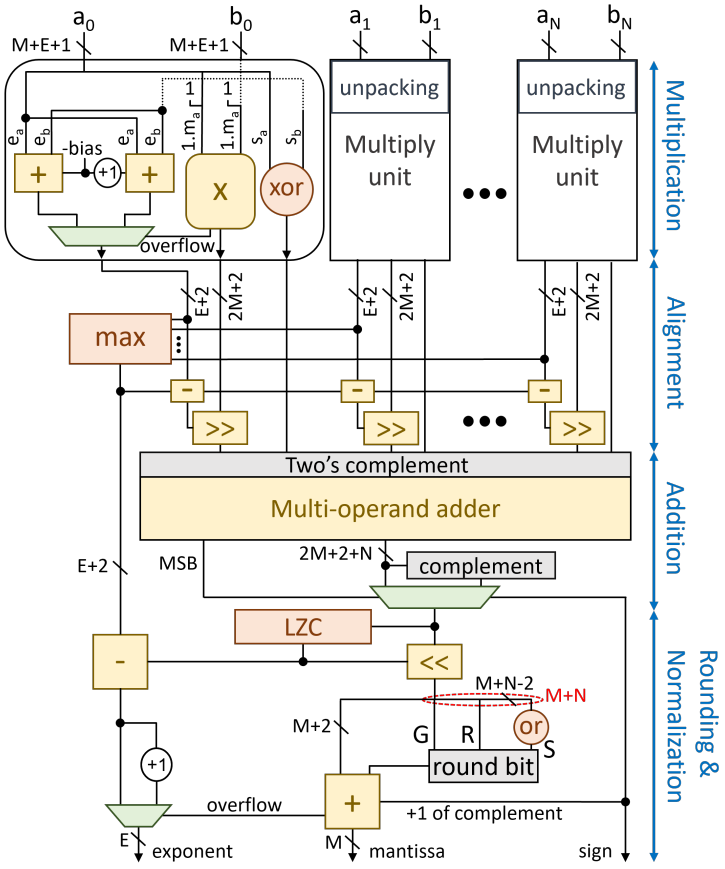


Figure 5.5: The unrolled architecture of the fused many-term dot product unit modeled for HLS. The organization resembles other fixed-pipeline organizations, such as these architectures [50, 62].

one constant bias should be removed. Thus, the exponent of the product is calculated as $e_Y = e_A + e_B - bias$.

This result is correct unless the multiplication operation overflows. Each FP number represents a value in the range [1,2). This means that by multiplying two numbers, the range of the product will be [1,4). As a normalized value is always in the range [1,2), a result that is greater than, or equal to, 2 is denoted as an overflow and needs to be re-normalized. For example, when multiplying $1.010_2 \times 1.110_2$ the result of the multiplication is 10.001100_2, which is greater than 2. To normalize the product, the fraction should be shifted to the right by one position, while the exponent should be increased by 1. Since we do not want this correction of the exponents to be performed after the multiplication, we pre-compute speculatively the overflowed value for the exponents in parallel to the multiplication. Based on the outcome of the multiplication (i.e., if the product is larger than 2), the normal of the overflowed exponent value is selected.

Figure 5.6 depicts a snippet of the C++ model that describes the functionality of the multiplication unit. The `hls_unroll` pragma guides the HLS tool to generate `N` parallel instances, equal to the amount of individual multiplications that are defined by the template parameter `N`. The two versions

of the exponent and the multiplication of the two fractions are computed in parallel, as all three operations are independent to each other. The correct exponent is selected depending on the value of the product, when this becomes available.

```
#pragma hls_unroll
for (int i=0; i<N; i++) {
  mul_exp[i] = a[i].exponent + b[i].exponent - bias;
  mul_exp_overf[i] = a[i].exponent + b[i].exponent - bias + 1;

  // mul_overf is 1 when the MSB of the res_prod is 1
  res_prod[i] = fracA[i]*fracB[i];
  res_expo[i] = (mul_overf[i]) ? mul_exp_overf[i] : mul_exp[i];
}
```

Figure 5.6: A snippet of the C++ description that implements the N parallel multiplication units.

Alignment of Products Each multiplication unit produces the product of the corresponding fractions and the sum of their exponents. To add these intermediate results, we first need to align them, since each one is associated with a different exponent.

To perform the necessary alignment, we need to find the maximum of the N exponents and then right-shift the fraction of each operand so that their exponents become equal to the maximum. Each right shift on the fraction equals to an increase by 1 in the exponent, meaning that each operand should be right-shifted as many times as the difference between the corresponding exponent and the maximum one. To compute the difference between the maximum exponent and the exponent of each part, the maximum exponent value is forwarded to all subtraction units, where the amount of shifting for each individual fraction is calculated.

To design a tree-based max function that finds the maximum exponent of the `N` intermediate exponents, a recursive template meta-programming [61] approach was adopted, as recommended in [37] and shown in Figure 5.7. In each iteration of the recursion, the function searches for the maximum value in each half of the input vector.

```
template<int N>
struct max_s {
  template<typename T>
  static T max(T *a) {
    T m0 = max_s<N/2>::max(a);
    T m1 = max_s<N-N/2>::max(a+N/2);
    return m0 > m1 ? m0 : m1;
  }
}
```

```
template<>
struct max_s<1> {
  template<typename T>
  static T max(T *a)
    return a[0];
}

template<int N, typename T>
T max(T *a)
  return max_s<N>::max(a);
```

Figure 5.7: A recursive template meta-programming approach for the design of a maximum-element identification hardware unit with logarithmic depth.

Addition After alignment, we need to transform each pair of (sign, unsigned fraction) to its appropriate signed representation, in order to add them with the remaining fractions. After this transformation, the N product fractions are reduced to one using an N-to-1 addition tree, which is generated through the C++ code depicted in Figure 5.8.

State-of-the-art designs [49, 50, 62, 115] utilize carry-save adders to implementing this multi-operand addition. Since we are working at the C++ level, this is not a preferred choice. In our case, multi-operand addition is abstractly represented as an unrolled reduction C++ loop. Even with

```
#pragma hls_unroll
for (int i=0; i<N; i++)
  acc += (mulSign[i]) ? -shifted_prod[i] : shifted_prod[i];
```

Figure 5.8: The C++ description that generates the addition tree by adding the positive or the negative value of the intermediate shifted product, depending on its sign.

this abstract form, this approach does not limit the efficiency of the final hardware, since carry-save arithmetic will be enabled after all post-HLS processes by the bit-level transformations of the RTL logic synthesis tool.

If the result of the multi-operand adder is negative, we need to compute its absolute value. To do so, we need to complement the output of the multi-operand adder and increment it by 1. To save delay, we postpone the +1 increment for the rounding step. Therefore, the value that is forwarded to the rounding and normalization stage is selected between the original and the inverted version of the output of the multi-operand addition, depending on the sign of this output.

Normalization and Rounding At the end, the result of the dot product should be normalized and rounded. To complete normalization, we need to count the number of leading zeros of the fraction, and then shift the fraction to the left as many positions as the number of leading zeros. In parallel, the computed Leading Zero Count (LZC) should be subtracted from the exponent.

Leading-zero counting implements in C++ the fast design proposed in [27], using recursive templates, as depicted in Figure 5.9(a). In this way, LZC is not treated as a monolithic block, but it can be scheduled in a fine-grained manner along with the exponent update and mantissa alignment.

The operation initiates through the top `LZC` function, which receives the input A and starts the recursion by calling the `lzc_s` function. In each recursive step of `lcz_s`, `lzc_reduce` decides if the number of leading zeros is an odd, or even number. Initially, `lzc_reduce` is applied to the whole input and, in each one of the following steps, the input is reduced to half by computing the logic OR of neighbor bits. When only one bit remains, the recursion stops. If the input is the all-zero vector from the beginning, flag `ZF` is asserted. In this case, this implementation [27] treats the remaining bits of the leading-zero count as "don't care". Otherwise, the complementary value of the inverted sequence of the intermediate results, which gets returned by the top function `LZC` at the end of the operation, indicates the number of leading zeros in A.

The recursive template, combined with the unrolled loops in the C++ code, lead to the tree structure of the LZC unit, which is illustrated in Figure 5.9(b). The input is fed at the top level of the structure, where the first output is generated form the `lzc_reduction` unit, and, as it moves to the next level, its size is reduced to half, until a single bit remains. At each level, the produced output is inverted before its value is used.

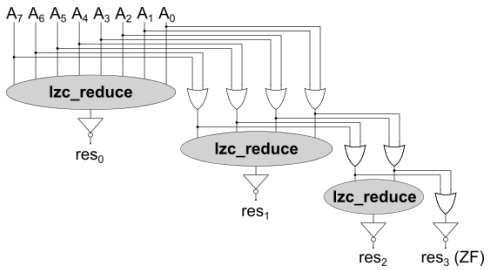
The $M+N$ least significant bits of the normalized fraction are used to compute the round bit that will be added to the LSB of the $M+2$ most significant bits. If the result of the addition was negative and the output was inverted, then the increment by 1 that was postponed from the addition step is also performed. As a single-bit value, this addition does not require an extra addition unit, but, instead, the value is pushed to the carry-in bit of the already existing adder. In the case that rounding overflows, the exponent is increased by 1 and the fraction is shifted right by one position.

At the end, the E less significant bits of the exponent, the M less significant bits of the fraction and the sign bit that was produced by the MSB of the multi-operand addition are packed and pushed to the output.

```
// Top LZC function
template<int N>
ac_int<ac::log2_ceil<N>::val+1,
       false> LZC(ac_int<N,false> A){
  int BW = ac::log2_ceil<N>::val+1;
  ac_int<BW,false> b,res;

  lzc_s<N>::lzc_s(A,b);// First call

  #pragma unroll yes
  for (int i=0; i<BW; i++)
    res[i] = b[BW-i];

  return  res.bit_complement();
}

// Recursion
template<int N>
struct lzc_s {
  template<typename T>
  static void lzc_s(
        ac_int<N,false> a, T &out){
    ac_int<N/2, false> a0;
    out[ac::log2_ceil<N>::val] =
                    lzc_reduce<N>(a);
    #pragma unroll yes
    for (int i=0; i<N/2; i++)
      a0[i] = a[2*i] | a[2*i+1];
    // Recursive call
    lzc_s<N/2>::lzc_s(a0,out);
  }
}
```

```
// Terminate recursion
template<>
struct lzc_s<1> {
  template<typename T>
  static void lzc_s(
        ac_int<1,false> a, T &out)
    out[0] = a[0];
}

bool lzc_reduce (ac_int<W,false> a) {
  ac_int<W/2,false> odd, even, out;
  #pragma unroll yes
  for (int i=0; i<W/2; i++) {
    even[i] = a[2*i];
    odd[i] = a[2*i+1];
  }
  even = even.bit_complement();
  ac_int<W/2,false> even_prefix;
  ac_int<1,false> t=true;
  #pragma unroll yes
  for (int i=W/2-1; i>=0; i--) {
    if (i == W/2-1) t = even[i];
    else t = t & even[i];
    even_prefix[i] = t;
  }
  even_prefix = even_prefix >> 1;
  even_prefix[W/2-1] = 1;
  out = even_prefix & odd;
  return out.or_reduce();
}
```

(a) Computing LZC using recursive template meta-programming in C++.



(b) The tree structure of the LZC unit.

Figure 5.9: The C++ model of the Leading Zero Count unit (a) and its structure (b).

5.4 Evaluation

The C++ model of the proposed FP many-term dot product unit was thoroughly tested through C++ simulation to ensure its correct functionality. Subsequently, the verified designs were synthesized with Catapult HLS after setting the appropriate architectural constraints. Our goal was to achieve a fully unrolled (parallel) architecture with the minimal latency, which operates with initiation interval of one at a specified clock frequency target. Catapult HLS utilized the available resources, produced an optimally pipelined architecture for the design and generated the corresponding Verilog RTL. To verify its correct functionality, we tested the generated RTL through the SCVerify flow using the Questa Advanced Simulator [112].

Since there is no HLS model that implements a *fused* many-term FP dot product, we include two different set of comparisons: On the one hand, we compare the proposed fused many-term FP dot product unit synthesized with Catapult HLS with state-of-the-art architectures of *non-fused* FP

vector dot products designed directly in RTL. Specifically, we compare against designs generated by the FloPoCo RTL generator [24], which is publicly available and considered a reference point in FP hardware designs. The RTL implementations of all designs under evaluation were mapped to the 45 nm Nangate standard-cell library and placed-and-routed using the Cadence digital implementation flow that is shown in Figure 5.10. Genus [14] was used for logic synthesis, while place-and-route was completed in Innovus [15].

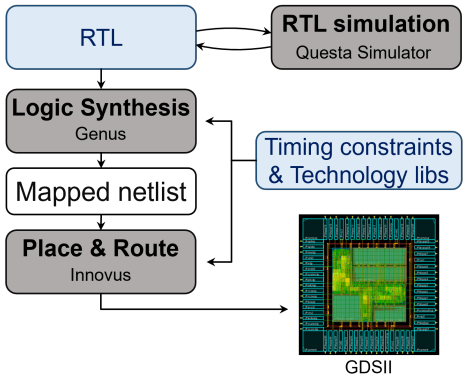


Figure 5.10: The digital implementation flow from RTL to GDSII.

On the other hand, for completeness, we summarize in a common table at the end of the Evaluation section the characteristics of state-of-the-art fused architectures, with respect to the proposed designs, using *as is* the data presented in the corresponding papers.

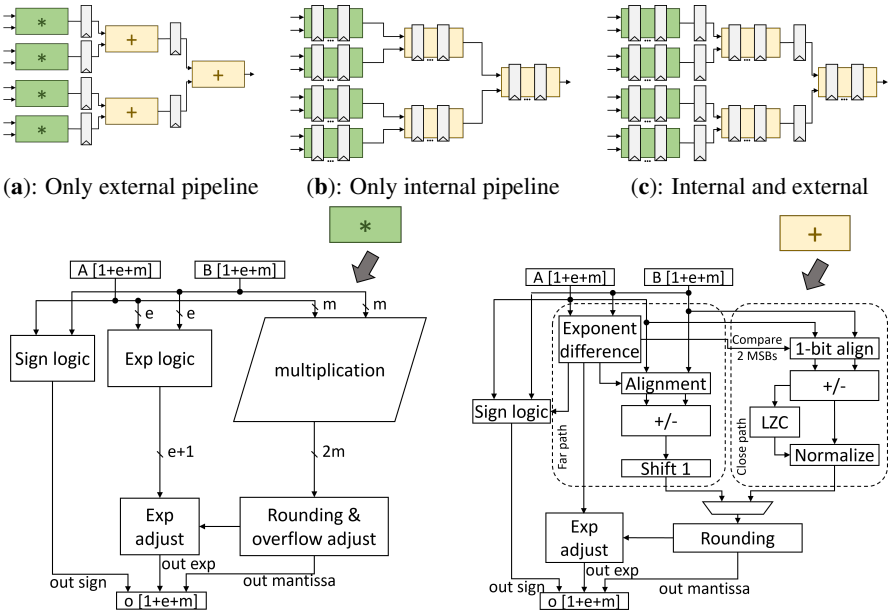
It should be noted that FPGA-based experimental evaluation is deemed beyond the scope of this paper, since the proposed approach targets ASIC implementations *exclusively*. In order to yield meaningful insights, experiments on an FPGA would require our C++ code to target FPGA-specific optimizations amenable to the underlying FPGA fabric, i.e., DSP blocks. Such transformations are left as future work.

5.4.1 Identifying State-of-the-Art Non-Fused FP Vector Dot Product Configurations

FloPoCo does not support fused vector dot product units. So, our goal is to use FloPoCo and identify an optimal non-fused architecture by combining the available FP multipliers and adders and the many possible pipeline configurations that FloPoCo allows for each operator without support for denormals. To better elucidate how we explored the available design space, we will use as an example the design of a 4-term dot product unit. This unit requires four multipliers to compute the four products in parallel, and three adders to construct the 4-to-1 reduction tree that will compute the final result of the dot product.

With the goal being to minimize the latency in cycles, the dot product unit can be designed using combinational multipliers and adders that are connected with intermediate pipeline stages. The performance of this approach, shown in Figure 5.11a, is bounded by the delay of the FP multipliers and adders. Alternatively, we could allow the multipliers and adders to be internally pipelined. Two alternatives emerge when using this approach. The one shown in Figure 5.11b pipelines the multiplier and the adders without using any extra pipeline registers between them. On the contrary, the second approach, shown in Figure 5.11c, allows for both internal and external pipelining, i.e., across

the FP units. FloPoCo employs efficient architectures for the design of each FP multiplier [39] and FP adder [108]. The high-level organization of each unit is depicted at the bottom of Figure 5.11.

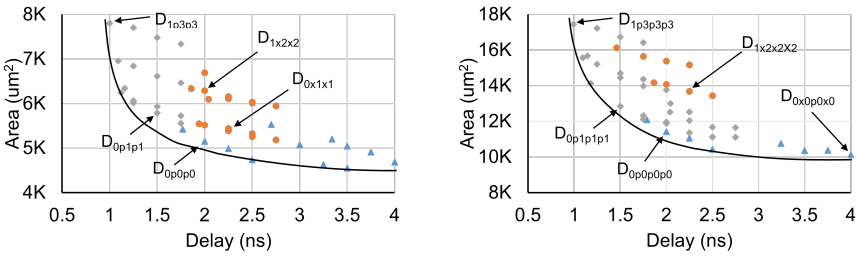


(**a**): Only external pipeline (**b**): Only internal pipeline (**c**): Internal and external

(**d**): Floating-point multiplier (**e**): Floating-point adder

Figure 5.11: Various pipelined organizations enabled by the FloPoco RTL generator for a 4-term dot product unit built from efficient FP multipliers and adders.

We examined 196 designs that cover 4 and 8-term dot products for the standard float and bloat16 formats targeting various clock frequencies. The results derived for the bloat16 format are summarized in Figure 5.12. The pareto curve inside each figure outlines the architectures with the least area at each delay target.



(**a**): 4-term products (**b**): 8-term products

Figure 5.12: The design-space exploration of 4- and 8-term dot products for the bfloat16 FP format generated with FloPoCo [24].

To uniquely identify each design choice in Figure 5.12, we follow a simple notation that describes the pipeline structure followed by each architecture. For instance, the design D0p0p0 that appears to be the best at 500 MHz in Figure 5.12a corresponds to the organization of Figure 5.11a for a 4-term

product. The three zeros denote that each level of the dot product has zero inner pipeline stages and the letters 'p' between the zeros state that external pipelining is used across stages. On the other hand, D1p3p3 – the best choice when targeting a clock frequency of 1 GHz in Figure 5.12a—corresponds to a design of the category shown in Figure 5.11c. More specifically, it is 4-term design with multipliers having 1 inner pipeline stage, the adders of both levels have 3 inner pipeline stages, and extra pipeline registers are placed between the multipliers and adders. When an external pipeline stage is missing, letter 'p' is replaced with an 'x'. The same structure is followed for the 8-term designs shown in Figure 5.12b, including more symbols per design due to the increased number of addition stages required.

5.4.2 Comparisons with the Proposed Fused Vector FP Dot Product Architecture

Having identified the best configurations of many design choices for 4- and 8-term dot products, we compare them with the equivalent proposed designs synthesized in RTL from Catapult HLS. The results obtained are summarized in Table 5.1. All designs are compared with respect to their area, power, and latency, under the same clock frequency constraint. The reported power was measured by running examples that cause, on average, a 25% toggle rate.

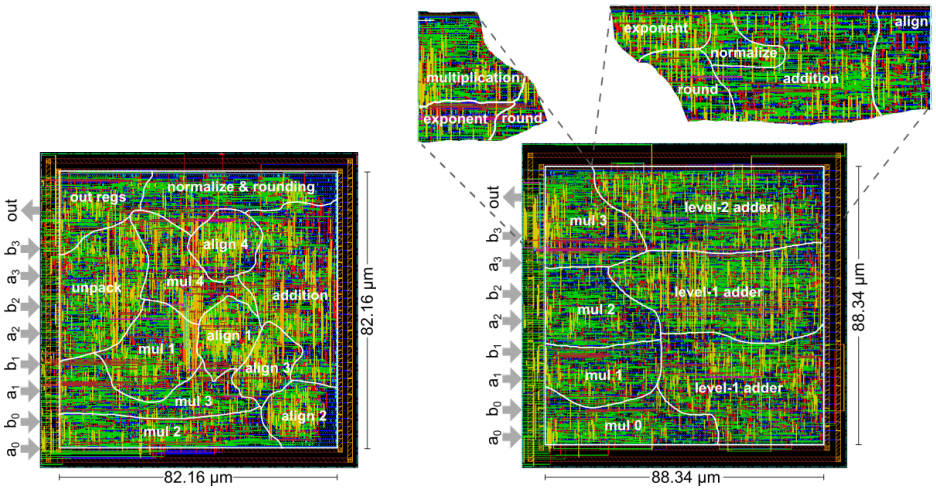
Table 5.1: Comparison of the proposed 4- and 8-term fused FP dot product units relative to state-of-the-art non-fused designs.

4-term		**Proposed**			**State-of-the-Art Non-Fused**		
		Area (um^2)	**Power (mW)**	**Lat.**	**Area (um^2)**	**Power (mW)**	**Lat.**
500 MHz	FP32	21,124	8.50	3	22,778	5.90	5
	BF16	5057	1.83	3	5151	2.47	3
1 GHz	FP32	31,518	13.26	6	32,568	11.32	12
	BF16	6750	4.62	6	7803	4.42	10
8-term		Proposed			State-of-the-art non-fused		
		Area (um^2)	Power (mW)	Lat.	Area (um^2)	Power (mW)	Lat.
500 MHz	FP32	50,847	14.17	3	51,304	13.66	7
	BF16	10,096	4.11	3	11,422	5.78	4
1 GHz	FP32	60,863	25.07	7	67,953	25.62	19
	BF16	14,405	9.39	6	17,614	9.44	14

For single-precision floats (FP32) and 4-term products, the proposed design achieves significantly lower latency and requires slightly less area than the corresponding state-of-the-art, at both clock frequency targets. For 8-term products, the proposed design achieves even higher reductions in latency. In general, the penalty for the comprehensive performance improvement achieved by the proposed design in FP32 is increased power consumption.

Instead, for bfloat16 (BF16), the proposed architecture excels in *all* three salient metrics. The improvement is greater at 1 GHz, where the latency is reduced by 40% and the area by around 14% for an 4-term unit, and by 57% and 18%, respectively, for an 8-term unit. Most importantly, these latency and area improvements at 1 GHz are achieved with similar power consumption. For instance, the layout of the two units under comparison for the case of a 4-term dot product of bfloats16 optimized for 1 GHz is shown in Figure 5.13.

At 500 MHz, the power consumption of the proposed design is lower by 26% and 29% for 4-term and 8-term products, respectively, as compared to the state-of-the-art. This reduction in power consumption at 500 MHz is achieved with slightly less area and with similar (or slightly lower) latency.

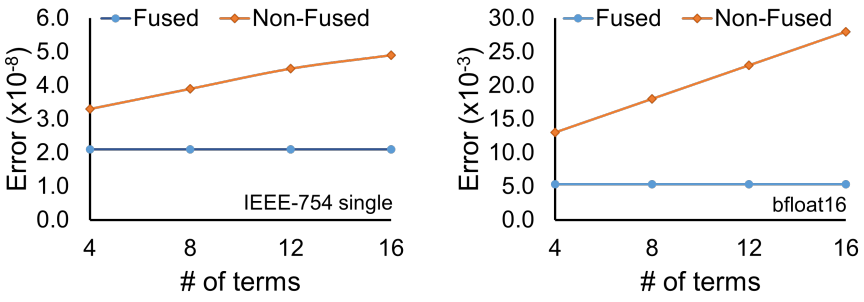


(**a**) Proposed (**b**) State-of-the-art non fused

Figure 5.13: The final layout for (**a**) the proposed and (**b**) the state-of-the-art non-fused 4-term dot product architectures, assuming a bfloat16 representation.

Part of the efficiency of fused dot-product architectures stems from the fact that alignment, normalization, and rounding are performed only once and are not repeated in each individual multiply and add operation. This characteristic also favorably affects the numerical accuracy of the computation.

To evaluate the accuracy of the proposed fused vector dot-product unit, compared to a non-fused architecture, we computed 1 million dot products for different implementations and FP formats. For each test, the inputs were generated randomly, following a Gaussian distribution. In each case, the computation error is compared relative to the "golden values" computed using the double precision floating point datatype.



(**a**) 32-bit single precision input data (**b**) 16-bit Bfloat16 input data

Figure 5.14: The error introduced by the fused and non-fused architectures for two different FP formats when computing dot products with increasing number of terms, relative to the same computation implemented with double-precision (64-bit) FP arithmetic.

Figure 5.14 illustrates how the relative error scales with the increasing number of the dot product terms for (a) IEEE-754 single precision data in, and (b) for bfloat16 data. When looking at the error

of the non-fused architecture, we can see that it scales with the number of terms in the dot product calculation. This is due to the increasing amount of intermediate rounding that is performed at the end of each individual adder or multiplier that comprise the design. On the other hand, the relative error of the proposed units remains constant and close to the minimum step between two consecutive floating point numbers. This behavior is due to the increased internal data width datapath used in all cases. which increases linearly with the number of terms of the dot product. A similar technique was used in Intel's Nervana NPP-T [50] fused architecture, where the width of the final accumulator was selected so that it could compute every dot product in its supported applications, without a loss in accuracy.

5.4.3 Performance Summary of Fused Dot Product Architectures

Even though fused vector dot product designs are not available as open-source, we summarize—for completeness—in Table 5.2 the characteristics of efficient fused vector dot product architectures and of the proposed design. The data for the competing approaches is taken verbatim from the corresponding papers. Obviously, one of the key strengths of the proposed approach is its templatized nature (unique in Table 5.2) that provides designers with unprecedented flexibility. This flexibility, in conjunction with HLS, fine-tune the resulting implementation for the specified design constraints and FP format. Consequently, as evidenced in Table 5.2, the proposed approach leads to balanced design configurations where the pipeline latency, clock frequency, and area are cohesively optimized.

Table 5.2: Comparison between the different fused floating-point vector dot product architectures.

Design	**Templatized**	**Open Source**	**#Terms**	**FP Format**	**Technology**	**Frequency (GHz)**	**Area (um^2) $\times$1000**	**Latency (Cycles)**
[63]	No	No	4	single	180 nm	0.08	$\sim$620	1
[105]	No	No	2	single	45 nm	0.37	16.10	1
[115]	No	No	2	single	45 nm	1.50	33.29	3
[62]	No	No	32	bfloat16	10 nm	1.11	$\sim$2.75	5
[50]	No	No	32	bfloat16	45 nm	N/A	N/A	10
Proposed	Yes	Yes *	4	single	45 nm	1.00	31.52	6
			8	bfloat16	45 nm	1.00	14.41	6

*: `https://github.com/ic-lab-duth/Fast-Float4HLS`.

5.5 Conclusions

Floating-point representations are very important for modern applications due to their increased precision compared to integers. Selecting the appropriate representation depending on the requirements and carefully designing the architecture of the operators is crucial for the overall performance and accuracy. Implementing the many-term FP dot product units required in the majority of deep learning applications with a *fused* architecture leads to highly efficient solutions, in terms of latency, area, and power. To enhance the scope of fused architectures on high-level synthesis design flows that support custom-precision FP arithmetic, we designed them as templatized soft FP cores that allow the designers to use them with arbitrary FP types in their C++ HLS models. The design is available under a permissive open-source license as part of the header-only FastFloat4HLS library [54]. The experimental results that compare the proposed approach to state-of-the-art non-fused many-term dot product architectures show that it can achieve lower latency and area at the same clock frequency, while its power consumption depends on the chosen FP format.

6 Reduced Precision Fused Multiply Add Operators for Systolic Arrays

Matrix multiplications are at the heart of deep learning algorithms and their computation in hardware maps naturally onto SAs [70]. TPUs [57] and other related architectures [20, 74, 93, 106] are characteristic examples of newly designed SAs. Matrix multiplication can be implemented in SAs using integer or FP arithmetic [58, 118]. For increased accuracy, the use of FP arithmetic dominates during the training of deep learning models. To increase energy efficiency, inference is typically executed using integer arithmetic, after appropriate data quantization and pruning [41]. However, recent studies have shown that FP arithmetic cannot be avoided, if one wishes to preserve the inference quality [58].

The introduction of reduced-precision FP formats has enabled the use of FP arithmetic also in the inference, as they offer the good precision of FP while maintaining low cost, close to integer arithmetic. The incorporation of these formats inevitably affects also the architecture of the corresponding FP operators. For instance, the operation of the traditional pipelined FP units used in SAs is dominated by the delay of the wide multipliers, while the logic dedicated to the exponent calculations is not time-critical. However, in reduced-precision FP operators this delay profile is partially flipped, since the bit-width of the mantissa (fraction) field is now equal to, or smaller than, the bit-width of the exponent field. Consequently, new architectures are required that must account for this new delay attribute of reduced-precision FP arithmetic, and, at the same time, tackle the chained structure of the SA's processing elements.

To address said challenges, this work proposes a novel pipeline architecture for SAs that operate on *reduced-precision* FP arithmetic, with the following salient characteristics:

- A new *skewed* pipeline micro-architecture is proposed that reorganizes the operations inside the pipelined organization of the FP fused multiply-add units, thereby enabling parallel execution of the pipeline stages of consecutive PEs within the SA. The proposed design minimizes the overall latency of matrix multiplication, as compared to traditional pipelined architectures, with minimal area and power overhead.
- Pipeline skewing is enabled by the introduction of new speculative forwarding paths within the exponent field's logic. These forwarding paths eliminate the restricting dependencies across pipeline stages and effectively increase pipeline parallelism.

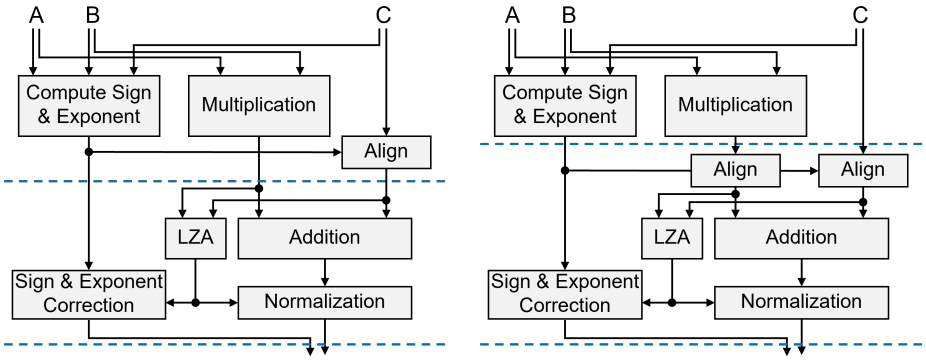
Experimental evaluation using state-of-the-art CNNs demonstrates the effectiveness of the proposed architecture. The overall execution latency is markedly reduced by 16% and 21% for MobileNet [52] and ResNet50 [48], leading to overall energy reductions of 8% and 11% respectively. These savings were achieved with a minimal area cost of 9%.

6.1 Systolic Arrays using Floating-Point Arithmetic

As it was discussed in Section 2.1, the typical SA hardware structure consists of an array of PEs, where each PE computes a MAC operation, and streams data to neighboring PEs. The structure and the flow of data inside the SA are defined by the implemented dataflow scheme.

Under the WS dataflow, a chain of multiply-add operations is computed in each column of the array. The FP multiply-add units (FMA) in each PE have a fused/cascaded structure [39, 49], whereby the product of the multiplication is passed directly to the adder, without intermediate normalization and rounding. Normalization occurs after each addition at the South border of each PE. To further reduce hardware cost, state-of-the-art implementations [33, 50, 62] do *not* perform rounding after *each* multiply-add step in each PE. Instead, the rounding is performed only once, at the South end of each *column*. To avoid precision loss, the intermediate results produced at the South output of each PE use double-width precision [58]. For instance, for Bfloat16 inputs, the reduction that occurs in the vertical direction is implemented with FP32 arithmetic.

State-of-the-art FP multiply-add units in each PE may adopt one of the two pipelined datapaths shown in Figure 6.1. The diagrams in the figure highlight only the most critical blocks involved in the multiply-add datapath and omit, for clarity, several logic-level details. Note that, for *reduced-precision* FP arithmetic, a two-stage pipeline – as depicted in Figure 6.1 – is sufficient to achieve the required clock frequency. On the contrary, traditional *full-precision* FP units rely on deeper pipelines for high clock frequencies [72].



(a) FMA for regular precision.

(b) FMA for reduced precision.

Figure 6.1: The two main pipeline organizations that may be employed by the FP multiply-add units in each PE of the SA. In reduced-precision FP arithmetic, two pipeline stages are sufficient to achieve the required clock frequency.

In the first pipeline stage of Figure 6.1(a), multiplication is performed in parallel with the exponent computation, which calculates the amount of alignment required for the incoming partial addition result. This approach is adopted by many multiply-add architectures [72, 144]. It is based on the fundamental assumption that the delay of the multiplier completely hides the computation on the exponents and the delay of alignment. However, this assumption is only true in full-precision FP arithmetic, where the delay of the multiplication dominates the delay of the exponent computations.

In the second pipeline stage of Figure 6.1(a), addition is performed. Leading-Zero Anticipation and counting (LZA) [27, 107], running in parallel to the addition, predicts the amount of shifting needed to normalize the adder's result. This shift amount is also used to correct the already computed exponent of the final result.

Since the delay of the multiplication cannot hide the delay of the exponent computations in reduced-precision FP arithmetic, it is preferable to move the alignment to the second pipeline stage, as shown in Figure 6.1(b). The alignment may involve either the output of the multiplier, or the incoming partial addition result [77, 81]. This approach is a more natural fit to the delay profiles observed with the new the pipeline of Figure 6.1(b) serves as the state-of-the-art reference FP multiply-

add design for reduced-precision FP arithmetic.

6.2 The Proposed Skewed Pipeline Architecture

The two-cycle latency incurred by either the pipelined FP multiply-add units shown in Figure 6.1 increases the number of cycles required to complete the reduction within each column of the SA. Also, the pipeline parallelism across PEs is limited since the computation in each PE can begin only after the previous PE in the same column has finished its operation.

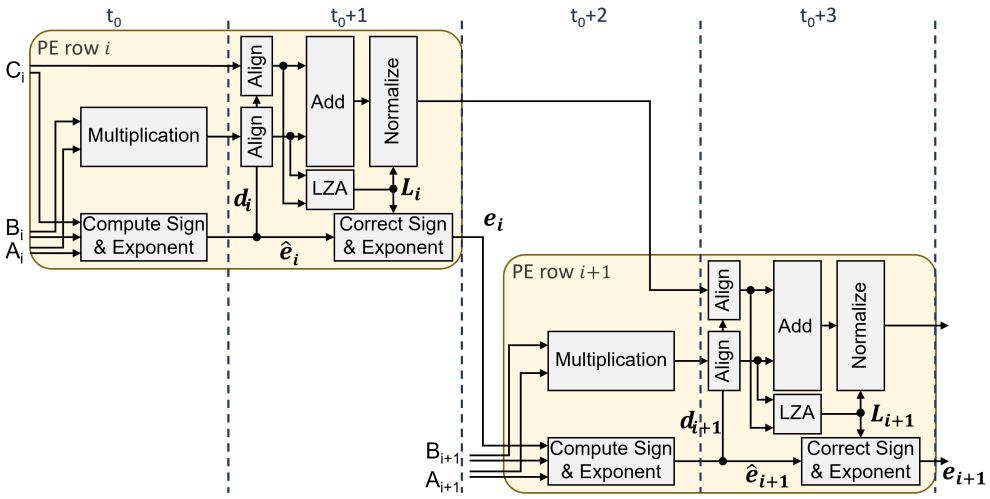


Figure 6.2: The dependencies arising in a chained FP multiply-add operation across two neighboring PEs of the same column of the SA. These dependencies prohibit the interleaving, in time, of the pipeline execution.

6.2.1 The serialization problem

The fundamental reason for this serial execution is the dependency that appears between the result of the second pipeline stage of the PE in row i of the SA and the first pipeline stage of the PE in row $i+1$ of the *same column*. This dependency is highlighted in Figure 6.2 across cycles t_0+1 and t_0+2. Recall that each PE employs the 2-stage pipelined organization of Figure 6.1(b).

To increase parallelism, we would like the first pipeline stage of the PE in row $i+1$ to execute in parallel with the second pipeline stage of the previous PE (i.e., both in cycle t_0+1). If this were allowed, it would create a new critical combinational logic path across the two neighboring PEs, emanating from the exponent output of the first PE: the alignment logic of the first PE would be connected *in series* with the LZA module of the first PE, the exponent correction logic of the first PE, and the exponent computation logic of the following PE.

To avoid the formation of this long path, the operation in each PE begins only after the previous PE has completed its entire computation at the end of its second pipeline stage.

6.2.2 Removing dependencies using speculative paths

To interleave, in time, the operation of the pipeline stages in each PE, a new pipelined organization for the FP multiply-add datapath is required, which relaxes the above-mentioned restricting dependencies and avoids the introduction of new combinational logic critical paths. The first step in optimizing the FP multiply-add pipeline is to decouple the exponent correction logic of the second pipeline stage of one PE from the exponent compute logic of the first pipeline stage of the next PE. This decoupling is achieved by the pipeline organization shown in Figure 6.3.

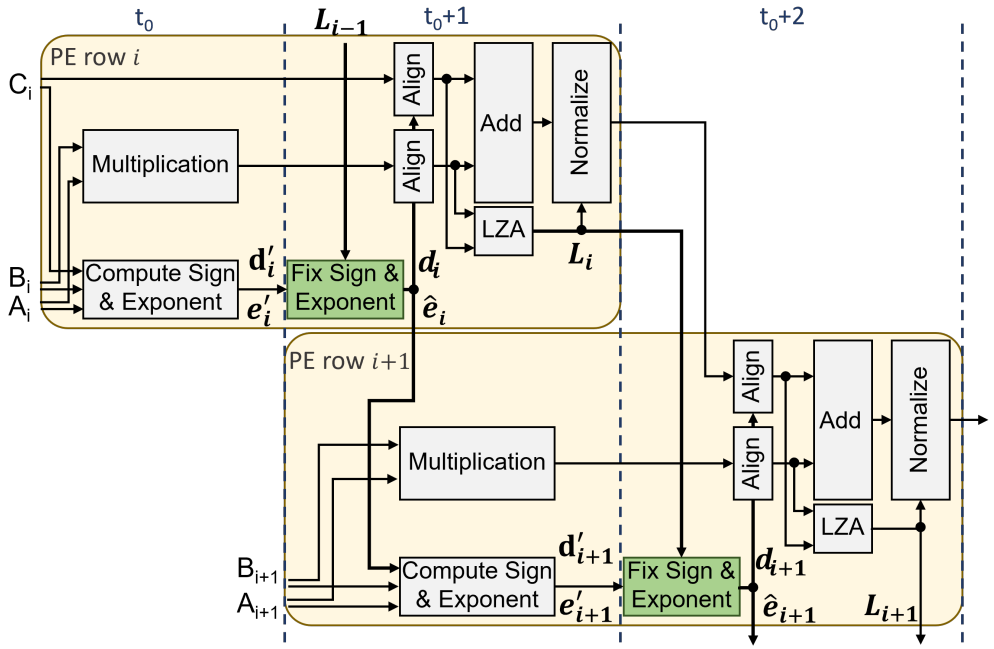


Figure 6.3: Removing the dependency across the exponent output of each PE. A speculative exponent is produced at the output of the first pipeline stage, which is corrected at the beginning of the second stage.

In this setup, the exponent correction logic is replaced by exponent *fix* logic and moved to the input of the second pipeline stage of each PE. This is the new module 'Fix Sign & Exponent' shown in green in Figure 6.3. To enable this relocation, the exponent fix logic no longer depends on the output of the LZA module of the current PE, but, instead, it receives the output of the LZA logic of the *previous PE*. This decoupling allows for the interleaving, in time, of the pipelined execution of the multiply-add operation in consecutive PEs of the same column of the SA.

The output of the exponent fix logic controls the alignment of the adder's inputs in the same pipeline stage and it is also given to the next PE in the place of the output exponent. This output is not the final exponent, but an intermediate and *partially correct* result. The correct exponent value will be computed in the exponent fix logic of the next PE.

In each PE, the exponent compute logic selects the maximum between the exponents of the multiplication that was just calculated and the input exponent that comes from the previous PE. This maximum value, which is denoted as $\hat{e}_i$, represents the exponent of the *unnormalized* result of the FMA's addition and it is calculated as $\hat{e}_i = max(e_{M_i}, e_{i-1})$, where $e_{M_i} = e_{A_i} + e_{B_i}$ is the exponent of the multiplication in the current PE. Furthermore, the difference of the two exponents $d_i = |e_{M_i} - e_{i-1}|$,

serves as the alignment value of the two addends. In the setup of Figure 6.2, $\hat{e}_i$ gets corrected by the value of the LZA L_i and the corrected exponent output $e_i = \hat{e}_i - L_i$, now referring to the *normalized* output, is forwarded to the next PE.

On the other hand, in the case of Figure 6.3, the compute exponent logic of the PE in row i receives the intermediate $\hat{e}_{i-1}$ exponent, instead of the corrected one, as L_{i-1} is not yet available to correct it. This means that the outputs of its first pipeline stage $e'_i = max(e_{M_i}, \hat{e}_{i-1})$ and $d'_i = |e_{M_i} - \hat{e}_{i-1}|$ are *speculative* values, as the exponent used refers to an unnormalized result and must be subsequently fixed. At the beginning of its second pipeline stage, L_{i-1} becomes available and is forwarded to the exponent fix logic, in order to correct the speculated values. The difference of the exponents required for the alignment is:

$$d_i = |e_{M_i} - e_{i-1}| = |e_{M_i} - (\hat{e}_{i-1} - L_{i-1})| = |(e_{M_i} - \hat{e}_{i-1}) + L_{i-1}|$$

As the value of L_{i-1} is always greater than, or equal to, zero, we can say that:

$$d_i = \begin{cases} |e_{M_i} - \hat{e}_{i-1}| + L_{i-1} = d'_i + L_{i-1} \text{ , if } e_{M_i} \geq \hat{e}_{i-1} \\ L_{i-1} - |e_{M_i} - \hat{e}_{i-1}| = L_{i-1} - d'_i \text{ , if } e_{M_i} < \hat{e}_{i-1} \end{cases}$$

Additionally, the fix logic generates $\hat{e}_i$ from e'_i. However, since $\hat{e}_i$ is either e_{M_i}, or e_{i-1} (see above), e'_i is not a computed quantity, but, instead, it comprises the two values e_{M_i} and $\hat{e}_{i-1}$ that are being forwarded from the first to the second pipeline stage. After the correction of $e_{i-1} = \hat{e}_{i-1} - L_{i-1}$ in the exponent fix logic, $\hat{e}_i$ is computed and forwarded to the next exponent compute logic block.

As both $\hat{e}_i$ and L_i are computed in the same pipeline stage, and because L_i becomes available at the end of the cycle, the correction of the *final* exponent result (i.e., at the South edge of each column) cannot happen in the same cycle. As a result, the correction for the exponent of the last PE of each column will happen during the rounding stage at the end of the column.

The presented re-organization of the exponent computations allows for the parallel execution of the pipeline stages of consecutive PEs. However, the placement of the exponent fix logic inevitably increases the combinational path delay of the second pipeline stage of each PE. To overcome this overhead, we can *retime* the normalization step.

This retiming is shown in Figure 6.4. Instead of normalizing the result of the addition in the same cycle, normalization occurs in parallel to the alignment logic at the input of the adder. The unnormalized value that arrives from the adder of the PE in row $i-1$ requires at most L_{i-1} left shifts to get normalized. In the meantime, the alignment value that is computed by the fix logic determines the amount of right-shifting that may be required, if the addend was already normalized. Depending on the relation between the alignment value and L_{i-1}, the addend would need to either shift to the left, or to the right. As only one of these options may occur, the two operations are completely in parallel, removing the serial dependency in the delay. The new alignment scheme also affects the alignment value of the second addend that comes from the multiplication. However, only a right shift may occur in that case. The unnormalized output of the final PE will be normalized at the rounding stage at the end of the column.

Overall, the proposed pipeline structure blurs what a PE actually is across the pipeline stages. In the new design, a PE implements, in parallel, part of the second pipeline stage of the first PE and part of the first pipeline stage of the next PE (in the same column). In fact, this new operational attribute of the PE is explicitly seen in cycle $t_0 + 1$ in Figure 6.4. Assuming that the highlighted PE of Figure 6.4 is the last of the column, an extra addition stage is needed for the operation to be complete. Similar to the baseline case, an extra stage is also needed to round the final result of each column.

Leveraging the recently proposed approach in [3] for online alignment and addition can simplify speculation hardware without significantly increasing overall hardware costs.

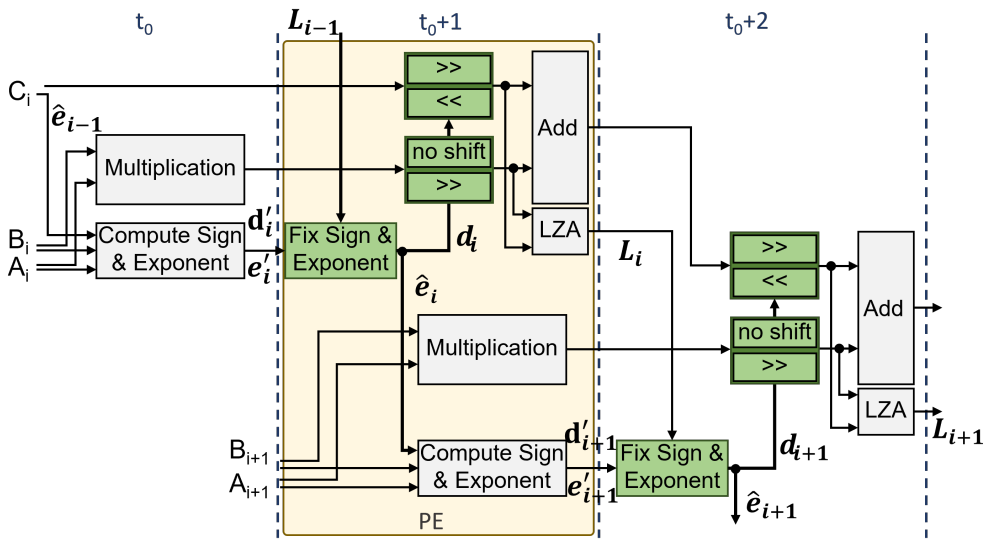


Figure 6.4: The normalization logic is retimed in parallel to the align logic of the next PE. The addition result that flows across PEs is properly shifted to the left, or right, according to the exponent fix logic of the same stage.

6.3 Evaluation

In this section, we demonstrate the effectiveness of the proposed architecture in reducing the energy requirements when computing CNNs, as compared to state-of-the-art FP multiply-add architectures employing the traditional two-stage pipeline organization of Figure 6.1(b). In both cases, we assume Bfloat16 inputs that are reduced in the vertical direction using single precision FP32 arithmetic.

Both designs under comparison were implemented in C++ and synthesized to Verilog RTL using Catapult HLS, driven by a commercial-grade 45-nm standard-cell library. Both SA architectures have an array size of 128×128 PEs. Final timing/area results are derived from the Oasys logic synthesis tool. Power was estimated after synthesis using the PowerPro power analysis and optimization tool.

The proposed design, depicted in Figure 6.4, requires 9% more area than the state-of-the-art FP multiply-add architecture shown in Figure 6.1(b). We assume that both designs have been optimized for a clock frequency of 1 GHz. This area overhead is due to the extra pipeline registers required by the proposed design to pass intermediate exponent and LZA output values across the two pipeline stages, and the extra combinational logic of the exponent fix module. Similarly, the proposed design consumes 7% more power, on average, when computing layers from state-of-the-art CNNs, such as MobileNet [52] and ResNet50 [48].

This marginal hardware area and power overhead is amortized by the latency savings reaped by the proposed approach, which allows for the parallel execution of the pipeline stages of consecutive PEs. Such latency savings allow the computation of each CNN layer to finish much sooner, thus yielding a *reduction* in the overall *energy* consumption of the computation.

To clarify this result, Figures 6.5 and 6.6 report the per-layer energy consumed when executing each layer of the MobileNet [52] and ResNet50 [48] CNNs. The energy reported refers to the average energy observed after computing MobileNet and ResNet50 on 100 randomly picked images from the ImageNet database [103].

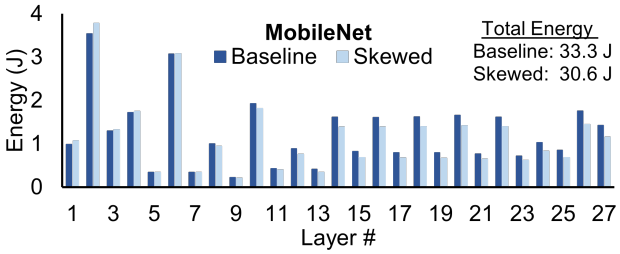


Figure 6.5: The per-layer energy consumption when executing MobileNet [52] with the two pipeline architectures under comparison.

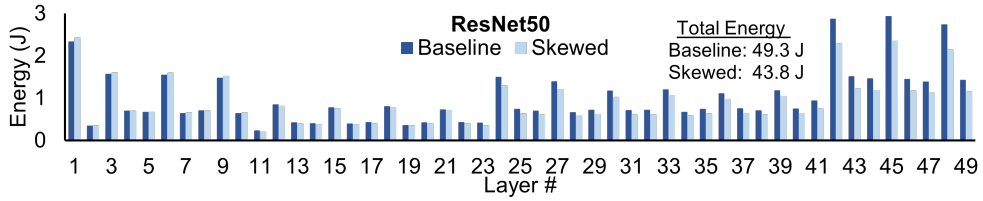


Figure 6.6: The per-layer energy consumption when executing ResNet50 [48] with the two pipeline architectures under comparison.

In both Figures 6.5 and 6.6, we observe that, in the first layers, the proposed approach actually leads to energy increases. The reason for this behavior is that the latency reduction cannot offset the small power overhead of the skewed pipeline organization. For the last layers, where the structure of the CNN layers changes, more latency is saved, thereby leading to significant per-layer energy savings. Most importantly, these per-layer savings translate to an *overall/total* energy reduction of 8% for MobileNet [52] and 11% for ResNet50 [48].

6.4 Conclusions

The design of balanced pipelined FP multiply-add units for the PEs of a SA should not stop at the boundaries of each PE, but it should also account for the dependencies arising across pipeline stages of consecutive PEs. The proposed skewed pipeline architecture focuses exactly on this aspect and effectively optimizes the latency of the reduction within each column of the SA. In effect, this reduces the overall latency of matrix multiplication. The small area and power overhead incurred by this pipeline reorganization is compensated by significant overall energy reductions when computing the layers of state-of-the-art CNNs.

7 Automatic Hardware Generation for CNN Accelerators

The need for custom CNN accelerators, that can easily adapt to the requirements of each application, has been increased as more applications have emerged, ranging from low power wearable devices to high performance data centers. To cover the wide variety of CNN applications and their constraints, the design of custom hardware accelerators has to be easily accessible and configurable.

This need has already been recognized in various cases. For instance, Gemmini [40], Angel-Eye [45] and DNNWeaver [109] allow the automated design of custom CNN accelerators derived from a highly parameterized architectural template. Other approaches instead of following a fixed architecture paradigm employ HLS for synthesizing the design from high-level C++ models. hls4ml [30] allows the translation of trained Python models to HLS-ready C++ models. Kalms et al. [60] proposed a HLS library that explores multiple parallelization schemes to improve efficiency and performance. Similarly, fpgaConvNet [126], Caffeine [143], and FlexCNN [8] use HLS to design CNN accelerators offering also various performance tuning strategies. FP-DNN [44] follows a hybrid approach using both HLS and traditional RTL-based design flows.

The automatic generation of CNN accelerators is often enhanced by architectural tuning techniques such as quantization and model pruning, which aim to reduce memory usage and execution latency. Quantization lowers the precision of the model's parameters and can be applied either after training [7, 22, 67] or during training [21, 56, 146]. Another well-studied technique that impacts the memory footprint is pruning [29, 38, 46], which involves removing specific parameters from the CNN model. When quantization or pruning is performed during training, the model can better adapt to these changes, thereby mitigating potential accuracy loss [56, 80].

Similar to state-of-the-art frameworks for the automated design and optimization of CNN accelerators, this work leverages the simplicity of Python-based CNN modeling and the flexibility of High-Level Synthesis (HLS) to automate the generation of CNN hardware accelerators. The approach emphasizes ease of use, enabling users to effortlessly create custom hardware accelerators, targeting either ASIC or FPGA platforms, directly from TensorFlow or PyTorch CNN models. These models are automatically translated into HLS-ready C++ code.

The synthesized designs follow the SPD template [123, 131], that was discussed in Section 2.2. The proposed generator, leveraging HLS and template metaprogramming, includes multiple optimization knobs, streamlining the automatic exploration of diverse architectural alternatives to boost performance and energy efficiency.

The novelty of the proposed approach can be summarized as follows:

- The introduced framework is highly configurable, allowing for the design of a wide-spectrum of CNN accelerators ranging from low-cost energy efficient designs up to high-performance variants. It supports all relevant CNN layers and enables the customization of the size and the parallelism offered by the synthesized datapath and the corresponding buffer stages.
- It supports low-power buffering architectures for the CNN dataflow architectural template that cover both traditional and non-traditional (such as strided or dilated) CNN layers.

- It integrates a low-power ABFT error-checking mechanism that performs a checksum checking for convolutional layers of arbitrary structure, enabling online functional safety at minimum extra cost, both in terms of buffering and error-checking logic.
- A wide spectrum of data representations and quantization models and their optimized hardware architecture are supported. Data representations include traditional FP representations, reduced precision FP formats [2, 87, 127] as well various forms of integer and fixed-point arithmetic.
- The experimental results highlight the hardware efficiency of three CNN accelerators synthesized automatically by the proposed framework directly from their Python models. Also, the proposed design framework is compared to other similar state-of-the-art CNN accelerator design frameworks and/or generators both in terms of their supported features as well as the hardware complexity of the derived designs for FPGA implementations.

7.1 Enabling Fast Acceleration Directly from Python

Machine learning engineers are constantly working on improving the accuracy and the runtime efficiency of their models. To support this effort, libraries such as Tensorflow and Pytorch have been developed that provide a rich collection of data structures and functions that simplify the process of developing and experimenting with new ML models.

On the contrary, this simplicity is not equally enjoyed by hardware engineers. Designing a custom hardware accelerator for a specific CNN model is much more complicated.

Table 7.1: The available layers in the developed C++ library for developing CNN applications.

Layer	Description
Conv2D	Implements the 2D convolution operation, similarly to the Conv2D of PyTorch, covering strided and dilated spatial variants. Activation functions (i.e. Linear, ReLU, Sigmoid, Softmax) are supported internally.
Pool2d	Implements the pooling function, that supports both "Max" and "Average" pooling, following an equivalent implementation to their PyTorch or Tensorflow counterpart.
Dense	Implements the fully-connected layer, similarly to the PyTorch Linear layer with support for an in-place activation function like in Conv2D.
Activation	Implements a stand-alone activation layer that supports the implementation of the activation layers like ReLU, Sigmoid and Softmax.
Flatten	Implements the conversion of a 3D matrix to a 1D vector, just like the PyTorch equivalent.
BatchNorm2d	Implements the BatchNorm2d layer from PyTorch.
Add_residual	Performs the addition of the outputs from two layers in residual connections.

With the goal to simplify the design and optimization of CNN hardware accelerators, we designed a library of C++ models for all relevant CNN layers that are ready for HLS and a API that automatically converts Tensorflow and PyTorch layers to their corresponding C++ versions. The available

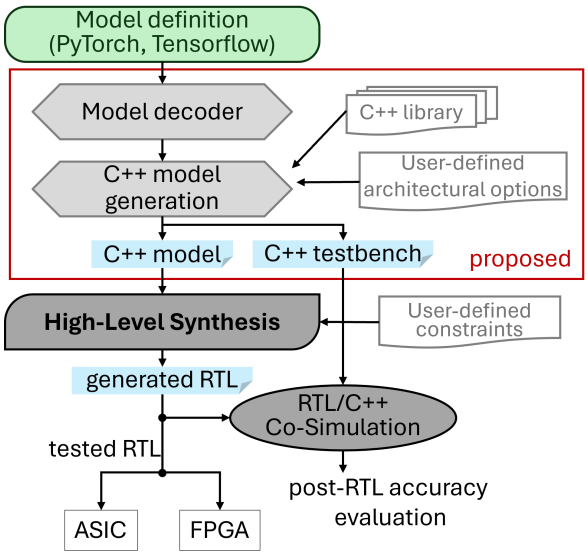


Figure 7.1: Overview of the design flow that converts Tensorflow or PyTorch models to their C++ equivalents, that are used to generate RTL though High-Level Synthesis.

layers summarized in Table 7.1, can be used to convert any CNN architecture, that can be defined by these, from Python to its corresponding HLS-ready C++ equivalent. The add_residual shown in the last row of Table 7.1 is not an actual layer, but it is used to implement residual connections similar to the add layers found in Pytorch and Tensorflow. In this way, we can leverage both the wide acceptance of Python-level development efforts with respect to ML and the versatility of HLS tools for efficient hardware synthesis.

7.1.1 Python to HLS-Ready C++ Conversion

Figure 7.1 illustrates the overall design flow. First, the Python-based CNN model is parsed to extract key characteristics such as the size and shape of convolution layers, activation functions, and spatial patterns (e.g., strided or dilated). Next, this extracted information, along with a user-defined Python dictionary specifying architectural options for each CNN layer, is used by the proposed framework to generate an equivalent HLS-ready C++ model that *exactly matches* the structure of the model. The layers in the Python model are replaced with corresponding layers from a custom-developed HLS C++ library.

In addition to the C++ model, a testbench is automatically generated. This testbench verifies the functional correctness of the model and ensures it is equivalent to its TensorFlow or PyTorch counterpart.

The generated C++ model is then synthesized to produce a custom CNN accelerator for either ASIC or FPGA deployment, guided by the user's architectural constraints. The resulting hardware design adheres to the SPD architectural template discussed in Section 2.2. To demonstrate the conversion process from a PyTorch model to its C++ equivalent, we use a version of LeNet [73] as an example. The Python implementation of LeNet is shown in Algorithm 5, and its structure is graphically depicted in Figure 7.2. This model includes a series of layers comprising convolution, max pooling, and fully connected layers, along with the appropriate activation functions at each stage.

Using the proposed framework, the Python model of LeNet is translated to an equivalent C++

Algorithm 5 Describing the LeNet architecture of Figure 7.2 in Python using the PyTorch API

```
1  import torch.nn as nn
2  class Net(nn.Module):
3   def __init__(self):
4    super().__init__()
5    self.conv_0=nn.Conv2d(in_channels=1,
6                          out_channels=6,
7                          kernel_size=5,
8                          padding=2)
9    self.relu=nn.ReLU()
10   self.max_0=nn.MaxPool2d(kernel_size=2,
11                           stride=2,)
12   self.conv_1=nn.Conv2d(in_channels=6,
13                          out_channels=16,
14                          kernel_size=5,
15                          padding=2)
16   self.max_1=nn.MaxPool2d(kernel_size=2,
17                           stride=2,)
18   self.flatten=nn.Flatten()
19   self.fc_0=nn.Linear(in_features=784,
20                       out_features=120)
21   self.fc_1=nn.Linear(in_features=120,
22                       out_features=84)
23   self.fc_2=nn.Linear(in_features=84,
24                       out_features=10)
25
26  def forward(self, x):
27   x=self.relu(self.conv_0(x))
28   x=self.max_0(x)
29   x=self.relu(self.conv_1(x))
30   x=self.max_1(x)
31   x=self.flatten(x)
32   x=self.relu(self.fc_0(x))
33   x=self.relu(self.fc_1(x))
34   x=self.fc_2(x)
35   return x
36
37 model=Net()
```

model. The result is depicted in Algorithm 6. Each layer is defined as a class with many template parameters, where some of these define the characteristics of the layer, similar to the PyTorch counterpart, while some others are used to configure the architecture of the synthesized hardware. The architecture-configuration parameters CNFG# are also passed as template parameters during the instantiation of each layer (lines 4–11 in Algorithm 6).

One notable difference between the Python and C++ model is the declaration of the activation functions. While in Python activation functions correspond to explicit function calls, in the proposed C++ models they are implemented within the layer itself, specified via a template parameter, and executed as a final operation before passing the output to the next layer. Additionally, unlike Python models, their C++ counterparts require the explicit definition of arrays that store the parameters, inputs and outputs for each layer and their corresponding data type. The shape of the arrays depends

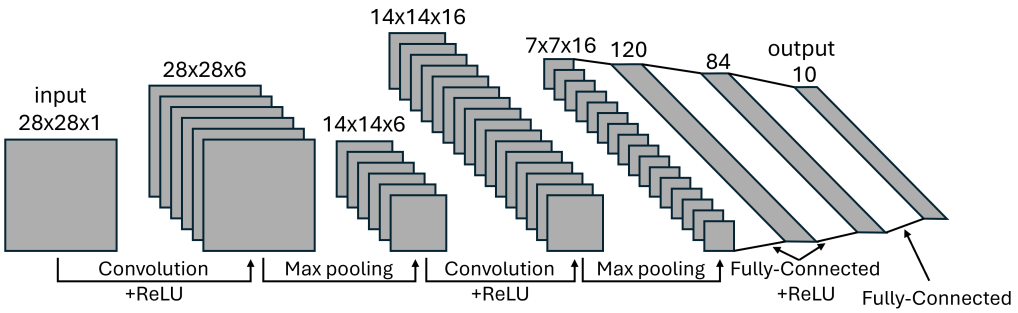


Figure 7.2: The structure of LeNet, implemented by 2 convolutional layers, 2 max pooling layers and 3 fully-connected layers.

on the shape of the layer's parameters as extracted from the Python model, while the structure of the interconnection channels depends on the customization of the hardware architecture, through the loop optimization techniques that were discussed in Section 2.3.

7.2 Configuration of the CNN Accelerator

The developed C++ models are designed so that they can support all the possible architectural choices that were discussed in Section 2.3. Through specifying template parameters, the model can generate arbitrary structures for each CNN layer. This makes the proposed framework a highly parameterizable solution for generating accelerators according to the available resources or required performance.

Apart from allowing the parameterization of the CNN engines, this framework incorporates all the different architectures discussed so far.

7.2.1 Optimized Buffering for Spatial Convolutions

In dataflow CNN accelerators, the stream of data between the engines implies that consecutive inputs are adjacent features of the IFM. In spatial convolution variants, however, each output is computed using features that are not adjacent. Thus, the engine should be able to carefully re-organize the inputs, to avoid redundant computations and data movements. The library utilizes the buffering optimizations for strided (LeapConv) and dilated (LazyDCStream) convolutions that were presented in Chapter 3 and manage to optimize the data movement inside the engine, thus resulting in more energy efficient hardware implementations. These architectures are selected over the standard convolution engine architecture, when spatial convolution layers are present in the CNN structure.

The architectural configurations that were presented in Section 2.3 are also applicable to both architectures of spatial variants, without affecting their functionality or performance. Figure 7.3 shows the organization of a Leapconv layer with stride 2, when an arbitrary unroll factor T_n or T_m is applied. Similarly, Figure 7.4 presents the respective organization of a LazyDCStream layer that is designed for a convolution with dilation rate 2.

7.2.2 Integrated Checksum Checker

For many applications it is important to ensure that the output of the computation is correct, thus the library integrates the power efficient error checking mechanism [31], that was presented in Chapter 4

Algorithm 6 Describing the CNN of Figure 7.2 in C++ using the designed library

```
class Net() {
private:
 // Layer instantiation
 Conv2d<1,6,5,5,CNFG0,relu> conv_0;
 Pool2d<6,2,2,CNFG1, max> max_0;
 Conv2d<6,16,5,5,CNFG2,relu> conv_1;
 Pool2d<16,2,2,CNFG3, max> max_1;
 Flatten<7,7,16,CNFG4> flatten;
 Dense<784, 120,CNFG5,relu> fc_0;
 Dense<120, 84,CNFG6,relu> fc_1;
 Dense<84, 10,CNFG7,linear> fc_2;

 // Define external memory for parameters
 weight_dtype w0[1][6][5][5], b0[6];
 weight_dtype w1[6][16][5][5], b1[16];
 weight_dtype w2[784][120], b2[120];
 weight_dtype w3[120][84], b3[84];
 weight_dtype w4[84][10], b4[10];

 // layer interconnection
 custom_dtype c0,c1,m0,m1,fl,f0,f1;

public:
 Net() {};
 ~Net() {};

 // model function
 void forward(inp, out) {
  conv_0(inp, c0, w0, b0);
  max_0(c0, m0);
  conv_1(m0, c1, w1, b1);
  max_1(c1, m1);
  flatten(m1, fl);
  fc_0(fl, f0, w2, b2);
  fc_1(f0, f1, w3, b3);
  fc_1(f1, out, w4, b5);
 }
};
```

and can be implemented alongside each convolution engine. Although this checksum method was introduced for a convolution between one IFM and a single kernel, its properties can be used to calculate the checksum of a whole CNN layer, where N IFMs of size $R \times C$ convolve with M filters of size $K \times L$ to produce M OFMs. In the general case, the explicit checksum for a whole CNN layer can be calculated using (7.1), where $a_{i,j,n}^{(k,l)}$ is a feature located at row i and column j of the n IFM and that during the convolution it gets multiplied with a weight that is located in row k and column l of a kernel. The sum of all $a_{i,j,n}^{(k,l)}$ is then multiplied with the sum of all weights with indexes k, l, n across all the M filters.

$$\sum_{k=0}^{K-1}\sum_{l=0}^{L-1}\sum_{n=0}^{N-1}\left[\left(\sum_{i=0}^{R}\sum_{j=0}^{C}a_{ijn}^{(kl)}\right)\sum_{m=0}^{M-1}h_{klnm}\right] \tag{7.1}$$

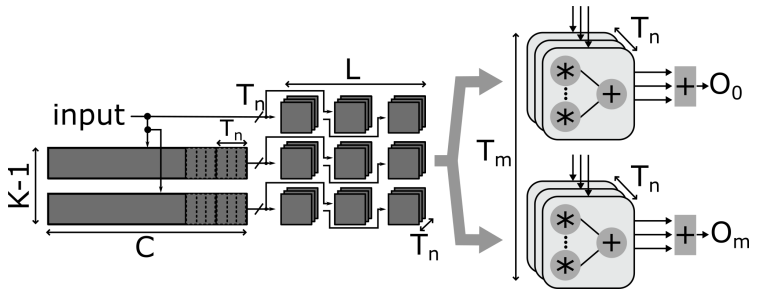


Figure 7.3: Organization of a LeapConv streaming convolution engine optimized for strided convolutions, and enhanced for computing a complete CNN layer.

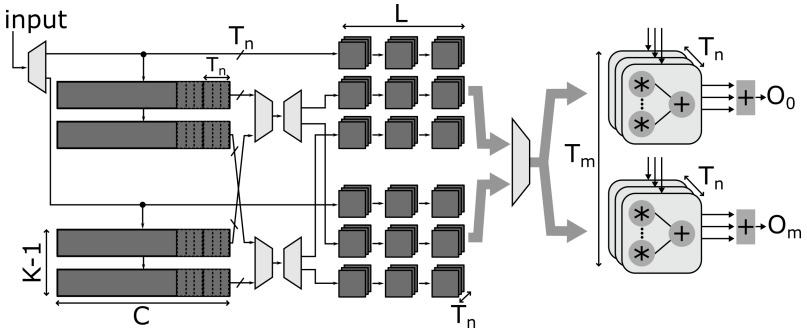


Figure 7.4: Organization of a LazyDCStream streaming convolution engine optimized for dilated convolutions, and enhanced for computing a complete CNN layer.

Similarly, the implicit checksum is given by (7.2), where the main difference with (7.1) is that the sum of the features that get multiplied with the same weight is computed implicitly as presented in [31].

$$\sum_{k=0}^{K-1}\sum_{l=0}^{L-1}\sum_{n=0}^{N-1}\left[\left(\sum_{i=0}^{R}\sum_{j=0}^{C}a_{ijn}-\sum_{i=0}^{R}\sum_{j=0}^{C}\hat{a}_{ijn}^{(kl)}\right)\sum_{m=0}^{M-1}h_{klnm}\right] \tag{7.2}$$

The proposed library supports checksum units that accompany convolutional layers and can compute the checksum either implicitly or explicitly, depending on the size of the IFM, resulting always in the most power-efficient version of the checker. In either case, the number of parallel checksum units utilized is defined by the unrolling of the architecture, as described in Section 2.3. These units compute the accumulation of features per IFM, multiply their values with the sum of weights from the M filters and add the individual results to produce the checksum.

7.2.3 Arithmetic Representation and Quantization

The proposed library supports both fixed- and floating-point arithmetic, giving the flexibility to the designer to select the best, based on the application's requirements. The FP operators integrated into the library are based on the FastFloat4HLS [54] open-source HLS library for FP arithmetic that support custom datatypes as well as dot-product fused operations that –to the best of our knowledge– have not yet been integrated into any other open-source HLS CNN model. The HLS implementation for fused dot-product operations that was discussed in Section 5.3 and is integrated in the FastFloat4HLS library, allows the design of any term fused dot-product units. This, combined with the

support for arbitrary datatypes, allows the implementation of custom optimized operators depending on the characteristics of the CNN layer.

In many cases, the limited memory resources or the need for small integer operators demand a quantization of the model. Quantization [135] is a method performed during or after the training phase of a CNN model, to convert the floating-point parameters into integers of either 32, 16 or 8 bit width. Apart from the quantization of its parameters, a quantized CNN model, requires that the input of each layer should be also quantized before used. This quantization of the inputs is performed between the layers of the CNNs, by scaling the output of each layer before it is moved to the next one. To support the inference of quantized models, the hardware should be enhanced to perform the appropriate quantization and de-quantization operations on the parameters and the data that are being passed between the CNN layers.

The proposed library is designed to support quantized models, by storing the quantization generated parameters in the local buffers dedicated to each layer, and by performing the scaling of the data as an extra operation at the output of each layer.

Overall, the representation of data is determined by the ML engineer's choice of quantization for the CNN. To reduce its complexity and improve its efficiency, the ML engineer may opt to quantize its parameters into either smaller floating-point or fixed-point representations. The quantization, however, comes with an impact to the accuracy of the CNN. By supporting both fixed- and floating-point representations, as well as quantized models, our proposed library facilitates easy design space exploration. This enables users to assess the hardware complexity associated with different representations and study the tradeoff between model accuracy and hardware complexity.

Table 7.2: The supported layers of the proposed library and their template parameters.

	Structural							Architectural				
Layer	FMAP	Kernel	Stride	Dilation	Padding	Pooling	Activation	Datatype	Unroll	Tilling	Forward Out	ABFT
Conv2D	R, C, N, M	K, L	-	-	P	-	A	fixed/float	T_n, T_m	T_r, T_c	-	True/False
Dense	N, M	-	-	-	-	-	A	fixed/float	T_n, T_m	-	-	-
Pool2D	R, C, N	K, L	-	-	-	Max, Avg	-	any	T_n	T_r, T_c	-	-
Activation	R, C, N	-	-	-	-	-	A	any	T_n	T_r, T_c	-	-
BatchNorm2D	R, C, N	-	-	-	-	-	-	fixed/float	T_n	T_r, T_c	-	-
Flatten	R, C, N	-	-	-	-	-	-	any	T_n, T_m	T_r, T_c	-	-
SConv2D	R, C, N, M	K, L	S	-	P	-	A	fixed/float	T_n, T_m	T_r, T_c	-	True/False
DConv2D	R, C, N, M	K, L	-	D	P	-	A	fixed/float	T_n, T_m	T_r, T_c	-	True/False
QConv2D	R, C, N, M	K, L	-	-	P	-	A	int	T_n, T_m	T_r, T_c	-	True/False
QDense	N, M	-	-	-	-	-	A	int	T_n, T_m	-	-	-
Residual connections												
Add	R, C, N	-	-	-	-	-	-	any	T_n	T_r, T_c	-	-
Copy	R, C, N	-	-	-	-	-	-	any	T_n	T_r, T_c	True/False	-
Duplicate	-	-	-	-	-	-	-	any	T_n	-	-	-

A: Linear, ReLU, Sigmoid Softmax

7.2.4 Available Configuration Options per Layer

All supported layers, along with their parameters through which the designer can configure the architecture, are summarized in Table 7.2. In addition to the conventional C++ implementation for the Conv2D layer, the optimized architectures for strided and dilated convolutions are realized through the SConv2D and DConv2D classes, respectively. Furthermore, the QConv2D and QDense classes handle the corresponding layers for quantized models.

The parameters for each layer can be categorized into two groups; the structural parameters, derived from the model definition, and the architectural parameters, which characterize the hardware architecture associated with each layer. Starting from the structural parameters, the "FMAP" and

"Kernel" parameters determine the size of the IFM and the kernel, respectively, bounding the computation loops within the layer. The parameters "Stride" and "Dilation" are exclusive to layers optimized for their corresponding convolutions, and define the named values. "Padding" is specific to convolutional layers, dictating the amount of padding on each side of the IFM. In the Pool2D layer, the "Pooling" parameter selects the pooling function to be implemented, either maximum or average. Finally, the "Activation" parameter determines which of the supported activation functions (i.e. Linear, ReLu, Sigmoid, Softmax) will be implemented in-place to the corresponding layer.

The architectural parameters are similar for all the supported layers, with slight variations due to the differences in input shapes. The "Datatype" parameter, defined as a C++ typename, specifies the arithmetic representation of input and output data for each layer. In quantized models, this parameter can be an integer representation of any precision, while for other variants of the Conv2D layer and for the Dense layer, it can be either a fixed- or floating-point datatype. The remaining layers support any representation as they do not involve complex mathematical operations. The "Unroll" and "Tilling" parameters determine the loop optimizations as detailed in Section 2.3. Unrolling generates T_n and T_m parallel units, where T_n and T_m denote the number of input and output feature maps, respectively. Layers that maintain the number of feature maps utilize only the T_n parameter. In the case of the Flatten layer, these parameters dictate the parallelization of data arriving at its input and those forwarded to its output, respectively. Tiling parameters are applicable to layers with 3D inputs, enabling the tiling of input feature arrival with respect to the size of the feature map R×C.

Finally, the instantiation of the available functional-safe architectures for the convolutional layers can be enabled by the "ABFT" parameter.

The Add, Copy, and Duplicate layers facilitate the hardware implementation of residual architectures. Within the C++ model, layer connectivity is achieved using the channel datatype (*ac_channel*) from the HLSLibs library [113]. To execute the Add operation on the outputs of two layers, both outputs must be accessible. However, since the pipeline structure may make one output available earlier than the other, it must be stored until required. This task is performed by the Copy layer. Meanwhile, the Duplicate layer, as its name suggests, forwards the value of one ac_channel, representing the output of a layer, to two others, serving as inputs for subsequent layers. The "Forward Out" parameter of the Copy layer, determines whether the stored data are necessary for another layer and should therefore be forwarded to it.

All these layers can be used to define any CNN architecture, which can be decomposed into these fundamental elements. However, when it comes to designing a hardware accelerator, the scalability of the CNN architecture depends on the available resources. Smaller architectures can be fully unrolled with ease, while larger ones may need to be partially unrolled or fully rolled to fit within the resource constraints. This constraint arises from the nature of dataflow accelerators, where each layer cannot be reused. Nonetheless, we are actively working on enhancing our framework to support implementations capable of reusing existing layers.

In addition to computational resources, larger CNN architectures may also entail a higher number of parameters. To fit these parameters into the local buffers, a tradeoff may be studied between the data width and the accuracy degradation. In any case, very big architectures utilize the external memory, thus leading to increased memory communication and potentially impacting overall throughput.

7.3 Evaluation

To showcase the capabilities of the proposed framework, first, we illustrate its effectiveness through three distinct application scenarios. In each case, we crafted and accelerated a unique machine learning model on an FPGA platform. Second, we compare the properties of the proposed C++

library and the efficiency of the hardware generated from it to other accelerator frameworks and generators found in the literature.

7.3.1 Evaluation Setup

The models for the three applications were first implemented and trained in PyTorch. We then converted them into C++ using our proposed Python API. The C++ code was then used to generate RTL code through Catapult HLS. Afterward, the Verilog code was synthesized using Vivado 2022.1 targeting a clock frequency of 100MHz, and embedded onto the Virtex-Ultrascale VCU108 Evaluation board for practical application.

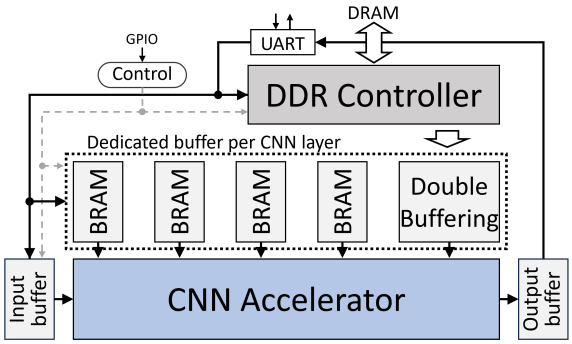


Figure 7.5: Overall architecture of the accelerator system implemented for the Virtex-Ultrascale VCU108 Evaluation board.

The overall design that is mapped in the FPGA and includes the proposed CNN accelerator is depicted in Figure 7.5. The overall accelerator block consists of four main parts: (a) the CNN accelerator; (b) a set of BRAMs that store the parameters per layer of the model; (c) the Double Data Rate (DDR) memory controller; and (d) a Universal Asynchronous Receiver-Transmiter (UART) interface for initializing externally on-chip buffering structures. The logic outside the "Accelerator" block is fixed and it is the same for all examined applications. When the number of parameters of specific layers exceeds the in-memory storage of the FPGA chip the "Double-buffering" module is instantiated to transfer online all necessary weight and biases parameters from the external DRAM. When the parameters of the CNN can be fully stored inside the on-chip memory of the FPGA "Double-buffering" and DRAM are not utilized.

7.3.2 Examined CNN Models

LeNet

For the first application we designed LeNet [73], a simple image classification model which was trained for the MNIST [26] dataset and its parameters were quantized to INT8 representation using quantization-aware training. The model receives input images of size 28×28 and returns a 10-element vector that classifies the input into one of the numbers from 0 to 9.

In this case, we designed the hardware without any specific latency requirements, targeting on a small area footprint. To achieve this, we unroll each convolutional layer only with respect to the size of the kernel $K \times L$, allowing the dot product of $K \times L$ elements to be performed in parallel. The remaining components of the architecture were not unrolled, instead they were designed as a fully pipelined architecture, resulting in a sequential computation for each input and output feature map.

Table 7.3: Layer characteristics and the unroll factors per layer for the convolutional and fully connected layers of the three applications under review.

Application	Layer	R	C	N	M	K	L	Tn	Tm
LeNet	conv-0	28	28	1	6	5	5	1	1
	conv-1	14	14	6	16	5	5	1	1
	fc-0	-	-	400	120	-	-	1	1
	fc-1	-	-	120	84	-	-	1	1
	fc-2	-	-	84	10	-	-	1	1
MLPerf Tiny	conv-0	32	32	3	16	3	3	1	1
	conv-1	32	32	16	16	3	3	1	1
	conv-2	32	32	16	16	3	3	1	1
	conv-3 (s2)	32	32	16	32	3	3	1	1
	conv-4	16	16	32	32	3	3	1	1
	conv-5 (s2)	32	32	16	32	1	1	1	1
	conv-6 (s2)	16	16	32	64	3	3	1	1
	conv-7	8	8	64	64	3	3	1	1
	conv-8 (s2)	16	16	32	64	1	1	1	1
	fc-0	-	-	64	10	-	-	1	1
Human activity recognition	conv-0	38	114	3	18	3	13	3	2
	conv-1	36	102	18	32	3	13	2	2
	conv-2	34	90	32	32	3	13	2	2
	fc-0	-	-	79872	164	-	-	8	4
	fc-1	-	-	164	2	-	-	4	2

The small size of the model allows the complete storage of its parameters at the on-chip memory. The key characteristics of each layer of the model that define its hardware architecture are summarized in Table 7.3. The selected architecture achieves a latency of 0.8ms for computing a single inference operation, and due to the pipelined design, it produces a new output result every 0.6ms.

After quantization-aware training, LeNet achieved an accuracy of 98% on 1000 samples from the MNIST dataset. The corresponding hardware implementation of LeNet, tested on the same dataset, showed a slight reduction in accuracy, achieving 95.5%.

MLPerf Tiny

For the second scenario, we choose to accelerate MLPerf Tiny [6], a TinyML model that belongs to the MLPerf [98] benchmarks collection. TinyML models have been proven crucial for edge devices as they manage to achieve high accuracy and throughput while maintaining a low power and area profile. MLPerf Tiny has multiple variants depending on the application, however, in this case we selected the image classification model, trained for the CIFAR10 [68] dataset and quantized to a 16-bit fixed-point representation.

Our goal is to design a hardware accelerator that can support a 60fps input frame rate, meaning that a new input will insert the architecture every 16.6ms. To achieve that, we have fully unrolled the kernel loops K, L, while the rest of the loops remain fully rolled and pipelined, as shown in Table 7.3. In addition, the convolutional layers with stride 2, which are found in the model and are denoted as (s2) on Table 7.3, have been implemented using the optimized architecture of LeapConv [32] that is integrated into the proposed library.

The selected architecture presents a 12.8ms latency, while its pipelined implementation enables a throughput of 11ms. After running 1000 inference operations, the accuracy of the implemented accelerator was measured to be 83%, very close to the 86% reported by the MLPerf Tiny benchmarks [6].

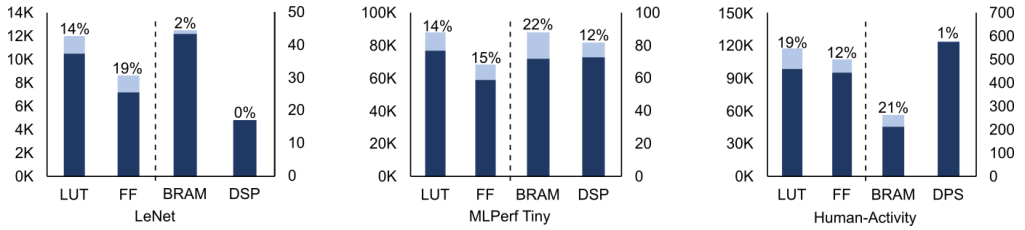


Figure 7.6: The resource utilization for the three applications implemented for the Virtex Ultrascale VCU108 Evaluation board.

Human-activity recognition

In the third scenario, we opted for a technique tailored for recognizing human activities based on joint awareness [89]. In this particular instance, a 2-dimensional CNN architecture is utilized to categorize five typical daily activities in indoor settings using 3-D skeletal data. The CNN model comprises three convolutional layers, followed by two fully connected layers, as shown in Table 7.3. In this approach, the input to the 2-D CNN model consists of a vector containing the x, y, and z coordinates of the human body's joints, identified as the most indicative for each action scenario.

To identify the five different activities, the model has been trained separately for each one, and during the inference, each input has to pass five times from the CNN, each time using a different set of parameters. Each input is constructed using 114 frames of a video. The data are being pre-processed and fed into the accelerator through the UART interface with a Baud Rate of 921600, meaning a new input will arrive almost every 80ms. For the accelerator to perform the five inference operations inside this time period, each inference should take at maximum 16ms, while the updating of the on-chip memory of the FPGA with the next set of parameters is performed in parallel to the inference operation, and after each layer is finished.

Every layer of the CNN, except the first fully connected layer, *fc-0*, has its parameters stored inside the on-chip memory. In the case of *fc-0*, the big size of its parameters, suggests that only a part of them will be stored inside the on-chip memory, which will update its content on-line during the inference operation.

Looking at the parameters outlined in Table 7.3, the kernel loops have been fully unrolled, while the loops that iterate of the N and M dimensions have been partially unrolled by the factors of T_n and T_m respectively. These architectural choices facilitate the implementation of a hardware structure adhering to latency requirements by achieving a latency of 16ms per inference operation.

In terms of accuracy, the implemented accelerator achieves an overall accuracy of 84.2%, showing a minimal reduction compared to the 86.4% accuracy reported by the corresponding software model.

The observed accuracy degradation in all application examples is attributed to the utilization of quantized fixed-point arithmetic in all operations, relative to the floating-point representation employed in the reference model. This accuracy loss can be ameliorated by allowing either in-training quantization or employ more sophisticated post-training quantization techniques. Selecting the appropriate quantization approach is orthogonal to the introduced HLS-design framework introduced in this work.

7.3.3 Hardware Complexity Results

Using the parameters from Table 7.3, we synthesized the three models for the Virtex-Ultrascale VCU108 Evaluation Board. For our evaluation, we designed each convolutional layer accompanied by a checksum checker to monitor the correctness of the computations. The resource utilization

for each of the three applications is summarized in Figure 7.6. Each bar represents the sum of resources required for the actual CNN accelerator and the resources associated with the checker component of the design. In each case, the checker component is relatively small compared to the overall architecture, contributing approximately 16% to the total resource utilization. The additional BRAMs, that are observed in the MLPerf Tiny and Human-activity recognition cases, are employed to build the accumulators within the design's checkers. This quantity of BRAMs correlates directly with both the number and size of convolutional layers within the CNN architecture, as the number of IFMs per layer directly impacts the count of accumulators in the checker's circuit. Furthermore, the increased count of convolutional layers in the MLPerf Tiny application results in an increased number of checkers, necessitating a greater number of DSPs to execute these operations efficiently.

Looking at the overall resource utilization, the compact size of LeNet results in very low resource usage, while its 8-bit quantization minimizes the demand for DSPs. MLPerf Tiny exhibits higher resource utilization due to its increased number of layers and its 16-bit quantization. Finally, the human-activity recognition model, despite its smaller architecture compared to MLPerf Tiny, consumes a significant amount of resources due to the increased hardware unrolling.

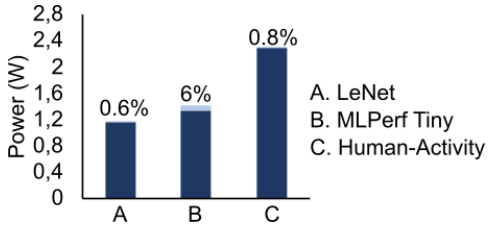


Figure 7.7: Power consumption for the three applications implemented on the Virtex-ultrascale VCU108 Evaluation board at 100MHz.

The power consumption of the three models is illustrated in Figure 7.7. Power measurements were conducted through post-implementation simulations using Vivado 2022.1, following the system organization described in Section 7.3.1. The input data utilized to estimate power consumption aligns with the data employed in our accuracy evaluation. For the first two cases, input data was sourced from the respective datasets, namely MNIST [26] and CIFAR10 [68]. In the third case, input data was collected from a group of healthy individuals with an average age of 69 ± 5 years. These individuals were free from mobility or cognitive impairments and participated in motion capture sessions [85].

For each application, we provide the power consumption for the accelerator and the associated BRAMs, excluding the UART interfaces and DDR memory. Additionally, we highlight the contribution of the checksum checker to the total power consumption. This power overhead is minimal, accounting for less than 1% in the cases of the LeNet and Human-activity recognition applications. However, in the case of MLPerf Tiny, it can reach up to 6%, a result of the higher DSP utilization. In summary, the power overhead introduced by the checker is relatively small. This demonstrates that the proposed library offers a low-power solution suitable for functional-safe architectures.

7.3.4 Comparison with State-Of-The-Art

To provide a comprehensive overview of the proposed automated CNN design framework, we compare it with similar state-of-the-art design frameworks both in terms of available features and in terms of the hardware complexity of the designed CNN accelerators for FPGAs.

Table 7.4 summarizes the properties of the most closely related design frameworks. Unlike the

Table 7.4: Summary of the various accelerator frameworks found in the literature, in comparison to the proposed library.

Framework, Library	Model Instantiation	Target Platform	Accelerator architecture	Design Method	Representation		Quantized models	Loop Unroll	Pipeline	Optimized Conv. layers	Residual	Functional Safety
					Floats	Fixed						
hls4ml [30]	Tensorflow, Torch	ASIC, FPGA	Dataflow	HLS	Any	Any	✓	✓	✓	✗	✓	✗
L. Kalms et al. [60]	C++	FPGA	Dataflow	HLS	FP32	Any	✓	✓	✓	✗	✗	✗
FlexCNN [8]	ONNX	FPGA	Systolic Array	HLS	FP32, FP16	FIX16, FIX8	✗	✗	✓	Dilate, Transpose	✓	✗
fpgaConvNet [126]	Caffe, Torch	FPGA	Dataflow	HLS	FP32	Any	✗	✓	✓	✗	✓	✗
Caffeine [143]	Caffe, Torch, Tensorflow	FPGA	Systolic-like	HLS	FP32, FP16	Any	✗	✓	✓	✗	✗	✗
DnnWeaver [109]	Caffe	FPGA	Systolic-like	RTL	✗	Any	✗	✓	✓	✗	✗	✗
Angel-Eye [45]	Caffe	FPGA	Dataflow	RTL	✗	FIX16, FIX8	✓	✓	✓	✗	✗	✗
FP-DNN [44]	Tensorflow	FPGA	Systolic Array	RTL-HLS	FP32	Any	✗	-	-	✗	✓	✗
Proposed	Tensorflow, Torch	ASIC, FPGA	Dataflow	HLS	Any	Any	✓	✓	✓	Stride, Dilate	✓	✓

Table 7.5: A comparison of the resources utilization for various FPGA accelerators designed for VGG-16 and derived from the different frameworks.

Framework	Platform	Precision	Frequency (MHz)	Resources				Performance			Power (W)
				LUT (K)	FF (K)	BRAM	DSP	GOPs	GOPs/kLUT	GOPs/DSP	
FlexCNN	Alveo U250	FIX16	241	682	359	1124	4463	1543.4	2.26	0.34	-
fpgaConvNet	Zynq 7020	FIX16	125	53.2	-	-	200	48.53	0.91	0.24	1.75
Caffeine	Virtex-7 690T	FIX16	150	300	-	-	2848	354	1.18	0.12	26
Angel-Eye	Zynq 7045	FIX16	150	182.6	127.7	486	780	187.8	1.03	0.24	9.63
DnnWeaver	Zynq 7020	FIX16	150	35.07	33.2	140	140	31.38	0.91	0.22	-
Proposed	UltraScale VCU108	FIX16	100	33.64	27.76	1728	176	31.5	0.93	0.18	3.62

majority of state-of-the-art, the proposed library supports arbitrary-precision floating-point arithmetic, by leveraging the –in house– developed open-source floating-point library FastFloat4HLS [54]. Similar to FlexCNN [8] that utilizes a versatile systolic array to optimize the computation of Dilated and Transpose convolutions, the proposed design framework integrates optimized buffering architectures that enable power efficient dataflow implementations for strided and dilated convolutions. Additionally, supporting the online self-checking of the output of convolution layers is a unique feature of the proposed approach.

Table 7.5 summarizes the performance of various FPGA implementations of VGG16. Each case has been synthesized by a different design framework including state-of-the-art and the proposed one. Resource utilization data for the state-of-the-art accelerators under comparison was sourced from the corresponding work's reported results, with varying FPGA platforms used for different accelerators. For FlexCNN the reported performance includes only convolution layers of VGG16, since this framework does not support fully-connected layers. To ensure a fair comparison, performance is evaluated relative to resource utilization, as indicated in the GOPs/kLUT and GOPs/DSP columns of Table 7.5. Overall, the proposed framework demonstrates comparable results to the majority of accelerators. Likewise, a similar observation can be made concerning power consumption. According to the reported power of the proposed accelerator, 0.91W accounts for device static power consumption, with the remaining 2.71W attributed to dynamic power consumption. Although the proposed accelerator demonstrates higher power consumption compared to fgpaConvNet, it still remains relatively low in comparison to the rest state-of-the-art accelerators.

7.4 Conclusions

CNNs have emerged as powerful tools in ML applications. This widespread adoption has underscored the necessity for specialized hardware accelerators to enhance their performance. In response to this demand, the framework presented here offers a unique solution by seamlessly integrating the simplicity of Python-based CNN modeling with the adaptability of HLS. This combination enables the automated generation of hardware accelerators tailored to specific CNN architectures. Through rigorous evaluation, the framework demonstrates its capability to produce efficient hardware solutions for various CNN models. Furthermore, compared to other state-of-the-art generators and frameworks, the proposed approach introduces new features while yielding hardware performance comparable to other state-of-the-art approaches.

8 Conclusions

8.1 Summary

This dissertation introduces a flexible and automated design framework that streamlines the development of hardware accelerators for Convolutional Neural Networks (CNNs) using High-Level Synthesis (HLS). Aimed at meeting the performance and efficiency demands of machine learning inference in edge and embedded systems, the framework creates a seamless path from high-level CNN models—developed in platforms such as TensorFlow or PyTorch—to optimized hardware implementations. By abstracting low-level hardware complexities, the framework accelerates development cycles and reduces manual design effort.

Central to the framework is a customizable accelerator generator capable of exploring a wide array of architectural configurations. It offers fine-grained control over key design dimensions, including parallelism, buffer organization, tiling schemes, and dataflow strategies. Unlike traditional solutions, the framework incorporates features that go beyond performance tuning, such as support for safety-critical applications and advanced memory management techniques. It also provides flexible options for arithmetic precision and data formats, catering to a broad spectrum of application needs and hardware budgets.

To address the inefficiencies in specialized convolution types, the framework includes two dedicated hardware engines. LazyDCstream targets dilated convolutions and employs a sliding-window dataflow optimized for structured reuse. Its defining innovation is a "lazy" data fetch mechanism that ensures consistent and minimal memory traffic, regardless of dilation parameters. LeapConv is tailored for strided convolution operations, which are typically irregular and inefficient in hardware. It restructures strided layers into multiple concurrent unity-stride computations that are then unified, enhancing data reuse and reducing redundant memory access.

The thesis also presents ConvGuard, a runtime fault detection mechanism based on Algorithm-Based Fault Tolerance (ABFT). ConvGuard leverages a newly identified invariance property in convolutions to compute checksums from input boundaries, enabling output verification without storing internal intermediate states. This makes the solution highly efficient in terms of area and power, while maintaining compatibility with various convolution configurations, including different kernel sizes and strides.

The framework also supports a broad spectrum of data representation options, from traditional floating-point to fixed-point and integer formats. Each format is paired with tailored arithmetic units to ensure efficient computation under varying precision requirements.

In conclusion, this research delivers a comprehensive and practical solution for the automated generation of CNN hardware accelerators. By uniting high-level design automation with low-level architectural optimizations, it enables scalable, safe, and efficient deployment of deep learning models in constrained environments, addressing critical challenges in modern machine learning hardware design.

8.2 Future Work

As machine learning models continue to grow in complexity and diversity, one of the key challenges for future dataflow ML accelerators lies in supporting highly irregular and dynamic computation patterns. Emerging models—such as transformers, dynamic neural networks, and sparse architectures—exhibit behavior that deviates significantly from the regular, statically scheduled computation patterns traditionally favored by dataflow architectures. Designing accelerators that can adaptively manage data dependencies, irregular memory access patterns, and variable execution paths without sacrificing energy efficiency or throughput remains an open and pressing problem. Future research must explore hybrid scheduling techniques, dynamic dataflow graph reconfiguration, and fine-grained runtime control to bring dataflow paradigms in line with the evolving needs of modern ML workloads.

Another major research challenge involves co-optimizing data movement, on-chip memory hierarchy, and interconnect topologies for next-generation accelerators. As dataflow architectures scale to support larger models and higher throughput, the cost of moving data—rather than computation—becomes the dominant factor in both energy consumption and latency. Achieving optimal data locality while maintaining flexibility requires advances in compiler-guided memory partitioning, intelligent buffering, and reconfigurable interconnects. Additionally, the growing importance of multi-tenant and cloud-edge deployment scenarios calls for dataflow architectures that can dynamically share and isolate resources without incurring significant performance degradation. Addressing these challenges will be critical for sustaining the scalability, efficiency, and adaptability of dataflow-based ML acceleration in the years ahead.

Designing ML accelerators for ultra-low-power Systems-on-Chip (SoCs) presents a unique set of research challenges that go beyond traditional performance-driven optimization. In these power-constrained environments—typical of wearable, implantable, or always-on edge devices—the energy cost of memory access, data movement, and leakage dominates computation, requiring a fundamental rethinking of accelerator microarchitecture. Key challenges include developing energy-proportional dataflows, aggressively minimizing off-chip memory usage, and exploiting approximate computing or in-situ processing to reduce power without compromising essential accuracy. Additionally, ultra-low-power ML accelerators must support fine-grained power gating, scalable voltage-frequency domains, and efficient wake-up mechanisms to adapt to intermittent workloads and energy harvesting conditions. Balancing programmability, model flexibility, and extreme energy efficiency remains an open problem, especially as applications demand increasingly complex inference tasks on sub-milliwatt budgets.

Finally, future research in functional safety for ML accelerators must evolve beyond basic fault detection mechanisms to address end-to-end system-level safety in increasingly autonomous and critical applications. While current approaches, such as error detection in convolutional layers or arithmetic units, provide localized protection, they fall short of guaranteeing correct behavior at the system level, especially under silent data corruptions or adversarial fault scenarios. Ensuring end-to-end safety requires a holistic approach that encompasses model-level robustness, data integrity across the memory hierarchy, and fault propagation analysis throughout the entire inference pipeline. Moreover, future ML accelerators must support adaptive safety mechanisms that can dynamically adjust protection levels based on workload criticality, real-time operating conditions, or detected anomalies. Integrating formal verification, resilience-aware compilation, and runtime monitoring into the accelerator stack will be essential to meet the stringent reliability demands of domains such as autonomous vehicles, medical devices, and industrial automation, where incorrect predictions, even if rare, can have catastrophic consequences.

Bibliography

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016.

[2] Ankur Agrawal, Silvia M. Mueller, Bruce M. Fleischer, Xiao Sun, Naigang Wang, Jungwook Choi, and Kailash Gopalakrishnan. DLFloat: A 16-b Floating Point format designed for Deep Learning Training and Inference. In *Int. Symp. on Computer Arithmetic (ARITH)*, 2019.

[3] Kosmas Alexandridis and Giorgos Dimitrakopoulos. Online alignment and addition in multiterm floating-point adders. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 33(4):1182–1186, 2025.

[4] AMD. Vitis HLS.

[5] Michael Andersch, Greg Palmer, Ronny Krashinsky, Nick Stam, Vishal Mehta, Gonzalo Brito, and Sridhar Ramaswamy. NVIDIA Hopper architecture, 2022.

[6] Colby Banbury, Vijay Janapa Reddi, Peter Torelli, Jeremy Holleman, Nat Jeffries, Csaba Kiraly, Pietro Montino, David Kanter, Sebastian Ahmed, Danilo Pau, et al. Mlperf tiny benchmark. *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*, 2021.

[7] Ron Banner, Yury Nahshan, and Daniel Soudry. Post training 4-bit quantization of convolutional networks for rapid-deployment. *Advances in Neural Information Processing Systems*, 32, 2019.

[8] Suhail Basalama, Atefeh Sohrabizadeh, Jie Wang, Licheng Guo, and Jason Cong. Flexcnn: An end-to-end framework for composing cnn accelerators on fpga. *ACM Transactions on Reconfigurable Technology and Systems*, 16(2):1–32, 2023.

[9] Robert C Baumann. Radiation-induced soft errors in advanced semiconductor technologies. *IEEE Trans. on Device and Materials Reliability*, 5(3):305–316, 2005.

[10] Ismet Bayraktaroglu and Alex Orailoglu. Concurrent test for digital linear systems. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 20(9):1132–1142, 2001.

[11] Michaela Blott et al. FINN-R: An end-to-end deep-learning framework for fast exploration of quantized neural networks. *ACM Trans. on Reconfigurable Technology and Systems*, 11(3):1–23, 2018.

[12] Shekhar Borkar. Designing reliable systems from unreliable components: the challenges of transistor variability and degradation. *IEEE Micro*, 25(6):10–16, 2005.

[13] Neil Burgess, Jelena Milanovic, Nigel Stephens, Konstantinos Monachopoulos, and David Mansell. Bfloat16 processing for neural networks. In *2019 IEEE 26th Symposium on Computer Arithmetic (ARITH)*, pages 88–91. IEEE, 2019.

[14] Cadence. Genus Synthesis Solution.

[15] Cadence. Innovus Implementation System.

[16] Narendra Chaudhary, Sanchit Misra, Dhiraj Kalamkar, Alexander Heinecke, Evangelos Georganas, Barukh Ziv, Menachem Adelman, and Bharat Kaul. Efficient and Generic 1D Dilated Convolution Layer for Deep Learning. *arXiv:2104.08002*, 2021.

[17] Chun-Fu Richard Chen, Rameswar Panda, Kandan Ramakrishnan, Rogerio Feris, John Cohn, Aude Oliva, and Quanfu Fan. Deep analysis of cnn-based spatio-temporal representations for action recognition. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 6165–6175, 2021.

[18] Jieyang Chen, Xin Liang, and Zizhong Chen. Online algorithm-based fault tolerance for cholesky decomposition on heterogeneous systems with GPUs. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 993–1002, 2016.

[19] Liang-Chieh Chen et al. Deeplab: Semantic image segmentation with deep convolutional nets, à trous convolution, and fully connected crfs. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 40(4):834–848, 2017.

[20] Yu-Hsin Chen, Tien-Ju Yang, Joel Emer, and Vivienne Sze. Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices. *IEEE JETCAS*, 9(2):292–308, 2019.

[21] Jungwook Choi, Zhuo Wang, Swagath Venkataramani, Pierce I-Jen Chuang, Vijayalakshmi Srinivasan, and Kailash Gopalakrishnan. Pact: Parameterized clipping activation for quantized neural networks. *arXiv preprint arXiv:1805.06085*, 2018.

[22] Yoni Choukroun, Eli Kravchik, Fan Yang, and Pavel Kisilev. Low-bit quantization of neural networks for efficient inference. In *2019 IEEE/CVF International Conference on Computer Vision Workshop (ICCVW)*, pages 3009–3018. IEEE, 2019.

[23] Steven Colleman, Man Shi, and Marian Verhelst. Coac: Cross-layer optimization of accelerator configurability for efficient cnn processing. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2023.

[24] Florent De Dinechin, Bogdan Pasca, and E Normale. Custom arithmetic datapath design for FPGAs using the FloPoCo core generator. *IEEE Design & Test of Computers*, 28(4):18–27, 2011.

[25] Allison McCarn Deiana et al. Applications and techniques for fast machine learning in science. *Frontiers in big Data*, 5, 2022.

[26] Li Deng. The mnist database of handwritten digit images for machine learning research. *IEEE Signal Processing Magazine*, 29(6):141–142, 2012.

[27] Giorgos Dimitrakopoulos, Kostas Galanopoulos, Christos Mavrokefalidis, and Dimitris Nikolos. Low-power leading-zero counting and anticipation logic for high-speed floating point units. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 16(7):837–850, 2008.

[28] Vincent Dumoulin and Francesco Visin. A guide to convolution arithmetic for deep learning. *arXiv:1603.07285*, 2016.

[29] Utku Evci, Trevor Gale, Jacob Menick, Pablo Samuel Castro, and Erich Elsen. Rigging the lottery: Making all tickets winners. In *International Conference on Machine Learning*, pages 2943–2952. PMLR, 2020.

[30] Farah Fahim, Benjamin Hawks, Christian Herwig, James Hirschauer, Sergo Jindariani, Nhan Tran, Luca P Carloni, Giuseppe Di Guglielmo, Philip Harris, Jeffrey Krupa, et al. hls4ml: An open-source codesign workflow to empower scientific low-power machine learning devices. *arXiv preprint arXiv:2103.05579*, 2021.

[31] Dionysios Filippas, Nikolaos Margomenos, Nikolaos Mitianoudis, Chrysostomos Nicopoulos, and Giorgos Dimitrakopoulos. Low-cost online convolution checksum checker. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 30(2):201–212, 2021.

[32] Dionysios Filippas, Chrysostomos Nicopoulos, and Giorgos Dimitrakopoulos. Leapconv: An energy-efficient streaming convolution engine with reconfigurable stride. In *2022 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 200–205. IEEE, 2022.

[33] Dionysios Filippas, Chrysostomos Nicopoulos, and Giorgos Dimitrakopoulos. Templatized fused vector floating-point dot product for high-level synthesis. *Journal of Low Power Electronics and Applications*, 12(4):56, 2022.

[34] Dionysios Filippas, Chrysostomos Nicopoulos, and Giorgos Dimitrakopoulos. Streaming dilated convolution engine. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2023.

[35] Dionysios Filippas, Christodoulos Peltekis, Giorgos Dimitrakopoulos, and Chrysostomos Nicopoulos. Reduced-precision floating-point arithmetic in systolic arrays with skewed pipelines. In *2023 IEEE 5th International Conference on Artificial Intelligence Circuits and Systems (AICAS)*, pages 1–5, 2023.

[36] Dionysios Filippas, Christodoulos Peltekis, Vasileios Titopoulos, Ioannis Kansizoglou, Georgios CH Sirakoulis, Antonios Gasteratos, and Giorgos Dimitrakopoulos. A high-level synthesis library for synthesizing efficient and functional-safe cnn dataflow accelerators. *IEEE Access*, 12:57194–57208, 2024.

[37] Michael Fingeroff. *High-level synthesis: blue book*. Xlibris Corporation, 2010.

[38] Jonathan Frankle and Michael Carbin. The lottery ticket hypothesis: Finding sparse, trainable neural networks. *arXiv preprint arXiv:1803.03635*, 2018.

[39] Sameh Galal and Mark Horowitz. Latency sensitive fma design. In *IEEE Symp. on Comp. Arithmetic (ARITH)*, pages 129–138, 2011.

[40] Hasan Genc, Seah Kim, Alon Amid, Ameer Haj-Ali, Vighnesh Iyer, Pranav Prakash, Jerry Zhao, Daniel Grubb, Harrison Liew, Howard Mao, et al. Gemmini: Enabling systematic deep-learning architecture evaluation via full-stack integration. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pages 769–774. IEEE, 2021.

[41] Amir Gholami, Sehoon Kim, Zhen Dong, Zhewei Yao, Michael W. Mahoney, and Kurt Keutzer. A survey of quantization methods for efficient neural network inference. *arXiv preprint arXiv:2103.13630*, 2021.

[42] R.C. Gonzalez and R.E. Woods. *Digital Image Processing*. Pearson, 3rd edition, 2007.

[43] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

[44] Yijin Guan, Hao Liang, Ningyi Xu, Wenqiang Wang, Shaoshuai Shi, Xi Chen, Guangyu Sun, Wei Zhang, and Jason Cong. Fp-dnn: An automated framework for mapping deep neural networks onto fpgas with rtl-hls hybrid templates. In *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 152–159. IEEE, 2017.

[45] Kaiyuan Guo, Lingzhi Sui, Jiantao Qiu, Jincheng Yu, Junbin Wang, Song Yao, Song Han, Yu Wang, and Huazhong Yang. Angel-eye: A complete design flow for mapping cnn onto embedded fpga. *IEEE transactions on computer-aided design of integrated circuits and systems*, 37(1):35–47, 2017.

[46] Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.

[47] Siva Kumar Sastry Hari, Michael Sullivan, Timothy Tsai, and Stephen W Keckler. Making convolutions resilient via algorithm-based error detection techniques. *IEEE Trans. on Dependable and Secure Computing*, 2021.

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

[49] Brian Hickmann and Dennis Bradford. Experimental analysis of matrix multiplication functional units. In *IEEE Symp. on Comp. Arithmetic (ARITH)*, pages 116–119, 2019.

[50] Brian Hickmann, Jieasheng Chen, Michael Rotzin, Andrew Yang, Maciej Urbanski, and Sasikanth Avancha. Intel Nervana Neural Network Processor-T (NPP-T) Fused Floating Point Many-Term Dot Product. In *IEEE Symp. on Comp. Arithmetic (ARITH)*, pages 133–136, 2020.

[51] Andrew Hopkins. Silicon evolution for the automotive revolution. ARM whitepaper, 2019.

[52] Andrew G Howard et al. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv:1704.04861*, 2017.

[53] Kuang-Hua Huang and Jacob A. Abraham. Algorithm-based fault tolerance for matrix operations. *IEEE Trans. on Computers*, C-33(6):518–528, 1984.

[54] IC-Lab-DUTH Repository. FastFloat4HLS C++ Library, 2022.

[55] Lenos Ioannou, Abdullah Al-Dujaili, and Suhaib A. Fahmy. High throughput spatial convolution filters on FPGAs. *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, 28(6):1392–1402, 2020.

[56] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2704–2713, 2018.

[57] Norman P. Jouppi et al. In-datacenter performance analysis of a tensor processing unit. In *Int. Symp. on Comp. Arch. (ISCA)*, page 1–12, 2017.

[58] Norman P. Jouppi, Doe Hyun Yoon, Matthew Ashcraft, Mark Gottscho, Thomas B. Jablin, George Kurian, James Laudon, Sheng Li, Peter Ma, Xiaoyu Ma, Thomas Norrie, Nishant Patil, Sushma Prasad, Cliff Young, Zongwei Zhou, and David Patterson. Ten Lessons From Three Generations Shaped Google's TPUv4i. In *International Symposium on Computer Architecture (ISCA)*, pages 1–14. IEEE, 2021.

[59] William Kahan. Ieee standard 754 for binary floating-point arithmetic. *Lecture Notes on the Status of IEEE*, 754(94720-1776):11, 1996.

[60] Lester Kalms, Pedram Amini Rad, Muhammad Ali, Arsany Iskander, and Diana Göhringer. A parametrizable high-level synthesis library for accelerating neural networks on fpgas. *Journal of Signal Processing Systems*, 93(5):513–529, 2021.

[61] Philipp Käsgen and Markus Weinhardt. *Using Template Metaprogramming for Hardware Description*. Universität Tübingen, 2018.

[62] Himanshu Kaul, Mark Anders, Sanu Mathew, Seongjong Kim, and Ram Krishnamurthy. Optimized fused floating-point many-term dot-product hardware for machine learning accelerators. In *IEEE Symp. on Comp. Arithmetic (ARITH)*, pages 84–87, 2019.

[63] Donghyun Kim and Lee-Sup Kim. A floating-point unit for 4d vector inner product with reduced latency. *IEEE Trans. on computers*, 58(7):890–901, 2008.

[64] Chen Kong and Simon Lucey. Take it in your stride: Do we need striding in CNNs? *arXiv e-prints*, December 2017.

[65] Israel Koren and C. Krishna. *Fault-Tolerant Systems*. Morgan Kaufmann, 2020.

[66] Mikhail V Koroteev. Bert: a review of applications in natural language processing and understanding. *arXiv preprint arXiv:2103.11943*, 2021.

[67] Raghuraman Krishnamoorthi. Quantizing deep convolutional networks for efficient inference: A whitepaper. *arXiv preprint arXiv:1806.08342*, 2018.

[68] Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. *Tech Report*, 2009.

[69] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Communications of the ACM*, 60(6):84–90, 2017.

[70] Hsiang-Tsung Kung. *Why systolic architecture?* Design Research Center, Carnegie-Mellon University, 1982.

[71] Hsiang Tsung Kung and Charles E Leiserson. Systolic arrays (for vlsi). In *Sparse Matrix Proceedings 1978*, volume 1, pages 256–282. Society for industrial and applied mathematics Philadelphia, PA, USA, 1979.

[72] Tomas Lang and Javier D Bruguera. Floating-point multiply-add-fused with reduced latency. *IEEE Transactions on Computers*, 53(8):988–1003, 2004.

[73] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[74] Jounghoo Lee, Jinwoo Choi, Jaeyeon Kim, Jinho Lee, and Youngsok Kim. Dataflow mirroring: Architectural support for highly efficient fine-grained spatial multitasking on systolic-array npus. In *Design Automation Conference (DAC)*, pages 247–252. IEEE, 2021.

[75] Sae Kyu Lee, Paul N. Whatmough, David Brooks, and Gu-Yeon Wei. A 16-nm always-on DNN processor with adaptive clocking and multi-cycle banked srams. *IEEE Journal of Solid-State Circuits*, 54(7):1982–1992, 2019.

[76] Guanpeng Li, Siva Kumar Sastry Hari, Michael Sullivan, Timothy Tsai, Karthik Pattabiraman, Joel Emer, and Stephen W. Keckler. Understanding error propagation in deep learning neural network (DNN) accelerators and applications. In *Proc. of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2017.

[77] Kai Li, Junzhuo Zhou, Boyu Li, Shuxing Yang, Sixiao Huang, Shaobo Luo, Wei Mao, and Hao Yu. A vector systolic accelerator for multi-precision floating-point high-performance computing. In *IEEE Intern. Conf. on Artificial Intelligence Circuits and Systems (AICAS)*, pages 226–229, 2022.

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

[79] Zhuang Liu, Hanzi Mao, Chao-Yuan Wu, Christoph Feichtenhofer, Trevor Darrell, and Saining Xie. A convnet for the 2020s. In *Proc. of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 11976–11986, 2022.

[80] Jian-Hao Luo, Jianxin Wu, and Weiyao Lin. Thinet: A filter level pruning method for deep neural network compression. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, Oct 2017.

[81] David R. Lutz. Fused multiply-add microarchitecture comprising separate early-normalizing multiply and add pipelines. In *IEEE Symp. on Computer Arithmetic*, pages 123–128, 2011.

[82] Yufei Ma et al. Optimizing the convolution operation to accelerate deep neural networks on FPGA. *IEEE Trans. on VLSI Systems*, 26(7):1354–1367, 2018.

[83] Thibaut Marty, Tomofumi Yuki, and Steven Derrien. Safe overclocking for CNN accelerators through algorithm-level error detection. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 39(12):4777–4790, 2020.

[84] Pascal Meinerzhagen, S. M. Yasser Sherazi, Andreas Burg, and Joachim Neves Rodrigues. Benchmarking of Standard-Cell Based Memories in the Sub-V_T Domain in 65-nm CMOS Technology. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 1(2):173–182, 2011.

[85] Dimitrios Menychtas, Nikolaos Petrou, Ioannis Kansizoglou, Erasmia Giannakou, Athanasios Grekidis, Antonios Gasteratos, Vassilios Gourgoulis, Eleni Douda, Ilias Smilios, Maria Michalopoulou, et al. Gait analysis comparison between manual marking, 2d pose estimation algorithms, and 3d marker-based system. *Frontiers in Rehabilitation Sciences*, 4, 2023.

[86] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, et al. Mixed precision training. *arXiv preprint arXiv:1710.03740*, 2017.

[87] Paulius Micikevicius, Dusan Stosic, Neil Burgess, Marius Cornea, Pradeep Dubey, Richard Grisenthwaite, Sangwon Ha, Alexander Heinecke, Patrick Judd, John Kamalu, et al. Fp8 formats for deep learning. *arXiv preprint arXiv:2209.05433*, 2022.

[88] Mohammad Motamedi, Philipp Gysel, Venkatesh Akella, and Soheil Ghiasi. Design space exploration of fpga-based deep convolutional neural networks. In *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 575–580. IEEE, 2016.

[89] Katerina Maria Oikonomou, Ioannis Kansizoglou, Pelagia Manaveli, Athanasios Grekidis, Dimitrios Menychtas, Nikolaos Aggelousis, Georgios Ch Sirakoulis, and Antonios Gasteratos. Joint-aware action recognition for ambient assisted living. In *2022 IEEE International Conference on Imaging Systems and Techniques (IST)*, pages 1–6. IEEE, 2022.

[90] ONNX GitHub repository. ONNX: Open Neural Network Exchange.

[91] Junhao Pan and Deming Chen. Accelerate Non-Unit Stride Convolutions with Winograd Algorithms. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASPDAC)*, page 358–364, 2021.

[92] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

[93] Christodoulos Peltekis, Dionysios Filippas, Giorgos Dimitrakopoulos, Chrysostomos Nicopoulos, and Dionisios Pnevmatikatos. ArrayFlex: A Systolic Array Architecture with Configurable Transparent Pipelining. In *Design Automation and Test in Europe (DATE)*, 2023.

[94] Lucian Petrica et al. Memory-efficient dataflow inference for deep CNNs on fpga. In *IEEE Intern. Conf. on Field-Programmable Technology (ICFPT)*, pages 48–55, 2020.

[95] Stanislaw J. Piestrak and Piotr Patronik. Design of fault-secure transposed fir filters protected using residue codes. In *Euromicro Conference on Digital System Design*, pages 575–582, 2014.

[96] Md Aminur Rab Ratul et al. Skin lesions classification using deep learning based on dilated convolution. *BioRxiv*, page 860700, 2020.

[97] Brandon Reagen, Udit Gupta, Lillian Pentecost, Paul Whatmough, Sae Kyu Lee, Niamh Mulholland, David Brooks, and Gu-Yeon Wei. Ares: A framework for quantifying the resilience of deep neural networks. In *Design Automation Conference (DAC)*, 2018.

[98] Vijay Janapa Reddi, Christine Cheng, David Kanter, Peter Mattson, Guenther Schmuelling, Carole-Jean Wu, Brian Anderson, Maximilien Breughe, Mark Charlebois, William Chou, Ramesh Chukka, Cody Coleman, Sam Davis, Pan Deng, Greg Diamos, Jared Duke, Dave Fick, J. Scott Gardner, Itay Hubara, Sachin Idgunji, Thomas B. Jablin, Jeff Jiao, Tom St. John, Pankaj Kanwar, David Lee, Jeffery Liao, Anton Lokhmotov, Francisco Massa, Peng Meng, Paulius Micikevicius, Colin Osborne, Gennady Pekhimenko, Arun Tejusve Raghunath Rajan, Dilip Sequeira, Ashish Sirasao, Fei Sun, Hanlin Tang, Michael Thomson, Frank Wei, Ephrem Wu, Lingjie Xu, Koichi Yamada, Bing Yu, George Yuan, Aaron Zhong, Peizhao Zhang, and Yuchen Zhou. Mlperf inference benchmark, 2019.

[99] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You Only Look Once: Unified, Real-Time Object Detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 779–788, 2016.

[100] Esther Roorda, Seyedramin Rasoulinezhad, Philip HW Leong, and Steven JE Wilton. Fpga architecture exploration for dnn acceleration. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 15(3):1–37, 2022.

[101] Frank Rosenblatt et al. *Principles of neurodynamics: Perceptrons and the theory of brain mechanisms*, volume 55. Spartan books Washington, DC, 1962.

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

[103] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015.

[104] Rick Salay, Rodrigo Queiroz, and Krzysztof Czarnecki. An analysis of ISO 26262: Using machine learning safely in automotive software, 2017.

[105] Hani H Saleh and Earl E Swartzlander. A floating-point fused dot-product unit. In *2008 IEEE International Conference on Computer Design*, pages 427–431. IEEE, 2008.

[106] Ananda Samajdar et al. A systematic methodology for characterizing scalability of DNN accelerators using scale-sim. In *IEEE Int. Symp. on Perf. Analysis of Systems and Software (ISPASS)*, pages 58–68, 2020.

[107] Martin S Schmookler and Kevin J Nowka. Leading zero anticipation and detection-a comparison of methods. In *Proceedings of the 15th IEEE Symposium on Computer Arithmetic*, pages 7–12, 2001.

[108] Peter-Michael Seidel and Guy Even. On the design of fast ieee floating-point adders. In *Proceedings of the 15th IEEE Symposium on Computer Arithmetic*, pages 184–194, 2001.

[109] Hardik Sharma, Jongse Park, Divya Mahajan, Emmanuel Amaro, Joon Kyung Kim, Chenkai Shao, Asit Mishra, and Hadi Esmaeilzadeh. From high-level deep neural models to fpgas. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12. IEEE, 2016.

[110] Ahmad Shawahna, Sadiq M. Sait, and Aiman El-Maleh. FPGA-based accelerators of deep learning networks for learning and classification: A review. *IEEE Access*, 7:7823–7859, 2019.

[111] Siemens EDA. Catapult High Level Synthesis.

[112] Siemens EDA. Questa Advanced Simulator.

[113] Siemens EDA. Algorithmic C (AC) Datatypes Reference Manual, 2022.

[114] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In *International Conference on Learning Representations*, 2015.

[115] Jongwook Sohn and Earl E Swartzlander. Improved architectures for a floating-point fused dot product unit. In *IEEE Symp. on Comp. Arithmetic (ARITH)*, pages 41–48, 2013.

[116] Greg Stitt, Abhay Gupta, Madison N. Emas, David Wilson, and Austin Baylis. Scalable Window Generation for the Intel Broadwell+Arria 10 and High-Bandwidth FPGA Systems. In *Proc. of the Intern. Symp. on Field-Programmable Gate Arrays*, page 173–182, 2018.

[117] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S Emer. Efficient processing of deep neural networks: A tutorial and survey. *Proceedings of the IEEE*, 105(12):2295–2329, 2017.

[118] Thierry Tambe, En-Yu Yang, Zishen Wan, Yuntian Deng, Vijay Janapa Reddi, Alexander Rush, David Brooks, and Gu-Yeon Wei. Algorithm-hardware co-design of adaptive floating-point encodings for resilient deep learning inference. In *Design Automation Conference (DAC)*, pages 1–6, 2020.

[119] Keisuke Tateno, Federico Tombari, Iro Laina, and Nassir Navab. CNN-SLAM: Real-Time Dense Monocular SLAM with Learned Depth Prediction. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 6243–6252, 2017.

[120] David B Thomas. Templatised soft floating-point for high-level synthesis. In *IEEE Intern. Symp. on Field-Programmable Custom Computing Machines (FCCM)*, pages 227–235, 2019.

[121] Yohann Uguen, Florent De Dinechin, Victor Lezaud, and Steven Derrien. Application-specific arithmetic in high-level synthesis tools. *ACM Trans. on Architecture and Code Optimization (TACO)*, 17(1):1–23, 2020.

[122] Ultralytics. YOLOv5.

[123] Yaman Umuroglu, Nicholas J Fraser, Giulio Gambardella, Michaela Blott, Philip Leong, Magnus Jahre, and Kees Vissers. Finn: A framework for fast, scalable binarized neural network inference. In *Proceedings of the 2017 ACM/SIGDA international symposium on field-programmable gate arrays*, pages 65–74, 2017.

[124] Aravind Vasudevan, Andrew Anderson, and David Gregg. Parallel multi channel convolution using general matrix multiplication. In *2017 IEEE 28th international conference on application-specific systems, architectures and processors (ASAP)*, pages 19–24. IEEE, 2017.

[125] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[126] Stylianos I Venieris and Christos-Savvas Bouganis. fpgaconvnet: Mapping regular and irregular convolutional neural networks on fpgas. *IEEE transactions on neural networks and learning systems*, 30(2):326–342, 2018.

[127] Shibo Wang and Pankaj Kanwar. BFloat16: The secret to high performance on Cloud TPUs. *Google Cloud Blog*, 4, 2019.

[128] Sying-Jyan Wang and N.K. Jha. Algorithm-based fault tolerance for FFT networks. *IEEE Trans. on Computers*, 43(7):849–854, 1994.

[129] Zhengyang Wang and Shuiwang Ji. Smoothed dilated convolutions for improved dense prediction. *Data Mining and Knowledge Discovery*, 35(4):1470–1496, 2021.

[130] Neil Weste and David Harris. *CMOS VLSI Design a Circuits and Systems Perspective*. Addison Wesley (3rd Edition), 2010.

[131] Paul Whatmough et al. FixyNN: Energy-efficient real-time mobile computer vision hardware acceleration via transfer learning. *Machine Learning and Systems*, 1:107–119, 2019.

[132] Paul Whatmough, Sae Lee, David Brooks, and Gu-Yeon Wei. DNN engine: A 28-nm timing-error tolerant sparse deep neural network processor for iot applications. *IEEE Journal of Solid-State Circuits*, PP:1–10, 06 2018.

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

[134] Sanghyun Woo, Shoubhik Debnath, Ronghang Hu, Xinlei Chen, Zhuang Liu, In So Kweon, and Saining Xie. Convnext v2: Co-designing and scaling convnets with masked autoencoders. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 16133–16142, 2023.

[135] Hao Wu, Patrick Judd, Xiaojie Zhang, Mikhail Isaev, and Paulius Micikevicius. Integer quantization for deep learning inference: Principles and empirical evaluation. *arXiv preprint arXiv:2004.09602*, 2020.

[136] Panruo Wu, Qiang Guan, Nathan DeBardeleben, Sean Blanchard, Dingwen Tao, Xin Liang, Jieyang Chen, and Zizhong Chen. Towards practical algorithm based fault tolerance in dense linear algebra. In *Proc. of the ACM Intern. Symp.on High-Performance Parallel and Distributed Computing*, page 31–42, 2016.

[137] Xilinx. Vitis HLS Hardware Design Methodology - Arbitrary Precision Datatypes - Floats and Doubles, 2022.

[138] Chen Yang, Yizhou Wang, Xiaoli Wang, and Li Geng. A Stride-Based Convolution Decomposition Method to Stretch CNN Acceleration Algorithms for Efficient and Flexible Hardware Implementation. *IEEE Trans. on Circuits and Systems I*, 67(9):3007–3020, 2020.

[139] Rongtian Ye, Fangyu Liu, and Liqiang Zhang. 3D Depthwise Convolution: Reducing Model Parameters in 3D Vision Tasks. In *Canadian AI 2019: Advances in Artificial Intelligence*, 2019.

[140] Juan Yepez and Seok-Bum Ko. Stride 2 1-D, 2-D, and 3-D Winograd for Convolutional Neural Networks. *IEEE Trans. on VLSI Systems*, 28(4):853–863, 2020.

[141] Tom Young, Devamanyu Hazarika, Soujanya Poria, and Erik Cambria. Recent Trends in Deep Learning Based Natural Language Processing. *IEEE Computational Intelligence Magazine*, 13(3):55 – 75, 2018.

[142] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. Optimizing fpga-based accelerator design for deep convolutional neural networks. In *Proceedings of the 2015 ACM/SIGDA international symposium on field-programmable gate arrays*, pages 161–170, 2015.

[143] Chen Zhang, Guangyu Sun, Zhenman Fang, Peipei Zhou, and Jason Cong. Caffeine: Towards uniformed representation and acceleration for deep convolutional neural networks. In *Proceedings of the ACM Turing Award Celebration Conference-China 2023*, pages 47–48, 2023.

[144] Hao Zhang, Dongdong Chen, and Seok-Bum Ko. Efficient multiple-precision floating-point fused multiply-add with mixed-precision support. *IEEE Transactions on Computers*, 68(7):1035–1048, 2019.

[145] Kai Zhao, Sheng Di, Sihuan Li, Xin Liang, Yujia Zhai, Jieyang Chen, Kaiming Ouyang, Franck Cappello, and Zizhong Chen. FT-CNN: Algorithm-based fault tolerance for convolutional neural networks. *IEEE Trans. on Parallel and Distributed Systems*, 32(7):1677–1689, 2021.

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