IMPERIAL

Efficient AI Model Acceleration through Tensor Decomposition and Dataflow Architecture Design

Zhewen Yu

A Thesis submitted in part fulfillment of requirements for the degree of
Doctor of Philosophy in Electrical and Electronic Engineering
of Imperial College London and the Diploma of Imperial College London

October 2024

Abstract

The exponential growth of AI model size presents significant challenges in terms of computational resources and memory footprints. To address these challenges, both software and hardware engineers are exploring optimization strategies. On the software side, model compression techniques such as pruning and quantization aim to reduce the number of parameters, without significantly sacrificing accuracy. On the hardware side, specialized processors such as AI accelerators or Neural Processing Units (NPUs) are being developed, especially to optimize the execution of matrix multiplication, a key operation in AI workloads.

However, AI models are also becoming increasingly diverse, leading to varied computational and memory requirements. As such, uniform optimization strategies are no longer sufficient to handle this diversity effectively. To address this, it is crucial to expand the design space, allowing design choices to be customized based on the specific characteristics of each AI model.

This thesis proposes an expanded design space approach. On the software algorithm side, advanced tensor decomposition algorithms are introduced, allowing layerwise customization of decomposition schemes for a given AI model. This fine-grained approach significantly reduces accuracy degradation by 7 to 20 percentage points compared to prior methods. On the hardware architecture side, the thesis enhances novel dataflow designs using Field Programmable Gate Arrays (FPGAs). A novel memory management methodology intelligently allocates both on-chip and off-chip memory resources, achieving a latency reduction of 25 to 48%. Additionally, the thesis bridges the gap between software and hardware through fast and efficient design automation techniques, cutting the search time by at least 10×. This inte-

gration demonstrates significant potential for the efficient acceleration of AI models and reduces repetitive engineering efforts in the rapidly evolving AI landscape.

Declaration of Originality

I hereby declare that the work presented in this thesis is my own unless otherwise stated. To the best of my knowledge the work is original and ideas developed in collaboration with others have been appropriately referenced.

Copyright Declaration

Acknowledgments

I would like to express my gratitude to my supervisor, Prof. Christos-Savvas Bouganis, for his invaluable guidance, insight, and support throughout my PhD. His expertise has shaped my research, and his encouragement has been a constant motivation. Even before my PhD, he guided me through my Master's project and encouraged me to pursue a PhD. I am truly thankful for his continuous support, which has been vital to the completion of my PhD and this thesis.

I want to thank my examiners, Prof. Wayne Luk and Dr. Jose Cano Reyes, for their insightful feedback during my viva and their valuable suggestions for improving my thesis. I also wish to thank Prof. George Constantinides and Dr. Yiren Zhao, who provided early advice during the first and second years of my PhD, which was crucial in shaping the direction of my research.

I also wish to extend my thanks to all my friends and colleagues in the CAS research group. There are a lot of stimulating discussions that have greatly contributed to this work. In particular, I want to thank my collaborators Alex, Jianyi, Petros and Cheng for their assistance, and collaboration.

Finally, I would like to express my heartfelt thanks to my family, especially my parents, for their unwavering support, patience, and understanding throughout this journey.

Contents

List of Figures

List of Tables

1 Introduction

Contents

1.1 Motivation

The exploitation of bio-inspired Artificial Intelligence (AI), particularly neural networks, is not a new concept; its history can be traced back to the 1940s. However, the past 15 years have witnessed a remarkable resurgence in interest and significant advances in this technology, transforming neural networks into a cornerstone of modern Deep Learning (DL) methods.

Deep Neural Networks (DNNs) have pushed the state-of-the-art in numerous tasks, such as face recognition, image classification, machine translation, and sentiment analysis, beyond what was previously thought achievable through classical

methods. For example, in image classification, DNNs have drastically reduced error rates in the ImageNet competition [1], showcasing their ability to learn from vast amounts of visual data. In machine translation, DNNs have revolutionized the field by enabling real-time, high-quality translations across multiple languages, as seen in tools like Google Translate. More recently, the great success of GPT models [2] has further showcased the versatility and power of DNNs.

The impressive success of DNNs over the past 15 years is driven by key developments in both software and hardware domains.

- The availability of large datasets like ImageNet [1] provides millions of labeled samples to train complex and accurate models. These extensive datasets offer diverse and comprehensive examples, enabling models to learn robust representations and generalize better across various tasks and domains.

- Advancements in training techniques such as Stochastic Gradient Descent (SGD) [3], dropout regularization [4], batch normalization [5], data augmentation [6] and so on, significantly boosted the training efficiency and generalization capabilities for DNNs.

- The evolution of model architectures, from the introduction of residual connections [7] that mitigate the vanishing gradient problem and enable the training of very deep networks, to more recent transformer styles [8] that excel in capturing long-range dependencies and parallelizing training, enhances both the scalability and efficiency of neural networks.

- Developments in hardware technologies, have dramatically accelerated the training and deployment of AI models. With their massively parallel processing capabilities, GPUs and TPUs offer optimized performance and efficiency specifically for matrix multiplications which are the main AI workloads. The

hardware advancements have enabled more complex models and faster experimentation cycles, driving rapid progress in the field.

Many existing software [9]–[11] and hardware [12], [13] technology stacks for DL are adopting a "one-size-fits-all" approach, which refers to using a general-purpose solution designed to work across a wide range of models and applications without customization. The advantage of this universal approach lies in its simplicity and broad applicability. However, such a universal approach can lead to sub-optimal performance especially when deploying diverse AI models in edge applications that involve resource-constrained devices and performance-sensitive scenarios. Therefore, model-specific customization is desirable for accelerating AI workloads with enhanced efficiency.

To address the significant engineering efforts associated with customization, design automation techniques become essential. These techniques can streamline the customization process, making it more feasible to optimize software and hardware configurations for specific models, and deliver end-to-end solutions. Examples of such techniques include Intel OpenVINO [14], AMD Vitis AI [13], and Nvidia TensorRT [15], though they currently focus more on software and compiler optimizations to enhance performance, leaving hardware-specific customization as a challenging frontier.

1.2 Overview

The research contributions of this thesis can be summarized as follows:

- From a software algorithm perspective, this thesis introduces the first fine-grained tensor decomposition algorithms for compressing model parameters (weights), allowing layer-wise customization of decomposition schemes for a

given AI model. Compared to state-of-the-art solutions, our fine-grained approach significantly reduces the accuracy degradation by 7 to 20 percentage points. This marks the first time that tensor decomposition achieves results comparable to other orthogonal model compression approaches, such as quantization and pruning.

- From a hardware architecture perspective, the research focuses on enhancing dataflow AI accelerators targeting Field Programmable Gate Arrays (FPGAs). These accelerators are designed to customize data movement and computation for deep learning tasks, leveraging the reconfigurable nature of FPGAs to achieve superior performance. We propose a methodology that intelligently exploits both on-chip and off-chip memory for these accelerators, resulting in a latency reduction of 25 to 48%.

- From a co-design perspective, the thesis addresses the integration of software and hardware through fast and efficient design automation, achieving at least a 10× reduction in search time. Such a co-design process ensures that software algorithms and hardware implementations work seamlessly together, and moreover, it enables system-level co-optimization by allowing the hardware and software to be jointly designed and fine-tuned.

As Figure 1.1 shows, in terms of the structure of this thesis, it is organized into the following chapters:

Chapter 2 begins with an overview of the background and literature on optimization techniques for the AI model inference stage. It covers both software algorithm and hardware architecture designs, establishing the state-of-the-art, particularly in model compression and accelerator architecture designs. The chapter also discusses the challenges encountered in these areas, providing the motivation for the innovations introduced in the following chapters.

Tensor decomposition, particularly Singular Value Decomposition (SVD), is a widely used technique for compressing AI model parameters. However, traditional SVD algorithms [16]–[18] typically apply uniform decomposition across all layers in a network, which can result in significant accuracy degradation (ranging from 20 to 70 percentage points). This degradation often requires fine-tuning to recover the accuracy, which is not always practical.

To overcome this issue, **Chapter 3** introduces a novel SVD algorithm that enables layer-specific mixing of decomposition schemes, tailored to the needs of each layer. Additionally, this chapter proposes a streaming FPGA accelerator architecture designed to efficiently support this new method, improving both model performance and hardware efficiency.

Building on this, **Chapter 4** extends the discussion by addressing the limitations of prior research, which only considered extreme cases in terms of decomposition schemes and left much of the design space underexplored. We propose reimagining the SVD process as a layer-wise substitution of the original pre-trained network with a low-rank architecture design space, composed of parameterizable building blocks. We demonstrate how thoroughly exploring this design space using Neural Architecture Search (NAS) can significantly improve the model's accuracy while maintaining compression benefits.

While SVD is the most common tensor decomposition method, others, such as Canonical Polyadic Decomposition (CPD), are also available. However, existing research does not combine these techniques within the same AI model. To address this, **Chapter 5** explores the integration of SVD and CPD, showing that mixing them can enhance both model accuracy and performance. Alongside algorithmic improvements, this chapter introduces a co-design automation flow that optimizes resource-constrained accelerator design and incorporates hardware-awareness into the compression algorithm using throughput proxies, optimizing performance at

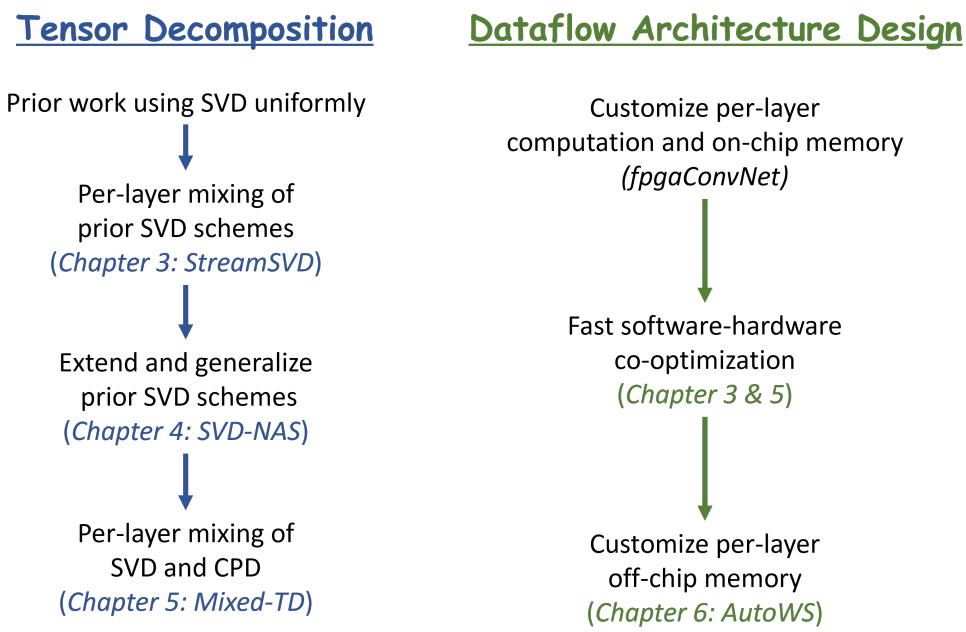


Figure 1.1: The research contributions of this thesis include: (1) advanced, fine-grained tensor decomposition algorithms that can significantly enhance the accuracy-performance trade-off on both many different devices such as FPGAs and mobile CPUs; (2) fast design automation techniques, specifically for model-specific computation and memory customization in dataflow architecture designs.

both the algorithmic and hardware levels.

The streaming, layer-pipelined FPGA accelerator is highly customizable, for example, it can efficiently accelerate decomposed AI models. However, implementing such a streaming architecture on resource-constrained devices remains challenging. Prior approaches [19]–[21] store all model weights on-chip, creating a bottleneck. This is especially problematic as modern CNNs can have hundreds of millions of model parameters [7], while most FPGA devices only offer tens of megabytes of on-chip memory [22]. This mismatch between model size and available on-chip memory limits the deployment of layer-pipelined FPGA accelerators to only small models.

Meanwhile, we found in all existing layer-pipelined FPGA accelerators, off-chip memory bandwidth is left underutilized. As such, in **Chapter 6**, we introduce a novel memory management methodology for layer-pipelined accelerators that leverages both on-chip and off-chip memory for model weights storage. This enables

the deployment of large-scale AI models on resource-constrained FPGA devices in a streaming manner, leading to high throughput and low latency.

1.3 Publications

The following is a list of publications that have resulted from the research conducted in this thesis:

1. **Zhewen Yu** and Christos-Savvas Bouganis, "StreamSVD: Low-rank Approximation and Streaming Accelerator Co-design", International Conference on Field-Programmable Technology (ICFPT), 2021

2. Alexander Montgomerie-Corcoran*, **Zhewen Yu*** and Christos-Savvas Bouganis, "SAMO: Optimised Mapping of Convolutional Neural Networks to Streaming Architectures", International Conference on Field Programmable Logic and Applications (FPL), 2022

3. **Zhewen Yu** and Christos-Savvas Bouganis, "SVD-NAS: Coupling Low-Rank Approximation and Neural Architecture Search", IEEE/CVF Winter Conference on Applications of Computer Vision (WACV), 2023

4. **Zhewen Yu** and Christos-Savvas Bouganis, "Mixed-TD: Efficient Neural Network Accelerator with Layer-Specific Tensor Decomposition", International Conference on Field-Programmable Logic and Applications (FPL), 2023

5. Alexander Montgomerie-Corcoran*, **Zhewen Yu***, Jianyi Cheng and Christos-Savvas Bouganis, "PASS: Exploiting Post-Activation Sparsity in Streaming Architectures for CNN Acceleration", International Conference on Field-Programmable Logic and Applications (FPL), 2023

6. Alexander Montgomerie-Corcoran, Petros Toupas, **Zhewen Yu**, Christos-Savvas Bouganis, "SATAY: A Streaming Architecture Toolflow for Accelerating YOLO Models on FPGA Devices", International Conference on Field-Programmable Technology (ICFPT), 2023

7. **Zhewen Yu** and Christos-Savvas Bouganis, "AutoWS: Automate Weights Streaming in Layer-wise Pipelined DNN Accelerators", Design, Automation and Test in Europe Conference (DATE), 2024

8. Petros Toupas*, **Zhewen Yu***, Christos-Savvas Bouganis and Dimitrios Tzovaras, "SMOF: Streaming Modern CNNs on FPGAs with Smart Off-Chip Eviction", IEEE International Symposium On Field-Programmable Custom Computing Machines (FCCM), 2024

9. **Zhewen Yu**, Sudarshan Sreeram, Krish Agrawal, Junyi Wu, Alexander Montgomerie-Corcoran, Cheng Zhang, Jianyi Cheng, Christos-Savvas Bouganis, Yiren Zhao, "HASS: Hardware-Aware Sparsity Search for Dataflow DNN Accelerator", International Conference on Field-Programmable Logic and Applications (FPL), 2024

* *indicates equal contribution*

1.4 Codebase Availability

To promote reproducibility and transparency, all the code used in this thesis is made available as open source. The following repositories contain the implementation and related scripts:

- Code for Chapter 3, available at `https://github.com/Yu-Zhewen/StreamSVD`

- Code for Chapter 4, available at `https://github.com/Yu-Zhewen/SVD-NAS`
- Code for Chapter 5, available at `https://github.com/Yu-Zhewen/Mixed-TD`
- Code for Chapter 6, available at `https://github.com/Yu-Zhewen/AutoWS`

2

Background

Contents

This chapter first reviews the trends in AI model design, followed by an introduction to common model compression techniques such as pruning, quantization, and decomposition. It then summarizes FPGA hardware accelerators, highlighting

their architecture. Finally, it discusses the design space exploration for optimizing these accelerators.

2.1 Model Design and Compression

2.1.1 Common Operations in AI Models

A typical AI model involves the forward pass, which propagates the input data through the network to generate predictions, and the backward pass, which calculates the gradients of the loss function to update the parameters (weights and biases) of the network. One of the simplest yet most fundamental AI Models is the Multi-Layer Perceptron (MLP). This model consists of multiple sequentially cascaded fully connected layers, where inside each layer, the multiplication operation occurs between a trainable weight matrix and the data vector. An optional bias vector may also be added to produce the results fed into the next layer. Between every two fully connected layers, non-linear activation functions are used to enhance the model's ability to capture complex relationships between inputs and output.

The fully connected layer is also known as the dense layer in some literature, as each element (neuron) of the output vector is influenced by the entire input data vector. This results in high computation and memory complexity, which grows quadratically with the increase in the size of the data vector. To reduce such complexity, the convolutional layer is widely used in recent AI models, especially when targeting computer vision tasks like image classification [23], object detection [24] and semantic segmentation [25]. Convolutional layers utilize weight-sharing across 2-D spatial dimensions and employ a sliding window technique. This allows them to efficiently capture local spatial features. Additionally, convolutional layers introduce spatial invariance, enabling the model to recognize objects regardless of their posi-

tion within the input. To further downsample the data size as the model deepens, pooling layers are often employed alongside convolutional layers.

Note that when evaluating an AI model, from the software perspective, metrics that are commonly used include prediction accuracy, which assesses the proportion of correct predictions made by the model. In addition, the number of operations and parameters involved are critical, as they represent the computational complexity and memory footprints respectively. Figure 2.1 demonstrates the common design process for AI models, highlighting the key stages such as model definition and model compression. Next, we will summarize previous literature related to these two aspects.

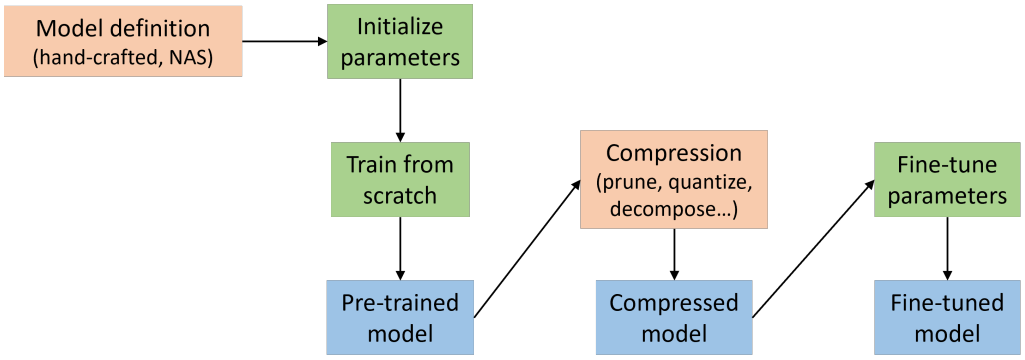


Figure 2.1: To design an efficient model with competitive accuracy, the process typically involves hand-crafting the model architecture or utilizing Neural Architecture Search (NAS), training the model from scratch, compressing it to enhance efficiency, and fine-tuning it to restore or improve accuracy.

2.1.2 Evolution of AI Models for Vision Tasks

AlexNet [26] and VGG [27] were among the early pioneers in deep convolutional neural networks for image classification. AlexNet, introduced in 2012, demonstrated the power of deep learning by winning the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) [1]. This architecture of AlexNet, combining convolution, pooling, and ReLU activation for feature learning, followed by fully-connected layers for classification, has since become a standard and influential structure in the design

of CNNs. Later in 2014, VGG contributed to the advancement of accuracy in image classification by adopting a strategy of increased depth. With 16 to 19 convolutional and fully-connected layers, VGG demonstrated that deeper networks could yield higher precision. Notably, VGG also employed small convolutional kernel sizes (3×3).

Instead of having a sequential network architecture, GoogleNet [28] and ResNet [7] introduce branch connections to the CNN design to make the networks even deeper. GoogleNet, in particular, introduced the Inception module, by leveraging multiple kernel sizes on the same input data simultaneously. The outputs of these diverse kernels are then concatenated, allowing the network to capture features at different scales effectively. ResNet, in 2015, introduced the Residual block to address the vanishing gradient problem when training extremely deep networks. The Residual block contains a skip connection allowing for elementwise addition between the block's input and output. This design innovation enables ResNet to scale up to 152 convolutional and fully-connected layers. Notably, ResNet achieved state-of-the-art accuracy at that time, reaching an impressive 80.62% on ImageNet. Furthermore, DenseNet [29] was introduced in 2017, with extra connections from one layer to all subsequent layers so that their feature maps are concatenated.

Beyond accuracy, the required computation effort and memory storage are also crucial considerations in CNN design. As such, the authors of SqueezeNet [30] suggest replacing 3×3 convolution filters with 1×1 filters, and reducing the number of channels fed into the remaining 3×3 filters. As a result, SqueezeNet achieves similar accuracy as AlexNet but reduces the number of parameters by $50\times$. Furthermore, ResNeXt[31] utilized grouped convolution which splits the input channels of each layer into multiple groups and each filter only needs to process the inputs from one of the groups. This approach was also enhanced in ShuffleNet [32] where the channels are occasionally shuffled between groups for improved accuracy.

MobileNet [33], introduced in 2017, targets mobile and edge devices with limited resources. Its design revolves around depthwise separable convolutions, splitting the standard convolution into depthwise and pointwise (1×1) convolutions. The depthwise convolution, as an extreme case of grouped convolution where each group only contains one channel, significantly reduces parameters and operations. More recently, EfficientNet [34], proposed by Tan *et al.* in 2019, takes a different approach by optimizing both depth (the number of layers), width (the number of channels), and resolution (spatial size) simultaneously to achieve better efficiency. It introduces a compound scaling method that uniformly scales these dimensions.

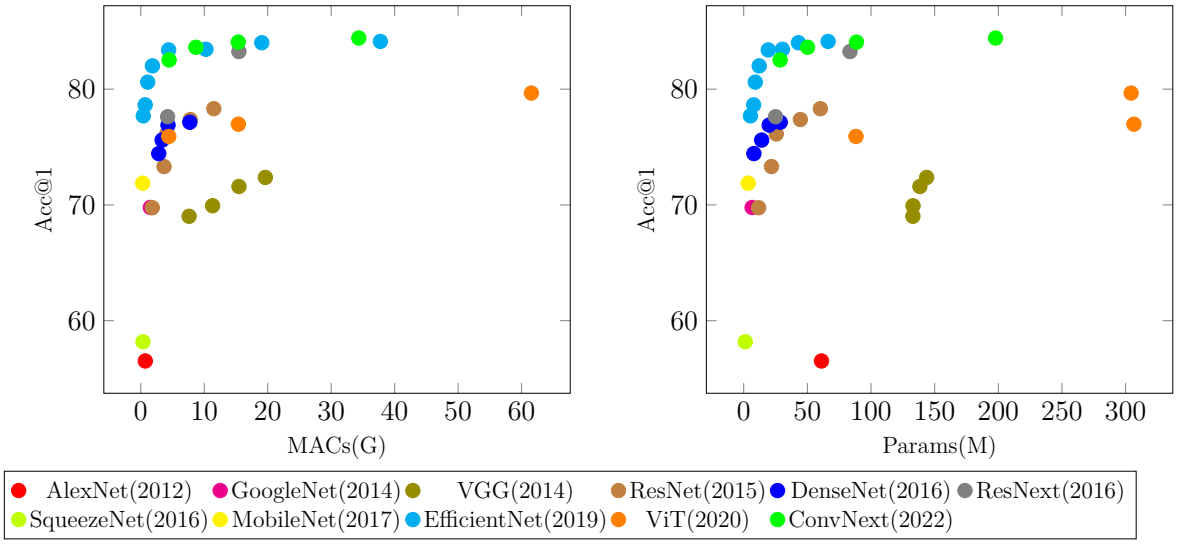


Figure 2.2: Benchmark AI models on ImageNet dataset [1]. Top-1 Accuracy, number of Multiply–accumulate operations, and number of parameters are reported.

Since 2020, the introduction of Vision Transformer [35] has attracted a lot of attention from the research community. However, as shown in two recent studies, RepVGG [36] and ConvNeXt [37], CNNs remain competitive in the vision domain. RepVGG introduces an innovative approach by separating the network architectures at the training and the inference time. During training, each 3×3 convolutional layer is accompanied by residual-style skip connections. Subsequently, during inference, these skip connections are fused into the 3×3 convolutional layers using the "Reparameterization" trick [38]. The fused RepVGG models exhibit an impres-

sive 83% improvement in speed compared to ResNet-50. In the case of ConvNeXt [37], it exploits techniques traditionally confined to language models, such as larger kernel size, GeLU activation [39] and layer normalization [40]. These techniques are strategically adapted to the vision domain, resulting in ConvNeXt achieving an outstanding 85.5% accuracy on the ImageNet dataset. This surpasses ViT's 79.7% accuracy while using fewer parameters and operations, as shown in Figure 2.2.

2.1.3 Model Pruning

From this section, we summarize the existing literature on model compression, focusing primarily on pruning, quantization, and tensor decomposition approaches. A high-level visual comparison of these techniques is provided in Figure 2.3.

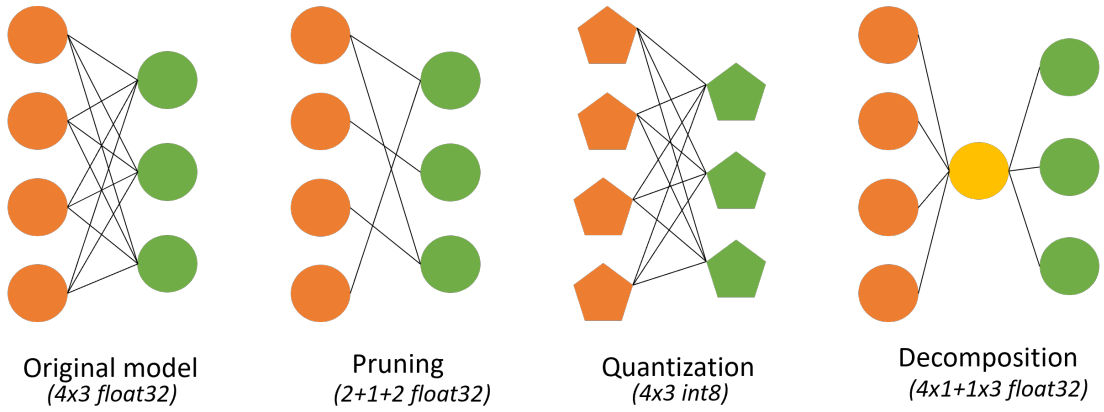


Figure 2.3: Visualization of model compression methods, including pruning, quantization, and decomposition, with the number of operations annotated inside the brackets.

Pruning is a technique to reduce the size of a neural network by eliminating certain connections or neurons. To minimize the impact of pruning on the network's accuracy, a typical approach is to sort the importance of connections by the L1 norm or L2 norm of the weights and activations, and then remove those unimportant connections in priority [41]–[44]. This pruning approach is also referred to as magnitude-based pruning in previous literature. However, magnitude-based pruning

may struggle when the data ranges of weights and activations vary a lot between layers, making the importance comparison across layers challenging and leading to sub-optimal pruning decisions. As an improvement, Molchanov *et al.* chose to normalize the "magnitude" metric locally within each layer before being sorted globally in the whole network [45].

Alternatively, Yu *et al.* introduced the "importance score" metric, which takes the output of the second-to-last layer before classification, and back-propagates it to previous layers by recursively multiplying the transpose of weight matrices [46]. Molchanov *et al.* proposed the "Taylor-expansion" metric that considers the change of the network's loss with respect to the values of weights [47]. The authors reported 0.19pp accuracy loss on ImageNet after pruning 50% neurons from VGG11-BN and 0.48pp accuracy loss after pruning 82% neurons from ResNet-34, claiming their approach is superior to the conventional magnitude-based pruning.

Although pruning individual connections between neurons brings a high sparsity ratio, it leaves the pruned neural network unstructured, making it difficult to be efficiently exploited on hardware for parallel computation. Instead, channel pruning, which removes an entire channel each time, is a structured and coarse-grained pruning method [42], [48], [49]. When combined with the various importance metrics discussed earlier, channel pruning has been widely adopted due to its structured nature, which makes it more efficient for parallel computation. The achievable compression ratios and their corresponding impact on accuracy are demonstrated in Table 2.1.

Channel pruning simplifies the deployment of the pruned model on various hardware accelerators, at the cost of not fully utilizing the sparsity [51]. As a result, channel pruning leads to more evident accuracy degradation than the unstructured approaches, when targeting the same compression ratio [52]. There have been also other researchers pruning along memory banks [53], blocks [54] or filter shapes [42]

Table 2.1: Results of various structured pruning techniques evaluated on ImageNet [1]. Accuracy results are reported after fine-tuning.

Method	Year	Model	Top-1 Acc (%)	Parameters (%)	Operations (%)
[45]	2016	VGG16	-2.3	N/A	-62.8
[46]	2018	ResNet-34	-0.9	-43.7	-43.8
		ResNet-50	-0.9	-43.8	-44.0
[50]	2021	ResNet-50	-0.9	N/A	-62.1
		MobileNet-V1	-2.8	N/A	-73.9

to compromise between unstructured pruning and channel pruning.

The above pruning approaches are applied to a pre-trained neural network. After pruning, the weights in the pruned network can be further fine-tuned to recover accuracy. The fine-tuning process can either be single-shot [55], which prunes the network and fine-tunes it once, or be iterative, which interleaves the pruning and the fine-tuning to progressively shrink the network. Instead, some other researchers such as Wen *et al.* [42] integrated the importance metrics into the loss function as the regularization, which directly encourages sparsity during network training.

2.1.4 Model Quantization

Quantization refers to the process of reducing the precision of the weights and activations in the network, since the original single-precision floating-point format (float32) requires significant memory and computational resources. After quantization, the new data format can take the form of integers and fixed-point numbers, such as int8 [56] or binary representations [57]. Alternatively, low-precision floating-point formats, such as bfloat16 [58] and minifloat [59] are also memory-efficient schemes for quantization.

A central challenge in the quantization process is identifying the data range of the original float32 values. A common practice is to utilize a calibration sub-dataset

for this purpose, as suggested by Migacz *et al.* [60]. After range identification, an optional clipping step can be applied to exclude outlier values. As such, previous studies determine the clipping threshold by minimizing the Mean Square Error (MSE) or Kullback-Leibler (KL) Divergence [61] over the whole data distribution. However, privacy and security concerns may impede the availability of the calibration sub-dataset. Therefore, recent studies [62], [63] generate synthetic images by utilizing batch normalization statistics from a pretrained CNN model. The synthetic images are then fed back into the same model to identify the data range in each layer. After the quantization process, if no further fine-tuning is applied to weight parameters, the process is referred to as post-training quantization. Otherwise, it is known as quantization-aware training in previous literature [64].

Another research question in quantization is regarding the granularity of this process. Previous literature [65] has pointed out that a uniform data type across all layers in a network may prove suboptimal, especially when the data range exhibits significant variation across different layers. One solution is adopting block floating point, where the data are divided into blocks. Each number has its own mantissa but the exponent is shared within the block. Lee *et al.* [66] claimed this block approach has no accuracy loss for 8-bit quantization on VGG16. Regarding block size, it can be set to a fixed configuration determined empirically through experimentation[67]. Specifically in CNNs, taking each channel as a block has become a popular choice, as Table 2.2 demonstrates.

Another fine-grained, non-uniform quantization approach is adopting mixed precision For example, Wang *et al.* [70] developed a framework that uses reinforcement learning to automatically chose the per-layer bitwidth. This mixed-precision quantization approach halved the latency of MobileNet-V1 with 0.42pp top-1 accuracy degradation. Furthermore, Banner *et al.* [61] proposed a per-channel bitwidth allocation algorithm by minimizing the overall MSE.

Table 2.2: Results of various post-training quantization techniques evaluated on the ImageNet dataset [1].

Method	Year	Data format	Granularity	Model	Top-1 Acc (%)
[61]	2018	int4	channel	VGG16	-1.1
				ResNet-18	-2.7
				ResNet-50	-2.3
				ResNet-101	-2.3
[68]	2019	int8	channel	ResNet-18	0.0
				MobileNetV1	-0.3
				MobileNetV2	-0.5
[69]	2022	int4	channel	ResNet-18	-0.2
				ResNet-50	-0.4

2.1.5 Tensor Decomposition

Tensor decomposition expresses one tensor as a set of elementary and simpler tensors that act on each other [71], [72]. The decomposition of a tensor $\mathcal{A}$ can be derived through the following optimization problem:

$$\min_{a_1,a_2\ldots a_M} \; r_1, r_2 \ldots r_M \quad s.t. \; ||\mathcal{A} - f(a_1, a_2 \ldots a_M)|| \leq \epsilon, \tag{2.1}$$

where $a_i, i \in [1, M]$ is the set of elementary tensors and f is the approximation function f so that the given M^{th}-order tensor $\mathcal{A}$ is approximated within the desired error boundary ϵ and the ranks of these elementary tensors r_i are also minimised.

Depending on the operations inside the function f, there are different formats of tensor decomposition available, where the most common one is Singular Value Decomposition (SVD). SVD targets the compression of a given 2^{nd}-order tensor, which is $M = 2$ and $\mathcal{A} \in \mathbb{R}^{d_1 \times d_2}$. As such, $\mathcal{A}$ is decomposed into the form of

$$\mathcal{A} \approx U_r \Sigma_r V_r^T, \tag{2.2}$$

where the rows of U_r and the columns of V_r are orthogonal bases, and Σ_r is a

diagonal matrix containing the top-r singular values in descending order. After absorbing the diagonal matrix Σ_r into U_r and V_r, the final format becomes the product of two tensors and the number of parameters remaining is $(d_1 + d_2)r$. r represents the rank of matrices after decomposition, where a smaller rank reduces the memory storage requirement at the cost of the error introduced.

SVD can be used to compress the weights of convolutional and fully-connected layers. Qiu *et al.* [73] utilized SVD to only decompose fully connected layers of VGG16. They reduced the number of parameters from 138.36M to 50.18M with only 0.04 percentage point (pp) top-5 accuracy loss. Zhang *et al.* [18] extended this idea and further applied SVD to convolutional layers of VGG16, which achieves 3.8× and 2.9× speed-up on CPU and GPU with 0.3pp top-1 accuracy loss. In these studies, the rank after decomposition is decided by hand-tuning [74]. However, it becomes impractical since modern CNNs are very deep in terms of the number of layers. As such, Zhang *et al.* [18] proposed an automatic one-shot rank selection method based on eigenvalues. The selection of ranks is formulated as an optimization problem, which maximizes the product-of-sum of eigenvalues from all weight matrices. Similarly, Chen *et al.* [75] selected ranks with the normalized spectral norm, which is the largest singular value of a given matrix. It is also possible to let the network itself learn the ranks during the training phase, by combining a low-rank regularization term into the loss function [76]–[78].

Apart from SVD, other tensor decomposition schemes such as Canonical Polyadic Decomposition (CPD) [79], Tucker [17], Tensor Train (TT) [80], Tensor Ring (TR) [81] are classified as Higher-Order Decomposition (HOD) methods where $M > 2$. Let us take CPD as an example, given the M^{th}-order tensor, $\mathcal{A} \in \mathbb{R}^{d_1 \times d_2 \times \ldots d_M}$, its CPD format can be represented as the sum of the outer product of 1^{st}-order tensors, leaving the remaining number of parameters to be $(d_1 + d_2 + \ldots d_M)r$.

$$\mathcal{A} \approx \sum_{i=1}^{r} a_{1,i} \otimes a_{2,i} \otimes \ldots a_{M,i} \tag{2.3}$$

This set of 1st-order tensors can be computed via the Alternating Least Squares (ALS) method [72], where the goal is to iteratively update one elementary tensor while fixing all others and solving a least squares problem. When comparing SVD and HOD methods for compressing weight matrices in CNNs, there is no conclusive evidence regarding which achieves a better accuracy versus compression ratio trade-off [17]. Therefore, both SVD and HOD methods remain viable options for application in this context. Figure 2.4 illustrates the process of SVD and CPD, where each node corresponds to a tensor.

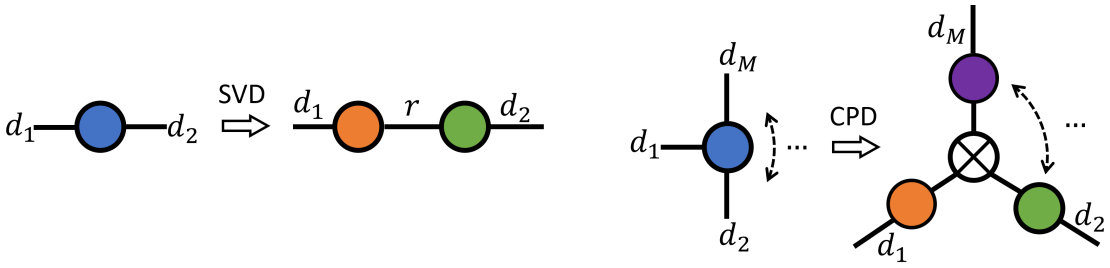


Figure 2.4: Visualize the decomposition process using the Tensor Diagram Notation [82]. Each solid node denotes a M^{th}-order tensor with M edges. Edges connected together represent the tensor contraction operation between two tensors. In CPD, the hollow node with a cross inside represents the sum of the outer product.

When exploiting the above tensor decomposition methods, the values inside each elementary tensor can be either derived through linear algebra approaches such as SVD and ALS, or obtained by training the decomposed layers from scratch [16], [78]. In the latter case, tensor decomposition is used to provide a compact building block structure only, which is not constrained by the linear relationships. For example, the design of MobileNet [33] is inspired by SVD, but non-linear activation functions were inserted between decomposed tensors so that training from scratch is required in this case.

Moreover, training from scratch is also often used for sparse matrix factorization methods, where tensor decomposition is incorporated with pruning together by

sparsifying those decomposed elementary tensors. One such example is butterfly factorization which efficiently represents matrices as products of block diagonal matrices with sparse structures [83], [84]. Wu *et al.* [85] instead explored unstructured sparse matrix factorization and the authors reported 85% parameter count reduction on ResNet-50, with 1pp accuracy loss on ImageNet. Furthermore, in Table 2.3, we present the accuracy degradation and the corresponding compression ratios achieved by prior tensor decomposition techniques.

Table 2.3: Comparison of prior tensor decomposition techniques evaluated on the ImageNet dataset [1]. Accuracy results are reported after fine-tuning, as those decomposition methods generally cause significant accuracy degradation without fine-tuning.

Method	Year	Model	Top-1 Acc (%)	Parameters (%)	Operations (%)
[16]	2018	VGG16	-1.1	N/A	-81.4
		ResNet-34	-1.7	N/A	-80.3
[78]	2020	AlexNet	0.0	N/A	-79.1
[86]	2021	ResNet-18	-0.4	-43.5	-66.7
		MobileNetV2	-1.5	-33.0	-11.0

2.1.6 Knowledge Distillation

Knowledge distillation is a specialized training technique that utilizes a large, well-trained model as the "teacher" to guide the training of a smaller, more compact "student" model. It can also help any existing model compression methods better recover accuracy during the fine-tuning stage [87].

Specifically, Hinton *et al.* [88] incorporated the Kullback-Leibler (KL) divergence into the training loss function to measure the difference in the prediction probability distributions between the "teacher" and the "student" models. This idea was enhanced by Romero [89] and Yim [90], where the L2 norm is further used to calculate differences in the activations of intermediate layers, not just the final output layers.

Moreover, distillation can even be used to transfer the knowledge from the "teacher" to the "student" model without accessing the whole training data [91], [92], a few-sample approach that is particularly valuable when the original data may be sensitive or proprietary, and cannot be easily shared. For instance, Li *et al.* [92] demonstrated that using just 500 images with knowledge distillation can deliver similar fine-tuned accuracy as using 50,000 images.

2.1.7 Neural Architecture Search

Neural Architecture Search (NAS) considers the automatic search process to identify the best network architecture, instead of relying on human intuition and expertise to manually design architecture. The key components associated with NAS are the search space, search algorithm, and performance evaluation.

The search space defines the set of possible neural network architectures that the algorithm can explore, which include both the size (number of layers, neurons, filters, etc.) and connectivity (how layers are connected, skip connections, branching structures, etc.). For example, DARTS [93] searched for the optimal layer connectivity inside the building block and stacked the optimal blocks together to build a deep network. During the design of MobileNetV3 [94], the authors fixed the structure of its basic building block and searched for the number of channels in each layer instead.

In terms of the search algorithm, a brute-force search is usually not realistic due to the large size of the search space. Instead, popular search algorithms include:

- Reinforcement Learning [95]. It treats the architecture search problem as a sequential decision-making process. It contains an agent (controller) to explore the search space and receive rewards based on the performance of sampled architectures.

- Evolutionary [96]. It mimics the process of natural selection by creating a population of architectures, which are recombined and mutated to create the next generation.

- Gradient Descent [97]. It designs a super-net that encompasses a diverse set of architectures within a single model, and uses backpropagation to learn a weighted sampling of these architectures.

When employing the search algorithm, the main bottleneck that prevents the fast iteration of the search is the performance evaluation which usually involves multiple objectives such as accuracy and latency. As such, proxy tasks are often used to provide fast performance estimation and speed up the NAS process. For example, the sampled architecture can be trained on a subset of data with a small number of epochs for fast accuracy estimation [98]. In addition, the latency of an architecture can be estimated using the number of operations involved [99].

2.2 Accelerator Design and Automation

2.2.1 CPU, GPU, TPU, and ASIC Accelerators

The Central Processing Unit (CPU) was originally designed for general-purpose computation, while the Graphics Processing Unit (GPU) was developed for rendering images and real-time graphics. Although neither was initially intended for AI tasks, both have been adapted to meet the growing demands of AI applications.

At the hardware level, AI-specific enhancements have improved the efficiency and performance of AI workloads. For example, NVIDIA's Tensor Cores optimize operations like mixed-precision arithmetic, balancing speed and accuracy for deep

learning tasks [100]. At the instruction level, modern CPUs and GPUs have introduced specialized instruction sets to accelerate critical AI operations, such as matrix multiplications, convolutions, and vector computations. Notable examples include Intel's AVX-512 VNNI and ARM's NEON [101]. Software frameworks also play a key role in optimizing AI workloads. NVIDIA's CUDA [9], Apple's MPS [102], and AMD's ROCm [103] provide specialized kernels tailored to their respective platforms, enabling developers to maximize hardware potential.

Due to the general-purpose nature of CPUs and GPUs, their efficiency can be limited. This is where Application-Specific Integrated Circuits (ASICs) prevail. Google's TPU [12] introduces a specialized Matrix Multiply Unit for high-efficiency AI computations. Cerebras' Wafer-Scale Engine [104] dedicates an entire wafer of silicon to AI tasks, offering unmatched power and bandwidth. Groq's Tensor Streaming Processor [105] is another ASIC optimized for high-speed AI processing.

2.2.2 Building Accelerator on FPGA

Field Programmable Gate Arrays (FPGAs) are hardware reconfigurable devices whose logic functions and interconnections can be reprogrammed to implement various digital circuits and systems. Compared to Application-Specific Integrated Circuits (ASICs), FPGAs have the advantages of lower initial cost and shorter development time. The basic component of FPGAs is known as the Logic Cell (AMD devices) or Logic Element (Altera devices) which consists of LookUp Tables (LUTs) and Flip-Flops (FFs) for implementing logic functions.

Modern FPGAs also integrate dedicated hard blocks to enhance the performance of common functions. For example, AMD 7 Series devices feature DSP blocks that combine an 18-bit by 25-bit multiplier with a 48-bit adder, which are programmable and can operate at a frequency of up to 800 MHz. Moreover, multiple low-precision

operations (such as 4-bit or 8-bit) can be packed into one DSP for an enhanced throughput in AI applications [106], [107]. Additionally, modern FPGAs also integrate on-chip Block RAM (BRAM) with configurable depth and width, offering high efficiency and memory bandwidth.

Instead of focusing on plain FPGAs, there is also a growing trend of building heterogeneous System-on-Chip (SoC) architecture. For instance, AMD Zynq and Intel Agilex devices incorporated ARM processors. Furthermore, the latest Versal ACAP devices from AMD further integrated the array of AI engines. These engines are single-instruction multiple-data (SIMD) and very long instruction word (VLIW) processors, designed especially for AI tasks.

When building AI accelerators on top of FPGAs, the typical design flow often involves crafting the design at the Register Transfer Level (RTL) [108]. Alternatively, High-Level Synthesis (HLS) allows designers to describe the functionality of the accelerator at a higher level of abstraction [21], [109]. Moreover, template-based modular design approaches [110], [111] have gained a lot of attention in FPGA-based AI accelerator development. These approaches leverage pre-designed, reusable modules for common AI tasks, facilitating rapid design iteration and customization. To further automate the design process, compilation toolflows [19], [20], [112] have been explored, aiming to translate models defined in ML frameworks such as PyTorch and Tensorflow into these hardware templates.

2.2.3 Layer-Sequential Architecture

In FPGA-based AI accelerators, the most common approach is layer-sequential architecture, as seen in designs like DnnWeaver [113], Angel-Eye [114], and Snowflake [115]. In these designs, convolutional layers, which typically form the primary computational workload in CNNs, are processed sequentially, with parallelism exploited

only within each layer. Figure 2.5 demonstrates such a design.

Moreover, in layer-sequential architectures, the entire activation feature map needs to be buffered due to its sequential scheduling, thereby putting pressure on memory resources [116]. As such, most layer-sequential implementations chose to fetch weights and activations from off-chip memory before processing, and subsequently write the outputs of the layer back to off-chip memory after computation. As a result, these intensive off-chip memory accesses become a performance bottleneck, because of the limited bandwidth and latency penalty associated [117].

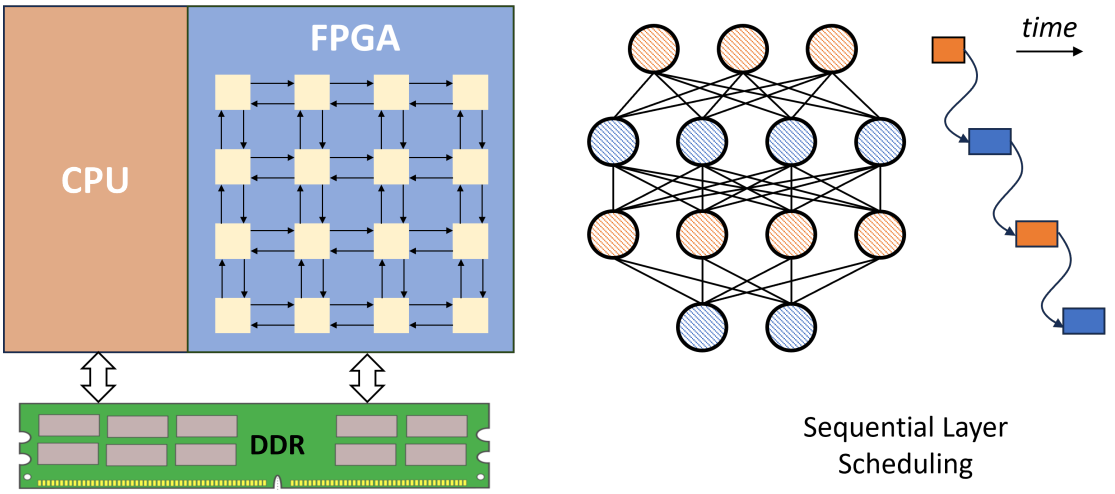


Figure 2.5: In a layer-sequential design, the model inference is scheduled across the CPU and the FPGA with intensive off-chip memory accesses.

Techniques such as double buffering and tiling are commonly employed to mitigate data communication bottlenecks [118]. Double buffering overlaps computation with data movement, which hides the memory access latency. Instead of processing the entire input data or feature map at once, the computation is broken down into smaller tiles to reduce the buffer size required. Moreover, tiling is often combined with a systolic array architecture [119], [120], to encourage data reusing and save bandwidth. In addition, by exploiting the redundancy inside data, encoding techniques can be utilized to further reduce the bandwidth requirement [121], [122].

Apart from memory optimizations, compute optimizations are also critical. Tech-

niques like FFT [123] and Winograd [124], [125] are often used to transform convolution operations into simpler elementwise multiplications, thereby reducing the overall computational load.

Accelerators adopting the layer-sequential architecture are often designed to be general-purpose. For instance, Vitis AI [13] introduced the programmable Deep Learning Processor Unit (DPU) IP which can accelerate a wide range of different CNNs, from ResNet models [7] in image classification to YOLO models [126] in object detection, by leveraging a dedicated instruction set. Specifically, Vitis AI can compile the CNN workload described in TensorFlow [10] or PyTorch [11], and convert these high-level descriptions to the DPU's instructions. Similarly, other works such as [127]–[129] also developed the overlay with layer-sequential architecture to have the flexibility of supporting a wide range of models.

2.2.4 Layer-Pipelined Architecture

An alternative approach is implementing the layer-pipelined architecture on FPGAs, which is also known as streaming or dataflow architecture sometimes. As shown in Figure 2.6, the idea is to build a customized layerwise pipeline for each specific workload. In this architecture, each layer such as convolution, pooling, activation and etc., is mapped to a dedicated computation engine. These engines are connected together with handshake interfaces, executed in a dataflow, pipelined manner. DeepBurning [130], fpgaConvNet [19], FINN [20], hls4ml [131] and HPIPE [132] are well-known examples in this line of research.

Due to the heterogeneity of the computation engines, deploying fine-grained compressed CNNs on layer-pipelined architecture presents unique advantages. FINN [20] focused on the computation of Binarized Neural Networks (BNN) by optimizing BNN-specific operations such as XNOR and Popcount, resulting in ultra-low

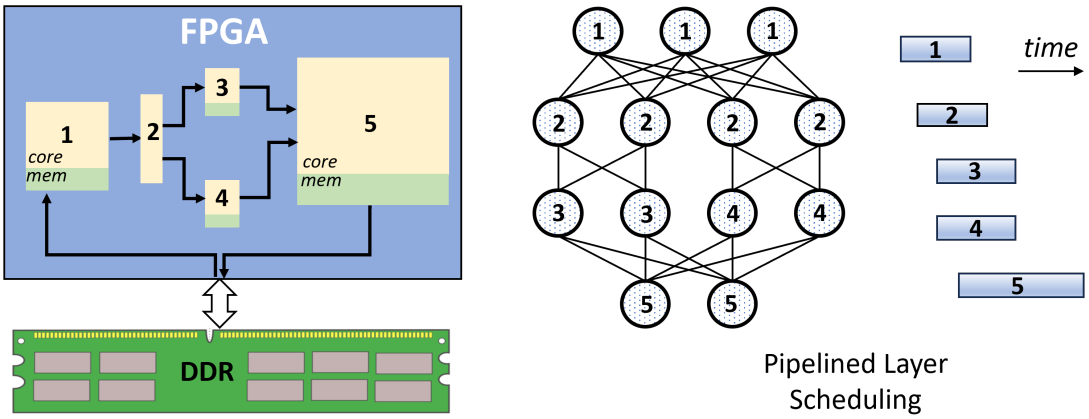


Figure 2.6: A customized layerwise pipeline is built for each specific workload.

resource utilization compared with normal fixed-point operations. DeepBurning [133] investigated DSP packing in order to support mixed-precision quantization, where the heterogeneous computation engines operate on different precisions per-layer. Recent work on HPIPE [132] and fpgaConvNet [134] exploit the data sparsity to skip any zero multiplication for performance gains. The skipping is controlled by either compile-time or run-time scheduling.

In terms of memory storage, most memory footprints are confined on-chip, including both weights and activations storage. The off-chip memory access is only required at the two ends of the pipeline. To efficiently squeeze on-chip memory resources, FINN [22] considered the overclocking of BRAMs, which provides virtual memory ports to mitigate the mismatches between desired and available memory shapes, and eventually improve the utilization efficiency of BRAMs. hls4ml [131], targeting the acceleration of particle physics tasks, considers the choice between LUTRAM and BRAM based on the latency requirement of specific tasks. The choice is made considering that BRAM may introduce longer delays. HPIPE [132] compressed the sparse weight parameters to the run-length encoded format, reducing on-chip storage with a small decoding overhead during computation.

Moreover, to overcome the on-chip memory bottleneck, recent works [135], [136] consider partitioning the computational pipeline across multiple FPGA devices, con-

nected through ethernet protocols in a data centre. However, such a multi-device system is at the expense of overall power consumption. Reconfiguration is another technique that has been exploited to schedule multiple partitions on a single device in a time-multiplexed manner, although this incurs a penalty in system latency.

In Table 2.4, we present the performance and energy efficiency results of various FPGA accelerators. It is important to note that layer-pipelined architectures generally achieve high performance and efficiency. However, they are constrained to deployment on larger FPGA devices due to the bottleneck of on-chip memory.

Table 2.4: Performance results of various FPGA accelerators, highlighting the advantages of the layer-pipelined architectures such as [19], [22], [133].

Method	Architecture	Year	Device	Model	Performance (GOP/s)	Energy Efficiency (GOP/s/W)
[114]	Layer-Sequential	2017	Z7045	VGG16	137	14
[129]	Layer-Sequential	2020	XCK325T	MobileNetV1	148	18
				MobileNetV2	99	12
				DenseNet-161	176	53
				SqueezeNetV1.1	134	16
[137]	Layer-Sequential	2022	Z7020	ResNet-18	101	29
				ResNet-50	109	31
				MobileNetV2	79	23
[19]	Layer-Pipelined	2017	ZC706	VGG16	156	N/A
[22]	Layer-Pipelined	2020	U250	ResNet-50	18300	258
[133]	Layer-Pipelined	2022	ZU3EG	VGG-Tiny	1490	1100

2.2.5 Accelerator Design Space Exploration

Once the FPGA accelerator architecture is decided in a parameterisable form, Design Space Exploration (DSE) can be used to evaluate different design choices and configurations. For example, in a layer-sequential architecture, the DSE process involves the decisions on tiling size, buffer size, systolic array size, and etc. For a layer-pipelined architecture, DSE involves the compute and memory resource allocation for each stage of the pipeline.

When evaluating these design choices, metrics of interest often include throughput [119], [138] and latency [139], [140] of the accelerator. Especially in a layer-pipelined architecture, throughput is determined by the slowest stage of the pipeline, while latency also considers the time taken for filling and emptying the pipeline. In addition, optimizing power [141] or energy consumption [142] can also be the goal of DSE, especially when targeting edge applications.

However, validating each design configuration on board is time-consuming, therefore, a common solution is to build resource and performance models and use predicted results to guide the process of DSE [19]. These models can either be analytical [128], [143], [144], offering insights into theoretical performance boundaries and potential bottlenecks, or regressional [145], providing predictions based on empirical data and statistical analysis.

Apart from optimizing these hardware performance metrics, the accelerator DSE process can be further combined with NAS or model compression, which is known as software-hardware co-design. For example, Abdelfattah *et al.* [146] constructs a joint search space that includes the design choices from both the CNN model and the FPGA accelerator. Their objective was to identify the Pareto front, optimizing the accuracy-latency tradeoff. Li *et al.* [147] formulated the search of the CNN channel number, quantization bitwidth, loop parallelism and tiling into one loss function which is optimized through stochastic gradient descent. Such software-hardware co-design provides an efficient solution for system-level exploration.

2.2.6 fpgaConvNet

The hardware implementation of this thesis builds upon the fpgaConvNet framework [19]. Accordingly, we provide a detailed overview of fpgaConvNet in this section to highlight its relevance to our work.

fpgaConvNet is an end-to-end toolflow capable of automatically generating a customized, layer-pipelined accelerator based on the description of a given CNN model architecture. In addition to inter-layer pipelining, the generated accelerator also explores various degrees of intra-layer parallelism, including *coarse-grained folding* and *fine-grained folding*, which correspond to channel and kernel parallelism inside a convolutional layer, respectively. To achieve these optimizations, fpgaConvNet employs a simulated-annealing-based DSE algorithm, enabling efficient balancing of performance and resource utilization.

Similar to other layer-pipelined architectures, fpgaConvNet generates accelerators that store all the weight parameters of a CNN permanently on-chip. To address resource limitations, fpgaConvNet further supports bitstream reconfiguration by partitioning the CNN into multiple subgraphs. It also enables partial offloading and reloading of weights, but only for the last layer of a network, to optimize resource usage [19].

Additionally, fpgaConvNet performs an almost faithful mapping of the original CNN model, applying only 16-bit fixed-point quantization without introducing further modifications. This allows the generated accelerator to closely match the behavior of the full-precision model.

In this thesis, we enhance fpgaConvNet with several key contributions. First, we extend its capabilities to support the acceleration of decomposed neural networks, enabling more efficient execution of models compressed using advanced tensor decomposition techniques (Chapter 3). Second, we introduce a greedy DSE algorithm to streamline the exploration of design configurations, ensuring optimal performance while reducing search complexity (Chapter 5). Finally, we incorporate the utilization of off-chip memory for weight transfer, addressing the limitations of on-chip memory capacity and enabling the deployment of larger models on resource-constrained FPGA devices (Chapter 6).

2.3 Summary

In this chapter, we have reviewed various techniques for efficiently accelerating AI model inference, covering aspects such as model design, compression techniques, and FPGA accelerator architectures.

- For model compression, fine-grained approaches tailored to specific models or even individual layers can achieve higher compression ratios while preserving model accuracy. However, compared to other techniques like pruning and quantization, tensor decomposition often lacks flexibility, making it less effective and popular. To address this, we are going to extend the design space of tensor decomposition in Chapter 3 and 4.
- The existing design flow typically involves a sequential approach, where software-level optimizations, such as model compression, are applied first, followed by hardware architecture-level optimizations. However, a more integrated, co-optimization approach could yield better overall efficiency, which will be explored in Chapter 5.
- When designing FPGA accelerators for AI models, layer-pipelined architectures offer high performance and energy efficiency. While these architectures are highly customizable, they are often constrained by limited on-chip memory capacity, which can restrict their broader application. As such, in Chapter 6, we are going to propose a novel memory management methodology for layer-pipelined architectures that exploits both on-chip and off-chip memory for weight storage.

3

StreamSVD: Low-rank approximation and streaming accelerator co-design

Contents

This chapter begins by explaining and comparing existing SVD compression algorithms. Afterwards, it innovatively combines these algorithms on a per-layer basis, customizing each layer to the most suitable method. Additionally, it introduces a specialized hardware design that efficiently supports the deployment of decomposed networks using a dataflow streaming architecture, demonstrating its performance advantages. This chapter is based on the conference paper StreamSVD [148] published in FPT 2021.

3.1 Introduction

SVD can be used to compress weight tensors and decompose them into their low-rank versions. However, existing studies [75], [149] develop decomposition algorithms that are driven solely by reducing the number of operations and the number of parameters in the model, without considering the actual throughput speed-up on hardware. As such, their solutions can lead to sub-optimal designs, when metrics of interest in many real applications are the accuracy-throughput trade-off [150]. Furthermore, existing SVD algorithms are coarse-grained, since the compression schemes are uniform across layers, leading to severe accuracy degradation. To address these two issues, we introduce StreamSVD, a framework that considers the co-design between fine-grained, per-layer SVD compression algorithm and the layer-pipelined, streaming architecture accelerator. The specific contributions of StreamSVD are:

- We propose a novel SVD compression algorithm that is fine-grained, per-layer, and also hardware-aware, by taking the acceleration parallelism and latency into account.

- We further demonstrate that SVD and quantization, these two compression approaches, can be efficiently combined through an iterative, training-free weight derivation, reducing the overall compression error introduced.

- We analyze the advantages of using a streaming architecture for mapping the SVD compressed CNN as opposed to other architectures, especially that streaming architecture introduces less overhead due to its pipeline structure and per-layer hardware customization.

- Our framework can automatically apply SVD compression to the CNN models in PyTorch, and generates an efficient layer-pipelined, streaming FPGA accelerator customized for this specific compressed model with the goal of throughput maximization.

3.2 SVD Compression Algorithm

3.2.1 Annotation

Let us consider an N-layer CNN and denote the weights of i^{th} convolutional layer as a 4^{th}-order tensor $\mathcal{W}_i \in \mathbb{R}^{c_{i+1} \times c_i \times k_{a,i} \times k_{b,i}}, i \in [1, N]$, where c_{i+1}, c_i and $k_{a,i} \times k_{b,i}$ are denoting the number of output channels, input channels and the kernel size of the i^{th} convolutional layer respectively. Its input and output are denoted as $\mathcal{X}_i \in \mathbb{R}^{b \times m_i \times n_i \times c_i}$ and $\mathcal{X}_{i+1} \in \mathbb{R}^{b \times m_{i+1} \times n_{i+1} \times c_{i+1}}$, where b denotes the batch size, m_i and n_i denote the spatial size of the feature map.

The original convolution operation can be represented as the tensor contraction along c_i, $k_{a,i}$ and $k_{b,i}$ dimensions,

$$\mathcal{X}_{i+1} = \sum_{c_i, k_{a,i}, k_{b,i}} (\mathcal{W}_i \odot \mathbb{S}_i[\mathcal{X}_i]), \tag{3.1}$$

where the sliding window function $\mathbb{S}_i$ performs the padding and striding, converting $\mathcal{X}_i$ into the shape of $b \times m_{i+1} \times n_{i+1} \times c_i \times k_{a,i} \times k_{b,i}$; and $\odot$ denotes the elementwise product.

By exploiting SVD to compress the convolution weights, Equation (3.1) can be decomposed into the following form, representing two consecutive convolutional operations instead.

$$\mathcal{X}_{i+1} = \sum_{c_{1,i}, k_{a1,i}, k_{b1,i}} (\mathcal{W}_{1,i} \odot \mathbb{S}_{1,i}[\mathcal{X}_{1,i}]), \quad \mathcal{X}_{1,i} = \sum_{c_{0,i}, k_{a0,i}, k_{b0,i}} (\mathcal{W}_{0,i} \odot \mathbb{S}_{0,i}[\mathcal{X}_i]) \tag{3.2}$$

$\mathcal{X}_{1,i}$ represents the intermediate result of these two convolutional operations. It is important to note that there are multiple possible schemes for defining $\mathcal{W}_{0,i}$, $\mathcal{W}_{1,i}$, $\mathbb{S}_{0,i}$ and $\mathbb{S}_{1,i}$. The choice of these schemes can significantly impact both the accuracy and efficiency of the model. In the following section, we will explore various SVD-based decomposition schemes, analyzing their effects on model performance.

3.2.2 Scheme Selection

Since the original weight $\mathcal{W}_i \in \mathbb{R}^{c_{i+1} \times c_i \times k_{a,i} \times k_{b,i}}$ is a 4^{th}-order tensor, SVD (defined in Equation (2.2)) which handles 2^{nd}-order tensors only cannot be directly used. Summarizing previous literature, we found the following decomposition schemes can be used, where we can derive the pair of low-rank tensors $\mathcal{W}_{0,i}$ and $\mathcal{W}_{1,i}$ from the original weight tensor $\mathcal{W}_i$ without retraining from scratch.

- **scheme 0** [151]: the original weight tensor $\mathcal{W}_i$ is completely sliced on its first dimension into c_{i+1} chunks, where each chunk $\mathcal{A}_i$ has the shape of $\mathbb{R}^{1 \times c_i \times k_{a,i} \times k_{b,i}}$. Afterwards, each individual chunk is reshaped into a 2^{nd}-order $\mathbb{R}^{(k_{a,i} k_{b,i}) \times c_i}$

where SVD can be applied to decompose the tensor into two parts, with the shape of $\mathbb{R}^{(k_{a,i}k_{b,i})\times r_i}$ and $\mathbb{R}^{r_i\times c_i}$ respectively, assuming the rank after decomposition is r_i. The decomposed tensors can be reshaped back to 4-D, and all c_{i+1} chunks are concatenated together on the first dimension. Eventually, we obtain the desired decomposed weight tensors $\mathcal{W}_{1,i} \in \mathbb{R}^{c_{i+1}\times(r_i c_{i+1})\times k_{a,i}\times k_{b,i}}$ and $\mathcal{W}_{0,i} \in \mathbb{R}^{(r_i c_{i+1})\times c_i\times 1\times 1}$.

- **scheme 1** [18]: the original 4-D weight tensor $\mathcal{W}_i$ is directly reshaped into the 2$^{\text{nd}}$-order form $\mathcal{A}_i \in \mathbb{R}^{c_{i+1}\times(c_i k_{a,i}k_{b,i})}$ by flattening its last three dimensions. SVD is then applied to this 2$^{\text{nd}}$-order tensor, producing two decomposed parts with the shape of $\mathbb{R}^{c_{i+1}\times r_i}$ and $\mathbb{R}^{r_i\times c_i k_{a,i}k_{b,i}}$. Same as before, r_i denotes the rank after decomposition. After bringing the pair of tensors back to 4-D, the final produced weight tensors are $\mathcal{W}_{1,i} \in \mathbb{R}^{c_{i+1}\times r_i\times 1\times 1}$ and $\mathcal{W}_{0,i} \in \mathbb{R}^{r_i\times c_i\times k_{a,i}\times k_{b,i}}$.

- **scheme 2** [74]: the original 4-D weight tensor is firstly transposed between its second and third dimensions, resulting in the shape of $\mathbb{R}^{c_{i+1}\times k_{a,i}\times c_i\times k_{b,i}}$. The transposed 4-D tensor is then reshaped to 2-D with the shape of $\mathcal{A}_i \in \mathbb{R}^{(c_{i+1}k_{a,i})\times(c_i k_{b,i})}$ so that SVD can be applied. After decomposition, the obtained tensors have the shape of $\mathbb{R}^{(c_{i+1}k_{a,i})\times r_i}$ and $\mathbb{R}^{r_i\times(c_i k_{b,i})}$ respectively. Through further reshaping and transposing, the final decomposed weight tensors are $\mathcal{W}_{1,i} \in \mathbb{R}^{c_{i+1}\times r_i\times k_{a,i}\times 1}$ and $\mathcal{W}_{0,i} \in \mathbb{R}^{r_i\times c_i\times 1\times k_{b,i}}$.

- **scheme 3** [152]: similar to scheme 0 but the original weight tensor $\mathcal{W}_i$ is completely sliced on its second dimension instead, producing c_i chunks, and each chunk $\mathcal{A}_i$ has the shape of $\mathbb{R}^{c_{i+1}\times 1\times k_{a,i}\times k_{b,i}}$. Each chunk is reshaped into $\mathbb{R}^{c_{i+1}\times(k_{a,i}k_{b,i})}$ which is then compressed by SVD, producing two 2-D tensors with the shape of $\mathbb{R}^{c_{i+1}\times r_i}$ and $\mathbb{R}^{r_i\times(k_{a,i}k_{b,i})}$ respectively. Therefore, after final reshaping and concatenating c_i chunks together, the obtained decomposed tensors in their 4-D forms will be $\mathcal{W}_{1,i} \in \mathbb{R}^{c_{i+1}\times(r_i c_i)\times 1\times 1}$ and $\mathcal{W}_{0,i} \in \mathbb{R}^{(r_i c_i)\times c_i\times k_{a,i}\times k_{b,i}}$.

Table 3.1: A summary of SVD schemes considered in StreamSVD. The original convolutional layer has c_i input channels, c_{i+1} filters with $k_{a,i} \times k_{b,i}$ kernel size. The original layer operates on the inputs whose width and height are m_i and n_i, and produces the outputs whose width and height are m_{i+1} and n_{i+1} in a batch size of b. Each original layer is decomposed into two consecutive convolution layers, as Equation (3.2) defined. The numbers for the first decomposed layer are highlighted in blue, while the second decomposed layer is highlighted in green.

	parameters count	operation count
original	$c_{i+1}c_ik_{a,i}k_{b,i}$	$bm_{i+1}n_{i+1}c_{i+1}c_ik_{a,i}k_{b,i}$
scheme 0	$c_{i+1}^2r_ik_{a,i}k_{b,i} + r_ic_{i+1}c_i$	$bm_{i+1}n_{i+1}c_{i+1}r_ik_{a,i}k_{b,i} + bm_in_ir_ic_{i+1}c_i$
scheme 1	$c_{i+1}r_i + r_ic_ik_{a,i}k_{b,i}$	$bm_{i+1}n_{i+1}c_{i+1}r_i + bm_{i+1}n_{i+1}r_ic_ik_{a,i}k_{b,i}$
scheme 2	$c_{i+1}r_ik_{a,i} + r_ic_ik_{b,i}$	$bm_{i+1}n_{i+1}c_{i+1}r_ik_{a,i} + bm_in_{i+1}r_ic_ik_{b,i}$
scheme 3	$c_{i+1}r_ic_i + r_ic_i^2k_{a,i}k_{b,i}$	$bm_{i+1}n_{i+1}c_{i+1}r_ic_i + bm_{i+1}n_{i+1}r_ic_ik_{a,i}k_{b,i}$

The produced low-rank weight tensors $\mathcal{W}_{0,i}$ and $\mathcal{W}_{1,i}$ are then used in Equation (3.2), which defines two consecutive convolution layers. By looking at the shapes of these tensors, we see that the decomposition may result in many pointwise and grouped convolutions. For example, both $\mathcal{W}_{0,i}$ in **scheme 0** and $\mathcal{W}_{1,i}$ in **scheme 1** have a 1×1 kernel size, which is characteristic of a pointwise convolution. In addition, $\mathcal{W}_{1,i}$ in **scheme 0** and $\mathcal{W}_{0,i}$ in **scheme 1**, on grouped channels as sliced chunks, essentially functioning as grouped convolutions. We will further discuss the impact of these decompositions on hardware performance in Section 3.4.1.

To obtain optimal compression results, our algorithm is able to determine the decomposition scheme $s_i \in \{0, 1, 2, 3\}$ and the decomposition rank r_i for each convolutional layer in the given CNN model. These parameters are selected individually for each layer, allowing for tailored optimization across the entire network.

Our algorithm firstly focuses on the layerwise scheme selection, and specifically, StreamSVD decides the most appropriate scheme as the one that introduces the smallest L2 norm error under the same number of operations.

$$\min_{s_i \in \{0,1,2,3\}} ||\mathcal{W}_i - \mathcal{W}_{1,i}\mathcal{W}_{0,i}||_2 \quad s.t.\ OP_{1,i} + OP_{0,i} \leq OP_{avail,i} \tag{3.3}$$

In the above equation, $OP_{1,i}$ and $OP_{0,i}$ denote the total operation count in the pair of generated convolutional layers, which are shown in green and blue in Table 3.1. $OP_{avail,i}$ specifies the operation budget allocated to these two decomposed layers. Since the operation count depends on the decisions of choosing the decomposition scheme s_i and the decomposition rank r_i. In StreamSVD, we take a two-step approach in order to decouple these two decisions.

First, we determine the decomposition scheme by running a series of sweeps where the operation budget $OP_{avail,i}$ is uniformly sampled between 0 and the original operation count. For each sampled budget, we solve Equation (3.3) by evaluating all candidate schemes and computing the maximum achievable rank within the given budget. The scheme that results in the smallest L2 norm error for each run is selected as the winner. Finally, after completing all runs, we apply a majority vote to decide the scheme. These decisions are made individually for each layer, ensuring a fine-grained and layer-specific approach.

3.2.3 Rank Selection

After determining the decomposition scheme per layer, the next step is to choose the appropriate decomposition ranks, where our approach is interpreting the rank selection problem as a special form of channel pruning.

Specifically, we replace each original convolutional layer with a pair of decomposed convolutional layers according to the schemes previously selected, and then initialize the values of ranks so that the operation count is identical as before the decomposition. As a result, a new network architecture is created with the number of layers doubled, but no compression has been achieved so far in terms of the operation count.

On this new network architecture, the number of channels between every pair

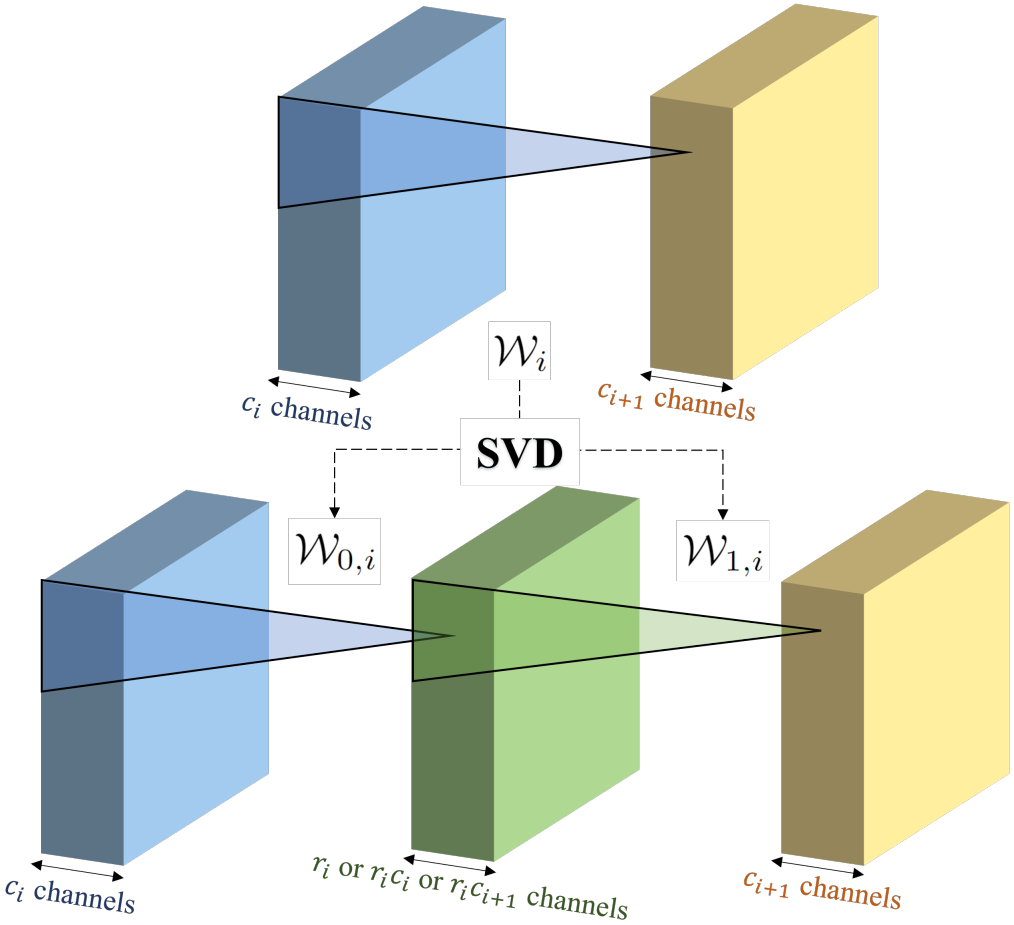


Figure 3.1: The weights decomposition of the original convolutional layer leads to the split to two new convolutional layers instead. Applying channel pruning between these two layers is the same as reducing the decomposition rank of weights.

of decomposed layers is directly related to the decomposition rank, which is shown as the depth of the green block in Figure 3.1. Therefore, reducing the rank of decomposed weight tensors $\mathcal{W}_{0,i}$ and $\mathcal{W}_{1,i}$ is equivalent to pruning channels between the corresponding two decomposed convolutional layers.

As such, on the new network architecture, we run the inference on a calibration set of data to evaluate the importance of these channels to be pruned, based on the Taylor criterion [47] which is denoted as I in Equation (3.4).

$$I(\mathcal{W}_{0,ij}) = \sum_{w \in \mathcal{W}_{0,ij}} (w \frac{\partial L}{\partial w})^2, \quad i \in [1, N], j \in \begin{cases} [1, r_i] & \text{if } s_i \in \{1, 2\} \\ [1, r_i c_i] & \text{if } s_i \text{ is } 3 \\ [1, r_i c_{i+1}] & \text{if } s_0 \text{ is } 0 \end{cases} \tag{3.4}$$

In the above equation, $\mathcal{W}_{0,ij}$ denotes the j^{th} part of weights after completely slicing the decomposed tensor $\mathcal{W}_{0,i}$ on its first dimension. s_i denotes the index of the selected decomposition scheme for the original i^{th} convolutional layer, and L denotes the entropy loss of the network. The measured importance metrics are then sorted globally across all the layers, and those channels with the least importance are pruned first until a target compression ratio regarding the network operation count has been reached. The above pruning process ultimately determines the decomposition rank for each layer.

3.3 Combine SVD and Quantization

Apart from the decomposition of weights, StreamSVD also employs fixed-point quantization to compress the CNN model. This section delves into the effective integration of SVD and quantization, comparing two distinct categories of approaches: sequential and iterative. Within each category, two variants are identified and denoted as “I” and “II”, respectively. As a result, four different methods are compared and analyzed in total, as Figure 3.2 shows.

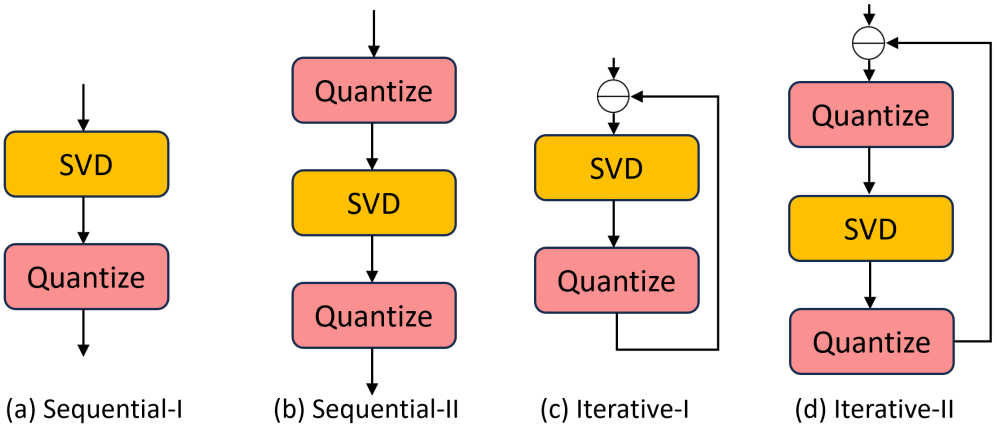


Figure 3.2: Apart from SVD, StreamSVD also employs fixed-point quantization. The fusion of these compression techniques can be applied in either a sequential or iterative manner, involving rank-1 refinements of SVD. The experiment in Section 3.5.2 will show **Iterative-I** offers the best result in terms of accuracy versus compression ratio.

3.3.1 Sequential Combination

Sequential-I: The original weight tensor $\mathcal{W}_i$ is firstly decomposed into a pair of tensors $\mathcal{W}_{0,i}$ and $\mathcal{W}_{1,i}$, both still in floating-point format, but with their ranks constrained to r_i. After this decomposition, the resulting tensors are quantized to a fixed-point format. This entire compression process can be expressed as:

$$\mathcal{W}_i \xrightarrow{\text{SVD}} \mathcal{W}_{0,i}, \mathcal{W}_{1,i} \xrightarrow{\text{Quantize}} (\mathcal{W}_{0,i})_q, (\mathcal{W}_{1,i})_q \tag{3.5}$$

Sequential-II: method alters the sequence of operations by applying quantization before tensor decomposition. Specifically, the original weight tensor $\mathcal{W}_i$ is firstly quantized to $(\mathcal{W}_i)_q$. The quantized tensor is then decomposed via SVD, which produces two floating-point tensors $\mathcal{W'}_{0,i}$ and $\mathcal{W'}_{1,i}$, as the decomposition involves continuous mathematical operations that inherently generate floating-point values. Consequently, these two tensors are re-quantized in the final step:

$$\mathcal{W}_i \xrightarrow{\text{Quantize}} (\mathcal{W}_i)_q \xrightarrow{\text{SVD}} \mathcal{W'}_{0,i}, \mathcal{W'}_{1,i} \xrightarrow{\text{Quantize}} (\mathcal{W'}_{0,i})_q, (\mathcal{W'}_{1,i})_q \tag{3.6}$$

Our experiment in Section 3.5.2 will show that **Sequential-I** introduces less compression error and is therefore the preferred approach.

3.3.2 Iterative Combination

The above two approaches both apply sequential, one-shot compression procedures. In this section, we further investigate the iterative combination of decomposition and quantization procedures.

As Algorithm 1 shows, the derivation of SVD (Equation (2.2)) can be converted to a refinement loop containing r iterations [144], [153]. Each iteration aims to approximate the target matrix using rank-1 approximations. The decomposition error which is the difference between the target matrix and its rank-1 product becomes the new approximation target in the next iteration. All produced rank-1 matrices are eventually concatenated together to constitute the desired rank-r approximations.

Algorithm 1 Iterative SVD

Require: $\mathcal{W}_i$ ▷ 4-D weights of i^{th} conv layer
$\mathcal{W}_i \Rightarrow \mathcal{A}_i$ ▷ transform to 2-D based on selected scheme
for $j \in 1$ to r **do** ▷ iterative r times, r is the selected rank
    $\mathcal{A}_i \Rightarrow U_1 \Sigma_1 V_1^T$ ▷ use SVD to obtain rank-1 approximations only
    $\mathcal{A}_i = \mathcal{A}_i - U_1 \Sigma_1 V_1^T$ ▷ take approximation error as target for next iteration
    $U_i(j) = U_1 \times sqrt(\Sigma_1)$, $V_i(j) = sqrt(\Sigma_1) \times V_1$
$U_i, V_i \Leftarrow U_i(j), V_i(j) \forall j \in 1tor$ ▷ concat all rank-1 tensors
return U_i, V_i

The quantization can be embedded into this iterative algorithm as well by applying quantization to the rank-1 tensors produced during each iteration. Furthermore, akin to the sequential combination, the order of applying quantization and SVD in the iterative process introduces two distinct variants: **Iterative-I** and **Iterative-II**,

where the differences are highlighted in Figure 3.2. We will compare the accuracy impacts of all these variants in Section 3.5.2.

3.4 Implementation Details

3.4.1 Accelerator Architecture

StreamSVD extends fpgaConvNet [19] as the backend tool to deploy the previously compressed CNN onto the FPGA. In existing fpgaConvNet, all the convolutional layers are pipelined, while the intra-layer parallelism is explored on the channel dimension and the kernel dimension, whose level of parallelism is referred to as *coarse-grained* and *fine-grained factors* respectively. The *coarse-grained factor* also decides the number of parallel data streams between layers, as activation data flow through these streams in the layout that channels are the last dimension.

Comparing the original CNN and the decomposed CNN, the main difference in workload is that the number of convolutional layers doubled but the total operations to compute reduced. In addition, considering benchmarks such as VGG16 [27] and ResNet-18 [7], more than 90% operations are in standard 3×3 convolutional layers. However, after decomposition, these layers can be replaced with pointwise (1×1) or grouped convolutions, depending on the specific decomposition scheme chosen. As a generalized form of depthwise convolution (Section 2.1.2), the grouped convolution splits input channels into multiple groups and each convolution filter is only responsible for processing one group [26], [31].

To adapt these changes in workload, StreamSVD extends the idea of *coarse-grained folding* to support the efficient parallelism inside the grouped convolution, which is required by **scheme 0** and **scheme 3** in Table 3.1. Let's take the implementation of **scheme 3** as an example, where the i^{th} original convolutional layer

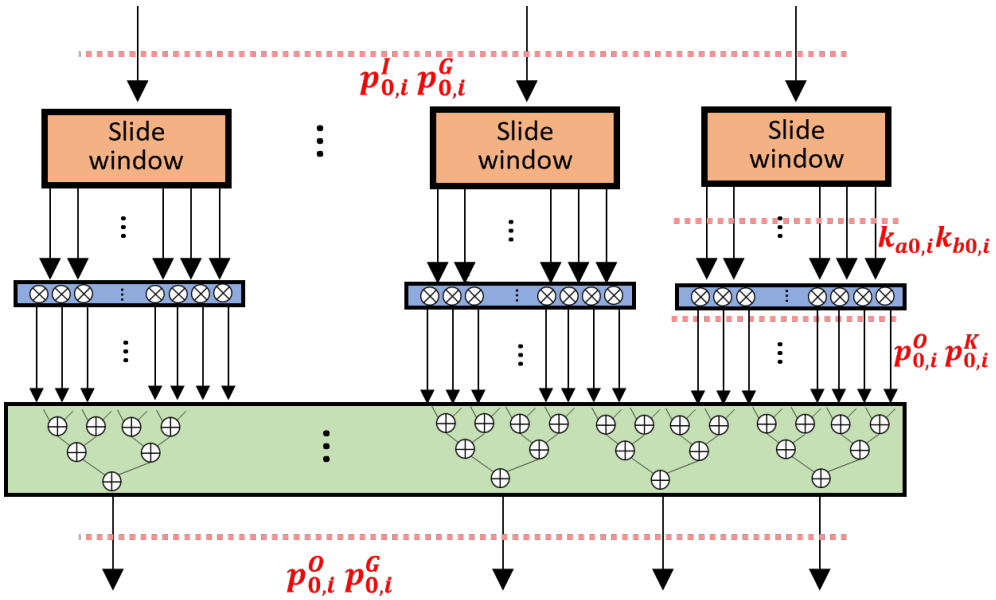


Figure 3.3: Hardware implementation example for the grouped convolution created by decomposition **scheme 3**, including sliding windows, multiplication arrays and adder trees. The parallelism on the group, output channel and kernel dimension are denoted as $p^G_{0,i}$, $p^O_{0,i}$ and $p^K_{0,i}$ respectively. Note that, the actual hardware has data interleaving between multiplication arrays and adder trees to add data belonging to different input channels together, which is not shown in this graph for simplicity.

with c_i input channels, c_{i+1} output channels and $k_{a,i} \times k_{b,i}$ kernel size is decomposed into the pair of:

1) a new grouped convolutional layer with c_i groups, with each group having 1 input channel and r_i output channels. Its kernel size is $k_{a,i} \times k_{a,i}$. For its mapping on the hardware, let us denote the parallelism on the group, output channel and kernel dimension as $p^G_{0,i}$, $p^O_{0,i}$ and $p^K_{0,i}$ respectively. As such, the corresponding hardware has $p^G_{0,i}$ data streams as the input, which are fed into $p^G_{0,i}$ sliding window modules to extract the $k_{a,i} \times k_{a,i}$ windows for each channel. Each sliding window is implemented as a line buffer and the produced data windows are shared across $p^O_{0,i}p^K_{0,i}$ units to be multiplied with on-chip weights. The outputs of these multiplication units are accumulated on the $p^K_{0,i}$ dimension, so the final output of this layer contains $p^G_{0,i}p^O_{0,i}$ parallel data streams in total.

2) a pointwise 1×1 convolutional layer, with $c_i r_i$ input channels and c_{i+1} output

channels respectively. Let us also denote its parallelism on the input channel and the output channel dimension as $p^I_{1,i}$ and $p^O_{1,i}$ respectively. Since the kernel size is 1×1, no sliding window is required. $p^I_{1,i}$ parallel input data streams are directly fed to a multiplication array containing $p^I_{1,i}p^O_{1,i}$ units in total. The accumulation happens across every $p^I_{1,i}$ multiplication units, and it waits for $c_i r_i / p^I_{1,i}$ cycles to gather all input channels. The output of this layer contains $p^O_{1,i}$ parallel data stream.

Note that there is a mismatch for the data order transferred between these two layers. The outputs from grouped convolution are $\mathbb{R}^{c_i/p^G_{0,i} \times r_i/p^O_{0,i} \times p^G_{0,i} \times p^O_{0,i}}$, while the pointwise convolution expects the inputs to be $\mathbb{R}^{c_i/p^G_{0,i} \times p^G_{0,i} \times r_i/p^O_{0,i} \times p^O_{0,i}}$. To avoid the overhead of inserting a transpose module in the dataflow, StreamSVD changes the order of weights accordingly at compile-time. Therefore, compared with existing fpgaConvNet, StreamSVD is able to parallelize between not only channels but also groups, by introducing the extra design parameter $p^G_{0,i}$.

3.4.2 Hardware-aware Adjustments

StreamSVD also leverages the simulated annealing-based Design Space Exploration (DSE) algorithm from fpgaConvNet [19], which identifies the optimal parallelism configuration to maximize accelerator performance. Here, we treat the DSE step as a black box, focusing instead on integrating it with the overall framework. Detailed analysis and innovations regarding the DSE algorithm itself will be presented later in Section 5.3.2.

Given a decomposed network, StreamSVD utilizes DSE to map the network onto the FPGA and identify the optimal pipeline configuration. However, our design flow extends beyond this initial mapping. After completing the DSE, StreamSVD feeds the identified configuration and the estimated performance metrics back into the SVD compression algorithm. Based on these inputs and the following heuristics,

StreamSVD adjusts the compression decisions and re-initiates the DSE process for a second iteration, ensuring further refinement and optimization.

Two hardware-aware heuristics are considered to adjust the compression decisions:

Parallelism Padding: For a convolutional layer with c_i input channels, if its parallelism on the input channel dimension is denoted as p_i^I, the latency of computing this layer will be proportional to the ceiling of the division between the workload and the parallelism $\left\lceil \frac{c_i}{p_i^I} \right\rceil$. To maximize the operation occupancy and leave no idle cycle for the computation units, it is desirable that c_i can be divisible by p_i^I. For a decomposed CNN, the rank r_i controls the number of channels between the pair of decomposed convolutional layers, as shown in Figure 3.1. Therefore, StreamSVD uses the DSE feedback to constrain the choice of the rank, by padding the value of r_i to the nearest integer that is divisible by both the output channel parallelism $p_{0,i}^O$ and the input channel parallelism $p_{1,i}^I$.

$$r_{pad,i} = \left\lceil \frac{r_i}{lcm(p_{0,i}^O, p_{1,i}^I)} \right\rceil \tag{3.7}$$

In Equation (3.7), "lcm" stands for Least Common Multiple. The incremental update of the decomposition rank encourages the efficient utilization of computational resources and also reduces the compression error while maintaining the throughput of the design. Moreover, compared to other heuristic padding methods, such as constraining r_i to the nearest power of two, our method is more device- and model-specific considering the DSE result.

Layer Reverting:

Although SVD can reduce the number of operations and parameters in the CNN, when such a decomposed network is mapped to hardware, it also introduces the

overhead by doubling the number of convolutional layers. Compared to the layer-sequential architecture (Section 2.2.3), the layer-pipelined streaming architecture (Section 2.2.4) avoids extra off-chip accesses between the pair of decomposed layers. However, the overhead remains in the extra control logic and buffers to pipeline these extra layers. In addition, the reduction in weight parameters does not necessarily lead to the reduction in the BRAM utilization on FPGA because BRAM can only be configured into certain shapes [154].

Therefore, StreamSVD captures the individual BRAM utilization and throughput for each layer of the network based on the Design Space Exploration (DSE) predictions. It then compares those metrics before and after the compression. In cases where there is neither performance gain or BRAM reduction, StreamSVD will automatically revert the SVD compression for that particular layer, preserving the original convolutional layer configuration for better accuracy.

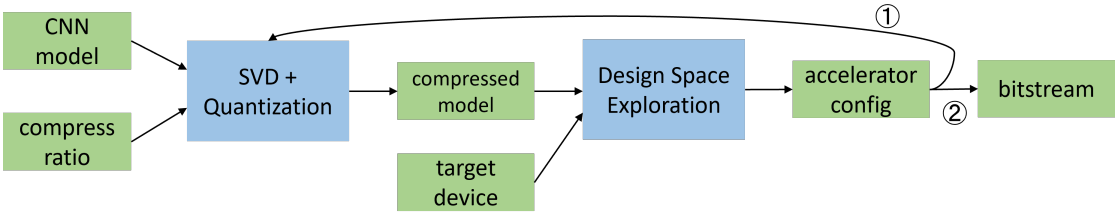


Figure 3.4: The hardware-aware workflow of StreamSVD. The compression and DSE are run twice, with the result of the first run guiding adjustments for the second. The final hardware accelerator configuration from the second run is then sent through a synthesis tool to generate the bitstream.

3.4.3 Co-design Summary

After introducing our compression algorithm and accelerator design, the complete automated workflow of StreamSVD can be described as follows (see Figure 3.4):

1. Given a pretrained CNN model and a target compression ratio (both typically specified by the user or determined by the application's requirements),

StreamSVD automatically selects the proper decomposition scheme and decomposition rank for every original convolutional layer (Section 3.2), where quantization is also applied as well (Section 3.3).

2. StreamSVD maps the compressed CNN model to FPGA, and utilizes DSE to identify the optimal pipeline configuration that maximizes the system throughput (Section 3.4.1). In addition, we gather the accelerator resource and performance predictions to adjust the compression decisions (Section 3.4.2).

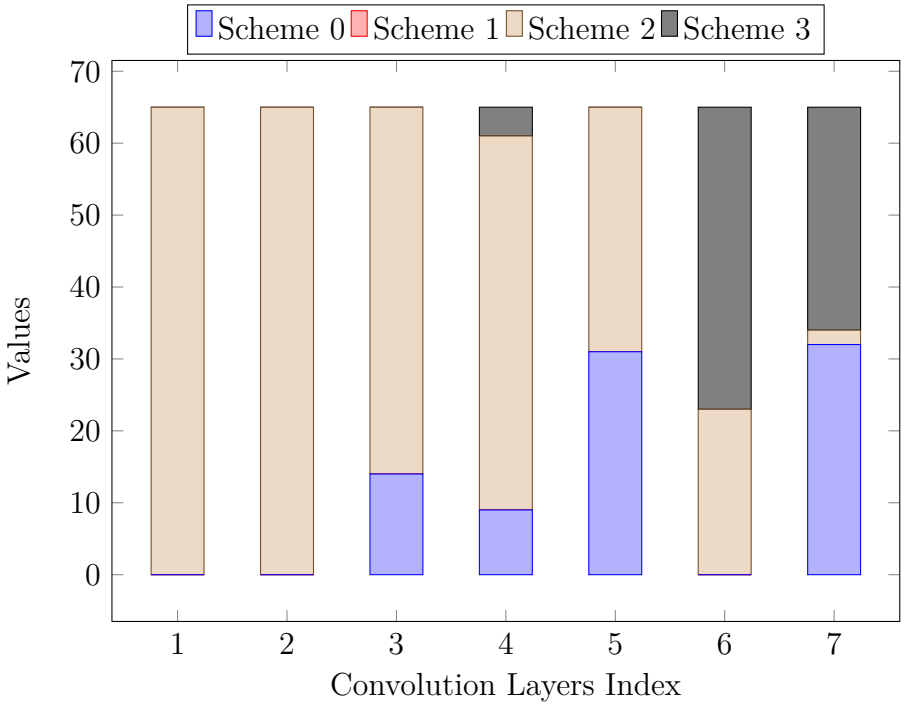
3. Based on the adjusted compression decisions, StreamSVD generates the final compressed model and its corresponding accelerator by repeating the DSE process. The accelerator design is synthesized and deployed on the target FPGA device.

3.5 Evaluation

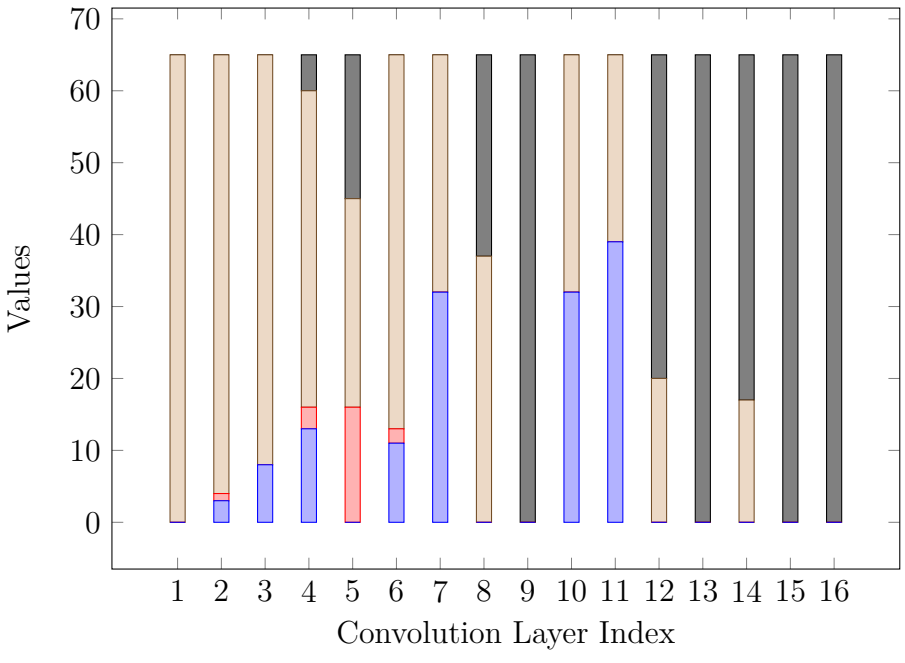
To evaluate the performance of StreamSVD, we consider the benchmark set including VGG11-BN, VGG16 and ResNet-18 which are pre-trained models provided by PyTorch TorchVision [11]. The post-training accuracy results are reported on the ImageNet validation set [1] without fine-tuning, while the throughput numbers are reported on a Xilinx ZC706 board.

3.5.1 Effectiveness of Scheme and Rank Selection

Notably, all prior work [18], [74], [151], [152] in the area of tensor decomposition has exclusively relied on uniform scheme selection, that is, applying the same decomposition scheme across all layers of CNN. In contrast, our work is the first to introduce a fine-grained, layerwise scheme selection algorithm that tailors the decomposition scheme to each individual layer. This innovative approach allows us



(a) VGG11-BN



(b) ResNet-18

Figure 3.5: Majority voting results illustrating our scheme selection method. For example, for the third convolutional layer in VGG11-BN, **scheme 2** is shown as the preferred choice. Notably, each layer exhibits a distinctive preference for a decomposition scheme, which varies not only between layers but also across different networks.

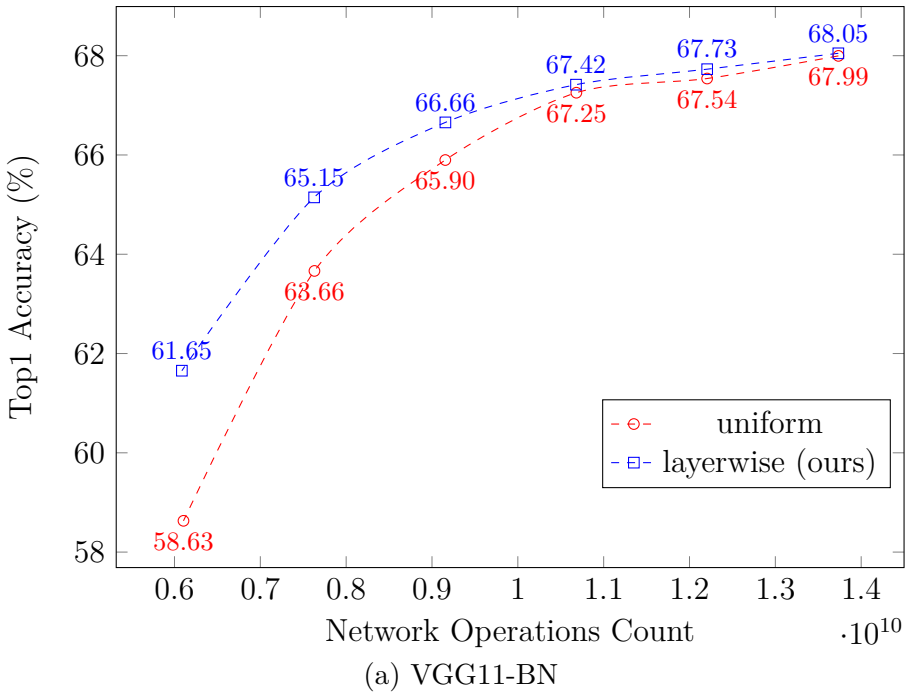
to exploit the unique properties of each layer, resulting in a more effective trade-off between accuracy and operation count.

Specifically, following Equation (3.3), StreamSVD runs a sweep test by sampling 65 different values of OP_{avail} between 0 and the original operation count, and uses a majority vote to decide the decomposition scheme per-layer. As such, we visualize the voting results for VGG11-BN and ResNet-18 in Figure 3.5. Our observation is that each layer exhibits a distinctive preference for a decomposition scheme. For example, for VGG11-BN, the first few layers (index 1 to 4) prefer **scheme 2**, while the last few layers (index 6 and 7) prefer **scheme 3** instead. The results for VGG16 are not shown here, as it follows a similar trend to VGG11-BN, and both models belong to the same family.

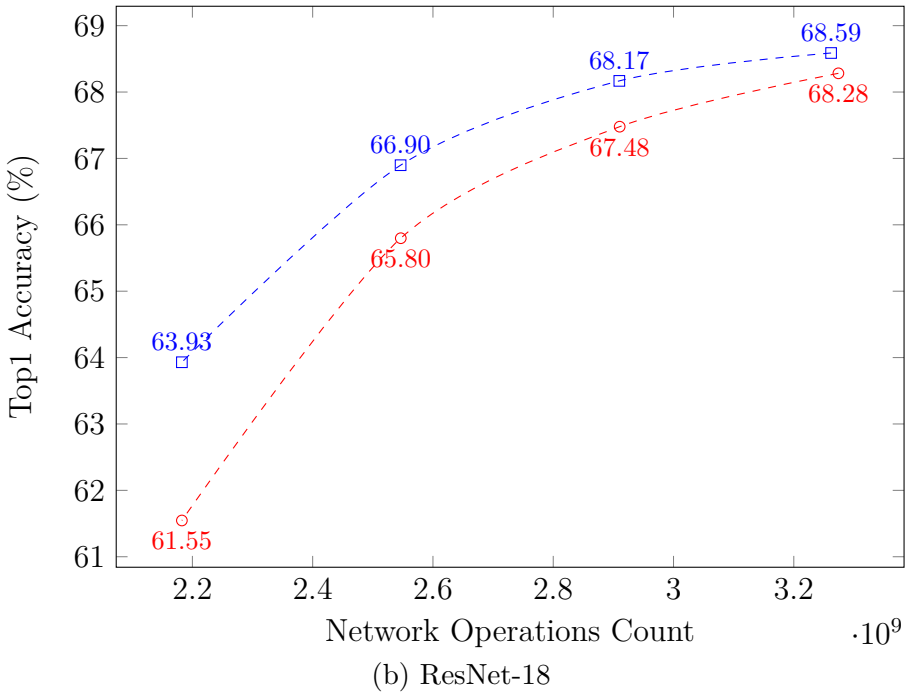
We further conduct a comparative analysis between our accuracy versus operation count frontier against the prior uniform scheme selection approaches. Since we consider four different decomposition schemes in total, each model yields four distinct trade-off curves under uniform scheme selection. For clarity and visual simplicity, we present only the optimal curve among these in Figure 3.6, ensuring the comparison focuses on the best-case performance of the uniform approach.

The results demonstrate that, under the same operation count target, our layerwise method consistently achieves higher accuracy. These findings suggest that there is no universally optimal decomposition scheme for all layers in a CNN. Consequently, the fine-grained and layerwise selection of schemes, as demonstrated by our approach, proves to be beneficial.

In terms of the selection of decomposition rank, StreamSVD converts the problem to a special form of channel pruning where the importance of rank is evaluated by the Taylor importance criterion (Equation (3.4)). In Figure 3.7, we compared our Taylor pruning approach against the following methods:

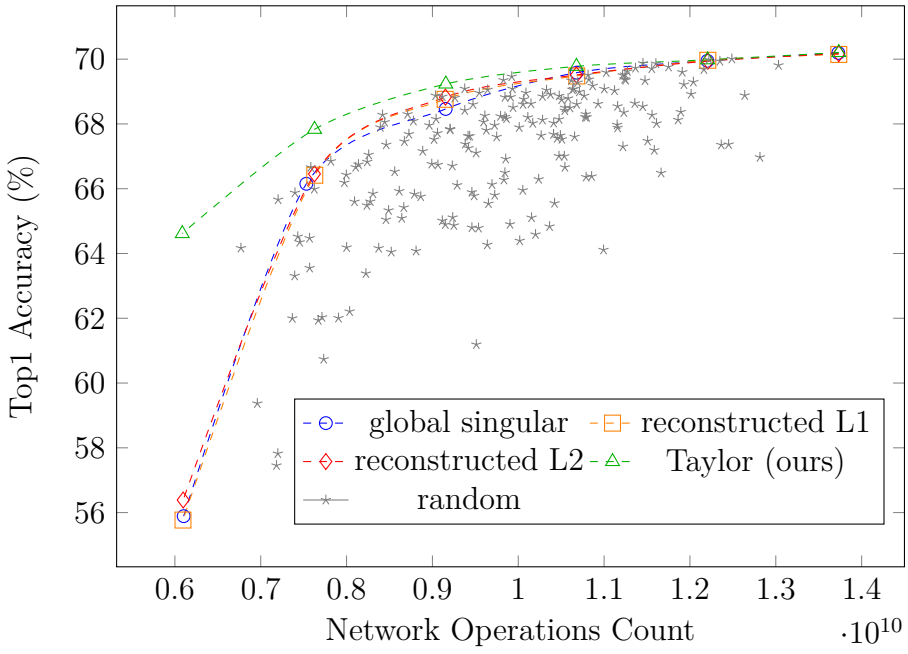


(a) VGG11-BN

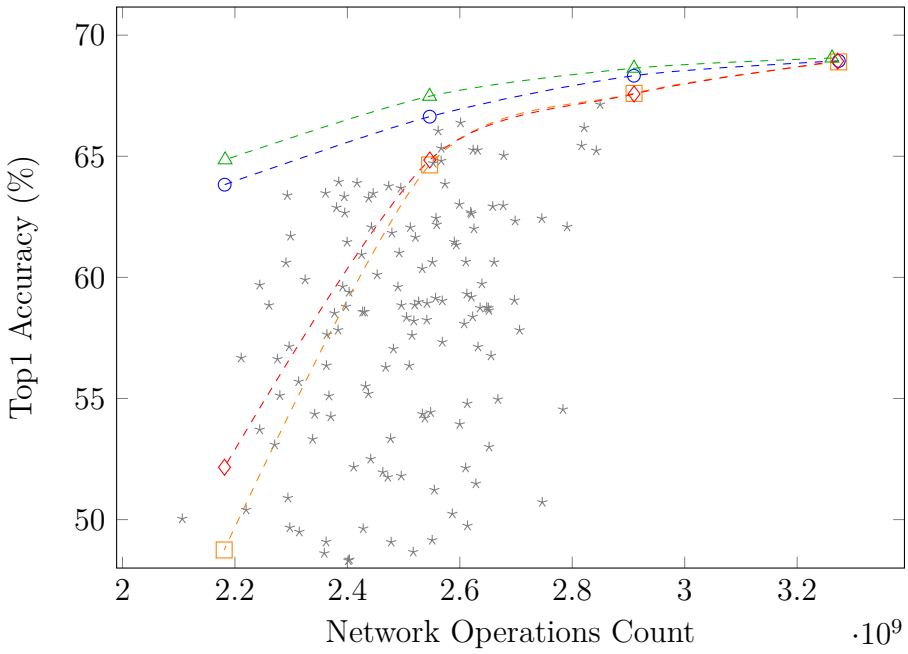


(b) ResNet-18

Figure 3.6: Accuracy-operations trade-off curves with accuracy results annotated. The existing literature [18], [74], [151], [152] adopts uniform scheme selection by forcing all the convolutional layers in the network to adopt the same decomposition scheme. We demonstrate the advantage of layerwise scheme selection.



(a) VGG11-BN



(b) ResNet-18

Figure 3.7: Selecting decomposition ranks with different criteria.

global singular [155]: The significance of rank r_i is assessed using the ratio $\frac{\Delta\varepsilon/\varepsilon}{\Delta OP}$. Here, $\Delta\varepsilon$ denotes the r_i-th largest eigenvalue while ε represents the sum of all eigenvalues. The term ΔOP quantifies the reduction in operations achieved by decreasing the rank by one.

reconstructed L1/2 [78]: The significance of rank r_i is estimated by the reconstruction error between the original weight tensor and the decomposed tensors in L1 or L2 norm, which can be represented as $||\mathcal{W}_i - \mathcal{W}_{1,i}\mathcal{W}_{0,i}||_1$ or $||\mathcal{W}_i - \mathcal{W}_{1,i}\mathcal{W}_{0,i}||_2$ respectively.

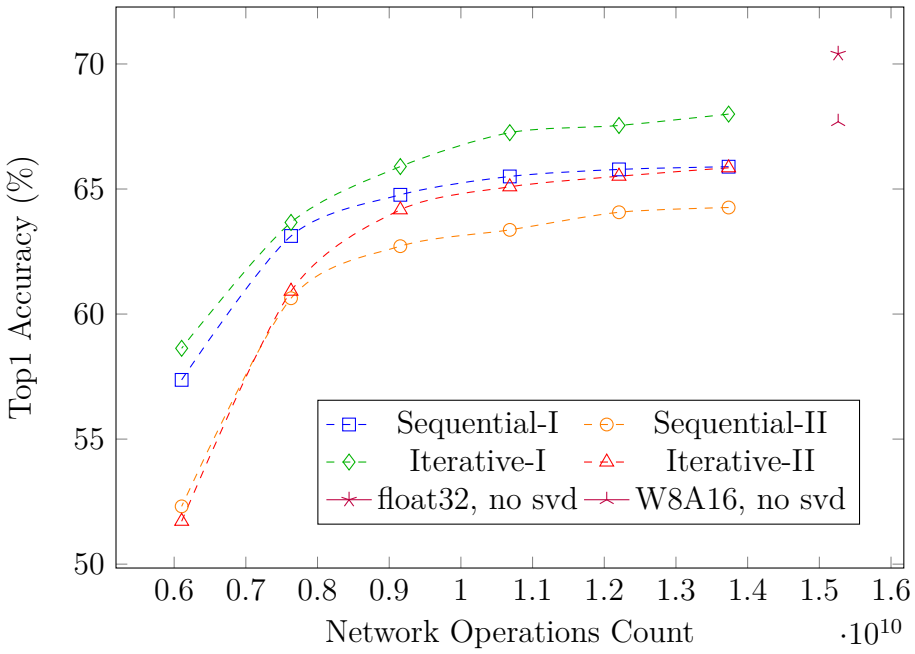
random: Randomly selecting the rank in each layer is theoretically capable of finding the optimal rank given sufficient search time. However, in practice, this method becomes impractical as the search time grows exponentially with the depth of the network.

In Figure 3.7, we present 200 data points for the random selection approach across both models. Our observations show that the Taylor method significantly outperforms the other approaches, with its advantage becoming more pronounced at lower operation counts (i.e., higher compression ratios).

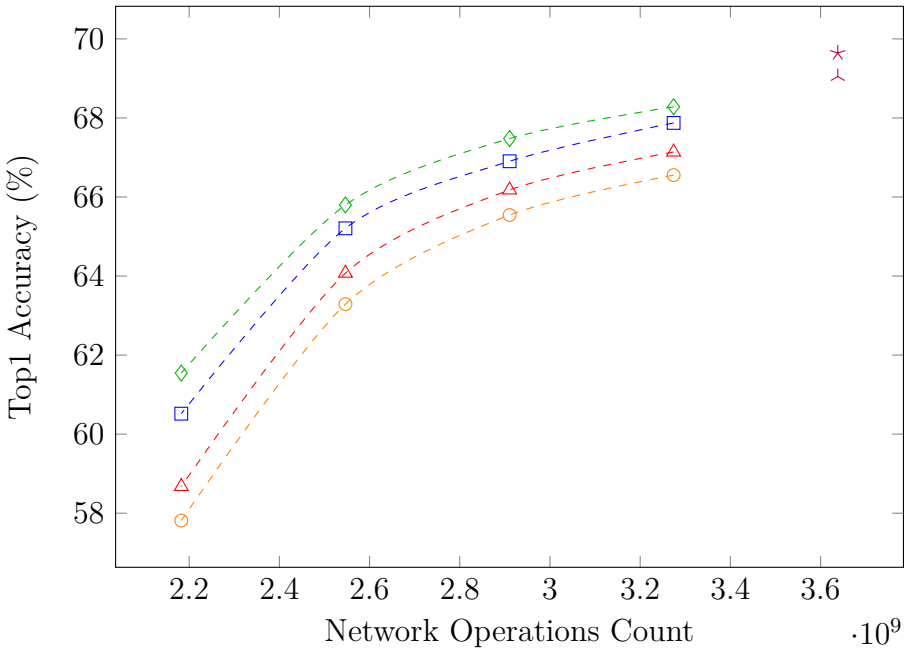
3.5.2 Combined Effect of SVD and Quantization

In Section 3.3, we have introduced four variants of algorithms to combine SVD and quantization, which are **Sequential-I**, **Sequential-II**, **Iterative-I** and **Iterative-II**. We evaluated the accuracy and operation count trade-off for these four methods in Figure 3.8, and we made the following observations:

- **Iterative** methods are superior to **Sequential** methods by iteratively compensating for quantization errors through the subtraction operation within the refinement loop, as illustrated in Algorithm 1.



(a) VGG11-BN



(b) ResNet-18

Figure 3.8: Combining SVD and quantization, where **Iterative-I** algorithm offers the highest accuracy under the same operation count budget.

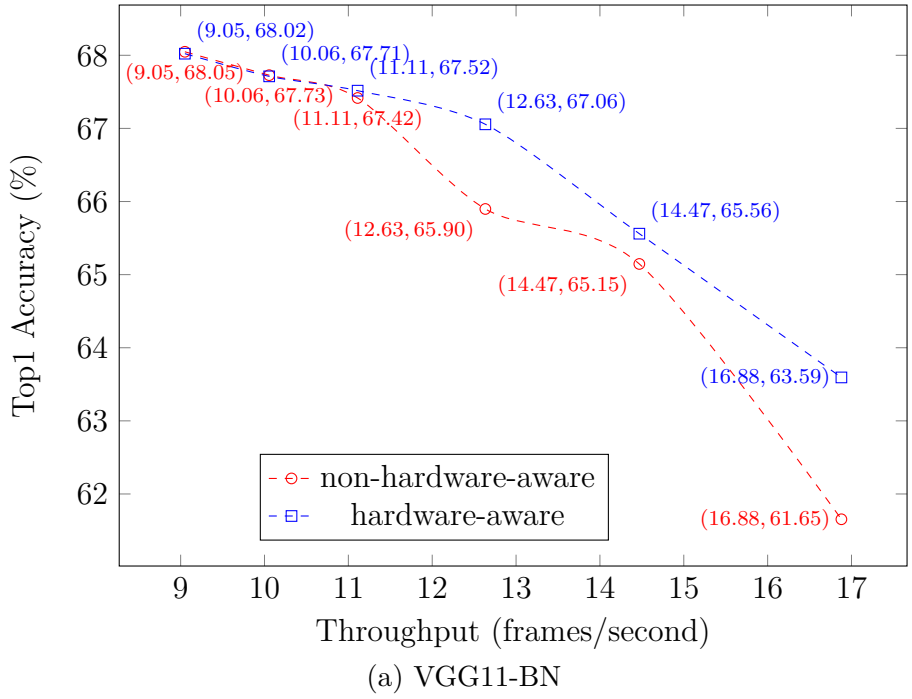
- methods **I**, which apply quantization after SVD, demonstrate superior performance compared to methods **II**, which apply quantization before SVD. This highlights the importance of the order in which quantization and low-rank approximation are applied.
- Overall, we found **Iterative-I** is the best approach, at least for the models we evaluated.

3.5.3 Impact of Hardware-awareness

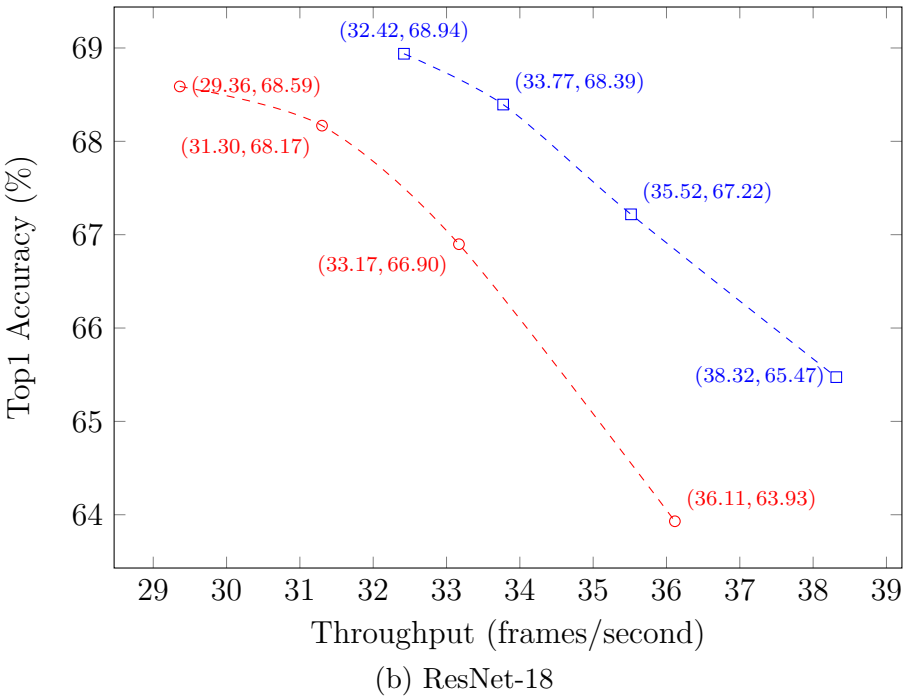
StreamSVD enhances the compression algorithm by integrating hardware awareness, as detailed in Section 3.4.2. We conducted an ablation study to compare model accuracy and accelerator throughput with and without these techniques. The results are presented in Figure 3.9.

For VGG11-BN, we observe that the introduction of hardware-aware techniques maintains the same throughput while improving accuracy. The consistent throughput indicates that the two DSE iterations (Figure 3.4) produce identical hardware accelerator configurations. However, during the second DSE iteration, the decomposition decisions become hardware-aware through parallelism padding and layer-reverting techniques. Consequently, the network undergoes less compression while preserving the same throughput, effectively reducing accuracy degradation for VGG11-BN.

The situation for ResNet-18 differs slightly, as both accuracy and throughput improve after the second iteration of DSE. In this case, hardware-awareness helps avoid compression decisions where the benefits are overwhelmed by the overhead of decomposition. By rejecting these overheads, the saved resources can be reallocated to other parts of the accelerator through the DSE algorithm, resulting in further speed-up.



(a) VGG11-BN



(b) ResNet-18

Figure 3.9: Accuracy-Throughput trade-off achieved, where top-right points are preferred. For VGG11-BN, hardware-awareness benefits accuracy, while for ResNet-18, both accuracy and throughput get boosted.

3.5.4 Comparison with Direct Channel Pruning

In Section 3.2.3, we demonstrate that reducing the rank of weight matrices is a special form of channel pruning, where we decompose the network by doubling the number of convolutional layer and only prune between any pair of decomposed layers. As such, we further compare StreamSVD's performance to direct channel pruning where no decomposition is utilized at all. For direct channel pruning, the same Taylor pruning criterion is applied and it is hardware-aware as well.

Our finding is that in the post-training scenario, direct channel pruning causes evident accuracy degradation (>10.0 percentage points) and brings a limited throughput improvement (<1.1×) on both models. Meanwhile, StreamSVD provides up to 1.9× throughput with 4.1 percentage points (pp) accuracy loss on VGG11-BN, and 1.6× throughput with 3.6 pp accuracy loss on ResNet-18. As such, pruning the decomposed network is superior to directly pruning the original network, particularly under the post-training scenario where no fine-tuning is allowed.

3.5.5 Comparison with Other Accelerators

This work integrates the SVD algorithm with a layer-pipelined, streaming architecture accelerator. To thoroughly evaluate the impact of the architecture, our algorithm is also tested on SCALE-Sim [120], which employs a layer-sequential, single computation engine design based on systolic arrays. The results are shown in Figure 3.10.

In an ideal scenario, halving the number of operations in a model could result in up to a 2× speed-up in throughput. In the case of targeting a streaming accelerator, StreamSVD achieves a notable speed-up of more than 1.7× on VGG11-BN. However, the speed-up achieved on the layer-sequential SCALE-Sim is 1.4×, where

we also observe a notable 30% increase in off-chip DRAM accesses after tensor decomposition. This additional off-chip overhead is attributed to the fact that the number of convolutional layers effectively doubles after decomposition, necessitating the storage of their intermediate activation data off-chip. In contrast, the streaming architecture of StreamSVD avoids this issue through a dataflow pipeline.

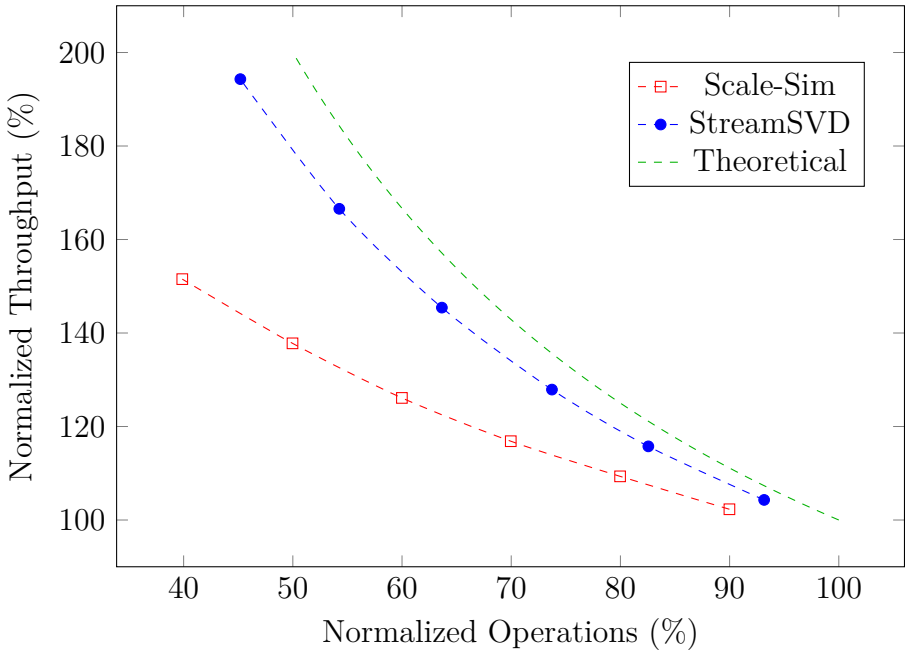
Table 3.2: Comparison with other FPGA accelerators for decomposed CNNs.

Method	Model	Quantize	Acc. (%)	FPS (frames/sec)	Device	Clock (MHz)	DSP	BRAM	Efficiency $\times 10^{-9}$ (FPS/DSP/MHz)
[156]	ResNet-18	W2A8	68.2	20.5	Ultra96	250	202	298	0.41
Ours		W8A16	68.4	33.8	ZC706	125	576	752	0.47
Ours	VGG11-BN	W8A16	67.5	11.1	ZC706	125	755	576	0.12
[157]	VGG16	float16	70.5	6.6	7V690T	200	1728	393	0.02
[73]		W16A16	64.6	4.5	ZC706	150	780	992	0.04
[158]		W16A16	-	5.5	7Z045	140	864	640	0.05
Ours		W8A16	70.2	5.6	ZC706	125	603	784	0.07

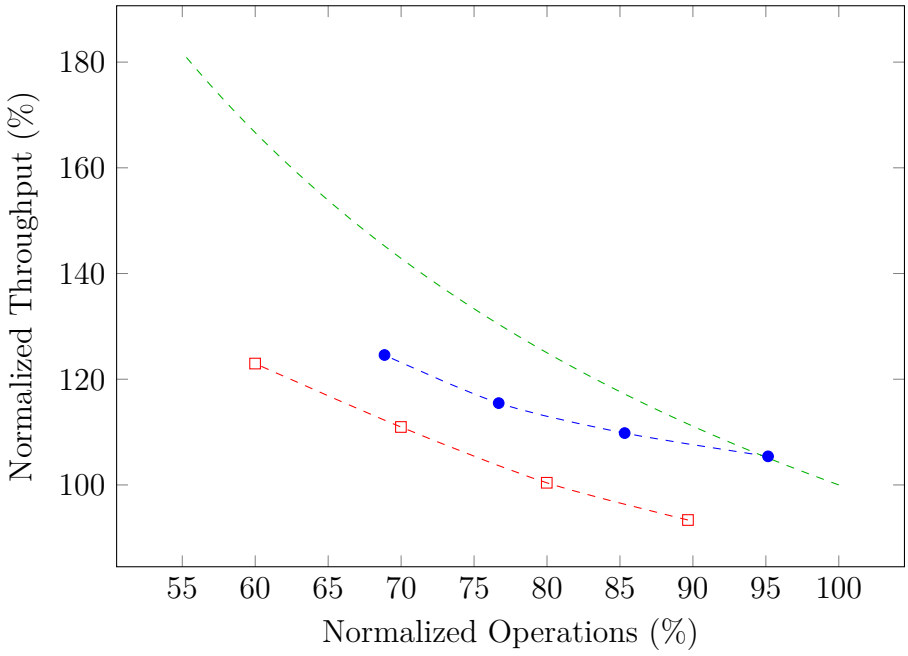
Finally, Table 3.2 compares StreamSVD with other FPGA accelerators that support decomposed CNNs, from which we have made the following observations:

- In the ResNet-18 comparison, our approach achieved a throughput of 1.65× compared to 3U-EdgeAI[156]. Additionally, our accuracy slightly surpassed that of 3U-EdgeAI, with respective values of 68.4% and 68.2%.
- Similarly, on the comparison of VGG16, our approach is also superior to VGG16-SVD [73] and NEURAghe [158] on both accuracy and throughput.
- In comparison with Mei *et al.* [157], it's important to note that their implementation utilizes float16 on a relatively large FPGA device (Xilinx 7V670T). Despite these disparities, when normalizing throughput by DSP utilization and clock frequency, our efficiency surpasses theirs.

In summary, we find StreamSVD outperforms existing accelerator designs that incorporate tensor decomposition for compressing CNN models. StreamSVD achieves superior efficiency and a more favorable accuracy-throughput trade-off.



(a) VGG11-BN



(b) ResNet-18

Figure 3.10: The relationship between network operation count reduction and actual throughput speed-up. All values are normalized with respect to the baseline, which is the quantized-only, no SVD configuration. The 'Theoretical' curve represents the scenario where throughput is the reciprocal of the operation count.

3.6 Summary

In this chapter, we demonstrate that SVD can be effectively used to compress AI models in a fine-grained, per-layer, and hardware-aware manner. Specifically, we show accuracy improvements by combining existing decomposition schemes and customizing decomposition ranks on a per-layer basis.

However, deploying such decomposed models on traditional hardware is inefficient due to the frequent use of pointwise and grouped convolutions, as well as the doubling of the total number of layers. To address these challenges, we extend fpgaConvNet to produce a streaming, dataflow accelerator that overcomes these bottlenecks, enabling efficient deployment of decomposed models. As a result, the system we produce delivers a superior accuracy-throughput trade-off compared to related work.

Regarding limitations, this chapter focuses solely on mixing existing decomposition schemes. In the following chapter, we will explore further extensions beyond these approaches.

4

SVD-NAS: Coupling Low-Rank Approximation and Neural Architecture Search

Contents

In the previous chapter (Section 3.2.2), we demonstrated existing solutions for SVD-based tensor decomposition schemes. In this chapter, we will generalize and extend these schemes by constructing a novel low-rank network architecture design space. By leveraging this extended low-rank design space, we will demonstrate that a better model accuracy and compression ratio can be achieved over prior work. Additionally, we will show that these gains hold even when the models are deployed on CPU or GPU devices. This chapter is based on the conference paper SVD-NAS [159] published in WACV 2023.

4.1 Introduction

Tensor decomposition serves as a technique to compress the CNN weights by replacing them with a set of low-rank tensors, effectively reducing the number of parameters. However, existing solutions [18], [74], [151], [152] have been limited in considering a restricted number of design choices concerning the decomposition schemes (Section 3.2.2), leaving a significant number of potential schemes unexplored. For instance, additional schemes can be generated by manipulating how the 4-D convolution weight tensor is transposed and sliced to 2-D so that SVD can then be applied.

Furthermore, in previous work, the determination of decomposition schemes is either dependent on extensive manual effort [18] or relies on using the Mean Squared Error (MSE) of the weights as a proxy for assessing the network's overall accuracy degradation [148]. We will show that these existing approaches can lead to sub-optimal decisions. Instead, we will demonstrate choosing the decomposition schemes can be interpreted as a Neural Architecture Search (NAS) problem, which allows for a more comprehensive exploration of design choices and leads to a better model

accuracy and compression ratio compared to prior work.

Figure 4.1 demonstrates the design flow of SVD-NAS, and its key novel aspects are summarized as follows:

- We describe the process of tensor decomposition as a per-layer substitution of the original pre-trained network. For every layer, we introduce a Low-Rank model architecture design space, *LR-space*, which is defined by a set of parameterisable building blocks. We demonstrate that searching the design parameters of these building blocks is equivalent to exploring different decomposition schemes and ranks.

- As the *LR-space* grows exponentially with the number of layers, brute-force exploration becomes impractical. To address this, we adopt a gradient-descent-based NAS approach for efficient navigation, enabling co-optimization of accuracy and performance, using a single Nvidia 2080 or 1080Ti GPU.

- In real-world scenarios, access to the original training dataset might not be easily granted. Thus, we evaluate the effectiveness of our approach in three different scenarios: post-training, where no training data are available; few-sample training, where only a tiny subset of training data are shared; and full training, where the complete training data are available. Under these conditions, and at a similar compression ratio, our proposed method, SVD-NAS, achieves 2.06-12.85 percentage points (pp) higher accuracy on ImageNet compared to state-of-the-art methods [16], [78], [86].

- To further evaluate the actual impact on the execution time reduction on a device, we present an example of latency-aware tensor decomposition tailored for deployment on a Pixel 4 phone. For this example, our approach achieves up to 45% latency reduction with at most a 2 pp accuracy degradation.

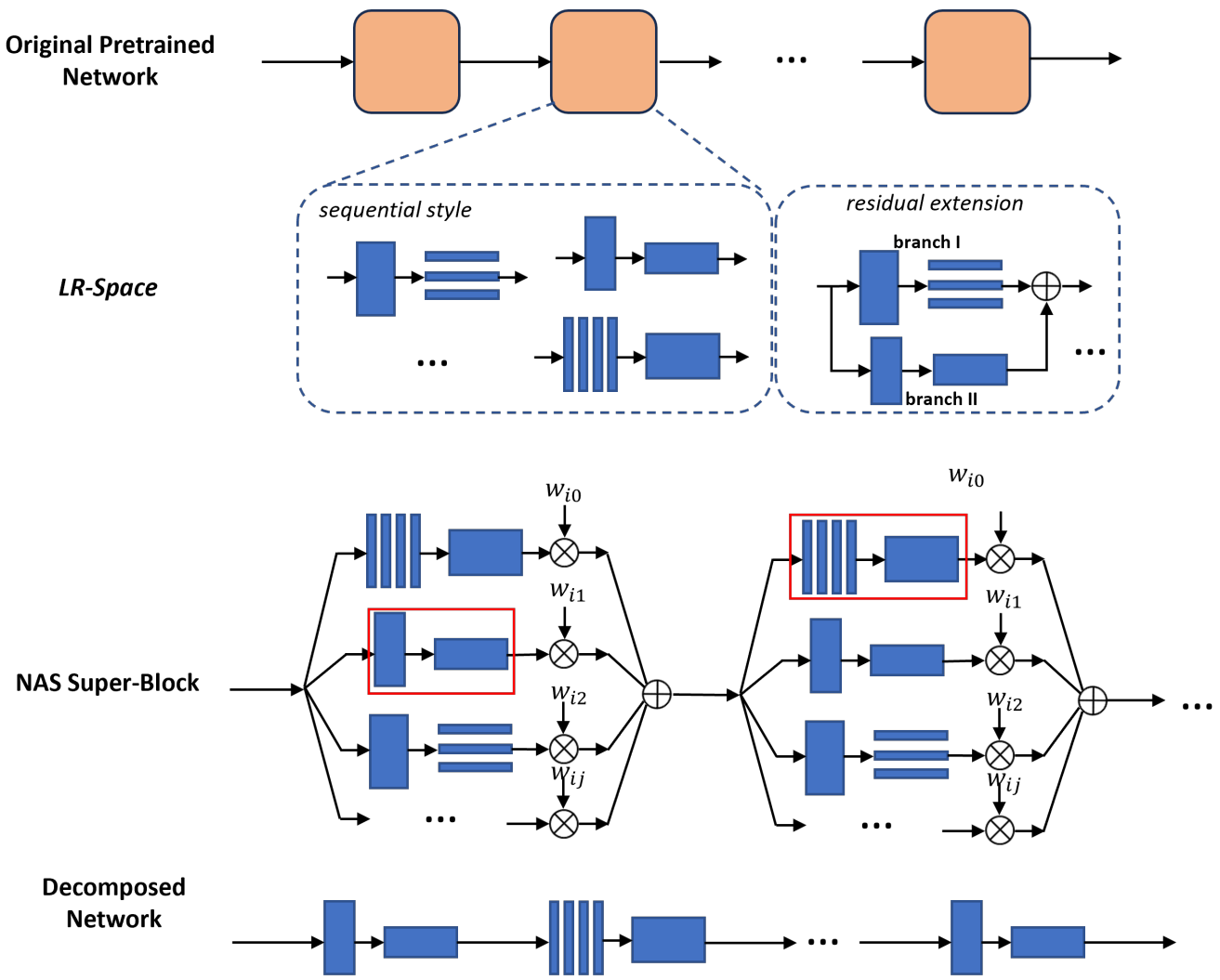


Figure 4.1: SVD-NAS starts with a pre-trained CNN model, where each layer's weights are represented by orange tensors. For tensor decomposition, it defines a per-layer *LR-space*, and the decomposed layers are visualized as blue tensors. The optimal decomposition configuration is identified through a gradient-descent-based NAS process, with the final solution highlighted using red bounding boxes.

4.2 *LR-space*

Let's consider a pre-trained CNN, where the i^{th} convolutional layer processes c_i input channels using $f_i = c_{i+1}$ filters with a kernel size of $k_{a,i} \times k_{b,i}$. The process of tensor decomposition substitutes this original convolutional layer with a parameterisable building block, where the design space is named as *LR-space*.

4.2.1 Sequential Building Block

In the sequential form of *LR-space*, each building block contains two consecutive convolutional layers that are sequentially connected. Similar to the previous notations, the input channels of these two layers are represented by $c_{0,i}$ and $c_{1,i}$, the number of filters by $f_{0,i}$ and $f_{1,i}$, and the kernel size by $k_{a0,i} \times k_{b0,i}$ and $k_{a1,i} \times k_{b1,i}$, respectively. Moreover, these two layers can also be grouped convolutions [26], [31] where the number of groups is denoted as $g_{0,i}$ and $g_{1,i}$ respectively. Regarding these design parameters of the building block, the following constraints are introduced so that the decomposed weight tensors can be derived with SVD without the need to retrain from scratch:

Constraint 1: $(k_{a0,i}k_{a1,i} = k_{a,i}) \wedge (k_{b0,i}k_{b1,i} = k_{b,i})$. The product of these two kernel sizes is constrained to be equal to the original kernel size, which maintains the same receptive field as the original pre-trained convolutional layer.

Constraint 2: $(min(g_{0,i}, g_{1,i}) = 1) \wedge (g_{0,i} \mid c_i) \wedge (g_{1,i} \mid f_i)$, where the symbol "|" expresses divisibility. This constraint ensures the two consecutive layers inside the building block cannot be grouped convolutions at the same time. As such, it maintains the connectivity of the original, standard convolution where each filter was connected to all input channels.

Constraint 3: $(c_{0,i} = c_i) \wedge (f_{1,i} = f_i) \wedge (f_{0,i} = c_{1,i} = max(g_{0,i}, g_{1,i})r_i)$. This set of conditions ensures the amount of activation data entering or exiting the building block remains the same as the original layer. Also, the number of channels between two decomposed layers is controlled by the grouped convolution as well as the decomposition rank r_i.

After decomposition, the number of floating-point operations (FLOPs) is proportional to the number of parameters as well as to the feature width and height. the number of weight parameters involved in each sequential form of building block can be expressed as:

$$\text{Params} = max(g_{0,i}, g_{1,i})r_i \left(\frac{c_i}{g_{0,i}} k_{a0,i} k_{b0,i} + \frac{f_i}{g_{1,i}} k_{a1,i} k_{b1,i} \right) \tag{4.1}$$

and the number of floating-point operations can be represented as:

$$\text{FLOPs} = max(g_{0,i}, g_{1,i})r_i m_i n_i \left(\frac{c_i}{g_{0,i}} k_{a0,i} k_{b0,i} s_{a1,i} s_{b1_i} + \frac{f_i}{g_{1,i}} k_{a1,i} k_{b1,i} \right) \tag{4.2}$$

where m_i and n_i denote the feature map height and width at the output of the building block. In addition, $s_{a1,i}$, $s_{b1,i}$ denote the 2-D kernel stride for the second layer inside the building block.

4.2.2 Weight Derivation

In this section, we demonstrate how to use SVD to derive the weights of the parameterisable building block from the original convolutional layer without retraining from scratch. Same as before, we denote the 4^{th}-order weight tensor of the original convolutional layer as $\mathcal{W}_i \in \mathbb{R}^{c_{i+1} \times c_i \times k_{a,i} \times k_{b,i}}$.

Step 1: For this 4^{th}-order weight tensor, if we slice and split on its first and second dimension into $g_{1,i}$ and $g_{0,i}$ groups respectively, we will obtain $g_i^0 g_i^1$ chunks in total, where the dimensions of each chunk are $\mathbb{R}^{\frac{c_{i+1}}{g_{1,i}} \times \frac{c_i}{g_{0,i}} \times k_{a,i} \times k_{b,i}}$. Due to the previously defined constraint on design parameters, the last two dimensions can also be represented as $k_{a1,i}k_{a0,i} \times k_{b1,i}k_{b0,i}$.

Step 2: Next, we reshape each chunk from 4-D into 2-D, where the new tensor has the shape of $(\frac{c_{i+1}}{g_{1,i}} k_{a1,i}k_{b1,i}) \times (\frac{c_i}{g_{0,i}} k_{a0,i}k_{b0,i})$. By applying SVD to each 2-D chunk and keeping only the top-r_i singular values, two low-rank chunks are obtained whose shapes are $(\frac{c_{i+1}}{g_{1,i}} k_{a1,i}k_{b1,i}) \times r_i$ and $r_i \times (\frac{c_i}{g_{0,i}} k_{a0,i}k_{b0,i})$ respectively.

Step 3: The obtained low-rank chunks are reshaped again, but this time, they are transformed into 4-D tensors: $\mathbb{R}^{\frac{c_{i+1}}{g_{1,i}} \times r_i \times k_{a1,i} \times k_{b1,i}}$ and $\mathbb{R}^{\frac{c_i}{g_{0,i}} \times r_i \times k_{a0,i} \times k_{b0,i}}$, which represent the weights of two consecutive layers inside the building block. In case of $g_{1,i}$ or $g_{0,i}$ is larger than one, the corresponding chunks are combined to represent the weights for grouped convolutions. [26], [31].

A visualization of the above steps is shown in Figure 4.2, where the orange tensors represent the uncompressed weights and the blue tensors represent the low-rank approximations.

4.2.3 Case Study

To further understand the design choice in the proposed *LR-space*, let us look at a specific example of the second convolutional layer in the ResNet-18 model. This convolutional layer uses $f_i = c_{i+1} = 64$ filters to process $c_i = 64$ input channels, where each filter has the kernel size of $k_{a,i} \times k_{b,i} = 3 \times 3$. In the proposed *LR-space*, the decomposition scheme is characterized by the design parameters $k_{a0,i}$, $k_{b0,i}$, $k_{a1,i}$, $k_{b1,i}$, $g_{0,i}$ and $g_{1,i}$.

Considering **Constraint 1**, and given that $k_{a,i} = k_{b,i} = 3$ are prime numbers,

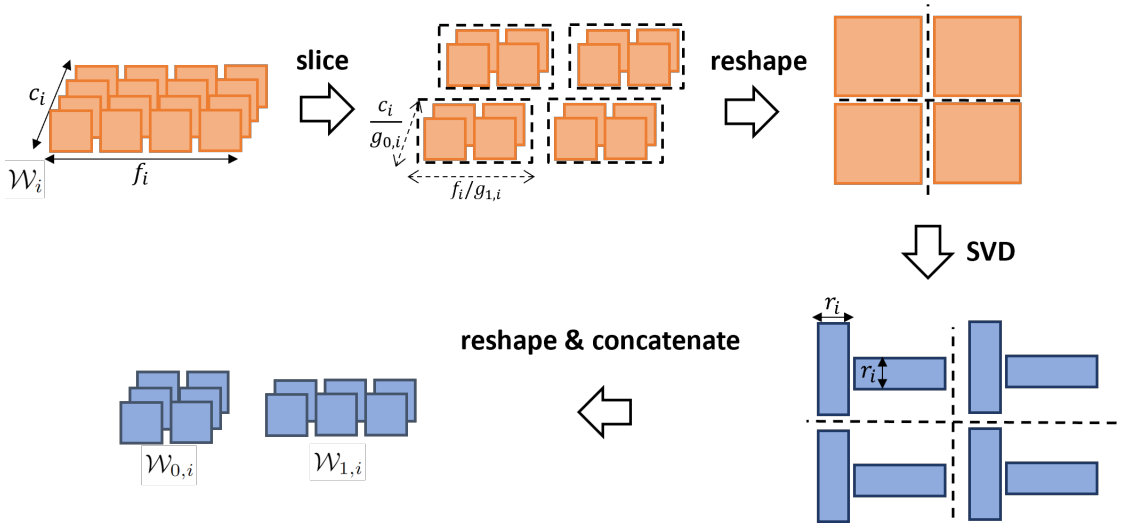


Figure 4.2: The weight derivation process for the *LR-space* involves transformations between tensors, where operations on tensors shown in orange are lossless. However, the SVD process introduces a lossy transformation by reducing the tensor ranks. As a result, post-SVD tensors are represented in blue to indicate their status as low-rank representations.

the feasible choices for $k_{a0,i} \times k_{b0,i}$ are restricted to $\{1 \times 1, 3 \times 3, 3 \times 1, 1 \times 3\}$. Once $k_{a0,i} \times k_{b0,i}$ is determined, $k_{a1,i} \times k_{b1,i}$ becomes uniquely defined, adhering to the constraint on the product of kernel sizes and producing 4 different combination.

Moving to **Constraint 2**, where at least one of the design parameters g_0 and g_1 is set to one, and both g_0 and g_1 are constrained to be factors of 64. Consequently, considering the combination of these two design parameters, there are $log_2(64) + log_2(64) + 1 = 13$ choices available.

Hence, in the discussed example, there exist $4 \times 13 = 52$ potential decomposition schemes in total. It is noteworthy that, as discussed in Section 3.2.2, previous research introduced only 4 decomposition schemes. We demonstrate that these are essentially the corner cases within the comprehensive design space of *LR-space*, as detailed in Table 4.1.

The **top** of Figure 4.3 shows the accuracy vs. FLOPs trade-off for the possible decomposition schemes in our expanded design space when considering a single layer. As the figure demonstrates, the Pareto front is populated with a number of different

Table 4.1: The proposed *LR-space* generalizes previous works in terms of the decomposition scheme, offering an expanded design space.

	$k_{a0,i} \times k_{b0,i}$	$k_{a1,i} \times k_{b1,i}$	g_0	g_1
scheme 0 [151]	1×1	3×3	1	64
scheme 1 [18]	3×3	1×1	1	1
scheme 2 [74]	1×3	3×1	1	1
scheme 3 [152]	3×3	1×1	64	1

schemes. In addition, as shown at the **bottom** of Figure 4.3, using the Mean Square Error (MSE) of the weight tensor to estimate the impact on accuracy degradation may also lead to a sub-optimal solution. In fact, the problem of choosing the optimal decomposition scheme becomes even more complicated when considering the entire network. As highlighted in previous literature[70], the contribution of each layer to the overall model accuracy varies.

To address these challenges, we employ Neural Architecture Search (NAS) to guide for the the exploration of decomposition scheme and rank, which will be detailed in Section 4.3.1.

4.2.4 Residual Extension

We further extend our exploration by considering the replacement of each convolutional layer in the original network with a residual-style building block (Figure 4.1). This residual block consists of two sequential building blocks that share the same input and combine their outputs using element-wise addition. The design parameters of these two sequential building blocks are still constrained by the conditions described in Section 4.2.1, but these two blocks can have different configurations, making the total number of combinations squared.

The weight tensors of the residual block can still be derived using SVD, and without model retraining. Specifically, we first decompose the pre-trained weight

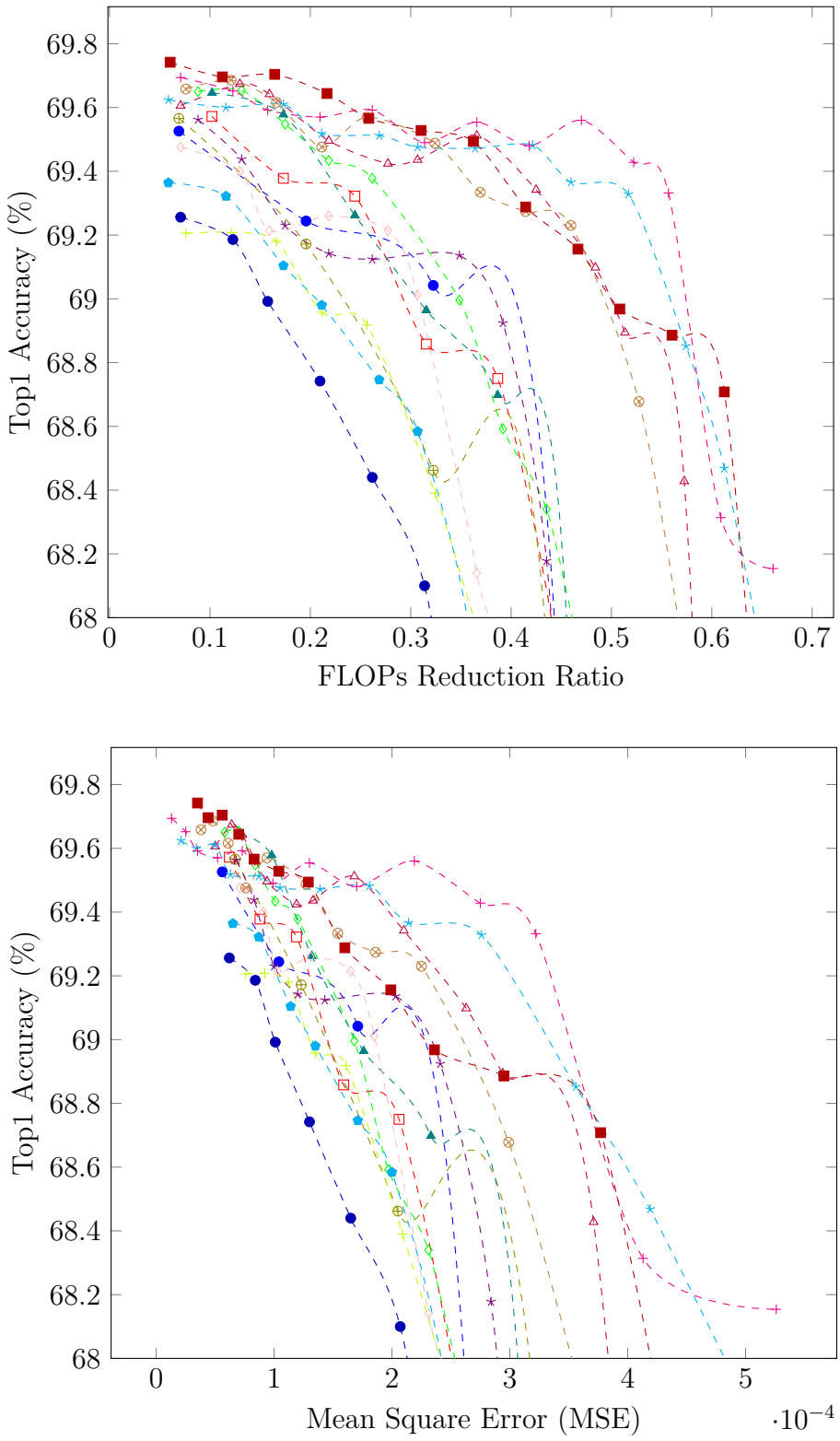


Figure 4.3: Explore the *LR-space* of the second convolutional layer in ResNet-18. All other layers are not compressed. Each type of marker corresponds to a specific decomposition scheme. The figures display 15 out of the 52 available schemes, with the remaining schemes omitted due to consistently low accuracy.

tensor $\mathcal{W}_i$ in the original network according to the design parameters in **branch I**, then calculate the decomposition error $\mathcal{E}_i$ which was introduced during the SVD process, and finally decompose $\mathcal{E}_i$ according to the design parameters specified to **branch II** (Figure 4.1).

Note that the incorporation of the residual extension is an optional feature within the *LR-space*. As demonstrated in the evaluation section, our results reveal that the effectiveness of this residual extension in achieving a superior performance trade-off varies on a case-by-case basis.

4.3 Search Algorithm

4.3.1 Gradient-descent NAS

With the design space defined by per-layer design parameters, we formulate a multi-objective optimization problem. The goal is to minimize the computational requirements while simultaneously maximizing the accuracy of the network. Notably, in the example discussed in Section 4.2.3, a single convolutional layer can present 52 available decomposition schemes. The design space expands significantly when considering the entire network, especially when combined with the choice of decomposition ranks.

In fact, brute-force exploration is not realistic given the size of the design space and the time required to evaluate the accuracy of individual design points. Therefore, we use a gradient-descent NAS approach [97] to solve this optimization problem.

The procedure is to take a pre-trained CNN model and replace every convolutional layer with a super-block. As Figure 4.1 shows, the super-block computes a weighted sum of candidate building blocks drawn from *LR-space*. For the j^{th} can-

didate in the i^{th} super-block, its weight w_{ij} is a scalar, and it is generated using the Gumbel-Softmax function [160], where

$$w_{ij} = \frac{\exp\left(\log(\pi_{ij}) + G_i\right)}{\sum_j \exp\left(\log(\pi_{ij}) + G_j\right)} \tag{4.3}$$

G_j is a sample from the Gumbel distribution, while π_{ij} is referred to as the sampling parameter in previous literature. π_{ij} is a learnable parameter whose value suggests how much the j^{th} candidate is favored in the i^{th} super-block. Throughout the search process, the sampling parameter π_{ij} is updated via gradient descent to minimize the following multi-objective loss function, where $\mathcal{L}_{entropy}$ and ΔFLOPs represents the entropy loss and the reduction in FLOPs respectively. Note that ΔFLOPs is calculated as the weighted sum of per-candidate FLOPs, using the same weights as in Equation (4.3), minus the FLOPs of the original convolutional layer. β is a hyperparameter to balance the objectives of reducing the FLOPs while maintaining the accuracy.

$$\mathcal{L}_{nas} = \mathcal{L}_{entropy} \cdot [log(-\Delta\text{FLOPs})]^{\beta} \tag{4.4}$$

4.3.2 Reduce Search Cost

The above algorithm is a standard gradient-descent NAS process, however, we found that the standard process can be time-consuming and GPU-demanding to be explored, given the huge size of *LR-space*. Therefore, as Figure 4.4 shows, we further introduce the following three techniques for more efficient exploration.

Design Space Pruning: Before the search starts, we prune the *LR-space* to reduce the number of candidate configurations considered. Candidates are selected based on their FLOPs being close to predetermined percentages γ (e.g., 95%, 90%,

85%, ..., 5%) of the FLOPs of the original layer. In addition, we observe the accuracy by compressing only one layer using a candidate configuration, while the remaining layers are left uncompressed. If the accuracy degradation exceeds a predefined threshold τ, such a candidate configuration is also excluded for that layer.

Selective Candidate Sampling: After pruning the design space, the remaining candidate configurations are constructed as a per-layer super-block. During every iteration of the search, we randomly sample a subset of candidates in each super-block, and only compute the weighted sum of this subset, while masking out the remaining candidates temporarily. In our experiment in Section 4.4, we sample two candidates per super-block per iteration, as this ensures that the NAS process can fit efficiently on a single GPU device. The probabilities of each candidate getting selected are given by the softmaxed sampling parameters, which is similar to Equation (4.3) but without the Gumbel sampling term G_j [161]. Consequently, in each search iteration, only the sampling parameters of the two selected candidates are updated according to Equation 4.4.

Greedy Branch Search: Our approach firstly determines the per-layer configurations with the sequential building blocks. Once these configurations are fixed, we then exploit the residual extension of the *LR-space*. Specifically, two additional convolutional layers are incorporated into each building block, and a second round of search is conducted to explore the configurations for these newly added layers.

4.4 Experiments

We conduct the experiments on the ImageNet dataset [1] using pre-trained ResNet-18 [7], MobileNetV2 [162] and EfficientNet-B0 [34] coming from torchvision [163]. We selected these models because they are widely used in related works [78], [86], [164] for benchmarking. Thanks to the techniques discussed in the previous section,

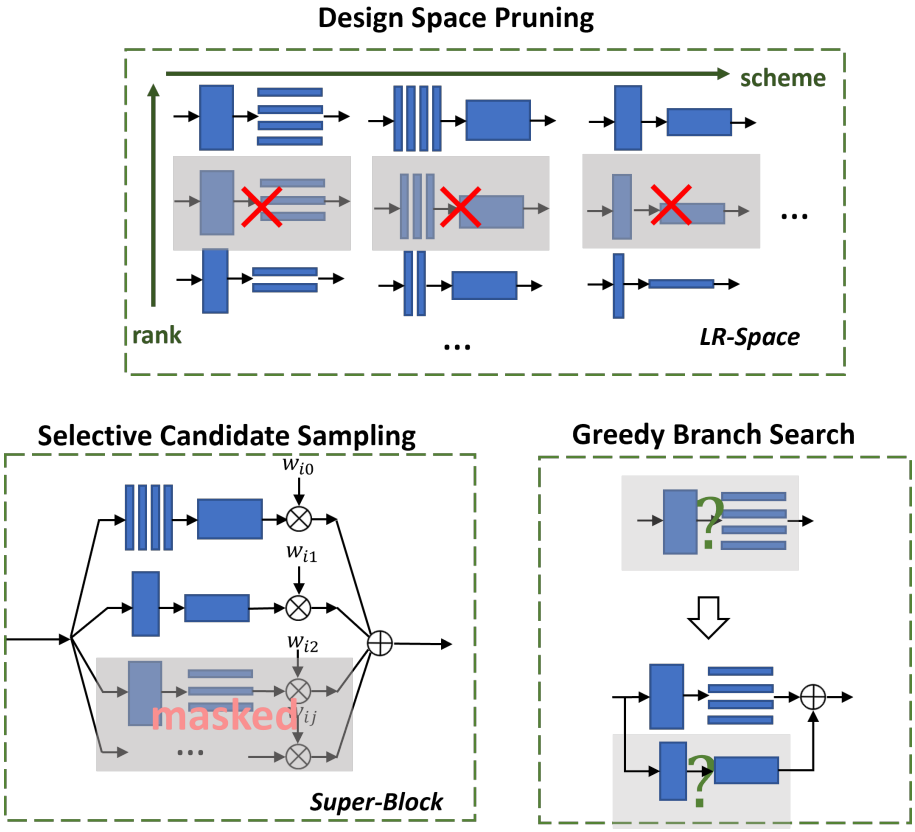


Figure 4.4: Given the huge size of *LR-space*, the above techniques are introduced to enable more efficient exploration of the design space.

we can perform the search on a single NVIDIA GeForce RTX 2080/1080 Ti GPU, and the search takes 9-18 hours to complete.

To evaluate the effectiveness of the proposed model compression approach, we assess its impact under three settings: **post-training**, where no training data are available; **few-sample training**, where only a small subset of the training data are accessible; and **full training**, where the complete training dataset is available. The key metrics of interest include the percentage reduction in FLOPs and parameters, along with the degradation in Top-1 and Top-5 accuracy, measured in percentage points (pp).

Note that, due to limitations in available data and the reproducibility of prior works, the choice of baseline methods may vary across different models. Consequently, we selected the most relevant baselines for each model to ensure a fair and

meaningful comparison.

4.4.1 Post-training Comparison

In some real-world scenarios, access to the original training dataset might not be easily granted, especially when the training dataset is of value or contains sensitive information. In this situation, compressing a pre-trained model without further fine-tuning becomes of interest.

In Table 4.2, we present the post-training compression results of our approach, where no fine-tuning is applied at all. For each model, multiple design points are showcased to illustrate the tradeoff between accuracy and compression. Additionally, results for both the sequential building block and the residual version are provided, highlighting that the advantage of the residual extension varies case by case.

We then pick our best results and contrast them to current state-of-the-art approaches [16], [78], [86] in Table 4.3. All these existing methods employ a uniform decomposition scheme across all layers in the network. Moreover, ALDS [86] and LR-S2 [78] automate rank selection based on the MSE heuristics, while F-Group [16] manually tunes the ranks.

The results in Table 4.3 clearly demonstrate that our SVD-NAS approach significantly outperforms existing methods when no fine-tuning is applied. Specifically, for ResNet-18 and EfficientNet-B0, our method achieves the highest compression ratios in terms of both FLOPs and parameters. For instance, SVD-NAS reduces the FLOPs of ResNet-18 by 58.60% and the parameters by 68.05%, while maintaining a 13.35% drop in Top-1 accuracy and a 9.14% drop in Top-5 accuracy. This represents a considerable improvement over other methods such as ALDS and LR-S2, which show much higher accuracy degradation (at least 18.70pp Top-1 and 13.38 Top-5).

Table 4.2: Post-training results of SVD-NAS, without any fine-tuning. Δ F/P denotes the percentage reduction in FLOPs and parameters, while Δ Top1/5 represents the accuracy degradation in percentage points (pp). The hyperparameter β balances the cross-entropy term and computation cost term in Equation (4.4). γ and τ control the level of design space pruning (see Section 4.3.2).

ResNet-18	β=48, γ=[0.3,0.7]		β=48, γ=[0.4,0.8]		β=32, γ=[0.4,0.8]	
	Δ F/P	Δ Top1/5	Δ F/P	Δ Top1/5	Δ F/P	Δ Top1/5
sequential	-59.17/-66.77	-17.58/-12.79	-53.64/-59.79	-10.55/-7.22	-50.39/-58.79	-6.75/-4.41
residual	-62.53/-72.61	-20.73/-18.88	-60.74/-69.21	-16.20/-11.12	-58.60/-68.05	-13.35/-9.14
	β=16, γ=[0.5,0.9]		β=4, γ=[0.5,0.9]		β=2, γ=[0.5,0.9]	
	Δ F/P	Δ Top1/5	Δ F/P	Δ Top1/5	Δ F/P	Δ Top1/5
sequential	-38.91/-43.71	-3.32/-2.50	-25.50/-36.70	-1.12/-0.84	-17.30/-27.19	-0.64/-0.24
residual	-45.16/-53.64	-4.20/-2.52	-28.86/-42.19	-1.38/-0.75	-13.98/-26.94	-0.67/-0.30
MobileNetV2	β=88, γ=[0.6,0.95], τ=5		β=80, γ=[0.6,0.95], τ=5		β=64, γ=[0.6,0.95], τ=5	
	Δ F/P	Δ Top1/5	Δ F/P	Δ Top1/5	Δ F/P	Δ Top1/5
sequential	-14.17/-10.66	-19.82/-13.15	-12.54/-9.00	-15.09/-7.79	-7.73/-6.70	-7.66/-4.46
residual	-16.13/-13.24	-23.16/-15.75	-13.64/-9.70	-17.59/-11.39	-8.09/-6.60	-8.06/-4.70
	β=48, γ=[0.6,0.95], τ=5		β=32, γ=[0.6,0.95], τ=5			
	Δ F/P	Δ Top1/5	Δ F/P	Δ Top1/5		
sequential	-6.53/-5.82	-5.26/-3.08	-1.92/-2.74	-1.90/-0.86		
residual	-6.60/-5.71	-4.69/-2.72	-2.20/-2.34	-1.62/-0.84		
EfficientNet-B0	β=88, γ=[0.5,0.95], τ=5		β=80, γ=[0.5,0.95], τ=5		β=64, γ=[0.5,0.95], τ=5	
	Δ F/P	Δ Top1/5	Δ F/P	Δ Top1/5	Δ F/P	Δ Top1/5
sequential	-26.53/-17.69	-24.65/-15.92	-22.85/-16.06	-13.15/-7.45	-21.04/-15.60	-10.77/-5.95
residual	-29.83/-20.60	-23.47/-15.02	-28.52/-20.38	-18.65/-11.20	-25.72/-18.86	-13.96/-8.10
	β=48, γ=[0.5,0.95], τ=5		β=32, γ=[0.5,0.95], τ=5		β=16, γ=[0.5,0.95], τ=5	
	Δ F/P	Δ Top1/5	Δ F/P	Δ Top1/5	Δ F/P	Δ Top1/5
sequential	-18.10/-13.23	-6.99/-3.79	-15.32/-12.08	-5.98/-3.05	-8.97/-6.11	-2.98/-1.52
residual	-22.17/-16.41	-10.11/-5.49	-15.95/-12.85	-6.03/-3.16	-8.85/-6.19	-2.91/-1.54

Table 4.3: Comparison with related work without fine-tuning.

Model	Method	Δ FLOPs (%)	Δ Params (%)	Δ Top-1 (pp)	Δ Top-5 (pp)
ResNet-18	SVD-NAS(Ours)	-58.60	-68.05	-13.35	-9.14
	ALDS [86]	-42.31	-65.14	-18.70	-13.38
	LR-S2 [78]	-56.49	-57.91	-38.13	-33.93
	F-Group[16]	-42.31	-10.66	-69.34	-87.63
MobileNetV2	SVD-NAS(Ours)	-12.54	-9.00	-15.09	-7.79
	ALDS [86]	-2.62	-37.61	-16.95	-10.91
	LR-S2 [78]	-3.81	-6.24	-17.46	-10.34
EfficientNet-B0	SVD-NAS(Ours)	-22.17	-16.41	-10.11	-5.49
	ALDS [86]	-7.65	-10.02	-16.88	-9.96
	LR-S2 [78]	-18.73	-14.56	-22.08	-14.15

For EfficientNet-B0, our approach reduces FLOPs by 22.17% and parameters by 16.41%, with only a 10.11% drop in Top-1 accuracy and 5.49% drop in Top-5 accuracy. This again outperforms ALDS and LR-S2, which exhibit larger accuracy losses (e.g., 22.08% drop in Top-1 accuracy for LR-S2 on EfficientNet-B0). In the case of MobileNetV2, while our method achieves the best accuracy-FLOPs trade-off, it does not lead to the highest reduction in parameters compared to ALDS and LR-S2. This is due to the fact that we do not explicitly prioritize parameters reduction in the loss function (Equation (4.4)), which shall be extended in our future work.

4.4.2 Fine-tune with Synthetic Data

Even though our framework outperforms the state-of-the-art approaches, we still observe a significant amount of accuracy degradation under the problem setting where training data are not available for fine-tuning. Under such a setting, an alternative solution is to generate an unlabelled synthetic dataset and then use knowledge distillation for fine-tuning.

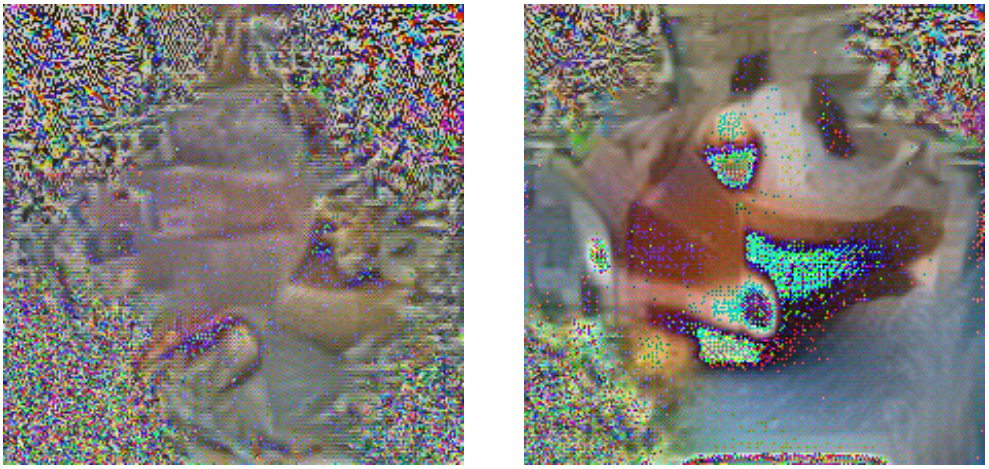
This kind of solution has been explored before to recover quantization error [62], but we are the first to show its successful application on tensor decomposition

as well. Specifically, building on prior work, the synthetic data are generated by feeding randomly initialized images $\boldsymbol{\mathcal{I}}$ to the uncompressed pre-trained network, and optimizing these images using back-propagation, with the target of minimizing the following loss function:

$$\begin{aligned}\mathcal{L}_{syn}(\boldsymbol{\mathcal{I}}) = \alpha_{syn}[(\mu(\boldsymbol{\mathcal{I}}))^2 + (\sigma(\boldsymbol{\mathcal{I}}) - 1)^2] \\ + \sum_i \frac{1}{f_i} \sum_j [(\mu_{ij}(\boldsymbol{\mathcal{I}}) - \mu'_{ij})^2 + (\sigma_{ij}(\boldsymbol{\mathcal{I}}) - \sigma'_{ij})^2]\end{aligned} \tag{4.5}$$

In the above equation, $\mu(\boldsymbol{\mathcal{I}})$ and $\sigma(\boldsymbol{\mathcal{I}})$ represent the mean and standard deviation of the current image itself. We aim for these two values to be close to 0 and 1, respectively, ensuring that the pixel values follow a standard normal distribution. μ'_{ij} and σ'_{ij} are the running mean and running standard deviation stored in the j^{th} channel of the i^{th} batch normalization layer from the pre-trained model, while $\mu_{ij}(\boldsymbol{\mathcal{I}})$ and $\sigma_{ij}(\boldsymbol{\mathcal{I}})$ represent the corresponding statistics recorded when the current image is fed into the original pre-trained network. We aim to match these numbers to ensure consistency in per-layer activation statistics. α_{syn} is a hyperparameter that balances two terms.

Moreover, compared to ZeroQ [62], our objective function introduces a key modification: we average the loss for each batch normalization layer by scaling it with the number of channels f_i. We found this small change leads to better fine-tuning results, especially on those compact models such as MobileNetV2 [162] and EfficientNet [34] where the number of channels in these compact models varies a lot between layers. Figure 4.5 shows two sample images taken from the original ZeroQ implementation and our method respectively on MobileNetV2. Without the introduced scaling term, the synthetic image becomes noisy, which we found, is more likely to cause overfitting during the later fine-tuning.



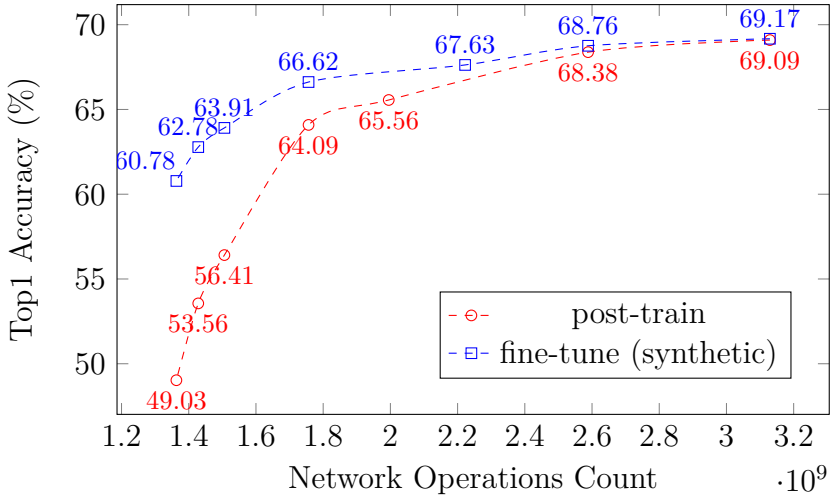
(a) ZeroQ[62] (b) Ours

Figure 4.5: Synthetic images of MobileNetV2 generated by (a) ZeroQ [62] and (b) our approach. The difference between the two images is caused by the scaling term f_i in Equation (4.5), which helps reduce distortion and improves the quality of the synthetic images generated by our method.

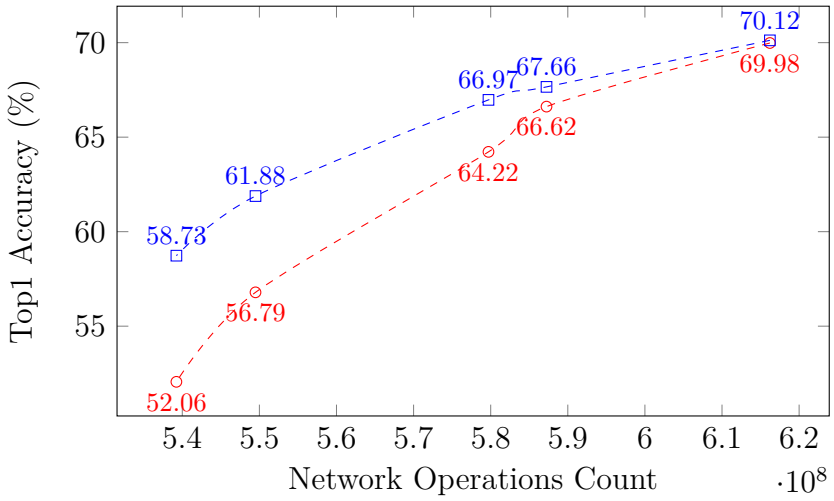
The number of synthetic images also affects the fine-tuning result, and empirically, we observe the accuracy improvement saturates when using 25k synthetic images. Once the synthetic dataset is generated, we treat the original pre-trained model as the teacher and the decomposed model as the student. Since the synthetic dataset is unlabelled, the knowledge distillation focuses on minimizing the MSE of per-layer outputs. Figure 4.6 compares model accuracy before and after distillation. Overall, using the synthetic dataset for knowledge distillation results in an average accuracy improvement of 3.76 pp across the three target models, without using any real training data.

4.4.3 Few-sample Fine-tune

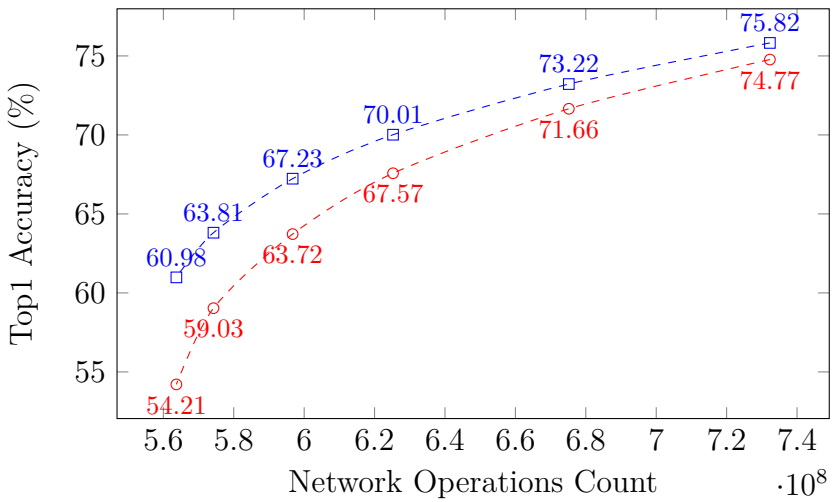
Fine-tuning with few samples differs from the previous problem setting in that now a small proportion of training data becomes available. Specifically, for evaluation, we randomly select 1k images from the ImageNet training set as a subset and fix it throughout the experiment. During fine-tuning, we found that using the entropy



(a) ResNet-18



(b) MobileNetV2



(c) EfficientNet-B0

Figure 4.6: Fine-tune the decomposed models using 25k synthetic images.

loss alone can lead to overfitting due to the limited number of samples available. Instead, applying knowledge distillation methods leads to better accuracy results. The loss function for knowledge distillation, as proposed by previous work [89], can be represented as:

$$\mathcal{L}_{kd} = \sum_{i} \text{MSE}(\hat{y}_i, y_i) + \alpha_{kd} \cdot T_{kd}^2 \cdot \mathcal{L}_{KLdiv} + (1 - \alpha_{kd}) \cdot \mathcal{L}_{entropy} \tag{4.6}$$

This includes the Mean Squared Error (MSE) on the outputs of convolutional layers, where $\hat{y}_i$ and y_i represent the per-layer outputs on the decomposed model and the original model, respectively. Additionally, it involves the Kullback-Leibler (KL) divergence on logits, which are softened by temperature T_{kd} (set at 6), and the cross-entropy loss on the decomposed model where the hyperparameter α_{kd} is set at 0.95.

Table 4.4: Comparison with few-sample pruning.

Model	Method	Structured	Δ FLOPs (%)	Δ Params (%)	Δ Top-1 (pp)	Δ Top-5 (pp)
ResNet-18	SVD-NAS(Ours)	✓	-59.17	-66.77	-3.95	-2.36
	FSKD [92]	✓	-59.01	-64.64	-6.01	-
	Reborn [165]	✓	-33.33	-	-	-4.24
	POT [166]	✗	-	-50.00	-	-1.48
MobileNetV2	SVD-NAS(Ours)	✓	-14.17	-10.66	-6.63	-3.61
	MiR [167]	✓	-13.30	-7.70	-1.80	-
	POT [166]	✗	-	-40.00	-	-2.87

As no previous work has reported results on few-sample tensor decomposition, we compare our framework with existing works on few-sample pruning instead. From Table 4.4, our SVD-NAS achieves a competitive accuracy-FLOPs trade-off on ResNet-18, particularly when compared to structured channel pruning methods. Specifically, our SVD-NAS achieves similar FLOPs and parameters reduction as FSKD [92]. Meanwhile, our accuracy degradation is significantly less, with a Top-1 accuracy drop of -3.95pp compared to -6.01pp for FSKD.

It is noteworthy that the compression ratio of MobileNetV2 achieved by our method is relatively less pronounced compared to pruning methods. This observation can be attributed to MobileNetV2's extensive use of depthwise and pointwise convolutions, which exhibit less redundancy in terms of weight tensor ranks [33]. Thus, pruning methods may offer more drastic reductions in such cases.

4.4.4 Full-training Fine-tune

We further employ the entire training set for fine-tuning the decomposed models. The results of this process are presented in Table 4.5. All these related works employ a single decomposition scheme uniformly across layers and models, whereas our SVD-NAS extends and generalizes them through the proposed *LR-space*.

Table 4.5: Fine-tune decomposed networks on the full training set.

Model	Method	Year	Δ FLOPs (%)	Δ Params (%)	Δ Top-1 (pp)	Δ Top-5 (pp)
ResNet-18	SVD-NAS(Ours)	2022	-59.17	-66.77	+0.03	+0.10
	ALDS [86]	2021	-43.51	-66.70	-0.40	-0.05
	ADMM-TT [168]	2021	-59.51	-	-	0.00
	CPD-EPC [169]	2020	-67.64	-73.82	-0.69	-0.15
	S-Conv [164]	2020	-51.23	-52.18	-0.63	-
	MUSCO [170]	2019	-58.67	-	-0.47	-0.30
	TRP [171]	2019	-68.55	-3.76	-2.33	
MobileNetV2	SVD-NAS(Ours)	2022	-14.17	-10.66	-1.66	-1.90
	ALDS [86]	2021	-11.01	-32.97	-1.53	-0.73
	Shared [172]	2021	0.00	-7.43	+0.39	0.37
	S-Conv [164]	2020	-19.67	-25.14	-0.90	-

For ResNet-18, our SVD-NAS achieves a remarkable 59.17% reduction in FLOPs and 66.77% decrease in parameters, all without any compromise on accuracy. Notably, other comparable methods like MUSCO [170] and S-Conv [164] report a minimum accuracy degradation of 0.47pp. In the case of MobileNetv2, ourSVD-NAS demonstrates competitive results compared to other state-of-the-art approaches.

We also observe that the advantage of our SVD-NAS over state-of-the-art is correlated with the problem setting on data availability. Under comparable compression ratios, our SVD-NAS demonstrates a significant accuracy gain in post-training scenarios (up to 6.77 pp) and few-sample settings (up to 2.06 pp). However, this advantage becomes less pronounced in full-training scenarios, where the improvement is up to 0.43 pp.

This finding suggests that when the data access is limited, the design choices of tensor decomposition should be more carefully considered, while when a lot of training data are available, the performance gap between different design choices can be compensated through fine-tuning.

4.4.5 Latency-aware Search

So far, the decisions of tensor decomposition have focused on reducing the FLOPs without considering the actual impact on the execution time reduction on a device. To address this, we target a Pixel 4 phone equipped with an ARM Cortex-A76 CPU. We integrate the open-source tool nn-Meter [173] to create a lookup table for latency estimation specific to this device. The latency lookup table is then used to replace the FLOPs term in Equation (4.4). Once the latency-aware search completes, we deploy the decomposed model on the device using the Tensorflow Lite 2.1 native benchmark binary [174].

The experiments were conducted using a single thread, with a fixed batch size of 1, and the reported result is an average of 50 runs. Our decomposed networks successfully reduce latency by 49.46%, 4.75% and 6.46% for ResNet-18, MobileNetV2, and EfficientNet-B0, respectively. Notably, this is achieved with an accuracy degradation within 2pp after fine-tuning on the entire ImageNet training set.

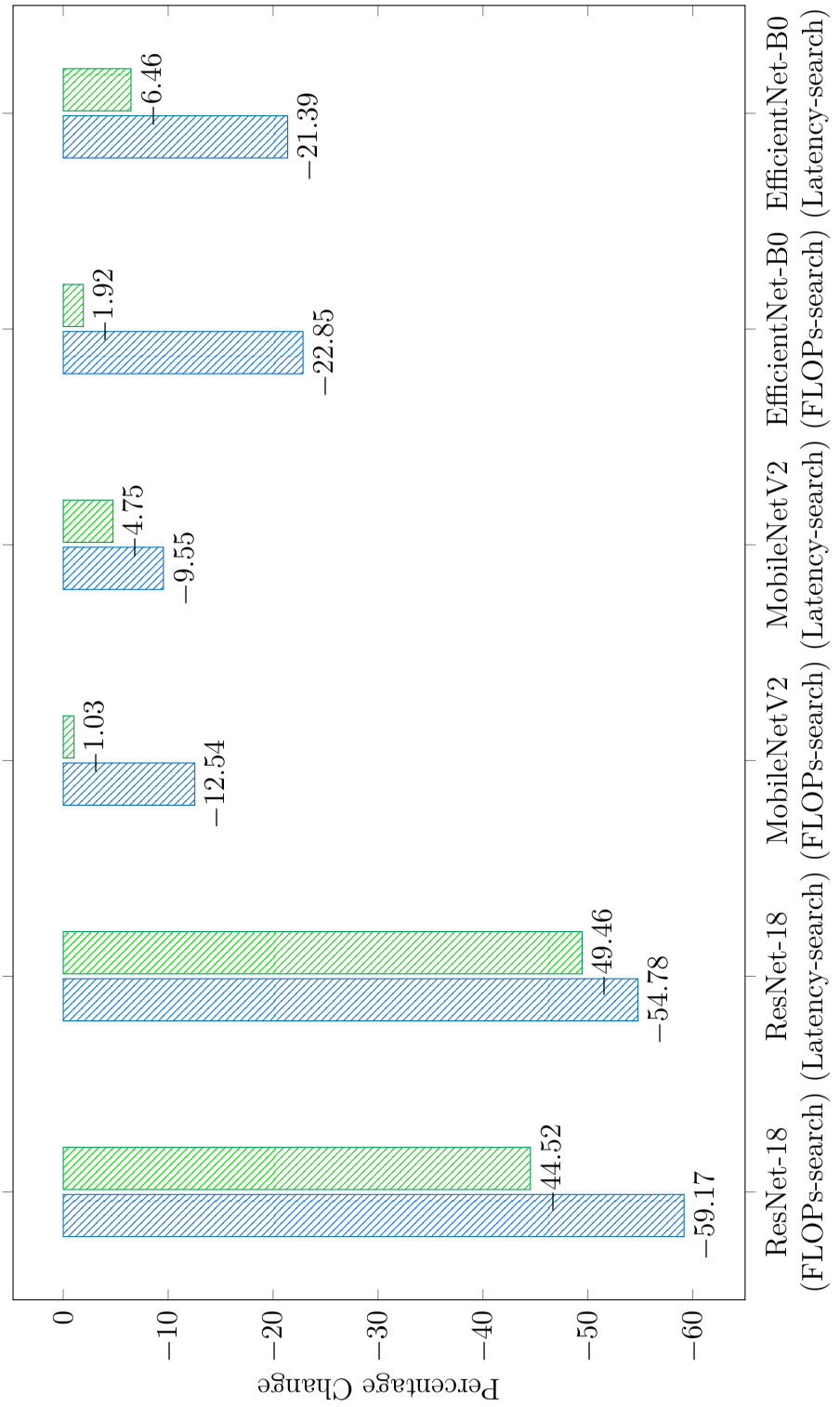


Figure 4.7: Latency-aware search results on Pixel 4, where the accuracy degradation is within 2pp after fine-tuning.

We further visualize the detailed FLOPs and latency reductions for different experimental setups in Figure 4.7, leading to the following observations:

- Upon incorporating the latency metric into the search process, the gap between FLOPs and latency is reduced to just 5% for both ResNet and MobileNet. For instance, in the case of ResNet-18, the latency-aware search yields a decomposed network that achieves a remarkable 54.78% reduction in FLOPs and a substantial 49.46% decrease in latency, compared to the original network.

- In contrast, EfficientNet includes SiLU and squeeze-and-excitation operations [34], which currently lack efficient optimization for CPU, thereby becoming a computational bottleneck. For instance, the SiLU activation function involves the exponential operation, which is not captured in the FLOPs metric. As a result, there is a more significant discrepancy between latency (-21.39%) and FLOPs reductions (-6.46%).

Such latency-aware tensor decomposition can also be extended to other computational platforms, including CPUs, GPUs, ASICs, and FPGAs, as long as a latency lookup table specific to the target device is available.

4.5 Summary

This chapter introduces *LR-space*, an SVD-based Low-Rank model architecture design space that unifies the decomposition schemes proposed in our work and existing research. Therefore, searching the proposed *LR-space* is equivalent to exploring all design choices involved in the tensor decomposition algorithm. To facilitate the automatic exploration of such space, we then incorporated a gradient-descent-based neural architecture search approach which balances the accuracy and compression

ratio in a multi-objective loss function, using back-propagation to learn the optimal decomposition design choice. Results demonstrate that the SVD-NAS achieves 2.06-12.85pp higher post-training accuracy on ImageNet than other state-of-the-art model compression methods, at a similar compression ratio.

So far, we have focused on SVD-based decomposition schemes, which require the transformation of 4-D convolution weight tensors into 2-D before rank reduction. An alternative approach is to explore Higher-Order Decomposition (HOD) methods, such as Canonical Polyadic Decomposition (CPD), which can be applied directly to 4-D tensors. This will be discussed in the next chapter.

5

Mixed-TD: Efficient Neural Network Accelerator with Layer-Specific Tensor Decomposition

Contents

So far, we have focused on tensor decomposition methods that are SVD-based only. In this chapter, we incorporate additional methods, particularly Canonical Polyadic Decomposition (CPD), demonstrating how a mixed usage of these techniques can enhance performance. We then co-explore this extensive tensor decomposition design space along with hardware customization, investigating techniques that can lead to a fast and efficient co-exploration. This chapter is based on two conference papers: SAMO [175] published in FPL 2022 and Mixed-TD [176] published in FPL 2023.

5.1 Introduction

In the case of a pre-trained deep neural network, the data distribution and error tolerance often vary across different parts of the network [51]. As such, it is necessary to make the compression method fine-grained and layer-specific to avoid significant accuracy degradation. For example, in previous work on weights pruning, unstructured ways of pruning bring a larger compression ratio with negligible accuracy degradation compared with the channel-wise structured pruning [177], [178]. Similarly, in weights quantization, mixed precision and block floating point methods are favored compared with uniform bitwidth and range [135], [179], [180].

In this chapter, we extend our exploration of fine-grained, layer-specific tensor decomposition techniques as a complementary dimension for model compression alongside pruning and quantization. This time, we incorporate the mixing of SVD with other alternative decomposition methods to enhance flexibility and performance. However, when integrating these advanced tensor decomposition techniques with layer-pipelined, dataflow accelerators, new challenges arise. Historically, exploring accelerator configurations has heavily relied on manual design space exploration, a

process that is both time-consuming and prone to yielding sub-optimal results [131], [181]. The joint design space of decomposition strategies and accelerator configurations is now even larger, rendering the two-iteration hardware-awareness approach presented in Chapter 3 insufficient for meeting this increased complexity.

The specific contributions of this chapter are as follows:

- Building on the work presented in Chapter 4 which focused solely on SVD, we further propose Mixed-TD, a novel approach that for the first time mixes Singular Value Decomposition (SVD) and Canonical Polyadic Decomposition (CPD) with per-layer customization for improved performance and flexibility.

- To address the time-consuming DSE process, we introduce a greedy optimizer aimed at efficiently exploring accelerator configurations while adhering to resource constraints. Additionally, to further expedite the co-search of decomposition decisions and accelerator configurations, we also employ an evolutionary search algorithm complemented by a random-forest-based throughput predictor. These combined techniques achieve over a 10× reduction in search time.

- Through the proposed fine-grained Mixed-TD compression and the efficient co-search, our performance evaluation shows that the proposed approach can achieve low latency, high throughput, and negligible accuracy loss at the same time. In terms of "Throughput per DSP", our approach demonstrates substantial gains, ranging from 1.73× to 10.29× compared to the results presented in previous chapters and existing literature.

5.2 Fine-grained Mixed-TD

5.2.1 Algorithm

The proposed Mixed-TD method is an extension of the *LR-space* which was previously introduced in Section 4.2.1. Let's still consider a pre-trained CNN, where the i^{th} convolutional layer processes c_i input channels using $f_i = c_{i+1}$ filters with a kernel size of $k_{a,i} \times k_{b,i}$.

LR-space was characterized by the channel grouping parameters $g_{j,i}$, the kernel parameters $k_{aj,i}, k_{bj,i}$ where $j \in \{0, 1\}$ and also the decomposition rank r_i. It is important to note that the range of j is limited to binary values in the *LR-space* due to its exclusive utilization of SVD. This condition ensures that the 4-D convolution weight tensor is transformed into a 2-D representation so that SVD can be applied for weight derivation. The Mixed-TD method builds upon this foundation, extending the range of j to $\{0, 1, 2, 3\}$ and incorporating even more decomposition schemes.

The expanded set of the decomposition design parameters, which is denoted as $\{g_{j,i}, k_{aj,i}, k_{bj,i}, r_i\}, j \in \{0, 1, 2, 3\}$ in the proposed Mixed-TD method, continues to be layer-specific. This design choice adheres to similar constraints outlined in Section 4.2.1, ensuring that the low-rank tensors can be derived from the original weight tensors in a post-training manner.

Specifically, the constraints dictate that $\prod_j k_{aj,i} = k_{a,i}$ and $\prod_j k_{bj,i} = k_{b,i}$, ensuring that the product of kernel parameters aligns with the original convolutional layer's kernel size. Additionally, the constraint specifies that at least one element of $g_{j,i}$ must be equal to one to preserve the connectivity between channels and filters. Moreover, specifically for CPD, $g_{1,i}$ and $g_{2,i}$ will be at least equal to or greater than the decomposition rank r_i. This requirement arises because each rank component

corresponds to a unique outer product, as described in Equation (2.3), and these outer products will be implemented as grouped convolutions [72].

In contrast to the previously introduced *LR-space*, Mixed-TD demonstrates the capability to decompose the 4-D weights of each original convolutional layer into a maximum of four tensors, adopting the format known as Canonical Polyadic Decomposition (CPD) in the existing literature [79]. It is noteworthy that, in instances where any decomposed tensor contains only one value, a fusion can be performed with others. Consequently, the CPD format may collapse into configurations involving three or two decomposed tensors. In the former case, this collapse corresponds to Tucker Decomposition [17], while in the latter case, it reverts to the SVD-based *LR-space*.

It is also important to note that, besides CPD, other decomposition methods, such as Tensor Train [182] and Tensor Ring [81], are also capable of decomposing 4-D convolutional weights into four tensors. However, these methods require derivation through retraining from scratch, so they are not considered in our work.

Therefore, the contribution of Mixed-TD is unifying the choice of SVD, Tucker and CPD of convolution weights under one design space. The main difference between SVD, Tucker and CPD is the number of tensors left after decomposition as well as their representation abilities. Empirically, we found that only SVD and CPD are actually favored to be mixed together on modern CNN benchmarks, and we will focus on the discussion of these two formats in the rest of the chapter.

In Section 4.2.2, we have already introduced the weight derivation for the SVD-based *LR-space*. In terms of CPD, the derivation of weights can be achieved using the Alternating Least Squares (ALS) method [72]. We show such an example in Algorithm 2, where we assume the design parameters $g_{j,i} = 1$. In the case of $g_{j,i} > 1$, we will slice the original 4-D weight tensor into chunks and apply ALS independently to each chunk.

Algorithm 2 Alternating Least Squares for Canonical Polyadic Decomposition

Require: $\mathcal{W}_i$ ▷ 4-D weights of i^{th} conv layer

1: $\mathcal{W}'_i \in \mathbb{R}^{c_i \times k_{a,i} \times k_{b,i} \times f_i} \Leftarrow \mathcal{W}_i \in \mathbb{R}^{f_i \times c_i \times k_{a,i} \times k_{b,i}}$ ▷ transpose

2: $\mathcal{T}_i(0) \in \mathbb{R}^{c_i \times (k_{a,i} k_{b,i} f_i)} \Leftarrow \mathcal{W}'_i$ ▷ $\mathcal{T}_i(j)$, keep j^{th} dim of $\mathcal{W}'_i$ while flatten others

3: $\mathcal{A}'_{0,i} \in \mathbb{R}^{c_i \times r_i}$, $\mathcal{A}'_{1,i} \in \mathbb{R}^{k_{a,i} \times r_i}$, $\mathcal{A}'_{2,i} \in \mathbb{R}^{k_{b,i} \times r_i}$, $\mathcal{A}'_{3,i} \in \mathbb{R}^{f_i \times r_i}$ ▷ random initialize

4: **while** not converged **do**

5: **for** $j = 0$ to 3 **do**

6: $\mathcal{T}_i(j) \Rightarrow U_{ri} \Sigma_{ri} V_{ri}$ ▷ use SVD

7: $\mathcal{A}'_{j,i} = U_{ri} \times sqrt(\Sigma_{ri})$

8: $\mathcal{T}_i(j+1) \Leftarrow U_{ri} \Sigma_{ri} V_{ri}$

9: $\mathcal{T}_i(0) = \mathcal{T}_i(4)$

10: **return** $\mathcal{W}_{0,i} \in \mathbb{R}^{r_i \times c_i \times 1 \times 1} \Leftarrow \mathcal{A}'_{0,i}$, $\mathcal{W}_{1,i} \in \mathbb{R}^{1 \times r_i \times k_{a,i} \times 1} \Leftarrow \mathcal{A}'_{1,i}$, $\mathcal{W}_{2,i} \in \mathbb{R}^{1 \times r_i \times 1 \times k_{b,i}} \Leftarrow \mathcal{A}'_{2,i}$, $\mathcal{W}_{3,i} \in \mathbb{R}^{f_i \times r_i \times 1 \times 1} \Leftarrow \mathcal{A}'_{3,i}$ ▷ reshape back to 4-d

The process begins by reshaping the original 4-D weight tensor $\mathcal{W}_i$ into 2-D form $\mathcal{T}_i(0)$ by flattening all but one dimension at a time (line 2). Then four 2-D low-rank tensors $\mathcal{A}'_{j,i}$ are defined and random initialized (line 3). The ALS algorithm proceeds iteratively, optimizing each 2-D low-rank tensor one at a time. At each iteration, SVD is applied to the current reshaped tensor $\mathcal{T}_i(j)$ (line 6), and the 2-D low-rank tensor is updated based on the resulting decomposed matrices (line 7). This process continues for each dimension until convergence is reached, at which point the 2-D low-rank tensors are reshaped back into their 4-D forms (line 10).

5.2.2 Hardware Design

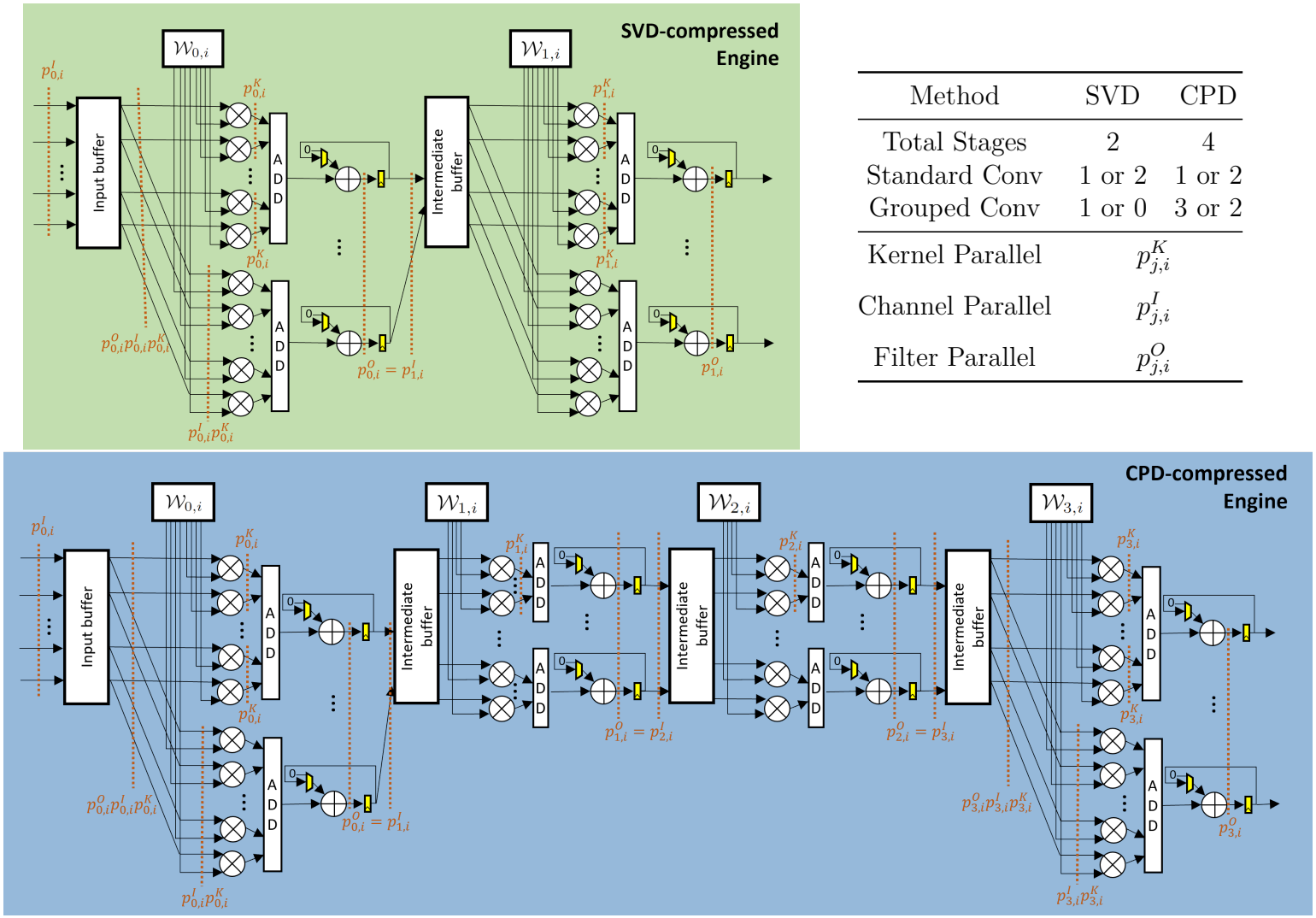
In terms of the accelerator design, we extend StreamSVD (Chapter 3) to further support any CPD-compressed computation and maintain the layer-pipelined architecture. Figure 5.1 highlights the differences for SVD-compressed and CPD-compressed computation.

In the case of SVD decomposition, the number of layers is effectively doubled, as each original convolution weight tensor is decomposed into **two** low-rank tensors. Conversely, when CPD is applied, there are **four** new convolutional layers per engine

instead. Each of these layers is implemented with a data buffer, multiplier arrays, adder trees, and accumulation units. In addition, for each new layer, tunable design parameters $p^I_{j,i}, p^O_{j,i}, p^K_{j,i}$ are available to control the parallelism across input channel, output filter, and kernel dimensions, where $j \in \{0, 1, 2, 3\}$ for CPD, and $\{0, 1\}$ for SVD.

Apart from the number of layers, another difference between SVD-compressed and CPD-compressed engines is the number of grouped convolutions each contains, which significantly affects the data broadcast pattern of the input buffers. In the case of SVD, the number of grouped convolutions could be between 0 and 1, while for CPD, that number will be between 2 and 3 depending on the choices of $g_{j,i}$ (Section 5.2.1).

Figure 5.1 shows an example where the SVD-compressed engine contains 0 grouped convolutions, while the CPD-compressed engine includes 2. As a result, we can observe, for the middle two layers of the CPD-compressed engine, the activation data are sent only to a subset of multiplier arrays due to the partitioning of filters into distinct groups, representing the occurrence of grouped convolutions. In contrast, for all other layers, the input activation data are broadcast to all multiplier arrays simultaneously as a standard convolution.



Method	SVD	CPD
Total Stages	2	4
Standard Conv	1 or 2	1 or 2
Grouped Conv	1 or 0	3 or 2
Kernel Parallel	$p^K_{j,i}$	
Channel Parallel	$p^I_{j,i}$	
Filter Parallel	$p^O_{j,i}$	

Figure 5.1: Internal architecture of the SVD-compressed and CPD-compressed Engine. Both architectures implement the convolution on the decomposed weight tensors, but they differ in the number of stages of computation as well as the dataflow between stages. N While the CPD-compressed engine features more pipeline stages, this does not necessarily imply a larger circuit area, as the area also depends on the level of parallelism specified for each stage.

5.3 Design Space Exploration Process of Dataflow Accelerator

5.3.1 Problem Formulation

Figure 5.2 demonstrates the end-to-end toolflow, showing how an AI model written in PyTorch is transformed into a layer-pipelined, dataflow accelerator design, ultimately producing a bitstream that can be deployed on an FPGA.

Regardless of whether the AI model is decomposed or not, during such transformation, one common problem is the identification of the optimal hardware configuration, which is also known as the Design Space Exploration (DSE) process [183]. Specifically, the DSE process can be formalized using the following terms:

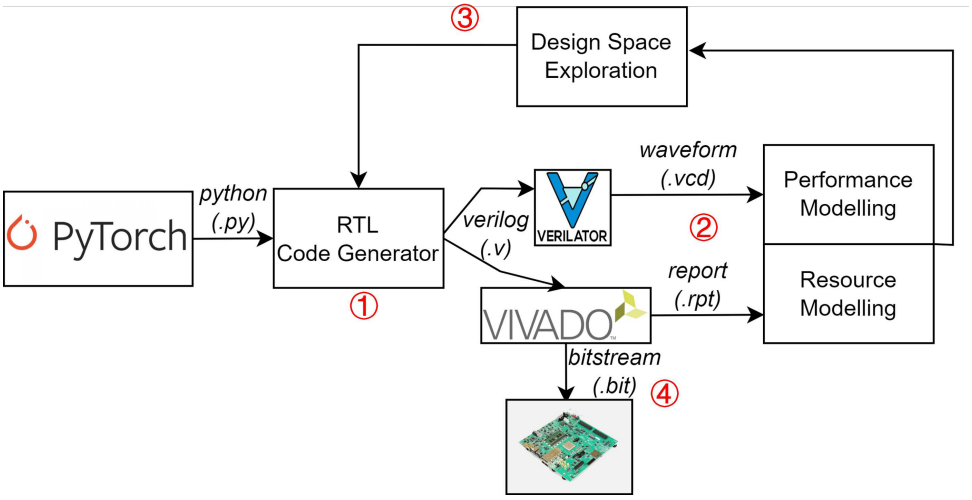


Figure 5.2: The end-to-end toolflow from PyTorch to bitstream includes DSE to identify the optimal accelerator design using resource and performance models (analytical or regression-based). These models are built by sampling a few design points and running them through simulation or synthesis. Once created, the models allow for fast performance estimation of other design points, avoiding the need for repeated simulation and synthesis, and significantly speeding up the DSE process.

- $\mathcal{L} : \{l_0, l_1, l_2...\}$ denotes all the layers in a given network, including convolution, pooling, activation, elementwise operations, and etc.

- $\mathcal{P} : \{p_0, p_1, p_2...\}$ denotes the feasible parallelism dimensions for each layer.
- $\mathcal{D} : \{d_0, d_1, d_2...\} \leftarrow \mathcal{L} \times \mathcal{P}$ represents the accelerator design space.
- $\mathcal{O}$ represents the objective of DSE. In this context, we assume a minimization objective, which could be latency or negative throughput, for example.
- $\mathcal{A}$ represents the area of a given design point.

As such, the goal is to

$$\min_{d \in D} \mathcal{O}(d), \quad s.t.\ \mathcal{A}(d) \leq \mathcal{A}_{max} \tag{5.1}$$

where $\mathcal{A}_{max}$ denotes the area upper bound of the design. The accelerator design space is often huge, especially when it grows exponentially with the number of layers in the network, making it infeasible to synthesize every design point and compare its on-device performance. In fact, solving this optimization problem heavily relies on the estimations from resource and performance models, as Figure 5.2 shows.

5.3.2 Optimization Algorithms

The optimization problem described in Equation (5.1) is generally non-convex. This is because the design space $\mathcal{D}$, which arises from the combination of layers and parallelism dimensions, is often highly discrete and non-linear, especially when considering constraints like resource allocation and performance metrics (e.g., latency or throughput).

To address this non-convex optimization problem, the following three types of algorithms are explored, leveraging the trade-off between the optimization evaluation time and the optimality of the solution.

Algorithm 3 Simulated Annealing Optimization Algorithm

```
1: K = K_0, d = d_0                                   ▷ initialization
2: while K > K_min do
3:     d' ← d                                         ▷ apply random change
4:     if A(d') ≤ A_max then                          ▷ resource constraint
5:         if ψ(d', d, K) then
6:             d = d'                                 ▷ accept new design
7:     K = K * λ                                      ▷ reduce temperature
```

- **brute-force**: This method enumerates all possible design points, discarding any that violate resource constraints. The remaining points are evaluated based on the objective function, and the optimal design is then identified. The primary advantage of this approach is that it guarantees finding the optimal solution. However, even with resource and performance models to bypass the need for a full synthesis of each design, the brute-force method can still lead to prohibitively long execution times. For instance, when using fpgaConvNet [19] to implement a VGG-style network with 11 layers, the design space consists of approximately 2×10^{42} points. On a modern single-threaded CPU, it would take around 5×10^{30} centuries to complete this enumeration, making the brute-force approach impractical.

- **simulated annealing**: it is a well-known stochastic optimization algorithm [184] , and its implementation is outlined in Algorithm 3. In the initialization phase, the algorithm identifies the resource-minimal design point d_0 by setting the computation inside each layer to follow a sequential order. It then enters the main optimization loop where it performs a random change to the current design point d in each iteration and evaluates the following function to decide whether to accept or abort the new design point d':

$$\psi(d', d, K) = exp\left(min\left(0, \frac{\mathcal{O}(d) - \mathcal{O}(d')}{K}\right)\right) > x \sim U(0, 1) \tag{5.2}$$

  K is referred to as the annealing temperature, a hyperparameter that decays

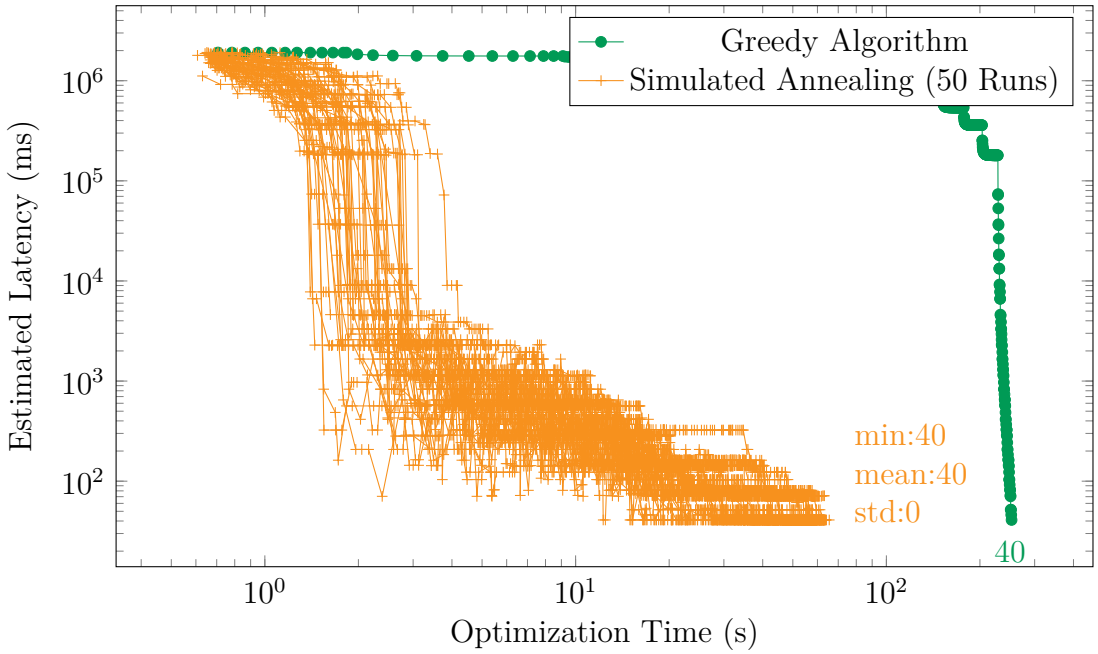
with the process. x denotes the stochastic decision threshold which is a random variable sampled from a uniform distribution. Due to this stochastic approach, a new design point might be accepted even when the objective temporarily worsens, facilitating exploration to escape local optima.

- **greedy**: As illustrated in Algorithm 4, the optimizer selects the layer whose parallelism offers the smallest increase in resource usage but the largest improvement in the objective function (latency or throughput). This selection is based on the $\frac{\Delta\mathcal{A}}{\Delta\mathcal{O}}$ metric. The process is repeated iteratively until either a resource constraint is reached or the slowest layer in the pipeline is fully unrolled. While the deterministic nature of the greedy algorithm makes it computationally efficient, it may not explore the full design space as thoroughly as other optimization methods.

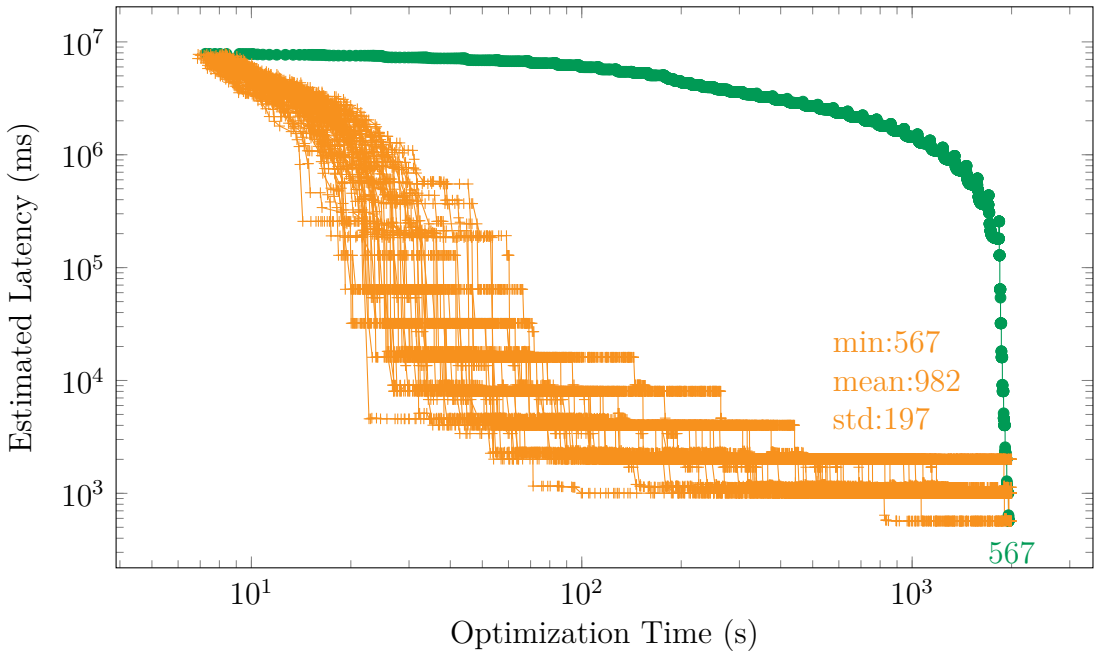
Algorithm 4 Greedy Optimization Algorithm

```
d = d_0                                              ▷ initialization
while A(d) ≤ A_max do
    i ← argmin(ΔA/ΔO), p_i ∈ P                       ▷ pick smallest resource impact
    p_i ↑                                            ▷ increment parallelism
```

In Figure 5.3, we compare simulated annealing and greedy algorithms, focusing on both optimization time and the minimal latency found through DSE. Since simulated annealing is stochastic, we run it 50 times with different random seeds. For the CNV network [20], which has 11 layers, all 50 runs of simulated annealing reach the same latency as the greedy algorithm and converge quickly. However, with MobileNetV1, a deeper network with 29 layers, the average latency from simulated annealing is almost twice as high as that of the greedy approach. This means multiple runs of simulated annealing are necessary, and since each run takes about 30 minutes, the greedy algorithm becomes the more efficient choice. Therefore, for the remainder of this thesis, we opt for the greedy algorithm due to its faster, more stable performance, especially for larger networks.



(a) CNV [20]



(b) MobileNetV1 [33]

Figure 5.3: Comparison between the Simulated Annealing and Greedy optimization algorithm. Networks are mapped to an AMD U250 device with a latency minimization objective. The final optimization results are annotated in the graphs, where the greedy algorithm yields one deterministic solution but simulated annealing may produce a distribution of results depending on different random seeds.

5.4 Model-Accelerator Co-search

The previous section demonstrates the DSE process for the dataflow accelerator configuration. In this section, we will focus on integrating this process with fine-grained Mixed-TD, aiming to co-optimize both the model decomposition setup and the dataflow accelerator configuration. Note that in Chapter 4, we had a gradient-descent search algorithm for exploring model decomposition decisions when targeting CPU devices. However, since we are no longer targeting fixed hardware in this chapter, the performance metrics between multiple candidates can no longer be linearly combined (as a weighted sum), as assumed in Section 4.3.1. Therefore, we adapted the evolutionary algorithm [185] for searching the model decomposition decisions instead.

5.4.1 Evolutionary Search

As Algorithm 5 demonstrates, initially, we randomly apply the decomposition design parameters $\{g_{j,i}, k_{aj,i}, k_{bj,i}, r_i\}, j \in \{0, 1, 2, 3\}$ (defined in Section 5.2.1). For each generated compressed model, we use the greedy algorithm to identify its optimal dataflow accelerator design, when its estimated throughput is also logged. The obtained N design points form the first "generation" of the "population" (line 1). For each generation, the algorithm uses mutation (line 7) and crossover (line 11) to sample new design points, which are called offsprings. The design points are then ranked together based on their accuracy and estimated throughput. During the ranking, the user can provide a throughput target and design points fail to achieve that will be removed (line 4 and 5). The remaining design points are sorted by the accuracy metric, where the top-N samples are retained to become the next generation. This ensures the continuation of high-performing design points throughout the evolution process.

Algorithm 5 Evolutionary Search

Require: Population size N, Offspring size M, Number of generations G
Initialize population $\mathcal{D}_{pop}$ by random sampling from design space
for $i \leftarrow 1$ to G **do**
    Generate offspring $\mathcal{D}_{off}$ by crossover and mutation
    Query accuracy and throughput for $d \in \mathcal{D}_{pop} \cup \mathcal{D}_{off}$
    Sort d, and keep the top-N as the new generation of the population
return Best $d \in \mathcal{D}_{pop}$

procedure MUTATION(sample d)
    for $l \leftarrow \mathcal{L}_d$ **do**
        randomly change the decomposition at location l
    return new sample d'

procedure CROSSOVER(samples d_0, d_1)
    for $l_{d0}, l_{d1} \leftarrow \mathcal{L}_{d0}, \mathcal{L}_{d1}$ **do**
        randomly exchange the decomposition of l_{d0}, l_{d1}
    return new samples d'_0, d'_1

5.4.2 Performance Predictor

The search speed of the evolutionary algorithm is bottlenecked by the time spent on evaluating the accuracy and throughput of each design point.

For accuracy evaluation, we use the decomposed network without fine-tuning and evaluate it on a single batch of validation data, rather than the entire ImageNet validation dataset. This significantly reduces the time required for accuracy queries, reducing the duration from minutes to just seconds on a desktop GPU.

To expedite throughput evaluation, we previously introduced performance modeling and a greedy optimization algorithm (Section 5.3.2) to facilitate fast DSE. However, even with these enhancements, the DSE process can still take minutes or hours to run for networks with 10 to 50 layers. Additionally, this process must be repeated whenever there are changes in the decomposition decisions.

To further address this challenge, we introduce another level of performance estimation, by building a proxy that predicts the achievable throughput for a specific

configuration based on network characteristics and the optimizer used. For this purpose, a random forest is selected as the proxy model.

Our approach is as follows: In the first few generations of the evolutionary search, we use the greedy solver from Section 5.3.2 to estimate the optimal throughput of each design. The results are saved and used to build a dataset. At a predetermined point in the search, we use this dataset to train a random forest regressor, which is then used to predict the throughput for subsequent designs during the evolutionary search, without further running the greedy solver. Training the regressor takes only a few minutes on a desktop CPU, and its inference time is less than a second. As a result, the evolutionary search can iterate more quickly, allowing for a broader exploration of decomposition decisions.

Our work differs from previous study [186], which aimed to build performance predictors for general-purpose computing architectures, such as CPUs, GPUs, or systolic array accelerators, where the architecture is fixed and only the workload changes during the search. In contrast, our work targets dataflow architectures on the FPGA, where the hardware is changed and customized for each design point.

5.5 Experiments

Our experiments are conducted on a server equipped with an Intel Xeon Bronze 3104 processor, which runs the search algorithm, along with an Nvidia GTX 1080 Ti GPU for accuracy queries and final model fine-tuning. The hardware accelerator obtained from the design process is evaluated using Vivado 2020.1, targeting an AMD Alveo U250 device.

The system flow, shown in Figure 5.4, consists of two main stages: *search* and *deployment*. In the *search* stage, we use CPUs and GPUs to estimate the accuracy

and throughput of each design point and explore the design space to identify the optimal configuration. This design is then fine-tuned to improve accuracy before being synthesized and deployed on the FPGA. The time spent on each step is also highlighted in Figure 5.4.

5.5.1 Benchmarks

As a case study, we focused on the ImageNet dataset [1] and evaluated two state-of-the-art models, ResNet-18 [7] and RepVGG-A0 [36]. ResNet-18 features representative residual block designs, while RepVGG-A0 is the latest model from the VGG family. Models were quantized to 8-bit Block Floating Point (BFP) format [61] for both activations and weights. Table 5.1 summarizes the results of our investigation. Our proposed method successfully produced a model with a significantly reduced number of parameters while maintaining an accuracy loss of less than 0.4pp compared to a model that uses 8-bit BFP only.

Table 5.1: Compression results on two CNN benchmarks trained on ImageNet dataset. Models have been fine-tuned after the compression. BitOPs are counted as two times of the number of MACs multiplied by the product of weight and activation bitwidth.

	Precision	Float32	BFP-W8A8	BFP-W8A8
	Decomposed	✗	✗	✓
ResNet-18	Top-1 Accuracy (%)	69.7	69.3	69.1
	Memory Size (Mb)	374	94	35
	BitOPs (G)	3715	232	97
	Throughput (FPS)	N/A	702	1041
RepVGG-A0	Top-1 Accuracy (%)	72.4	71.9	71.5
	Memory Size (Mb)	266	67	35
	BitOPs (G)	2789	174	87
	Throughput (FPS)	N/A	864	1162

We then compare our results against state-of-the-art work, as well as our own implementation from the previous chapter. While direct comparisons may not always

be feasible due to differences in models or devices used, it is useful to position our work against the state-of-the-art on the ImageNet task within the space of achieved throughput and accuracy.

The results are shown in Table 5.2, where resources of Intel devices have already been converted to the equivalent ones on AMD devices, as 1 Intel ALM = 1.8 AMD LUT [187], and 1 Intel DSP = 2 AMD DSP [188]. We observe our Mixed-TD approach outperforms non-dataflow accelerator designs [137], [177], [189] significantly, with respect to both peak FPS, and latency (equal to the inverse of FPS at batch size 1). Since dataflow accelerators typically target larger FPGA devices, we also normalize performance by resources for a fair comparison. Specifically, in terms of FPS/DSP, our approach outperforms non-dataflow accelerator designs between 1.73× and 10.29×.

With respect to dataflow accelerator designs, we compared our Mixed-TD approach to the previously proposed StreamSVD (Chapter 3), as both methods leverage tensor decomposition. Our work achieves a 3.02× improvement in peak FPS/DSP compared to StreamSVD, due to the combination of SVD and CPD, along with more efficient co-optimization of the decomposition decision and accelerator configuration.

In addition, we considered two other state-of-the-art dataflow accelerator designs: HPIPE [188] and FCMP [22]. While these approaches focus on weight sparsity and binary quantization, respectively, they do not employ tensor decomposition. In terms of batch size 1 performance, our design outperforms HPIPE, but we fall short in peak performance because their design operates at a much higher clock frequency (580 MHz vs. our 200 MHz). FCMP, due to binarization, uses significantly fewer DSPs than our design, but their classification accuracy on ResNet is 1.8 percentage points lower than ours.

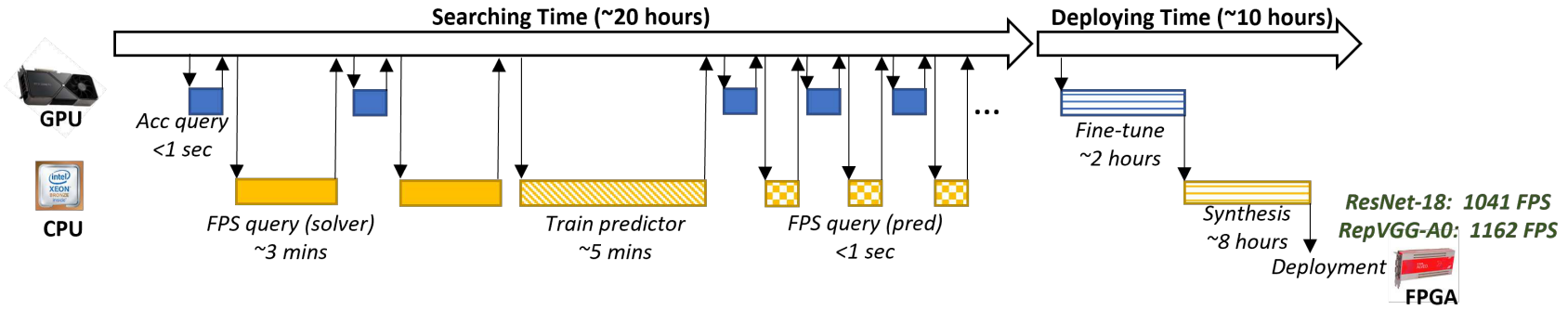


Figure 5.4: The system flow includes the *search* and the *deployment* stages. During the *search* stage, we query the accuracy and the throughput of each design point and perform the design space exploration to identify the optimal design. The optimal design is then fine-tuned to improve its accuracy before being synthesized and deployed on the FPGA device.

Table 5.2: Performance and resource comparison with existing work. "Q", "S" and "TD" stands for quantization, sparsity (weights pruning) and tensor decomposition respectively. Regarding performance metrics, both FPS for batch size 1, and peak FPS for a large batch size (we and [148] use 256, [22] uses 1024, and others did not provide the details) are reported. As pipeline is emptied between batches, the pipeline depth impacts differently those two FPS metrics.

	Dataflow	Network	Year	Method	Acc.	Device	Freq. (MHz)	URAM	BRAM	kLUT	DSP	FPS *batch 1*	FPS *peak*
MCBBS [177]	✗	2021	VGG16	Q, S	64.8	Arria GX1150	242	-	1785	605	2704	54	-
N3H-Core [189]	✗	2022	ResNet-18	Q	70.4	XC7Z045	100	-	541	153	900	31	123
FILM-QNN [137]	✗	2022	ResNet-18	Q	70.5	ZCU102	150	-	881	180	2092	-	215
HPIPE [188]	✓	2020	ResNet-50	Q, S	71.9	S10 2800	580	-	11278	1064	10044	909	4550
FCMP [22]	✓	2020	ResNet-50	Q	67.3	Alveo U250	195	109	3870	1027	1611	526	2703
StreamSVD [148]	✓(partial)	2021	ResNet-18	Q, TD	68.4	XC7Z045	125	-	752	-	576	-	34
Ours	✓	2023	ResNet-18	Q, TD	69.1	Alveo U250	200	0	3564	1550	6394	1041	1138
Ours	✓	2023	RepVGG-A0	Q, TD	71.5	Alveo U250	200	0	3550	1555	5652	1162	1288

Overall, the results highlight the significant advantages of our Mixed-TD approach over non-dataflow accelerator designs. Additionally, we extend the performance frontier established by StreamSVD, demonstrating that our mixed tensor decomposition strategy (SVD and CPD) can lead to highly efficient dataflow designs. When compared to sparsity and quantization-based approaches like HPIPE and FCMP, our method offers competitive performance with similar task accuracy. This reinforces the complementary role tensor decomposition can play in achieving high-performance, resource-efficient accelerator designs.

5.5.2 Benefits on Mixing SVD and CPD

Compared to previous chapters and literature, one main innovation in this chapter is the introduction of mixing SVD and CPD. To understand the specific benefits of such a mixture, we conduct an ablation study in Figure 5.5.

The results of three methods are compared, which are SVD-only, CPD-only, and the proposed Mixed-TD. At the beginning of the search, Mixed-TD does not show a clear advantage due to the larger design space that needs to be explored. However, the benefits of Mixed-TD become evident after the 90-hour mark.

It is important to note that in this part of the ablation study, the random-forest-based performance predictor is deliberately disabled. This decision ensures that the evaluation focuses exclusively on the benefit of combining SVD and CPD in the Mixed-TD approach, without being influenced or interfered with by the predictor's guidance. This isolation allows for a clearer assessment of how the mixing strategy itself contributes to the observed performance improvements.

In fact, from Figure 5.5, we observe that Mixed-TD demonstrates the highest accuracy compared to SVD-only and CPD-only approaches, with the advantage of up to 4.83pp. This finding confirms the benefits of mixing SVD and CPD on a

per-layer basis.

5.5.3 Benefits of Performance Predictor

In this section, we conduct another ablation study, to understand how the random-forest-based performance predictor helps speed up the search process.

When disabling the random-forest-based performance predictor and using the greedy solver alone, the evolutionary search algorithm can explore only 22 designs per hour, resulting in a convergence time of over 150 hours. In contrast, with the help of the random-forest-based predictor, the search algorithm can explore 4,969 designs per hour, achieving convergence in less than 20 hours. Although the predictor slightly compromises on the design quality by identifying a design with 1,041 FPS, which is 3.5% below the target of 1,079 FPS, this trade-off is minimal compared to the significant reduction in search time. Moreover, training the random forest takes only a few minutes, which is negligible compared to the overall time savings during the search process.

We further examine the possibility of using the number of Multiply-Accumulate (MAC) operations to estimate the system throughput, and we compare this baseline with our random-forest-based predictor in Figure 5.6. Although the MAC-based approach is even 10 times more faster, it leads to designs with significantly lower throughput than the target. In contrast, our performance predictor converges to a design point that better aligns with the desired throughput.

From Figure 5.6, we can also observe that the initial steps of the predictor's execution are slower compared to later stages. This slowdown occurs because the greedy solver is launched to gather training data for the random-forest-based predictor. Additionally, while the predictor exhibits some fluctuations during its operation, it ultimately converges to a design point that is only 3.5% lower than the

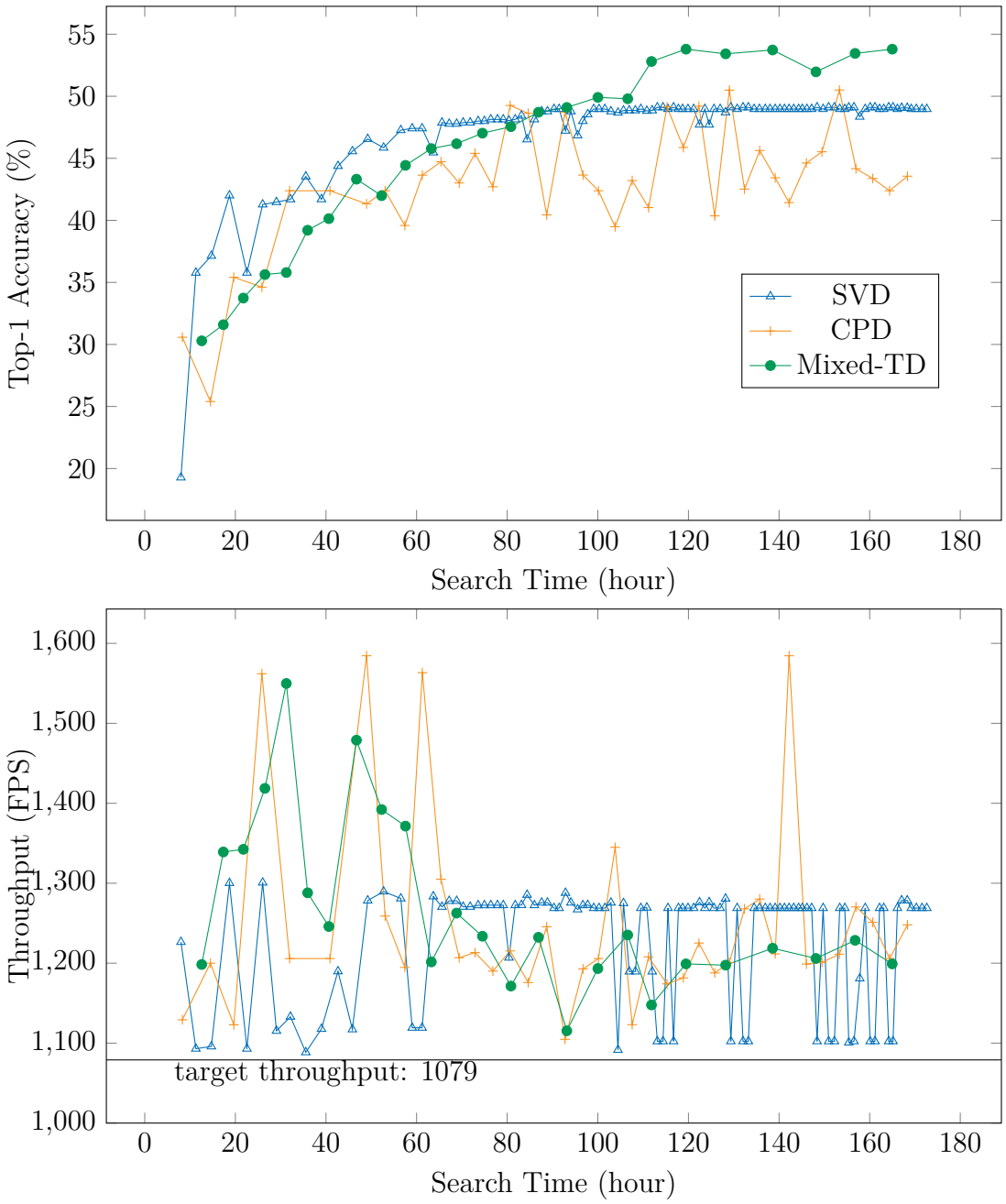


Figure 5.5: Compare tensor decomposition methods on ResNet-18. The search is without the involvement of the performance predictor. Curves do not start from zero points as initialization is required to obtain the first generation of design points.

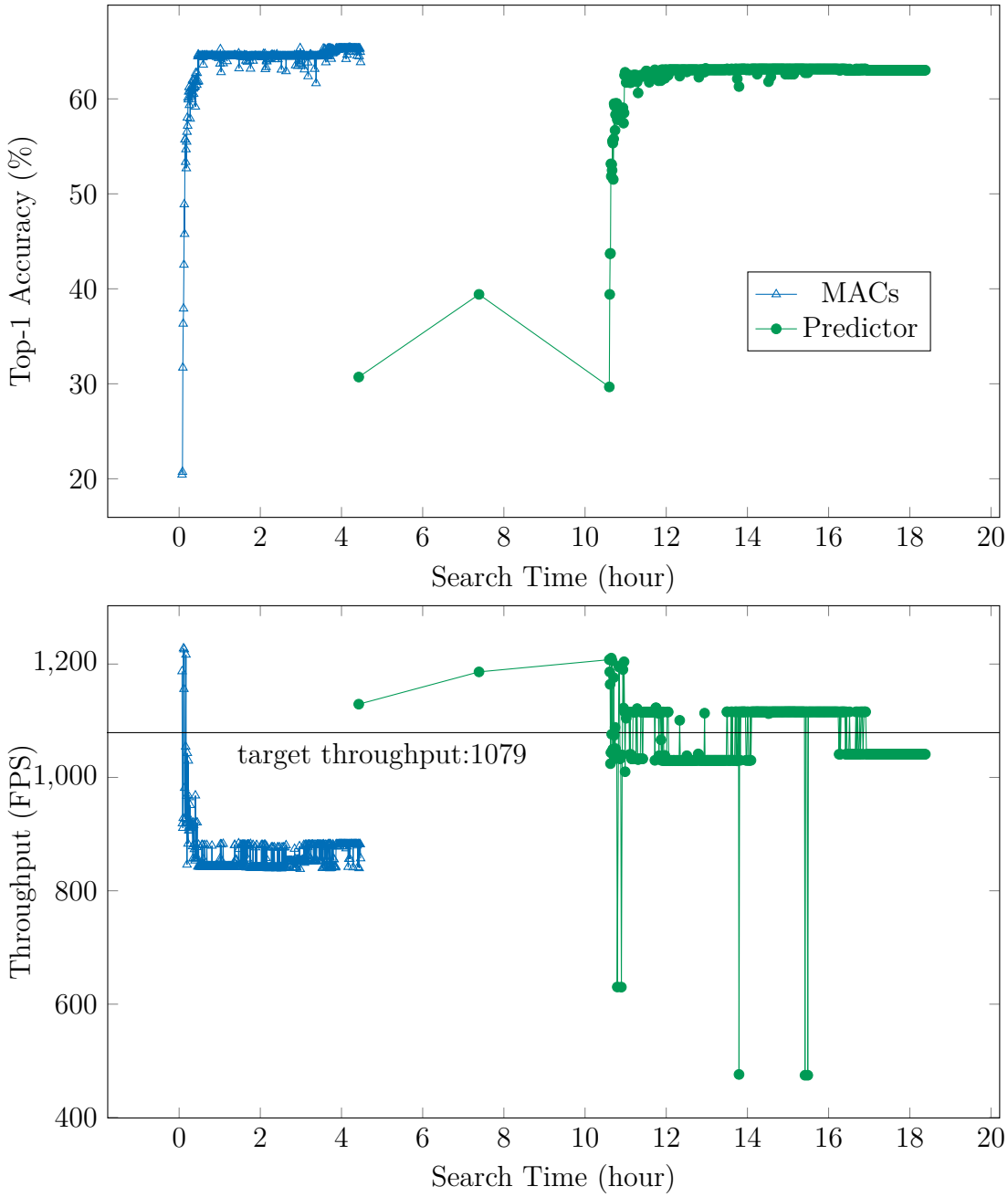


Figure 5.6: Compare searching results on ResNet-18, Mixed-TD, when using the predictor to estimate throughput v.s. using MACs for estimation. The first few steps of the predictor run are slow as the greedy solver is launched to gather training data. Our predictor converges to a design point that better meets the target.

target throughput. This convergence is possible because the evolutionary algorithm iterates through generations, each containing multiple design points, allowing it to refine its search despite the presence of some standalone mispredicted outliers.

5.6 Summary

The chapter presents a novel method, called Mixed-TD, for fine-grained and layer-specific model compression by applying the layer-specific mixture of SVD and CPD. Our result demonstrate that Mixed-TD achieves substantial weight compression while preserving high accuracy and considering the target hardware. To efficiently navigate the extended design space, we introduced an evolutionary search with a random-forest-based throughput predictor. This chapter demonstrates the benefits of tensor decomposition methods in the space of mapping CNN models onto FPGAs, as well as the need for proxies in order to navigate quickly the large design space. In terms of the efficiency metric "Throughput per DSP", our approach demonstrates substantial gains, ranging from 1.73× to 10.29× compared to existing FPGA accelerators.

6

AutoWS: Automate Weights Streaming in Layer-wise Pipelined DNN Accelerators

Contents

In previous chapters, we demonstrated the capability of tensor decomposition to compress AI models, leading to reductions in both computation and memory

requirements. This chapter approaches the problem from a different angle, focusing on hardware-only solutions. Specifically, we address the limited on-chip memory resources of FPGAs and introduce a methodology for the smart management of both on-chip and off-chip memory. This approach is particularly useful in problem settings where model compression cannot be performed, such as when maintaining a faithful implementation of the model is critical for accuracy. In such cases, hardware-based optimization offers a valuable alternative or can be used in conjunction with the model compression approaches discussed in earlier chapters. This chapter is based on the conference paper AutoWS [190] published in DATE 2024.

6.1 Introduction

Efficient management and utilization of memory resources are always crucial for enhancing overall performance in hardware accelerators, regardless of the underlying architecture.

For the layer-sequential architecture [113]–[115], both weight and activation data are stored off-chip, leading to intensive off-chip memory access. Techniques such as tiling and double buffering are commonly employed to mitigate data communication bottlenecks by maximizing data reuse and hiding latency.

On the other hand, prior research on the layer-pipelined accelerator [21], [22], [191], restricts off-chip memory access to the two endpoints of the computation pipeline, with intermediate activation data streamed on-chip. Furthermore, all model parameters (weights) are preloaded into on-chip memory before inference begins. However, the substantial demand for on-chip memory impedes the deployment of large-scale networks on resource-constrained devices. Figure 6.1 demonstrates the memory structures of prior layer-sequential and layer-pipelined architectures. In this chapter, we will refer to prior work on the layer-pipelined architecture as the “vanilla

layer-pipelined architecture" to distinguish it from our approach.

Let's consider the AMD Alveo U250, a data-center acceleration card. Despite its high performance, this card has a limited internal SRAM capacity of only 54MB. ResNet-50, a popular neural network architecture with 23 million parameters, requires a storage capacity of approximately 92MB in IEEE Single-precision floating-point format, which exceeds the SRAM capacity. Apart from the memory capacity, the on-chip memory bandwidth also becomes a limitation when the architecture requires access to a large number of the parameters concurrently to support parallel computation for enhanced performance.

To overcome these limitations, this chapter introduces a novel memory management methodology for layer-pipelined architectures that exploits both on-chip and off-chip memory for weight storage, as Figure 6.1 shows. Our approach introduces memory fragmentation and we address the challenges of deciding the memory fragmentation parameters and bandwidth allocation when dealing with multiple pipelined Compute Engines (CE), offering an automated solution to these critical issues. Furthermore, we bundle the above memory subsystem with a parameterised layer-pipelined architecture resulting in the first such accelerator that supports partial weight streaming.

Our contributions can be summarized as follows:

- A scalable CE template featuring tunable unroll factors, coupled with a flexible memory structure supporting static and dynamic weights loading, tunable in a per-layer basis.
- A Design Space Exploration (DSE) process, which balances the processing rates of CEs and optimizes the allocation of off-chip bandwidth in an iterative and greedy way, based on our performance and resources modeling.

- A deterministic DMA scheduler that links off-chip memory to multiple CEs in a time-multiplexed manner, allowing the efficient exploitation of off-chip bandwidth with two clock domains.

- Overall, the proposed methodology enables the deployment of larger-scale AI models on more resource-constrained FPGA devices in a layer-pipelined manner, which was not possible in prior works. In terms of performance, our solution reduces latency compared to state-of-the-art methods by 25 to 48%.

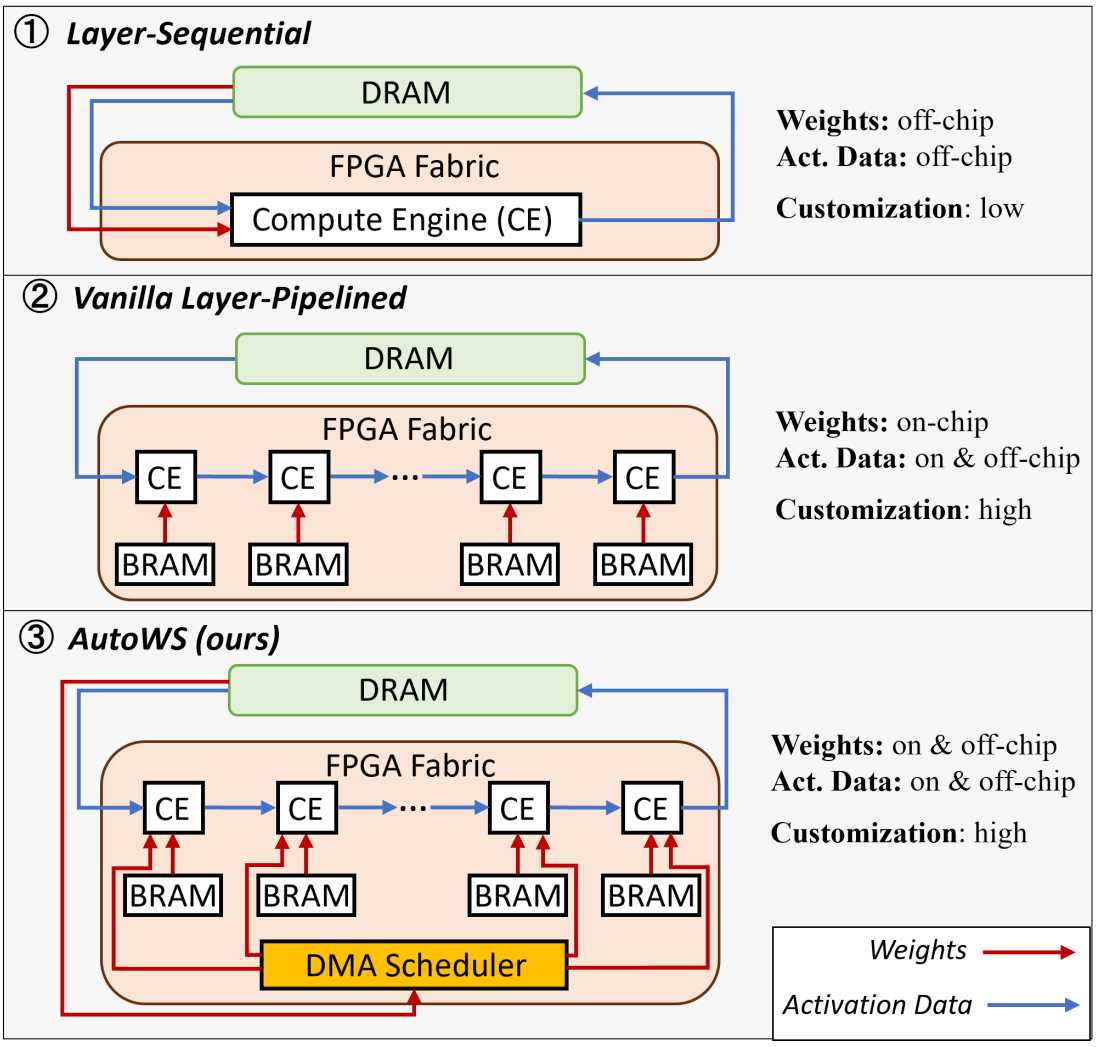


Figure 6.1: Comparison of prior layer-sequential designs [13], [137], [192], vanilla layer-pipelined designs [19], [20], [130], and our proposed architecture, emphasizing differences in the memory structure for weight and activation storage.

6.2 Compute Engines

In this section, we begin by introducing the internal structure of the Compute Engine (CE) in a vanilla layer-pipelined architecture, covering its computational dataflow and memory organization. These aspects are customizable for each layer to optimize performance. We then provide an in-depth analysis of how on-chip memory utilization affects the performance of the CE, highlighting our motivation. Following this, we introduce the core technique of our AutoWS framework, which employs both on-chip and off-chip memory for efficient weight storage through a novel fragmentation technique. Finally, we adapt resource and performance modeling to illustrate the impact of this technique.

6.2.1 Dataflow and Parallelization

Inside each Compute Engine (CE), the dataflow involves the following interconnected components, facilitated with handshake interfaces:

- **Input buffer**: used for sampling or rearranging the incoming data from the previous layer. For example, for convolutional and pooling layers, line buffers are used to implement sliding windows.
- **Weight memory**: stores the weights of convolutional and fully connected layers. In a vanilla layer-pipelined architecture, all the weights are stored statically in on-chip memory.
- **Processing array**: an array of parallel processing elements that handle elementwise operations such as multiplication, addition, and ReLU activations. In cases where weights memory is not involved, this array may consume multiple activation data streams.

- **Output buffer**: used for rearranging the output data and also performing cross-element operations when required. In a convolutional layer, for example, the outputs are accumulated across the 2-D kernel window and channel dimensions.

Each CE is responsible for executing the computations of a single layer within the target network. Consequently, all symbols defined in this section are specific to that layer, and we will omit layer indices for simplicity. Additionally, we will use the terms "layer" and "CE" interchangeably, as they refer to the same computational unit in this context. Within this CE or layer, on-chip memory is required to implement the input and output buffers, as well as the weight memory, which we will consider separately.

For the input and output buffers, their buffer width can be expressed as the product of the processing array's parallelism p and the activation bitwidth l_a. Furthermore, d_a refers to the depth of input and output buffers, which is selected as the minimum number of activations required to prevent pipeline stalls or backpressure, according to the previous literature[22]. Typically, this number involves only a subset of the layer's activations, due to the streaming nature of architecture. As such, the on-chip memory usage of the input and output buffer can be represented as $\mathcal{M}(p \cdot l_a, d_a)$. Here, $\mathcal{M}$ is defined as an FPGA vendor-specific function that can calculate the number of on-chip primitives required, taking both width and depth as inputs.

Similarly, for the weight memory, the memory width can also be expressed as the product of parallelism and bitwidth, which is $p \cdot l_\omega$, representing the number of bits of weights required per cycle to prevent the processing array from being idle. If we assume the total number of weight parameters for this layer is denoted by e_ω, then the depth of weight memory can be represented as d_ω where $p \cdot d_\omega = e_\omega$. Therefore, the on-chip memory usage of the weight memory can be represented as

$\mathcal{M}(p \cdot l_\omega, d_\omega)$. As we are going to demonstrate in the experiment section (Table 6.2), in a layer-pipelined architecture, the weight memory often becomes the bottleneck in on-chip memory usage. For example, in a ResNet-18 design, the BRAM usage for weight memory can be 20× greater than that of the input and output buffers. Consequently, the optimization of weight memory will be the primary focus of the remainder of this chapter.

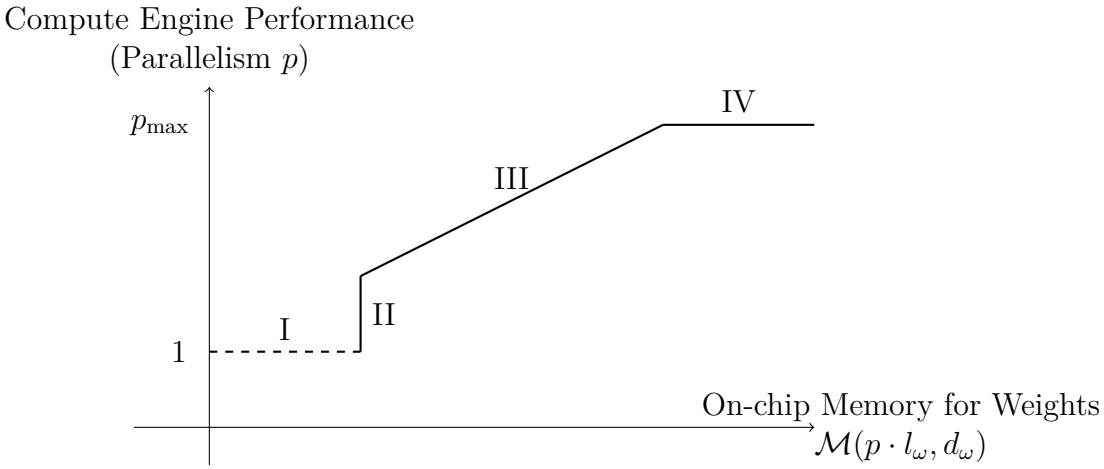


Figure 6.2: The piecewise relationship between Compute Engine (CE) performance (parallelism) and weight memory usage, divided into four distinct segments, annotated as I to IV.

To understand the relationship between the CE performance (parallelism) and the weight memory usage, we demonstrate it as a piecewise function in Figure 6.2, which can be divided into four different segments (I to IV). When parallelism $p = 1$, it indicates the minimum on-chip memory requirement for weight storage. Therefore, **Segment I**, represented by the horizontal dashed line, is considered invalid, as it is not feasible to reach below this minimum requirement in a layer-pipelined architecture. **Segment II** and **III** reflect the scenarios when the CE's parallelism p is increasing. As a result, the shape of the weight memory is effectively changed, with its width $p \cdot l_\omega$ becoming larger and the depth $d_\omega = e_\omega/p$ decreasing. For **Segment II**, the number of on-chip memory primitives required is actually unchanged, due to most FPGA vendors providing on-chip Block RAM (BRAM) that has fixed capac-

ity but configurable depth and width. However, this configurability comes with a limit. Once the maximum width is reached, the weight memory must be split across multiple BRAMs, even though it can result in each BRAM being underutilized in terms of capacity. In **Segment III**, the relationship is represented as a linear line, reflecting the increasing number of BRAMs as parallelism rises. Finally, **Segment IV** is characterized by a horizontal line, indicating that parallelism has reached its maximum value (p_{max}). At this point, further increases in on-chip memory do not yield any performance benefits.

As previously mentioned in Section 6.1, modern FPGA devices have limited on-chip memory resources, which presents challenges for effectively utilizing layer-pipelined architectures. The achievable parallelism is constrained, especially when targeting relatively large-scale networks on resource-constrained devices. Figure 6.7 later will demonstrate this trend using an AMD U250 device, providing a concrete example.

6.2.2 Weight Fragmentation

One main novelty of the proposed AutoWS is in the introduction of weight fragmentation while preserving the layer-pipelined architecture. In this scheme, the original weight memory structure is divided into the **Static** and **Dynamic** regions along the depth dimension d_ω. Specifically, weights belonging to the **Static** region are stored permanently on-chip, consisting of n fragments, each with a depth of u_{on}. On the other hand, weights in the **Dynamic** region are stored in the off-chip memory, also consisting of n fragments, but each with a depth of u_{off} instead. The n fragments in the **Dynamic** region share the same on-chip physical memory structure in a time-multiplexed manner, ensuring that only one fragment is active on-chip at any given time. The relationships among the symbols are expressed as follows:

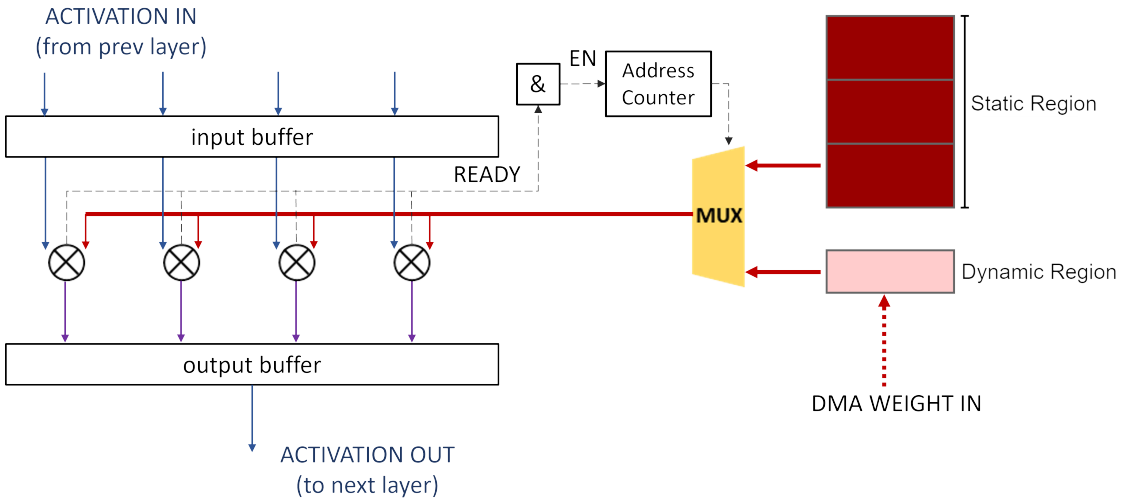


Figure 6.3: Dataflow inside each Computation Engine (CE), with weights storage fragmented into the static region which stays permanently on-chip, and the dynamic region which is stored off-chip and only loaded upon requirement.

$$d_{on} = u_{on}n, \quad d_{off} = u_{off}n, \quad d_{\omega} = d_{on} + d_{off} \tag{6.1}$$

As such, by exploiting the above fragmentation technique, the saving of the on-chip weight memory can be expressed as:

$$\Delta\mathcal{M}_{\omega} = \mathcal{M}(p \cdot l_{\omega}, d_{\omega}) - \mathcal{M}(p \cdot l_{\omega}, d_{on}) + \mathcal{M}(p \cdot l_{\omega}, u_{off}) \tag{6.2}$$

The above equation contains three terms, which correspond to the original on-chip weight memory usage in a vanilla design, the remaining on-chip usage for the **Static** region only, and the overhead of the on-chip buffer required to store one fragment of the **Dynamic** region. Moreover, in our AutoWS, we select the value of u_{off} in order to minimize this overhead, by making $\mathcal{M}(p \cdot l_{\omega}, u_{off}) = 1$.

Figure 6.3 demonstrates the implementation of weight fragmentation at the CE level. On the left side, the architecture consists of a pipeline that includes the input buffer, a processing array (represented as a multiplier array), and an output buffer.

On the right side, the weight storage is divided into two distinct regions: **Static** (shown in dark red) and **Dynamic** (shown in light red). A **MUX** (multiplexer) is responsible for selecting between the weights from the **Static** and **Dynamic** regions based on the control logic. This selection is controlled by an **Address Counter**, which is driven by an **EN** signal that enables the counter. Each element from the processing array generates a **READY** signal, and these **READY** signals are combined (ANDed) to produce that **EN** signal.

Let's consider an example, where the current CE is implementing a convolutional layer. In this case, b denotes the batch size, $\hat{h}$ and $\hat{w}$ represent the spatial feature map height and width out of the layer. During run-time, the processing array first requests and reads a fragment from the **Static** region with a depth of u_{on}, which takes u_{on} cycles, assuming no stalls in the processing array. Simultaneously, during this time, a fragment from the **Dynamic** region, with a depth of u_{off}, is being loaded from off-chip memory into the on-chip buffer. Once u_{on} cycles have elapsed, the **Address Counter** controls the **MUX** to switch the input of the processing array to read from the **Dynamic** region instead, which takes another u_{off} cycle.

Since there are n fragments in both **Static** and **Dynamic** regions, and each fragment as the weight of the convolutional layer, is reused across the batch size b as well as the spatial dimensions of the feature map $\hat{h}$ and $\hat{w}$. Therefore, the above process that switches between **Static** and **Dynamic** occurs r times in total, which is defined as

$$r = b \cdot \hat{h} \cdot \hat{w} \cdot n \tag{6.3}$$

Importantly, the on-chip buffer for the **Dynamic** region is implemented as dual-port memory, one for reading and the other for writing, allowing it to continue loading data from off-chip memory while simultaneously providing data to the pro-

cessing array. As a result, the total time budget for loading a fragment of the **Dynamic** region from off-chip is $u_{on} + u_{off}$ cycles, rather than just u_{on}. Moreover, this dual-port on-chip buffer is operating on two clock domains:

- clk_{comp}: controls the read port of this on-chip buffer, and also the execution of various computations within the CE.
- clk_{dma}: manages the write port of this on-chip buffer, and also the process of loading weights from the off-chip memory.

By taking these into account, the average off-chip memory bandwidth requirement $\overline{\beta_\omega}$ for loading the fragment of **Dynamic** region can be expressed as:

$$\overline{\beta_\omega} = p \cdot l_\omega \cdot \frac{u_{off}}{u_{on}+u_{off}} \cdot clk_{comp} \qquad \text{(bits per second)} \tag{6.4}$$

In the above equation, $p \cdot l_\omega$ and u_{off} represent the width and depth of this fragment, while $u_{on}+u_{off}$ denotes the time budget for loading this fragment. Moreover, the above equation together with Equation 6.2, establishes the trade-off between on-chip memory savings and off-chip memory bandwidth. By reducing on-chip memory usage through weight fragmentation, additional off-chip memory bandwidth is required to load the fragments. This approach exploits the fact that, in prior vanilla layer-pipelined architectures, the off-chip memory bandwidth is often underutilized. By leveraging this unused bandwidth, the system can effectively address the on-chip memory bottleneck.

6.3 Macro-Architecture

After examining the weight fragmentation within each individual computation engine (CE), let us now shift our focus to the macro-architecture of the design. The

weight fragmentation technique introduces additional memory-related design parameters per layer: u_{on}, u_{off} and n, as defined in Equation (6.1). We will demonstrate that the selection of these parameters depends on the DMA scheduling and the bandwidth allocation across all the CEs, both requiring system-level exploration.

In the previous section, we omitted the indices of the symbols for simplicity, even though they are defined layer-wise. However, since we are discussing macro-architecture, we will now clearly denote the layer indices associated with the symbols. For example, let $\mathcal{L} : \{l_0, l_1, l_2, ...\}$ represent all the layers in a given network, and then the per-layer parallelism can be denoted as $\mathcal{P} : \{p_0, p_1, p_2, ...\}$.

6.3.1 DMA Connection and Scheduling

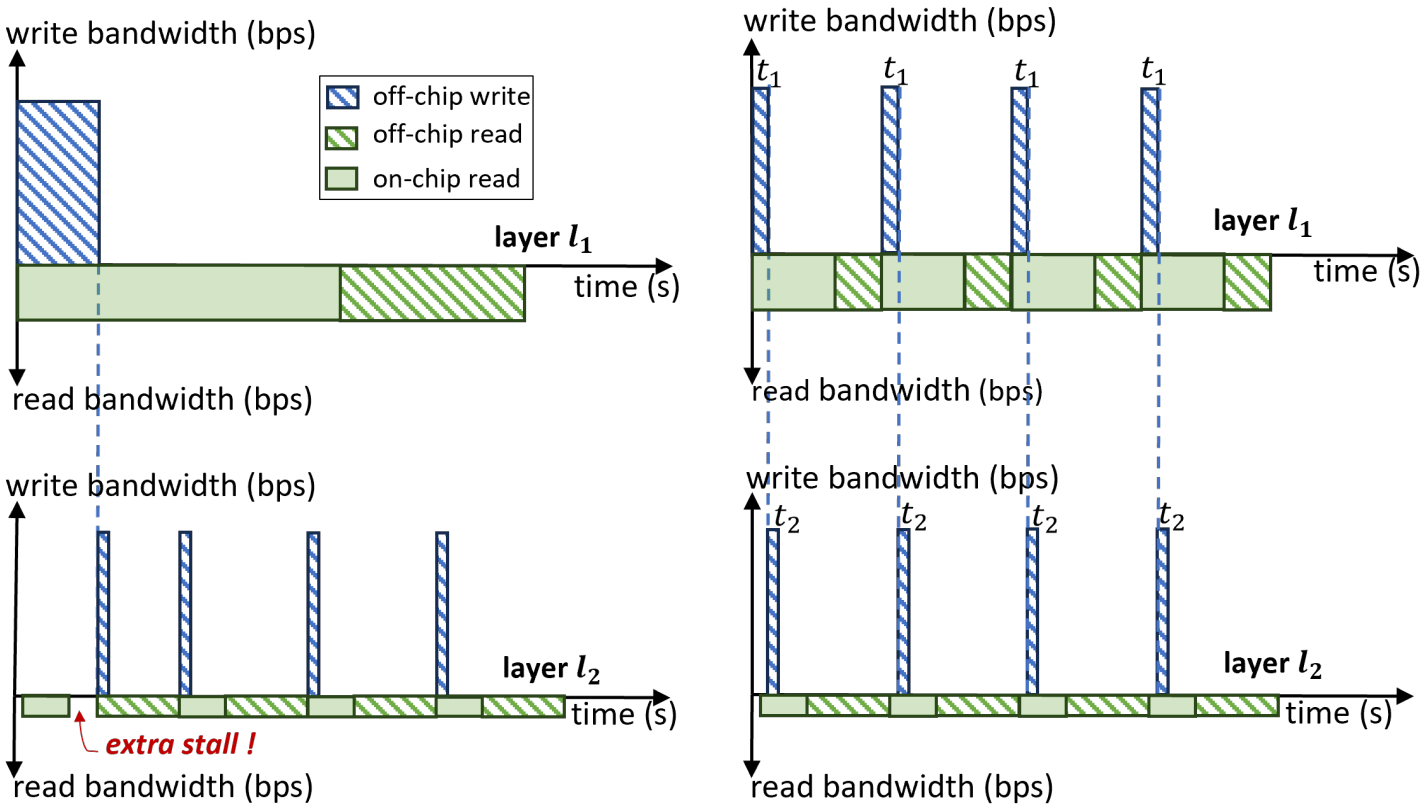
In a layer-pipelined architecture, the number of CEs in the accelerator is equal to the number of layers in the target AI model. After applying weight fragmentation, each CE may require a DMA connection for loading the **Dynamic** region from the off-chip memory. However, as the number of CEs increases, the system may run out of available memory banks or DMA channels to support all CEs simultaneously. Here we consider an extreme case that only a single off-chip memory bank is available, and we will demonstrate a connection scheduling strategy to ensure that each CE can still access the off-chip memory in a timely manner.

Our solution at the microarchitecture level is to employ a **demultiplexer** to manage the routing between the single off-chip memory bank and the multiple DMA ports required by the CEs. At any moment, the off-chip memory bank only serves one CE at a time, fully utilizing the available off-chip memory bandwidth to fill the on-chip buffer for the **Dynamic** region. The **demultiplexer** is controlled by a compile-time generated configuration sequence that outlines the **order** and the **duration** of serving each individual CE. The configuration sequence is generated

at compile time because the workload is static, meaning the dataflow and computational requirements for each CE are predetermined and do not change during runtime.

The **order** of this configuration sequence is determined by the per-layer parameter r (defined in Equation (6.3)), which controls how often we switch between **Static** and **Dynamic** regions in each layer. To illustrate this, let us consider an example involving two consecutive layers/CEs (l_1 and l_2) in the pipeline. Both layers have the same batch size b and feature map size $\hat{h}, \hat{w}$. We will demonstrate two different scenarios, as well as their performance impact, as shown in Figure 6.4.

Scenraio 1: In the **top-left** figure, we present the write and read scheduling for layer l_1, whose **Dynamic** region only contains one fragment ($n_1 = 1$). As such, its entire **Dynamic** region will be loaded in burst mode, represented as a single large rectangle shaded in blue. Moving to the **bottom-left** figure which displays the scheduling of the layer l_2 instead, the assumption is layer l_2 follows l_1 in the computation pipeline so that the computation inside layer l_2 always lags behind layer l_1 due to the pipeline depth between them. The assumption for l_2 is its **Dynamic** region contains four fragments in total ($n_2 = 4$), represented as four slim rectangles shaded in blue. Given that we already assume that batch size b and feature map size $\hat{h}, \hat{w}$ are equal across layers, the repeat parameter r_2 is four times r_1 in this scenario, according to Equation (6.3). As a result, an undesired pipeline stall is introduced, when the DMA is busy with writing the substantial weight chunk for l_1, prohibiting the computation inside l_2 from moving forward. Moreover, the pipeline stall will occur every time l_1 loads weights from off-chip.



(a) Imbalanced DMA Scheduling

(b) Balanced DMA Scheduling

Figure 6.4: Two-layer example of write/read scheduling. The number of burst writes (represented by blue-shaded rectangles) is determined by the per-layer repeat parameter r (as defined in Equation (6.3)). The height and width of each blue-shaded rectangle can be represented as $\beta_{max} - \beta_a$ and t_i respectively (as defined in Equation (6.6)). In (a), an undesired pipeline stall occurs because the DMA is occupied with writing the substantial weight chunk for l_1, preventing the computation in l_2 from progressing. Such a pipeline stall will repeatedly occur for every pixel in the feature map, significantly impacting performance.

Scenraio 2: In contrast, looking at the **top right** and the **bottom right** of Figure 6.4, where n_1 is set to be equal to n_2 (both having four fragments), the repeat parameters also become equal as well ($r_1 = r_2$). This adjustment introduces a balanced order of serving computational elements (CEs), resulting in a smoother data flow throughout the architecture and diminishing the pipeline stall in the previous scenario. Therefore, we enforce this balancing by introducing the following constraint to the per-layer repeat parameters:

$$\forall i, j \in [1, N], \qquad r_i = r_j \tag{6.5}$$

where N denotes the number of layers in the target AI model. By applying this constraint in conjunction with Equation (6.3), we can determine the number of fragments n for each layer.

After establishing a balanced **order** of the **demultiplexer** configuration sequence, let's also look into its **duration** of serving each individual CE. Let β_{max} denote the maximum bandwidth of the off-chip memory bank, and β_a represent the off-chip memory bandwidth required for the first CE to read input samples and the last CE to write prediction results. As such, the time duration for serving the i^{th} CE in burst mode (the width of each blue-shaded rectangle in Figure 6.4) can be represented as:

$$t_i = \frac{p_i \cdot l_\omega \cdot u_{off,i}}{\beta_{max} - \beta_a} \tag{6.6}$$

where the numerator represents the size of each off-chip fragment in bits, while the denominator indicates the available off-chip memory bandwidth for transferring weights.

6.3.2 Design Space Exploration

In the previous chapter, specifically in Section 5.3.1, we defined the design space of the layer-pipelined architecture as $\mathcal{D} : \{d_0, d_1, d_2...\} \leftarrow \mathcal{L} \times \mathcal{P}$, where $\mathcal{L} : \{l_0, l_1, l_2, ...\}$ represents all the layers in a given network, and $\mathcal{P} : \{p_0, p_1, p_2, ...\}$ represents the per-layer parallelism.

After introducing weight fragmentation, the design space is now extended to $\mathcal{D}' \leftarrow \mathcal{L} \times \mathcal{P} \times \mathcal{F}$, where $\mathcal{F} : \{f_0, f_1, f_2, ...\}$ represents per-layer fragmentation ratios:

$$f_i = \frac{u_{off,i}}{u_{on,i} + u_{off,i}} \tag{6.7}$$

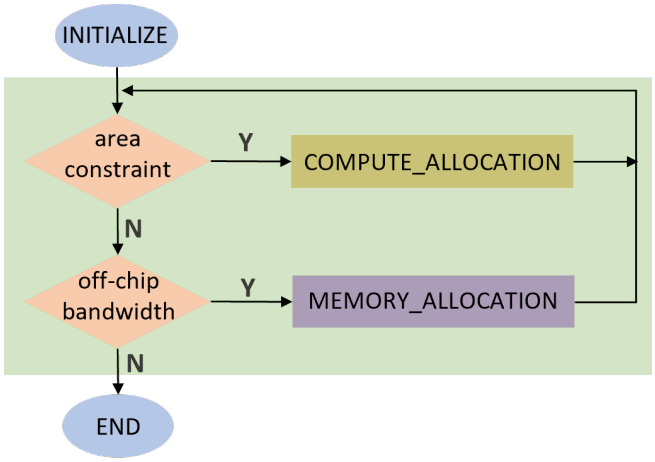
where a larger fragmentation ratio indicates a higher proportion of the weights that are stored off-chip.

It's important to note that in a layer-pipelined architecture, all design choices can be made on a layer-wise basis, resulting in $\mathcal{D}'$ being represented by the Cartesian product of $\mathcal{L}$, $\mathcal{P}$, and $\mathcal{F}$. Thus, the new DSE process can be formulated as follows:

$$\min_{d \in D'} \mathcal{O}(d), \quad s.t.\ \mathcal{A}(d) \leq \mathcal{A}_{max}, \ \ \beta(d) \leq \beta_{max} \tag{6.8}$$

Here, $\mathcal{O}$ represents the optimization goal, which could be minimizing latency or maximizing throughput. In this particular chapter, the focus is on minimizing latency, as we aim to deploy large-scale AI models on resource-constrained FPGA devices using the proposed AutoWS framework. Resource-constrained FPGAs are more aligned with edge applications, where low latency is critical [193].

Moreover, $\mathcal{A}$ represents the area estimation, β denotes the total off-chip memory bandwidth required, including the bandwidth for activations (β_a) and the bandwidth



procedure COMPUTE_ALLOCATION
▷ smallest area impact
$i \leftarrow \text{argmin}(\frac{\Delta\mathcal{A}}{\Delta\mathcal{O}}), p_i \in \mathcal{P}$
$p_i \leftarrow p_i + \phi$ ▷ increase parallelism

procedure MEMORY_ALLOCATION
▷ smallest bandwidth impact
$i \leftarrow \text{argmin}(\frac{\Delta\overline{\beta_\omega}}{\Delta\mathcal{M}_\omega}), f_i \in \mathcal{F}$
$f_i \leftarrow f_i + \mu$ ▷ increase eviction

Figure 6.5: Iterative, greedy optimization of the computation and memory.

sum of per-layer weights ($\overline{\beta_\omega}$, as defined in Equation (6.4)). These two constraints ensure that the chosen design d complies with the maximum area ($\mathcal{A}_{max}$) and bandwidth (β_{max}) limitations imposed by the hardware platform.

To solve this optimization problem, we extend the greedy algorithm, which was introduced in Algorithm 4, and further iteratively optimize the computation and memory with the following two passes, as Figure 6.5 shows:

- **greedy compute allocation**: In this pass of the optimization, we evaluate the ratio of area impact to latency reduction ($\frac{\Delta\mathcal{A}}{\Delta\mathcal{O}}$) for each $p_i \in \mathcal{P}$. This ratio can be estimated using the resource and performance modeling techniques elaborated in the previous chapter, and our goal is to identify the index i that minimizes this ratio. This corresponds to selecting the layer/CE that provides the greatest latency reduction for the least area impact. Once the index i is identified, we incrementally promote the corresponding parallelism p_i by a user-defined step size ϕ. We repeat the process until the required on-chip memory exceeds the limit, when we transition to the next phase.

- **greedy memory allocation**: Initially, the DSE begins with the setup where all weights reside on-chip. In case the required on-chip memory exceeds the limit, we evaluate the ratio of off-chip memory bandwidth impact to on-chip

memory reduction ($\frac{\Delta\overline{\beta_\omega}}{\Delta\mathcal{M}_\omega}$) for each $f_i \in \mathcal{F}$. The variables β_ω and $\mathcal{M}_\omega$ have been defined in Equation (6.4) and (6.2), respectively. The goal is also to identify the index i that minimizes this ratio. Afterward, we increment the corresponding fragmentation ratio f_i by another user-defined step size μ to evict more weights to the off-chip memory. We repeat this process until the remaining on-chip memory usage is within the allowable limit. Once this condition is met, we transition back to computation allocation to further increase parallelism.

In this DSE algorithm, two user-defined hyperparameters, ϕ and μ, control the step sizes of the exploration. The choice of these hyperparameters affects the trade-off between exploration time and solution optimality. A larger step size accelerates exploration, but may lead to sub-optimal solutions.

6.4 Evaluation

6.4.1 Experiment Setup

For evaluation, we target the AMD Xilinx FPGA devices and use Vivado 2019.1 for hardware synthesis. Regarding the DNN models, we considered MobileNetV2 [162], ResNet-18 and ResNet-50 [7], as these are widely used benchmarks in state-of-the-art layer-sequential architecture designs [13], [137], [192]. To ensure a fair comparison, we adopt the same quantization strategies employed by these state-of-the-art designs, quantizing the models to 4-bit or 8-bit precision.

Furthermore, we employ fpgaConvNet [191] to generate "vanilla layer-pipelined" designs where all weights are stored permanently on-chip. We extend fpgaConvNet by incorporating our own weights fragmentation (Figure 6.3), DSE method (Figure 6.5), and DMA scheduling (Figure 6.4) techniques. Note that our design

methodology is fully automated, so it can also be easily integrated into other layer-wise pipelined toolflows, such as FINN [22] and hls4ml [21], in the future.

6.4.2 Overall Results

Figure 6.6 provides a comparative analysis among our AutoWS, the "vanilla layer-pipelined" approach [19], [175], and other "layer-sequential" architectures [13], [137], [192]. We evaluate three different models, and for each model, we also include the results targeting three different devices. For easy illustration, we define the device size relative to model parameters; for example, ZCU102 is "large" for MobileNetV2 but "small" for ResNet-50 due to its larger parameter size.

Our key observations include:

- "Vanilla layer-pipelined" excels on "large" FPGA devices with ample on-chip memory. For example, in the **top-right** corner of Figure 6.6, where MobileNetV2 is deployed on the ZCU102 device, the "vanilla layer-pipelined" design (orange bar) achieves a latency of 2.3 ms, which is less than half the 5.3 ms latency of the "layer-sequential" approach (blue bar). Similar patterns are observed for ResNet-18 on U50 and ResNet-50 on U250.

- Our AutoWS maintains latency on these "large" devices, as our greedy DSE automatically identifies that there is no need to store weights off-chip, and the designs become primarily compute-bound. For example, as shown again in the **top-right** corner of Figure 6.6, our AutoWS achieves the same 2.3ms latency as the "Vanilla layer-pipelined" approach.

- On small to medium-sized FPGA devices, the "vanilla layer-pipelined" approach may underperform compared to the "layer-sequential" method. For example, as shown in the **top-middle** of Figure 6.6, where MobileNetV2 is

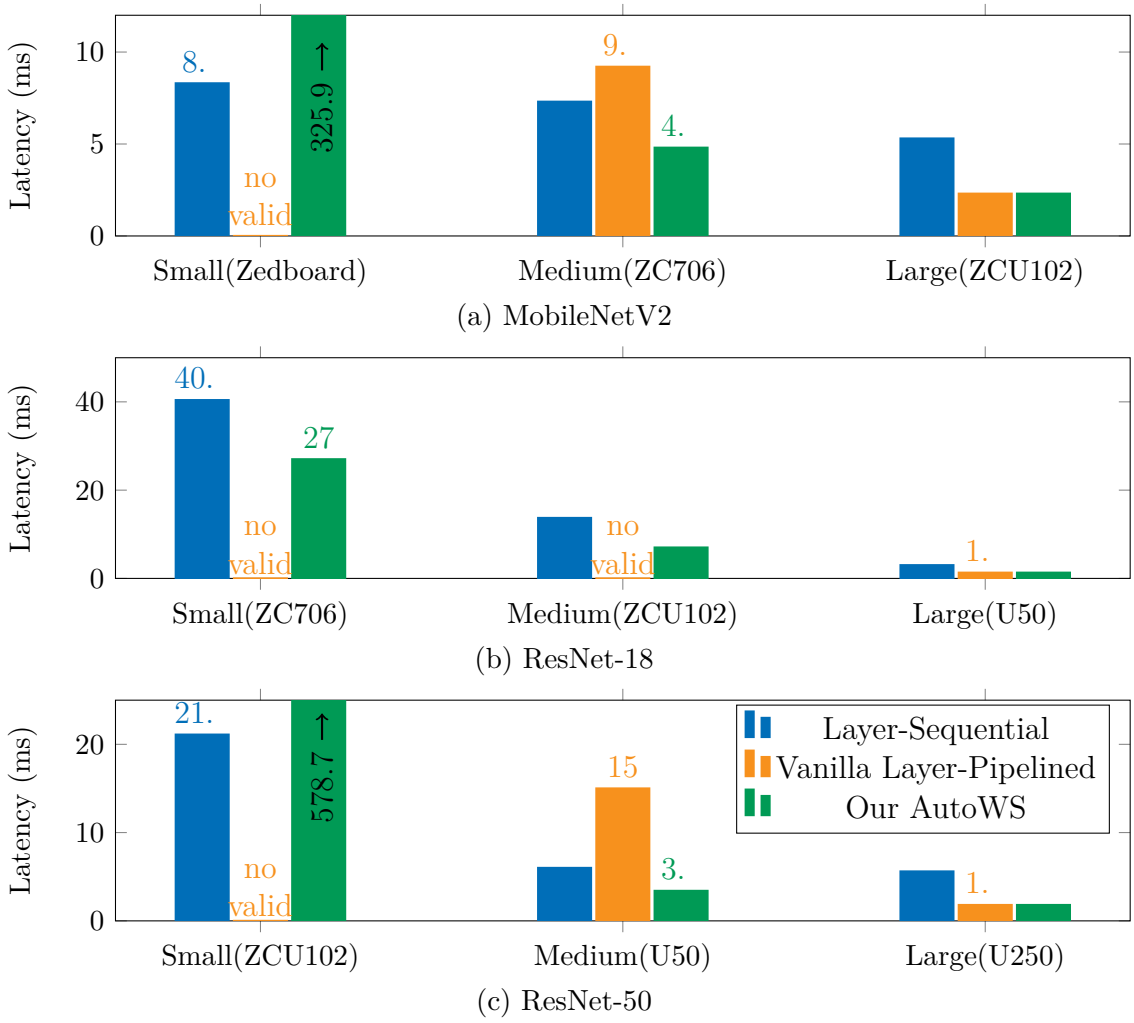


Figure 6.6: Latency comparison across different networks and devices. We define the size of the device (Small, Medium, Large) relative to the model parameters. The vanilla layer-pipelined solution excels on larger FPGA devices. However, on smaller FPGA devices, its performance is hindered due to limited on-chip memory bandwidth, and in some cases, the solution cannot be implemented at all, as indicated by "no valid".

mapped to the ZC706 device, the "vanilla layer-pipelined" design (orange bar) achieves a latency of 9.2 ms, which is worse than the 7.3 ms latency achieved by the "layer-sequential" approach (blue bar). This behavior aligns with **Segment III** in Figure 6.2, where limited on-chip memory resources restrict the achievable parallelism in the "vanilla layer-pipelined" design.

- In some extreme cases, it may not be feasible to implement the "vanilla layer-pipelined" approach on small to medium-sized FPGA devices, as indicated by the "no valid" labels in Figure 6.6. This behavior corresponds to **Segment I** in Figure 6.2, where the minimum on-chip memory required by the "vanilla layer-pipelined" approach already exceeds the device's capacity.

- The advantage of our AutoWS is particularly evident on small to medium-sized FPGA devices. For instance, in the **bottom-middle** of Figure 6.6, where ResNet-50 is deployed on the U50 device, AutoWS reduces the latency from 15.0 ms in the "vanilla layer-pipelined" approach (orange bar) to just 3.4 ms (green bar). Furthermore, this 3.4 ms latency also outperforms the "layer-sequential" approach, which requires 6.0 ms (blue bar).

- However, in some cases, such as MobileNetV2 on Zedboard and ResNet-50 on ZCU102, "layer-sequential" still achieves the lowest latency. This is because the off-chip memory bandwidth on these devices is limited compared to the number of parameters in those models. This bandwidth constraint restricts the full application of our proposed methodology.

In summary, the results in Figure 6.6 demonstrate that "layer-pipelined" approaches have a clear advantage over "layer-sequential" approaches on "large" devices with ample on-chip memory resources. Our work extends this advantage to more resource-constrained devices by considering the access requirements of the weights and leveraging off-chip memory bandwidth.

Table 6.1: Resource utilization of various designs generated by our AutoWS, covering different models and devices.

	MobileNetV2			ResNet-18			ResNet-50		
Quantization	W4A4	W4A4	W4A5	W4A4	W4A5	W8A8	W4A5	W8A8	W8A8
Accuracy(%)	65.6	65.6	65.7	70.3	70.5	70.0	77.3	76.0	76.0
Latency(ms)	325.9	4.8	2.3	27.0	7.0	1.3	578.7	3.4	1.8
Device	ZedBoard	ZC706	ZCU102	ZC706	ZCU102	U50	ZCU102	U50	U250
Frequency(MHz)	200	200	200	200	200	250	200	250	250
LUT	53k (99%)	219k (100%)	273k (100%)	216k (99%)	269k (98%)	704k (81%)	272k (99%)	869k (50%)	1714k (99%)
FF	48k (45%)	213k (49%)	251k (46%)	54k (12%)	97k (18%)	444k (25%)	86k (99%)	599k (34%)	1032 (30%)
DSP	94 (43%)	391 (43%)	1222 (48%)	150 (17%)	1251 (50%)	5817 (98%)	64 (3%)	3807 (64%)	7804 (64%)
BRAM	280 (100%)	1084 (99%)	1428 (78%)	1090 (100%)	1824 (100%)	2490 (93%)	1824 (100%)	2688 (100%)	4025 (75%)
URAM	-	-	-	-	-	576 (93%)	-	640 (100%)	967 (76%)
DRAM(Gbps)	11.8 (35%)	2.9 (3%)	0.3 (0%)	81.6 (80%)	149.6 (97%)	1.2 (0%)	53.8 (35%)	687.6 (41%)	0.8 (0%)

We also present the implementation details of designs generated by our AutoWS in Table 6.1. The results show that AutoWS prioritizes the use of on-chip BRAM for weight storage before utilizing off-chip DRAM for weight transfer. This continues until the design becomes constrained by other resources, such as LUTs or DSPs. For example, in the case of MobileNetV2, off-chip DRAM usage remains consistently under 35%.

6.4.3 Case Study

In this section, we further provide a case study that focuses on mapping ResNet-18 to ZCU102, offering more detailed insights into our implementation. Firstly, we conducted a parameter sweep, as depicted in Figure 6.7, systematically adjusting the budget of on-chip memory (BRAM utilization), while keeping the budgets of compute resource (LUT, DSP) and off-chip bandwidth fixed. All resource numbers are normalized to the specifications of a single ZCU102 device.

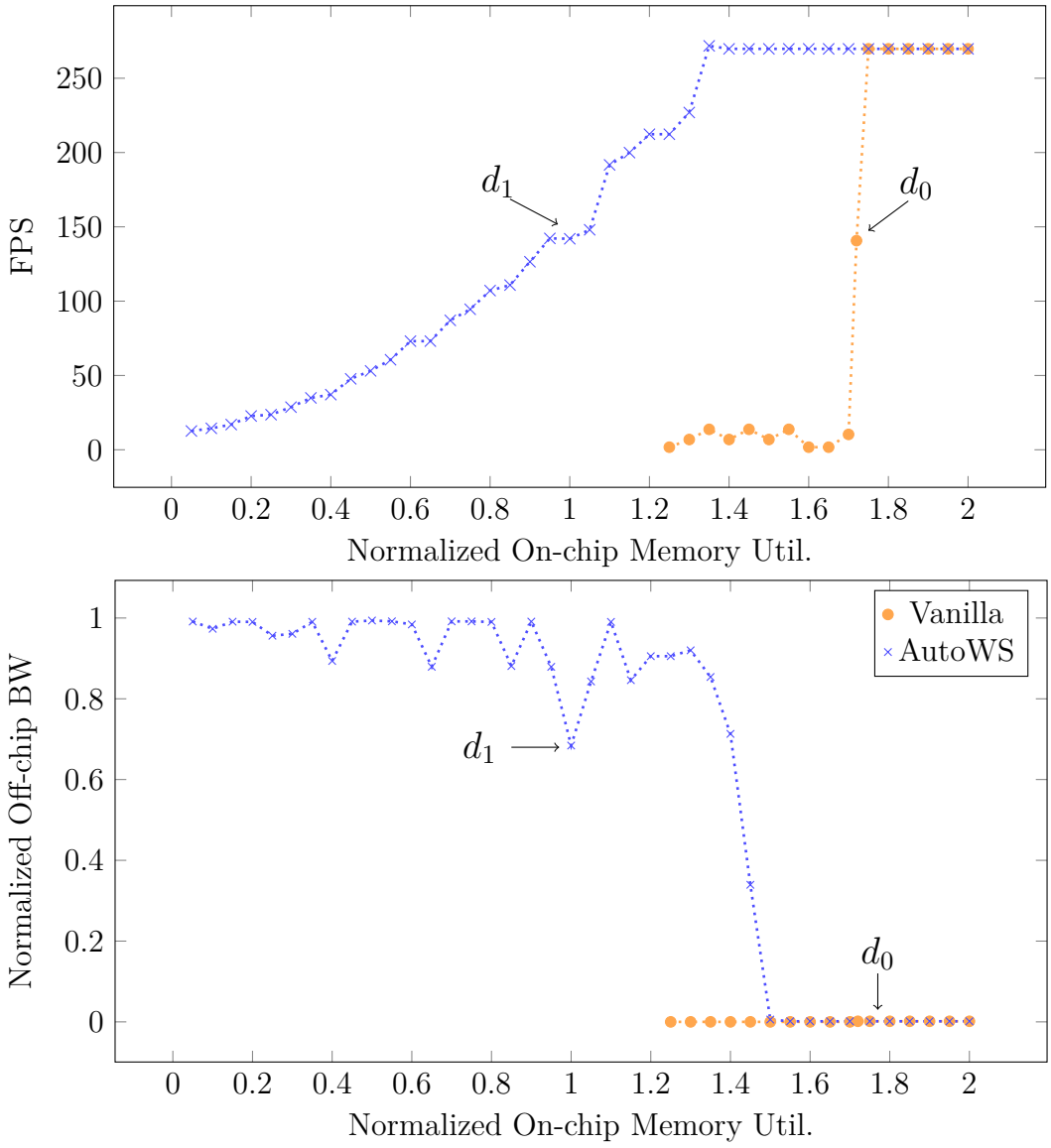


Figure 6.7: ResNet18-ZCU102, memory and performance trade-off. Normalization is against the maximum resource available on the device.

Based on the BRAM utilization, the **top** of Figure 6.7 can be split into three segments:

- $[0, 1.25)$: Here, the "vanilla" approach cannot fit, resulting in no valid design points. However, AutoWS exhibits steadily improved throughput as on-chip memory resources increase.
- $[1.25, 1.75)$: The "vanilla" approach is feasible but lags behind AutoWS in throughput, suggesting the bottleneck changes from the on-chip memory capacity to the off-chip memory bandwidth.
- $[1.75, 2)$: In this range, both the "vanilla" approach and AutoWS converge to the same set of design points as the accelerator becomes compute-bound instead.

Note that in the previous analysis in Figure 6.2, we illustrate the piecewise relationship between performance and weight memory usage. Figure 6.7 provides an actual example that demonstrates this trend in practice.

Table 6.2 offers a detailed resource breakdown for two specific design points, namely d_0 and d_1, selected from Figure 6.7. These two design points correspond to the "vanilla" approach and AutoWS, respectively. For d_0, its off-chip memory bandwidth is notably under-utilized, as the "vanilla" approach did not account for weight transfers (wt). Conversely, AutoWS (d_1) effectively utilizes this resource.

Regarding BRAM usage, it falls into three categories:

- **act_fifo**: FIFOs that connect CEs as well as the input/output buffers within each CE, responsible for holding intermediate activation data on-chip.
- **wt_buff**: shared on-chip buffers, used to load weights in the dynamic region from off-chip memory.

- **wt_mem**: the on-chip storage allocated for weights in the static region.

The costs of **act_fifo** and **wt_buff** are relatively minor (below 10%) compared to **wt_mem**. A comparison between design points d_0 and d_1, with similar throughput, reveals that AutoWS saves BRAM utilization by 70%, as highlighted in red in Table 6.2.

Table 6.2: ResNet18-ZCU102, memory resource breakdown.

Design Point	Off-chip BW (Gbps)			BRAM Usage (MB)				DSP	FPS
	act	wt	total (util.)	act_fifo	wt_buff	wt_mem	total(util.)		
Vanilla (d_0)	0.1	0.0	0.1 (0%)	0.4	0.0	8.3	8.7 (172%)	1113	141
AutoWS (d_1)	0.1	105.0	105.1 (68%)	0.4	0.1	4.6	5.1 (99%)	1180	142

Furthermore, Figure 6.8 shows the layer-wise memory allocation of our DSE algorithm. In this case, 5 out of 21 layers have part of the weights stored off-chip (layers 15 to 18 and 20). The selection of these layers prioritizes minimal bandwidth impact and large on-chip memory saving, as guided by the criterion $\frac{\Delta\overline{\beta_\omega}}{\Delta\mathcal{M}_\omega}$ outlined in Figure 6.5. We visualize this criterion as the red curve in Figure 6.8, with the corresponding values obtained at the end of DSE. As shown in the figure, layers 15 to 18 and 20 have the lowest values for this metric, leading to their weights being moved off-chip, represented by the green regions.

6.4.4 Object Detection

Furthermore, we evaluate the effectiveness of our proposed methodology in the context of the COCO object detection task [24], using the 8-bit YOLOv5n model [194], and targeting the ZCU102. AutoWS (8.7ms) achieves a 36% latency reduction compared to Vitis AI (13.7ms) [13], and a 9% reduction compared to the "vanilla layer-pipelined" (9.5ms).

Similar to the previous experiments, the advantage of AutoWS stems from the limitations in the "vanilla layer-pipelined" architecture, where constrained on-chip

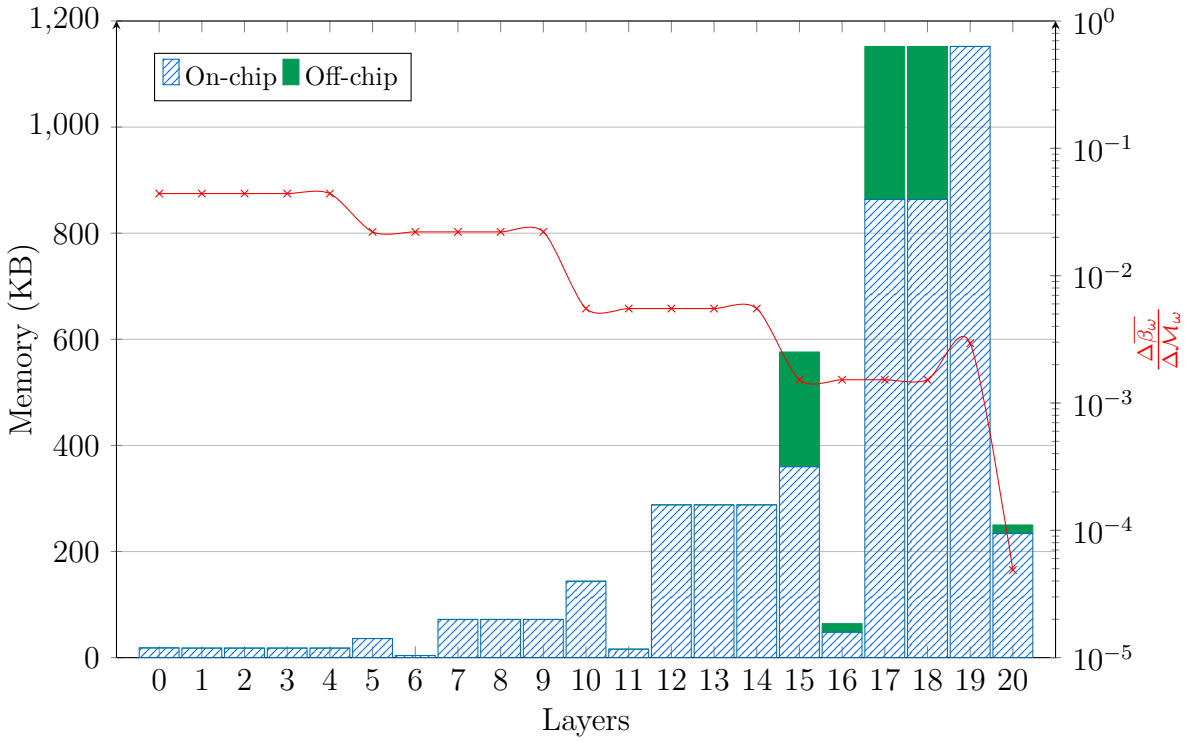


Figure 6.8: ResNet18-ZCU102, per-layer on-chip and off-chip memory allocation for the design point d_1. According to our DSE algorithm, layers with the smallest ratio $\frac{\Delta\overline{\beta_\omega}}{\Delta\mathcal{M}_\omega}$ will have their weights prioritized for movement to off-chip memory.

memory resources restrict the achievable parallelism. AutoWS overcomes these restrictions by optimizing weight access patterns and efficiently leveraging off-chip memory bandwidth. This also demonstrates the general applicability of the methodology beyond standard ImageNet classification tasks.

6.5 Summary

When designing FPGA accelerators for AI models, the capacity of on-chip memory is often the performance bottleneck. For accelerators designed as the traditional dataflow layer-pipelined architecture, such bottleneck is even more obvious since the model parameters are required permanently on-chip for high performance and easy scheduling. In this chapter, we introduce AutoWS which provides a novel memory management methodology for the layer-pipelined architecture, that efficiently exploits both on-chip and off-chip memory for weights storage. Specifically, it introduces memory fragmentation where the ratio of on-chip versus off-chip storage is automatically decided through a DSE process. As a result, AutoWS enables the deployment of larger-scale AI models on more resource-constrained FPGA devices in a layer-pipelined manner, which was not possible in prior works.

The proposed methodology is versatile and can be applied to a wide range of FPGA devices, regardless of whether DDR or HBM is used for off-chip memory, as long as the available maximum bandwidth is known. Moreover, it can seamlessly integrate with model compression techniques such as quantization, pruning, and decomposition, enabling further memory savings. Finally, our innovations are not confined to FPGAs but can also be extended to ASIC designs, potentially offering a comprehensive solution for next-generation AI accelerators.

7

Conclusion

Contents

7.1 Summary

This thesis investigated efficient AI model inference in both software algorithm and hardware architecture designs, with a particular focus on using Tensor Decomposition to compress the weights of AI models and designing customizable dataflow accelerators on resource-constrained FPGA devices. From the existing research, this thesis extends the design space of these two techniques and combines them with novel design automation approaches to reduce repetitive engineering efforts in the fast-evolving AI landscape. The specific contributions of this thesis are summarized as follows and they have all been open-sourced as Chapter 1 has stated:

Fine-grained Tensor Decomposition for CNN models

Within a Convolutional Neural Network (CNN) model, the data distribution of weight parameters along with their error tolerance can vary a lot across different parts of the network. Therefore, customizing a layer-wise, fine-grained compression scheme for weight parameters is highly desirable. This has led to the development of methods such as mixed precision quantization and unstructured pruning. However, when applying tensor decomposition for compressing weight parameters, existing literature remains coarse-grained in terms of their algorithm.

Starting from Chapter 3, this thesis introduces the first SVD-based algorithm that allows mixing existing tensor decomposition schemes on a per-layer basis. Then in Chapter 4, to extend the choices of decomposition schemes available, a novel low-rank network architecture design space *LR-space* is constructed, which generalizes from the corner cases considered in previous studies. Finally, Chapter 5 further incorporated CPD-based tensor decomposition into the design space, where the choice of applying SVD or CPD can also be model-specific and layerwise.

By leveraging these algorithms, our fine-grained approach significantly reduces accuracy degradation by 7 to 20 percentage points compared to prior work in the area of tensor decomposition. Furthermore, our experiments further demonstrate that fine-grained tensor decomposition can achieve results equivalent to or better than mixed precision quantization and unstructured pruning. Most importantly, networks decomposed using these methods can run much more efficiently on existing CPU/GPU devices, making them highly attractive for practical applications.

Dataflow Architecture Accelerators and Memory Optimization

For applications with critical performance demands, customized accelerator designs employing dataflow architectures are the optimal choice. Therefore, apart from deploying decomposed AI models on CPU/GPU devices, Chapters 3 and 5 intro-

duce hardware engines for efficiently running these decomposed models on FPGAs instead. These engines are interconnected with the rest of the computation graph to enable pipelined execution, releasing the full potential of tensor decomposition.

However, when designing these dataflow architectures, memory storage often becomes a bottleneck, especially when all the weight parameters are placed permanently on the chip. Model compression techniques such as tensor decomposition or quantization cannot always fully address this issue. Therefore, Chapter 6 focuses on a hardware-only solution. We introduce a methodology that smartly allocates both on-chip and off-chip memory resources, tailored for each specific combination of model workload and device constraints. This enables the deployment of large-scale AI models on resource-constrained FPGA devices in a layer-pipelined manner, which was not possible in previous work. As a result, we achieve a latency reduction of 25 to 48% on several modern CNN benchmarks.

Automated Software and Hardware Co-Design

When considering model compression and accelerator design, a shared problem is efficiently searching the design space to identify the optimal configuration. This challenge arises because the design space can be huge due to the fine-grained methodology employed in both the software algorithm and hardware architecture designs. To address this, this thesis investigated both advanced search algorithms and performance proxies.

In Chapter 4, we adapt gradient descent to search the model compression design space for tensor decomposition. To estimate the impact of compression on real-time performance, we first use the number of operations as a proxy, which is common in the machine learning community. To further promote the hardware-awareness of our algorithm, we integrate regressional performance models targeting mobile CPUs and GPUs.

Then, in Chapter 3 and 5, the focus is shifted to FPGAs. Since hardware in FPGA-based designs is not fixed, we employ the evolutionary algorithm to search for optimal model compression and hardware accelerator configurations simultaneously. This co-optimization can be imagined as a two-level nested loop, where the outer loop searches for compression decisions, while the inner loop explores accelerator configurations. In Chapter 5, we propose a greedy hardware design space exploration technique and a random forest performance predictor to further speed up the iteration of the inner loop and outer loop respectively. Overall, our experimental results have shown a significant reduction in search time by more than 10×, with a negligible impact on search quality.

7.2 Reflections

One of the key lessons learned during this research is the growing difficulty in exploring model compression techniques, especially tensor decomposition, as well as quantization and pruning. AI Model designs are fast evolving through sophisticated engineering efforts and automated Neural Architecture Search (NAS) processes, which optimize models for both accuracy and compactness. This evolution has led to the increasing difficulty of applying traditional compression methods without significant accuracy degradation.

For example, in Table 4.4, despite overcoming some of the state-of-the-art tensor decomposition techniques, we observed that accuracy degradation remains substantial for models like MobileNetV2 and EfficientNet. These models, being designed with efficiency in mind, are particularly sensitive to compression, which highlights the limitations of current compression techniques when applied to highly optimized model architectures.

This experience raises an important question: Is it still feasible to train a model

and then compress it afterward, or should more integrated approaches, such as quantization-aware training, be prioritized? Developing new strategies to address the compressibility of compact and efficient models is essential for the continued progress in AI model optimization.

Furthermore, regarding hardware accelerators, the research has highlighted the attractive potential for co-design between model compression and accelerator architectures. However, an open question that remains is how much automation can ultimately be achieved in this co-design process. For example, in terms of the design space and search algorithms, a significant amount of manual configuration is still required, such as setting hyperparameters or applying heuristics.

7.3 Potential Extensions and Improvements

7.3.1 Short-term Improvments

Throughout this thesis, the experiments for both software and hardware evaluation primarily focus on Convolutional Neural Network (CNN) models, which remain highly competitive for image classification tasks [195]. However, it would be interesting to examine the proposed methodologies on other types of models and tasks. For example, the low-rank tensor decomposition space (*LR-space*) can be applied to transformer models on Natural Language Processing (NLP) tasks. These transformer models contain a lot of matrix multiplications that are potentially suitable for decomposition. Similarly, in the hardware domain, the highly customizable dataflow accelerator architecture might also be appropriate for transformer models. Once hardware libraries are designed to support any additional operation types involved in transformer models, the proposed design automation (Chapter 5) and memory optimization (Chapter 6) technologies will also be applicable there.

In addition, the current evaluation throughout the thesis focuses on the latency and throughput of the accelerators. same as the objective function set in DSE (Equation 5.1). However, it would be valuable to also evaluate the impact on energy efficiency. By considering energy consumption as an additional factor in the optimization process, it would be possible to achieve a more holistic evaluation of the accelerator's performance. Of course, a key research challenge in this direction is modeling energy consumption for a fast DSE process. Otherwise, it would require running synthesis or even place-and-route for each design point, which will be time-consuming.

7.3.2 Long-term Research

Regarding the model compression algorithms, we have currently focused on tensor decomposition and explored its combination with the quantization approach (Chapter 3 and 5). However, in this combination, we employed a fine-grained decomposition scheme while using a coarse-grained quantization method with a fixed data width. Such a combination did not explore more advanced techniques such as mixed-precision quantization. By extending gradient descent and evolutionary search techniques proposed in Chapter 4 and 5, it is possible to co-search fine-grained compression decisions involving both tensor decomposition and quantization. Additionally, unstructured pruning can be included in the design space as well. Therefore, it is possible to build a tool that can explore multiple model compression approaches jointly on any given AI model.

Back to the FPGA accelerator designs, this thesis explores dataflow-style architectures with dedicated hardware units allocated per layer as a pipeline. This approach contrasts with traditional Neural Processing Units (NPUs), which typically exploit only intra-layer parallelism while exploiting sequential time-multiplexing between layers. Based on the analysis in Chapter 6, the dataflow style is more fa-

vored on middle and large-size FPGAs with enough on-chip resources available when the sequential style can be advantageous when the resources become really constrained. One possible direction is finding a middle ground where a configurable partial pipeline can be implemented. For example, a transformer model has cascaded attention blocks, each containing multiple matrix multiplication, transpose, softmax and other operations. In such a case, the attention block can be computed internally in a pipelined manner, while time-multiplexing can be applied externally between the blocks. Such an adaptable architecture could optimize performance by adjusting the granularity of parallelism and pipelining based on the specific constraints and requirements of the given application and hardware.

Bibliography

[1] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE conference on computer vision and pattern recognition*, Ieee, 2009, pp. 248–255.

[2] A. Radford, "Improving language understanding by generative pre-training," 2018.

[3] S. Ruder, "An overview of gradient descent optimization algorithms," *arXiv preprint: 1609.04747*, 2016.

[4] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *The journal of machine learning research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[5] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *International conference on machine learning*, pmlr, 2015, pp. 448–456.

[6] C. Shorten and T. M. Khoshgoftaar, "A survey on image data augmentation for deep learning," *Journal of big data*, vol. 6, no. 1, pp. 1–48, 2019.

[7] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[8] A. Vaswani, N. Shazeer, N. Parmar, *et al.*, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[9] N. Corporation, *CUDA Toolkit - Free Tools and Training — developer.nvidia.com*, `https://developer.nvidia.com/cuda-toolkit`, [Accessed 05-12-2024].

[10] M. Abadi, P. Barham, J. Chen, *et al.*, "{Tensorflow}: A system for {large-scale} machine learning," in *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, 2016, pp. 265–283.

[11] A. Paszke, S. Gross, F. Massa, *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *Advances in neural information processing systems*, vol. 32, 2019.

[12] N. P. Jouppi, C. Young, N. Patil, *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the 44th annual international symposium on computer architecture*, 2017, pp. 1–12.

[13] V. Kathail, "Xilinx vitis unified software platform," in *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2020, pp. 173–174.

[14] Intel, *Openvino toolkit: Open visual inference & neural network optimization*, 2024. [Online]. Available: `https://software.intel.com/content/www/us/en/develop/tools/openvino-toolkit.html`.

[15] Nvidia, *Tensorrt: High performance deep learning inference library*, 2024. [Online]. Available: `https://developer.nvidia.com/tensorrt`.

[16] B. Peng, W. Tan, Z. Li, S. Zhang, D. Xie, and S. Pu, "Extreme network compression via filter group approximation," in *Proceedings of the European Conference on Computer Vision (ECCV)*, 2018, pp. 300–316.

[17] Y.-D. Kim, E. Park, S. Yoo, T. Choi, L. Yang, and D. Shin, "Compression of deep convolutional neural networks for fast and low power mobile applications," *arXiv preprint: 1511.06530*, 2015.

[18] X. Zhang, J. Zou, K. He, and J. Sun, "Accelerating very deep convolutional networks for classification and detection," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 38, no. 10, pp. 1943–1955, 2015.

[19] S. I. Venieris and C.-S. Bouganis, "Fpgaconvnet: A framework for mapping convolutional neural networks on fpgas," in *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, IEEE, 2016, pp. 40–47.

[20] Y. Umuroglu, N. J. Fraser, G. Gambardella, *et al.*, "Finn: A framework for fast, scalable binarized neural network inference," in *Proceedings of the 2017 ACM/SIGDA international symposium on field-programmable gate arrays*, 2017, pp. 65–74.

[21] F. Fahim, B. Hawks, C. Herwig, *et al.*, "Hls4ml: An open-source codesign workflow to empower scientific low-power machine learning devices," *arXiv preprint: 2103.05579*, 2021.

[22] L. Petrica, T. Alonso, M. Kroes, N. Fraser, S. Cotofana, and M. Blott, "Memory-efficient dataflow inference for deep cnns on fpga," in *2020 International Conference on Field-Programmable Technology (ICFPT)*, IEEE, 2020, pp. 48–55.

[23] A. Krizhevsky, G. Hinton, *et al.*, "Learning multiple layers of features from tiny images," 2009.

[24] T.-Y. Lin, M. Maire, S. Belongie, *et al.*, "Microsoft coco: Common objects in context," in *Computer Vision–ECCV 2014: 13th European Conference, Zurich, Switzerland, September 6-12, 2014, Proceedings, Part V 13*, Springer, 2014, pp. 740–755.

[25] M. Everingham, S. A. Eslami, L. Van Gool, C. K. Williams, J. Winn, and A. Zisserman, "The pascal visual object classes challenge: A retrospective," *International journal of computer vision*, vol. 111, pp. 98–136, 2015.

[26] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Advances in neural information processing systems*, vol. 25, 2012.

[27] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *3rd International Conference on Learning Representations (ICLR 2015)*, Computational and Biological Learning Society, 2015.

[28] C. Szegedy, W. Liu, Y. Jia, *et al.*, "Going deeper with convolutions," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 1–9.

[29] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 4700–4708.

[30] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "Squeezenet: Alexnet-level accuracy with 50x fewer parameters and< 0.5 mb model size," *arXiv preprint: 1602.07360*, 2016.

[31] S. Xie, R. Girshick, P. Dollár, Z. Tu, and K. He, "Aggregated residual transformations for deep neural networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 1492–1500.

[32] X. Zhang, X. Zhou, M. Lin, and J. Sun, "Shufflenet: An extremely efficient convolutional neural network for mobile devices," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 6848–6856.

[33] A. G. Howard, M. Zhu, B. Chen, *et al.*, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *arXiv preprint: 1704.04861*, 2017.

[34] M. Tan and Q. Le, "Efficientnet: Rethinking model scaling for convolutional neural networks," in *International conference on machine learning*, PMLR, 2019, pp. 6105–6114.

[35] A. Dosovitskiy, L. Beyer, A. Kolesnikov, *et al.*, "An image is worth 16x16 words: Transformers for image recognition at scale," *arXiv preprint: 2010.11929*, 2020.

[36] X. Ding, X. Zhang, N. Ma, J. Han, G. Ding, and J. Sun, "Repvgg: Making vgg-style convnets great again," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2021, pp. 13 733–13 742.

[37] Z. Liu, H. Mao, C.-Y. Wu, C. Feichtenhofer, T. Darrell, and S. Xie, "A convnet for the 2020s," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2022, pp. 11 976–11 986.

[38] T. Salimans and D. P. Kingma, "Weight normalization: A simple reparameterization to accelerate training of deep neural networks," *Advances in neural information processing systems*, vol. 29, 2016.

[39] D. Hendrycks and K. Gimpel, "Gaussian error linear units (gelus)," *arXiv preprint: 1606.08415*, 2016.

[40] J. L. Ba, J. R. Kiros, and G. E. Hinton, "Layer normalization," *arXiv preprint: 1607.06450*, 2016.

[41] C. Louizos, M. Welling, and D. P. Kingma, "Learning sparse neural networks through l_0 regularization," in *International Conference on Learning Representations*, 2018.

[42] W. Wen, C. Wu, Y. Wang, Y. Chen, and H. Li, "Learning structured sparsity in deep neural networks," *Advances in neural information processing systems*, vol. 29, 2016.

[43] H. Hu, R. Peng, Y.-W. Tai, and C.-K. Tang, "Network trimming: A data-driven neuron pruning approach towards efficient deep architectures," *arXiv preprint: 1607.03250*, 2016.

[44] J. Ye, X. Lu, Z. Lin, and J. Z. Wang, "Rethinking the smaller-norm-less-informative assumption in channel pruning of convolution layers," in *6th International Conference on Learning Representations, ICLR 2018*, 2018.

[45] P. Molchanov, S. Tyree, T. Karras, T. Aila, and J. Kautz, "Pruning convolutional neural networks for resource efficient inference," in *International Conference on Learning Representations*, 2016.

[46] R. Yu, A. Li, C.-F. Chen, *et al.*, "Nisp: Pruning networks using neuron importance score propagation," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018, pp. 9194–9203.

[47] P. Molchanov, A. Mallya, S. Tyree, I. Frosio, and J. Kautz, "Importance estimation for neural network pruning," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2019, pp. 11 264–11 272.

[48] H. Li, A. Kadav, I. Durdanovic, H. Samet, and H. P. Graf, "Pruning filters for efficient convnets," *arXiv preprint: 1608.08710*, 2016.

[49] Y. He, X. Zhang, and J. Sun, "Channel pruning for accelerating very deep neural networks," in *Proceedings of the IEEE International Conference on Computer Vision*, 2017, pp. 1389–1397.

[50] X. Ding, T. Hao, J. Tan, *et al.*, "Resrep: Lossless cnn pruning via decoupling remembering and forgetting," in *Proceedings of the IEEE/CVF international conference on computer vision*, 2021, pp. 4510–4520.

[51] E. Wang, J. J. Davis, R. Zhao, *et al.*, "Deep neural network approximation for custom hardware: Where we've been, where we're going," *ACM Computing Surveys (CSUR)*, vol. 52, no. 2, pp. 1–39, 2019.

[52] Z. Liu, J. Li, Z. Shen, G. Huang, S. Yan, and C. Zhang, "Learning efficient convolutional networks through network slimming," in *Proceedings of the IEEE International Conference on Computer Vision*, 2017, pp. 2736–2744.

[53] S. Cao, C. Zhang, Z. Yao, *et al.*, "Efficient and effective sparse lstm on fpga with bank-balanced sparsity," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2019, pp. 63–72.

[54] B. Li, Z. Kong, T. Zhang, *et al.*, "Efficient transformer-based large scale language representations using hardware-friendly block structured pruning," in *Findings of the Association for Computational Linguistics: EMNLP 2020*, 2020, pp. 3187–3199.

[55] N. Lee, T. Ajanthan, and P. Torr, "Snip: Single-shot network pruning based on connection sensitivity," in *International Conference on Learning Representations*, 2018.

[56] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," *arXiv preprint: 1510.00149*, 2015.

[57] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "Xnor-net: Imagenet classification using binary convolutional neural networks," in *European conference on computer vision*, Springer, 2016, pp. 525–542.

[58] D. Kalamkar, D. Mudigere, N. Mellempudi, *et al.*, "A study of bfloat16 for deep learning training," *arXiv preprint: 1905.12322*, 2019.

[59] P. Gysel, "Ristretto: Hardware-oriented approximation of convolutional neural networks," *arXiv preprint: 1605.06402*, 2016.

[60] S. Migacz, "8-bit inference with tensorrt," in *GPU technology conference*, vol. 2, 2017, p. 7.

[61] R. Banner, Y. Nahshan, and D. Soudry, "Post training 4-bit quantization of convolutional networks for rapid-deployment," *Advances in Neural Information Processing Systems*, vol. 32, 2019.

[62] Y. Cai, Z. Yao, Z. Dong, A. Gholami, M. W. Mahoney, and K. Keutzer, "Zeroq: A novel zero shot quantization framework," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2020, pp. 13 169–13 178.

[63] M. Haroush, I. Hubara, E. Hoffer, and D. Soudry, "The knowledge within: Methods for data-free model compression," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2020, pp. 8494–8502.

[64] S. Zhou, Y. Wu, Z. Ni, X. Zhou, H. Wen, and Y. Zou, "Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients," *arXiv preprint: 1606.06160*, 2016.

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

[66] J. H. Lee, S. Ha, S. Choi, W.-J. Lee, and S. Lee, "Quantization for rapid deployment of deep neural networks," *arXiv preprint: 1810.05488*, 2018.

[67] B. Darvish Rouhani, D. Lo, R. Zhao, *et al.*, "Pushing the limits of narrow precision inferencing at cloud scale with microsoft floating point," *Advances in neural information processing systems*, vol. 33, pp. 10 271–10 281, 2020.

[68] M. Nagel, M. v. Baalen, T. Blankevoort, and M. Welling, "Data-free quantization through weight equalization and bias correction," in *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2019, pp. 1325–1334.

[69] E. Frantar and D. Alistarh, "Optimal brain compression: A framework for accurate post-training quantization and pruning," *Advances in Neural Information Processing Systems*, vol. 35, pp. 4475–4488, 2022.

[70] K. Wang, Z. Liu, Y. Lin, J. Lin, and S. Han, "Haq: Hardware-aware automated quantization with mixed precision," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2019, pp. 8612–8620.

[71] M. Heath, A. Laub, C. Paige, and R. Ward, "Computing the singular value decomposition of a product of two matrices," *SIAM Journal on Scientific and Statistical Computing*, vol. 7, no. 4, pp. 1147–1159, 1986.

[72] T. G. Kolda and B. W. Bader, "Tensor decompositions and applications," *SIAM Review*, vol. 51, no. 3, pp. 455–500, 2009.

[73] J. Qiu, J. Wang, S. Yao, *et al.*, "Going deeper with embedded fpga platform for convolutional neural network," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2016, pp. 26–35.

[74] C. Tai, T. Xiao, Y. Zhang, X. Wang, and E. Weinan, "Convolutional neural networks with low-rank regularization," in *4th International Conference on Learning Representations, ICLR 2016*, 2016.

[75] Z. Chen, Z. Chen, J. Lin, S. Liu, and W. Li, "Deep neural network acceleration based on low-rank approximated channel pruning," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 67, no. 4, pp. 1232–1244, 2020.

[76] J. M. Alvarez and M. Salzmann, "Compression-aware training of deep networks," *Advances in neural information processing systems*, vol. 30, 2017.

[77] W. Wen, C. Xu, C. Wu, Y. Wang, Y. Chen, and H. Li, "Coordinating filters for faster deep neural networks," in *Proceedings of the IEEE International Conference on Computer Vision*, 2017, pp. 658–666.

[78] Y. Idelbayev and M. A. Carreira-Perpinán, "Low-rank compression of neural nets: Learning the rank of each layer," in *Proceedings of the IEEE/CVF*

Conference on Computer Vision and Pattern Recognition, 2020, pp. 8049–8059.

[79] V. Lebedev, Y. Ganin, M. Rakhuba, I. Oseledets, and V. Lempitsky, "Speeding-up convolutional neural networks using fine-tuned cp-decomposition," in *3rd International Conference on Learning Representations, ICLR 2015-Conference Track Proceedings*, 2015.

[80] T. Garipov, D. Podoprikhin, A. Novikov, and D. Vetrov, "Ultimate tensorization: Compressing convolutional and fc layers alike," *arXiv preprint: 1611.03214*, 2016.

[81] Q. Zhao, G. Zhou, S. Xie, L. Zhang, and A. Cichocki, "Tensor ring decomposition," *arXiv preprint: 1606.05535*, 2016.

[82] *The tensor network*. [Online]. Available: `https://tensornetwork.org/diagrams/#Penrose:1971%201`.

[83] T. Dao, A. Gu, M. Eichhorn, A. Rudra, and C. Ré, "Learning fast algorithms for linear transforms using butterfly factorizations," in *International conference on machine learning*, PMLR, 2019, pp. 1517–1527.

[84] H. Fan, T. Chau, S. I. Venieris, *et al.*, "Adaptable butterfly accelerator for attention-based nns via hardware and algorithm co-design," in *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, IEEE, 2022, pp. 599–615.

[85] K. Wu, Y. Guo, and C. Zhang, "Compressing deep neural networks with sparse matrix factorization," *IEEE transactions on neural networks and learning systems*, vol. 31, no. 10, pp. 3828–3838, 2019.

[86] L. Liebenwein, A. Maalouf, D. Feldman, and D. Rus, "Compressing neural networks: Towards determining the optimal layer-wise decomposition," *Advances in Neural Information Processing Systems*, vol. 34, pp. 5328–5344, 2021.

[87] B. Zhao, Q. Cui, R. Song, Y. Qiu, and J. Liang, "Decoupled knowledge distillation," in *Proceedings of the IEEE/CVF Conference on computer vision and pattern recognition*, 2022, pp. 11 953–11 962.

[88] G. Hinton, O. Vinyals, and J. Dean, "Distilling the knowledge in a neural network," *arXiv preprint: 1503.02531*, 2015.

[89] A. Romero, N. Ballas, S. E. Kahou, A. Chassang, C. Gatta, and Y. Bengio, *Fitnets: Hints for thin deep nets*, 2015. arXiv: `1412.6550 [cs.LG]`.

[90] J. Yim, D. Joo, J. Bae, and J. Kim, "A gift from knowledge distillation: Fast optimization, network minimization and transfer learning," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 4133–4141.

[91] R. G. Lopes, S. Fenu, and T. Starner, "Data-free knowledge distillation for deep neural networks," *arXiv preprint: 1710.07535*, 2017.

[92] T. Li, J. Li, Z. Liu, and C. Zhang, "Few sample knowledge distillation for efficient network compression," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2020, pp. 14 639–14 647.

[93] H. Liu, K. Simonyan, and Y. Yang, "Darts: Differentiable architecture search," in *International Conference on Learning Representations*, 2018.

[94] A. Howard, M. Sandler, G. Chu, *et al.*, "Searching for mobilenetv3," in *Proceedings of the IEEE/CVF international conference on computer vision*, 2019, pp. 1314–1324.

[95] W. Jiang, X. Zhang, E. H.-M. Sha, *et al.*, "Accuracy vs. efficiency: Achieving both through fpga-implementation aware neural architecture search," in *Proceedings of the 56th Annual Design Automation Conference 2019*, 2019, pp. 1–6.

[96] H. Cai, C. Gan, T. Wang, Z. Zhang, and S. Han, "Once-for-all: Train one network and specialize it for efficient deployment," in *International Conference on Learning Representations*, 2019.

[97] B. Wu, X. Dai, P. Zhang, *et al.*, "Fbnet: Hardware-aware efficient convnet design via differentiable neural architecture search," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2019, pp. 10 734–10 742.

[98] M. S. Abdelfattah, A. Mehrotra, Ł. Dudziak, and N. D. Lane, "Zero-cost proxies for lightweight nas," in *International Conference on Learning Representations*, 2020.

[99] X. Dong, L. Liu, K. Musial, and B. Gabrys, "Nats-bench: Benchmarking nas algorithms for architecture topology and size," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 44, no. 7, pp. 3634–3646, 2021.

[100] S. Markidis, S. W. Der Chien, E. Laure, I. B. Peng, and J. S. Vetter, "Nvidia tensor core programmability, performance & precision," in *2018 IEEE international parallel and distributed processing symposium workshops (IPDPSW)*, IEEE, 2018, pp. 522–531.

[101] A. Ltd., *Neon – Arm® — arm.com*, `https://www.arm.com/technologies/neon`, [Accessed 05-12-2024].

[102] A. Inc., *Metal Performance Shaders: Optimize graphics and compute performance with kernels that are fine-tuned for the unique characteristics of each Metal GPU family.* `https://developer.apple.com/documentation/metalperformanceshaders`, [Accessed 05-12-2024].

[103] I. Advanced Micro Devices, *AMD ROCm Software: Open software stack that includes programming models, tools, compilers, libraries, and runtimes for AI and HPC solution development on AMD GPUs*, `https://www.amd.com/en/products/software/rocm.html`, [Accessed 05-12-2024].

[104] S. Lie, "Cerebras architecture deep dive: First look inside the hw/sw co-design for deep learning: Cerebras systems," in *2022 IEEE Hot Chips 34 Symposium (HCS)*, IEEE Computer Society, 2022, pp. 1–34.

[105] D. Abts, J. Ross, J. Sparling, *et al.*, "Think fast: A tensor streaming processor (tsp) for accelerating deep learning workloads," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, IEEE, 2020, pp. 145–158.

[106] T. Han, T. Zhang, D. Li, *et al.*, "Convolutional neural network with int4 optimization on xilinx devices," *Xilinx White Paper, WP521*, 2020.

[107] Y. Chen, J. Dotzel, and M. S. Abdelfattah, "M4bram: Mixed-precision matrix-matrix multiplication in fpga block rams," in *2023 International Conference on Field Programmable Technology (ICFPT)*, IEEE, 2023, pp. 69–78.

[108] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE journal of solid-state circuits*, vol. 52, no. 1, pp. 127–138, 2016.

[109] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. 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*, 2015, pp. 161–170.

[110] Y. Ma, N. Suda, Y. Cao, J.-s. Seo, and S. Vrudhula, "Scalable and modularized rtl compilation of convolutional neural networks onto fpga," in *2016 26th international conference on field programmable logic and applications (FPL)*, IEEE, 2016, pp. 1–8.

[111] Y. Guan, H. Liang, N. Xu, *et al.*, "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)*, IEEE, 2017, pp. 152–159.

[112] X. Zhang, J. Wang, C. Zhu, *et al.*, "Dnnbuilder: An automated tool for building high-performance dnn hardware accelerators for fpgas," in *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, IEEE, 2018, pp. 1–8.

[113] H. Sharma, J. Park, D. Mahajan, *et al.*, "From high-level deep neural models to fpgas," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, IEEE, 2016, pp. 1–12.

[114] K. Guo, L. Sui, J. Qiu, *et al.*, "Angel-eye: A complete design flow for mapping cnn onto embedded fpga," *IEEE transactions on computer-aided design of integrated circuits and systems*, vol. 37, no. 1, pp. 35–47, 2017.

[115] V. Gokhale, A. Zaidy, A. X. M. Chang, and E. Culurciello, "Snowflake: An efficient hardware accelerator for convolutional neural networks," in *2017 IEEE International Symposium on Circuits and Systems (ISCAS)*, IEEE, 2017, pp. 1–4.

[116] J. Yu, G. Ge, Y. Hu, *et al.*, "Instruction driven cross-layer cnn accelerator for fast detection on fpga," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 11, no. 3, pp. 1–23, 2018.

[117] S. I. Venieris, A. Kouris, and C.-S. Bouganis, "Toolflows for mapping convolutional neural networks on fpgas: A survey and future directions," *ACM Computing Surveys (CSUR)*, vol. 51, no. 3, pp. 1–39, 2018.

[118] Y. Shen, M. Ferdman, and P. Milder, "Maximizing cnn accelerator efficiency through resource partitioning," *ACM SIGARCH Computer Architecture News*, vol. 45, no. 2, pp. 535–547, 2017.

[119] X. Wei, C. H. Yu, P. Zhang, *et al.*, "Automated systolic array architecture synthesis for high throughput cnn inference on fpgas," in *Proceedings of the 54th Annual Design Automation Conference 2017*, 2017, pp. 1–6.

[120] A. Samajdar, Y. Zhu, P. Whatmough, M. Mattina, and T. Krishna, "Scale-sim: Systolic cnn accelerator simulator," *arXiv preprint: 1811.02883*, 2018.

[121] X. Wang, C. Wang, J. Cao, L. Gong, and X. Zhou, "Winonn: Optimizing fpga-based convolutional neural network accelerators using sparse winograd algorithm," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 11, pp. 4290–4302, 2020.

[122] A. Montgomerie-Corcoran and C. Savvas-Bouganis, "Def: Differential encoding of featuremaps for low power convolutional neural network accelerators," in *Proceedings of the 26th Asia and South Pacific Design Automation Conference*, 2021, pp. 703–708.

[123] B. Li, S. Pandey, H. Fang, *et al.*, "Ftrans: Energy-efficient acceleration of transformers using fpga," in *Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design*, 2020, pp. 175–180.

[124] D. Wu, W. Cao, and L. Wang, "Spwmm: A high-performance sparse-winograd matrix-matrix multiplication accelerator for cnns," in *2019 International Conference on Field-Programmable Technology (ICFPT)*, IEEE, 2019, pp. 255–258.

[125] H. Ye, X. Zhang, Z. Huang, G. Chen, and D. Chen, "Hybriddnn: A framework for high-performance hybrid dnn accelerator design and implementation," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*, IEEE, 2020, pp. 1–6.

[126] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 779–788.

[127] M. S. Abdelfattah, D. Han, A. Bitar, *et al.*, "Dla: Compiler and fpga overlay for neural network inference acceleration," in *2018 28th International Con-*

ference on Field Programmable Logic and Applications (FPL), IEEE, 2018, pp. 411–4117.

[128] R. Shi, Y. Ding, X. Wei, *et al.*, "Ftdl: A tailored fpga-overlay for deep learning with high scalability," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*, IEEE, 2020, pp. 1–6.

[129] Y. Yu, T. Zhao, K. Wang, and L. He, "Light-opu: An fpga-based overlay processor for lightweight convolutional neural networks," in *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2020, pp. 122–132.

[130] Y. Wang, J. Xu, Y. Han, H. Li, and X. Li, "Deepburning: Automatic generation of fpga-based learning accelerators for the neural network family," in *Proceedings of the 53rd Annual Design Automation Conference*, 2016, pp. 1–6.

[131] J. Duarte, S. Han, P. Harris, *et al.*, "Fast inference of deep neural networks in FPGAs for particle physics," *Journal of Instrumentation*, vol. 13, no. 07, Jul. 2018, ISSN: 1748-0221. DOI: 10.1088/1748-0221/13/07/P07027.

[132] M. Hall and V. Betz, "From tensorflow graphs to luts and wires: Automated sparse and physically aware cnn hardware generation," in *2020 International Conference on Field-Programmable Technology (ICFPT)*, IEEE, 2020, pp. 56–65.

[133] E. Luo, H. Huang, C. Liu, *et al.*, "Deepburning-mixq: An open source mixed-precision neural network accelerator design framework for fpgas," in *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, IEEE, 2023, pp. 1–9.

[134] A. Montgomerie-Corcoran, Z. Yu, J. Cheng, and C.-S. Bouganis, "Pass: Exploiting post-activation sparsity in streaming architectures for cnn accelera-

tion," in *2023 33rd International Conference on Field-Programmable Logic and Applications (FPL)*, IEEE, 2023, pp. 288–293.

[135] T. Alonso, L. Petrica, M. Ruiz, *et al.*, "Elastic-df: Scaling performance of dnn inference in fpga clouds through automatic partitioning," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 15, no. 2, pp. 1–34, 2021.

[136] M. Ibrahim, Z. Zhao, M. Hall, and V. Betz, "Extending data flow architectures for convolutional neural networks to multiple fpgas," in *2023 International Conference on Field-Programmable Technology (ICFPT)*, IEEE, 2023, pp. 126–135.

[137] M. Sun, Z. Li, A. Lu, *et al.*, "Film-qnn: Efficient fpga acceleration of deep neural networks with intra-layer, mixed-precision quantization," in *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2022, pp. 134–145.

[138] A. Kouris, S. I. Venieris, and C.-S. Bouganis, "Cascadê cnn: Pushing the performance limits of quantisation in convolutional neural networks," in *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, IEEE, 2018, pp. 155–1557.

[139] S. I. Venieris and C.-S. Bouganis, "Latency-driven design for fpga-based convolutional neural networks," in *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, IEEE, 2017, pp. 1–8.

[140] S. Basalama, A. Sohrabizadeh, J. Wang, L. Guo, and J. Cong, "Flexcnn: An end-to-end framework for composing cnn accelerators on fpga," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 16, no. 2, pp. 1–32, 2023.

[141] A. Montgomerie-Corcoran, S. I. Venieris, and C.-S. Bouganis, "Power-aware fpga mapping of convolutional neural networks," in *2019 International Con-*

ference on Field-Programmable Technology (ICFPT), IEEE, 2019, pp. 327–330.

[142] R. V. W. Putra, M. A. Hanif, and M. Shafique, "Drmap: A generic dram data mapping policy for energy-efficient processing of convolutional neural networks," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*, IEEE, 2020, pp. 1–6.

[143] A. Sohrabizadeh, J. Wang, and J. Cong, "End-to-end optimization of deep learning applications," in *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2020, pp. 133–139.

[144] A. Kouris, S. I. Venieris, M. Rizakis, and C.-S. Bouganis, "Approximate lstms for time-constrained inference: Enabling fast reaction in self-driving cars," *IEEE Consumer Electronics Magazine*, vol. 9, no. 4, pp. 11–26, 2020.

[145] P. Goswami, M. Shahshahani, and D. Bhatia, "Robust estimation of fpga resources and performance from cnn models," in *2022 35th International Conference on VLSI Design and 2022 21st International Conference on Embedded Systems (VLSID)*, IEEE, 2022, pp. 144–149.

[146] M. S. Abdelfattah, Ł. Dudziak, T. Chau, R. Lee, H. Kim, and N. D. Lane, "Best of both worlds: Automl codesign of a cnn and its hardware accelerator," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*, IEEE, 2020, pp. 1–6.

[147] Y. Li, C. Hao, X. Zhang, *et al.*, "Edd: Efficient differentiable dnn architecture and implementation co-search for embedded ai solutions," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*, IEEE, 2020, pp. 1–6.

[148] Z. Yu and C.-S. Bouganis, "Streamsvd: Low-rank approximation and streaming accelerator co-design," in *2021 International Conference on Field-Programmable Technology (ICFPT)*, IEEE, 2021, pp. 1–9.

[149] J. Zhu, Z. Qian, and C.-Y. Tsui, "Lradnn: High-throughput and energy-efficient deep neural network accelerator using low rank approximation," in *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*, IEEE, 2016, pp. 581–586.

[150] C. Hao, X. Zhang, Y. Li, *et al.*, "Fpga/dnn co-design: An efficient design methodology for iot intelligence on the edge," in *2019 56th ACM/IEEE Design Automation Conference (DAC)*, IEEE, 2019, pp. 1–6.

[151] E. L. Denton, W. Zaremba, J. Bruna, Y. LeCun, and R. Fergus, "Exploiting linear structure within convolutional networks for efficient evaluation," *Advances in neural information processing systems*, vol. 27, 2014.

[152] M. Wang, B. Liu, and H. Foroosh, "Factorized convolutional neural networks," in *Proceedings of the IEEE International Conference on Computer Vision Workshops*, 2017, pp. 545–553.

[153] S. Ribes, P. Trancoso, I. Sourdis, and C.-S. Bouganis, "Mapping multiple lstm models on fpgas," in *2020 International Conference on Field-Programmable Technology (ICFPT)*, IEEE, 2020, pp. 1–9.

[154] A. Xilinx, *7 series fpgas memory resources user guide*, Accessed: 2024-09-23, 2023. [Online]. Available: `https://docs.amd.com/v/u/en-US/ug473_7Series_Memory_Resources`.

[155] X. Zhang, J. Zou, X. Ming, K. He, and J. Sun, "Efficient and accurate approximations of nonlinear convolutional networks," in *Proceedings of the IEEE Conference on Computer Vision and pattern Recognition*, 2015, pp. 1984–1992.

[156] Y. Chen, C. Hawkins, K. Zhang, Z. Zhang, and C. Hao, "3u-edgeai: Ultra-low memory training, ultra-low bitwidth quantization, and ultra-low latency acceleration," in *Proceedings of the 2021 on Great Lakes Symposium on VLSI*, 2021, pp. 157–162.

[157] C. Mei, Z. Liu, Y. Niu, X. Ji, W. Zhou, and D. Wang, "A 200mhz 202.4 gflops@ 10.8 w vgg16 accelerator in xilinx vx690t," in *2017 IEEE Global Conference on Signal and Information Processing (GlobalSIP)*, IEEE, 2017, pp. 784–788.

[158] P. Meloni, A. Capotondi, G. Deriu, *et al.*, "Neuraghe: Exploiting cpu-fpga synergies for efficient and flexible cnn inference acceleration on zynq socs," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 11, no. 3, pp. 1–24, 2018.

[159] Z. Yu and C.-S. Bouganis, "Svd-nas: Coupling low-rank approximation and neural architecture search," in *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision*, 2023, pp. 1503–1512.

[160] E. Jang, S. Gu, and B. Poole, "Categorical reparameterization with gumbel-softmax," in *International Conference on Learning Representations*, 2016.

[161] H. Cai, L. Zhu, and S. Han, "Proxylessnas: Direct neural architecture search on target task and hardware," in *International Conference on Learning Representations*, 2018.

[162] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "Mobilenetv2: Inverted residuals and linear bottlenecks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 4510–4520.

[163] *GitHub - pytorch/vision: Datasets, Transforms and Models specific to Computer Vision — github.com*, `https://github.com/pytorch/vision`, [Accessed 29-02-2024].

[164] Y. Bhalgat, Y. Zhang, J. M. Lin, and F. Porikli, "Structured convolutions for efficient neural network design," *Advances in Neural Information Processing Systems*, vol. 33, pp. 5553–5564, 2020.

[165] Y. Tang, S. You, C. Xu, *et al.*, "Reborn filters: Pruning convolutional neural networks with limited data," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, 2020, pp. 5972–5980.

[166] I. Lazarevich, A. Kozlov, and N. Malinin, "Post-training deep neural network pruning via layer-wise calibration," in *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2021, pp. 798–805.

[167] H. Wang, J. Liu, X. Ma, Y. Yong, Z. Chai, and J. Wu, "Compressing models with few samples: Mimicking then replacing," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2022, pp. 701–710.

[168] M. Yin, Y. Sui, S. Liao, and B. Yuan, "Towards efficient tensor decomposition-based dnn model compression with optimization framework," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2021, pp. 10 674–10 683.

[169] A.-H. Phan, K. Sobolev, K. Sozykin, *et al.*, "Stable low-rank tensor decomposition for compression of convolutional neural network," in *European Conference on Computer Vision*, Springer, 2020, pp. 522–539.

[170] J. Gusak, M. Kholiavchenko, E. Ponomarev, *et al.*, "Automated multi-stage compression of neural networks," in *Proceedings of the IEEE/CVF International Conference on Computer Vision Workshops*, 2019, pp. 0–0.

[171] Y. Xu, Y. Li, S. Zhang, *et al.*, "Trained rank pruning for efficient deep neural networks," in *2019 Fifth Workshop on Energy Efficient Machine Learning and Cognitive Computing-NeurIPS Edition (EMC2-NIPS)*, IEEE, 2019, pp. 14–17.

[172] W. Kang and D. Kim, "Deeply shared filter bases for parameter-efficient convolutional neural networks," *Advances in Neural Information Processing Systems*, vol. 34, 2021.

[173] L. L. Zhang, S. Han, J. Wei, *et al.*, "Nn-meter: Towards accurate latency prediction of deep-learning model inference on diverse edge devices," in *Proceedings of the 19th Annual International Conference on Mobile Systems, Applications, and Services*, 2021, pp. 81–93.

[174] *Performance measurement | TensorFlow Lite — tensorflow.org*, `https://www.tensorflow.org/lite/performance/measurement`, [Accessed 29-02-2024].

[175] A. Montgomerie-Corcoran, Z. Yu, and C.-S. Bouganis, "Samo: Optimised mapping of convolutional neural networks to streaming architectures," in *2022 32nd International Conference on Field-Programmable Logic and Applications (FPL)*, IEEE, 2022, pp. 418–424.

[176] Z. Yu and C.-S. Bouganis, "Mixed-td: Efficient neural network accelerator with layer-specific tensor decomposition," *arXiv preprint: 2306.05021*, 2023.

[177] L. Liu and S. Brown, "Leveraging fine-grained structured sparsity for cnn inference on systolic array architectures," in *2021 31st International Conference on Field-Programmable Logic and Applications (FPL)*, IEEE, 2021, pp. 301–305.

[178] J. Meng, S. K. Venkataramanaiah, C. Zhou, P. Hansen, P. Whatmough, and J.-s. Seo, "Fixyfpga: Efficient fpga accelerator for deep neural networks with high element-wise sparsity and without external memory access," in *2021 31st International Conference on Field-Programmable Logic and Applications (FPL)*, IEEE, 2021, pp. 9–16.

[179] S. Rasoulinezhad, H. Zhou, L. Wang, and P. H. Leong, "Pir-dsp: An fpga dsp block architecture for multi-precision deep neural networks," in *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, IEEE, 2019, pp. 35–44.

[180] C. Latotzke, T. Ciesielski, and T. Gemmeke, "Design of high-throughput mixed-precision cnn accelerators on fpga," in *2022 32nd International Conference on Field-Programmable Logic and Applications (FPL)*, IEEE, 2022, pp. 358–365.

[181] R. DiCecco, G. Lacey, J. Vasiljevic, P. Chow, G. Taylor, and S. Areibi, "Caffeinated fpgas: Fpga framework for convolutional neural networks," in *2016 International Conference on Field-Programmable Technology (FPT)*, IEEE, 2016.

[182] I. V. Oseledets, "Tensor-train decomposition," *SIAM Journal on Scientific Computing*, vol. 33, no. 5, pp. 2295–2317, 2011.

[183] M. Motamedi, P. Gysel, V. Akella, and S. Ghiasi, "Design space exploration of fpga-based deep convolutional neural networks," in *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*, IEEE, 2016, pp. 575–580.

[184] C. R. Reeves, Ed., *Modern Heuristic Techniques for Combinatorial Problems*. USA: John Wiley & Sons, Inc., 1993, ISBN: 0470220791.

[185] X. Dai, P. Zhang, B. Wu, *et al.*, "Chamnet: Towards efficient network design through platform-aware model adaptation," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2019, pp. 11 398–11 407.

[186] L. Dudziak, T. Chau, M. Abdelfattah, R. Lee, H. Kim, and N. Lane, "Brp-nas: Prediction-based nas using gcns," *Advances in Neural Information Processing Systems*, vol. 33, pp. 10 480–10 490, 2020.

[187] L. Shannon, V. Cojocaru, C. N. Dao, and P. H. Leong, "Technology scaling in fpgas: Trends in applications and architectures," in *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*, IEEE, 2015, pp. 1–8.

[188] M. Hall and V. Betz, "Hpipe: Heterogeneous layer-pipelined and sparse-aware cnn inference for fpgas," *arXiv preprint: 2007.10451*, 2020.

[189] Y. Gong, Z. Xu, Z. He, *et al.*, "N3h-core: Neuron-designed neural network accelerator via fpga-based heterogeneous computing cores," in *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2022, pp. 112–122.

[190] Z. Yu and C.-S. Bouganis, "Autows: Automate weights streaming in layer-wise pipelined dnn accelerators," *arXiv preprint: 2311.04764*, 2023.

[191] S. I. Venieris and C.-S. Bouganis, "Fpgaconvnet: A toolflow for mapping diverse convolutional neural networks on embedded fpgas," *arXiv preprint: 1711.08740*, 2017.

[192] S.-E. Chang, Y. Li, M. Sun, *et al.*, "Mix and match: A novel fpga-centric deep neural network quantization framework," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, IEEE, 2021, pp. 208–220.

[193] *The challenge of batch size 1*, [Accessed 30-09-2024], 2020. [Online]. Available: `https://groq.com/wp-content/uploads/2020/05/GROQP002_V2.2.pdf`.

[194] G. Jocher, *ultralytics/yolov5: v3.1 - Bug Fixes and Performance Improvements*, `https://github.com/ultralytics/yolov5`, version v3.1, Oct. 2020. DOI: `10.5281/zenodo.4154370`. [Online]. Available: `https://doi.org/10.5281/zenodo.4154370`.

[195] D. Qin, C. Leichner, M. Delakis, *et al.*, "Mobilenetv4-universal models for the mobile ecosystem," *arXiv preprint arXiv:2404.10518*, 2024.