

Joint Optimization of Algorithms, Hardware, and Systems for Efficient Deep Neural
Networks

by

Shiyu Li

Department of Electrical and Computer Engineering
Duke University

Defense Date: April 2, 2024

Approved:

Yiran Chen, Supervisor

Hai Li, Co-Chair

James Morizio

Daniel J. Sorin

Lisa Wu Wills

Danyang Zhuo

Dissertation submitted in partial fulfillment of the requirements for the degree of
Doctor of Philosophy in the Department of Electrical and Computer Engineering
in the Graduate School of Duke University

2024

ABSTRACT

Joint Optimization of Algorithms, Hardware, and Systems for Efficient Deep Neural Networks

by

Shiyu Li

Department of Electrical and Computer Engineering
Duke University

Defense Date: April 2, 2024

Approved:

Yiran Chen, Supervisor

Hai Li, Co-Chair

James Morizio

Daniel J. Sorin

Lisa Wu Wills

Danyang Zhuo

An abstract of a dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Electrical and Computer Engineering
in the Graduate School of Duke University
2024

Copyright © 2024 by
Shiyu Li
All rights reserved

Abstract

Deep learning has enabled remarkable performance breakthroughs across various domains, including computer vision, natural language processing, and recommender systems. However, the typical deep neural network (DNN) models employed in these applications require millions of parameters and billions of operations, leading to substantial computational and memory requirements. While researchers have proposed compression methods, optimized frameworks, and specialized accelerators to improve efficiency, outstanding challenges persist, limiting the achievable gains.

A fundamental challenge lies in the inherent irregularity and sparsity of DNNs. Although these models exhibit significant sparsity, with a considerable fraction of weights and activations being zero or near-zero values, exploiting this sparsity efficiently on modern hardware is problematic due to the irregular distribution of non-zero elements. This irregularity leads to substantial overhead in indexing, gathering, and processing sparse data, resulting in poor utilization of computational and memory resources. Furthermore, recent research has identified a significant gap between the theoretical and practical improvements achieved by compression methods. Additionally, emerging DNN architectures with novel operators often nullify previous optimization efforts in software frameworks and hardware accelerators, necessitating continuous adaptation.

To address these critical challenges, this dissertation targets building a holistic approach that jointly optimizes algorithms, hardware architectures, and system designs to enable efficient deployment of DNNs in the presence of irregularity and sparsity. On the algorithm level, a novel hardware-friendly compression method based on matrix decomposition is proposed. The original convolutional kernels are decomposed into common basis kernels and a series of coefficients, with conventional pruning applied to the coefficients. This compressed DNN forms a hardware-friendly structure where the sparsity pattern is shared across input feature map pixels, alleviating sparse pattern processing costs.

On the hardware level, a novel sparse DNN accelerator is introduced to support the inference of the compressed DNN. Low-precision quantization is applied to sparse coeffi-

icients, and high-precision to basis kernels. By involving only low-precision coefficients in sparse processing, the hardware efficiently matches non-zero weights and activations using inverted butterfly networks. The shared basis kernels and sparse coefficients significantly reduce buffer size and bandwidth requirements, boosting performance and energy efficiency.

At the system level, a near-data processing framework is proposed to address the challenge of training large DNN-based recommendation models. This framework adopts computational storage devices and coherent system interconnects to partition the model into subtasks. Data-intensive embedding operations run on computational storage devices with customized memory hierarchies, while compute-intensive feature processing and aggregation operations are assigned to GPUs for maximum efficiency. This framework enables training large DNN-based recommendation models without expensive hardware investments.

Through joint optimization across algorithms, hardware architectures, and system designs, this research aims to overcome the limitations imposed by irregularity and sparsity, enabling efficient deployment of DNNs in a broad range of applications and resource-constrained environments. By addressing these critical issues, this work paves the way for fully harnessing the potential of deep learning technologies in practical settings.

Contents

| | |
|---|-----|
| Abstract | iv |
| List of Tables | x |
| List of Figures | xi |
| Acknowledgements | xiv |
| 1 Introduction | 1 |
| 1.1 Key Terms and Notations | 3 |
| 1.2 Sparsity in DNN Models | 4 |
| 1.3 Irregularity in DNN Computation | 7 |
| 1.4 The Necessity of Joint Optimization | 9 |
| 2 Hardware Friendly DNN Compression Algorithm | 12 |
| 2.1 Background and Related Works | 12 |
| 2.1.1 Compact DNN Model Design | 13 |
| 2.1.2 Low-Rank Approximation | 13 |
| 2.1.3 Model Pruning | 14 |
| 2.2 PENNI Framework [1] | 15 |
| 2.2.1 Filter Decomposition | 16 |
| 2.2.2 Retraining | 18 |
| 2.2.3 Model Shrinking | 19 |
| 2.2.4 Hardware Benefits | 21 |
| 2.2.5 Evaluations | 23 |
| 2.3 ESCALATE Algorithm [2] | 32 |
| 2.3.1 Computation Reorganization | 33 |
| 2.3.2 Hybrid Quantization | 35 |
| 2.3.3 Decomposing the Compact Model | 37 |
| 2.3.4 Evaluation | 37 |

| | | |
|-------|--|----|
| 2.4 | Conclusion | 41 |
| 3 | Sparse DNN Accelerator with Kernel Decomposition | 42 |
| 3.1 | Background and Related Works | 43 |
| 3.2 | Motivation | 44 |
| 3.3 | ESCALATE Architecture Overview | 45 |
| 3.4 | Basis-First Dataflow | 47 |
| 3.5 | Dilution-Concentration | 49 |
| 3.5.1 | Sparse Encoding | 49 |
| 3.5.2 | Dilution | 51 |
| 3.5.3 | Concentration | 53 |
| 3.6 | Buffer Design | 55 |
| 3.7 | Evaluation | 56 |
| 3.7.1 | Experiment Settings | 57 |
| 3.7.2 | Main Result | 59 |
| 3.7.3 | Layer-wise Analysis | 61 |
| 3.8 | Discussion | 64 |
| 3.8.1 | Design Trade-off | 64 |
| 3.8.2 | Overhead Analysis | 64 |
| 3.8.3 | Modern Compact CNNs | 65 |
| 3.9 | Conclusion | 66 |
| 4 | Near-Data Processing System for Large-Scale DNN-based Recommendation Model Training | 67 |
| 4.1 | Background and Related Works | 67 |
| 4.1.1 | DNN-based Recommendation Model | 67 |
| 4.1.2 | Computational Storage | 69 |
| 4.1.3 | Coherent Interconnect | 69 |
| 4.1.4 | CXL-enabled SSD | 70 |

| | | |
|-------|--------------------------------------|-----|
| 4.1.5 | Accelerating DLRM Training | 71 |
| 4.2 | Motivation | 72 |
| 4.3 | NDRec System | 74 |
| 4.3.1 | System Architecture | 74 |
| 4.3.2 | Lookahead Embedding | 76 |
| 4.3.3 | Task Coordination | 79 |
| 4.3.4 | Embedding Table Placement | 81 |
| 4.3.5 | Communication | 82 |
| 4.4 | Near-Storage Embedding | 83 |
| 4.4.1 | Embedding Cache Design | 83 |
| 4.4.2 | Embedding Kernel | 87 |
| 4.5 | Evaluation | 88 |
| 4.5.1 | Experiment Settings | 88 |
| 4.5.2 | End-to-End Results | 91 |
| 4.5.3 | Embedding Cache | 93 |
| 4.6 | Discussion | 95 |
| 4.6.1 | Bandwidth | 95 |
| 4.6.2 | Scalability | 96 |
| 4.6.3 | Latency Breakdown | 97 |
| 4.6.4 | Energy Consumption | 98 |
| 4.6.5 | SSD Endurance | 98 |
| 4.7 | Ablation Study | 99 |
| 4.7.1 | Alternative Architecture | 99 |
| 4.7.2 | Component Contribution | 100 |
| 4.8 | Conclusion | 101 |
| 5 | Conclusion | 102 |

| | |
|------------------------|-----|
| Bibliography | 104 |
| Biography | 116 |

List of Tables

| | | |
|-----|---|----|
| 2.1 | Compression Result on CIFAR10. ‘-’ denotes unavailable data from the original paper. | 25 |
| 2.2 | Compression Result of AlexNet on ImageNet. | 26 |
| 2.3 | Compression Result of ResNet50 on ImageNet. | 26 |
| 2.4 | Measured inference performance of VGG16-CIFAR10 on different devices. | 27 |
| 2.5 | Detailed Structure of the compressed VGG16-CIFAR10 | 31 |
| 2.6 | Compression result of ESCALATE algorithm. | 39 |
| 3.1 | Configurations of ESCALATE and baselines. | 56 |
| 3.2 | Unit energy cost per 8-bit integer operation extracted from a commercial 65nm technology. | 57 |
| 3.3 | Power and Area estimation of PE Block(65nm) | 57 |
| 4.1 | Hardware Setup for the evaluation | 90 |
| 4.2 | Model Configurations | 91 |

List of Figures

| | | |
|------|--|----|
| 1.1 | The taxonomy of sparsity in DNN models. | 5 |
| 2.1 | Four phases in PENNI framework: A. Filter Decomposition; B. Retraining with Sparsity Regularization; C. Coefficient Pruning; D. Model Shrinking. | 16 |
| 2.2 | Model shrinking procedure. | 20 |
| 2.3 | Shrinking a model containing skip connections. | 22 |
| 2.4 | The change in training and validation accuracy. The model is ResNet-50 with parameter setting R4. | 28 |
| 2.5 | Test accuracy, parameters, and computation reduction with different numbers of basis kernel d | 29 |
| 2.6 | Layer width after the model shrinking. | 30 |
| 2.7 | The computation process of one output feature map of the reorganized decomposed convolution. For simplicity, we assume $d = 3$ here. | 34 |
| 2.8 | The comparison of model size and accuracy with uniform, hybrid and basis-only quantization. | 40 |
| 3.1 | The over view of <i>ESCALATE</i> accelerator. | 46 |
| 3.2 | The Basis-First Dataflow. | 48 |
| 3.3 | The illustration of sparse encoding and bit-gather operation. | 50 |
| 3.4 | Dilution process. For simplicity, we omit the higher 4 bits during the process in the figure. | 51 |
| 3.5 | Rolling mask and implicit barrier. | 52 |
| 3.6 | Concentration through look-ahead and look-aside. | 54 |
| 3.7 | The design of input buffer. Each input buffer is implemented as a circular queue and uses the reference count to evict the finished chunks. | 55 |
| 3.8 | The normalized speedup and energy efficiency (<i>over Eyeriss</i>) of <i>ESCALATE</i> and baseline accelerators. | 58 |
| 3.9 | The normalized DRAM accesses (<i>relative to ESCALATE</i>) of baseline accelerators on all evaluated models. | 60 |
| 3.10 | The inference energy breakdown of all evaluated models. We omit the output buffer since its energy consumption is negligible. | 60 |

| | | |
|------|---|----|
| 3.11 | Layerwise sparsity and speedup over dense accelerator (Eyeriss) in ResNet-18 model. | 63 |
| 3.12 | The accuracy and latency/energy trade-off with different numbers of basis kernels (d). | 64 |
| 3.13 | The MAC idle cycles and sparsity of each layer in MobileNet. | 65 |
| 4.1 | <i>(left)</i> The general architecture of DLRM. <i>(right)</i> An example of the embedding operation. | 68 |
| 4.2 | SmartSSD Architecture | 69 |
| 4.3 | The architecture of the NDRec system and the allocation of the host physical Address Space (HPA). | 74 |
| 4.4 | The illustration of lookahead embedding assuming there are two samples per batch. | 76 |
| 4.5 | The illustration of the training timeline of naïve offloading and NDRec. | 79 |
| 4.6 | The task coordination in NDRec system. | 79 |
| 4.7 | The operational intensity (FLOPS/DRAM byte) with different table sizes and batch sizes. | 81 |
| 4.8 | An example of the software-managed cache. | 84 |
| 4.9 | An example of the embedding kernel configuration with the embedding dimension of 64. | 87 |
| 4.10 | The simulation framework. | 89 |
| 4.11 | Normalized speedup of NDRec and baseline approaches over UVM-based GPU training on 4 GPUs. | 92 |
| 4.12 | <i>(left)</i> Hit rate and cache planning overhead; <i>(right)</i> Average data movement size and latency to SSD per iteration. | 94 |
| 4.13 | The DRAM cache hit rate with different hot-entry thresholds <i>(left)</i> and hot-entry table sizes <i>(right)</i> | 95 |
| 4.14 | Achieved aggregated bandwidth with different configurations. The dashed lines illustrate the theoretical maximum bandwidth. | 95 |
| 4.15 | Average speedup over GPU-UVM with 2K and 16K batch size and various combinations of GPUs and SmartSSDs. | 96 |
| 4.16 | The execution time breakdown of NDRec and GPU-UVM. | 97 |
| 4.17 | The average energy consumption per iteration. | 98 |

| | | |
|------|--|-----|
| 4.18 | The comparison with alternative solutions. ‘CS-Caching’ stands for the use of a caching strategy like cDLRM directly with CXL-SSD. | 99 |
| 4.19 | The analysis of the contribution of each component. | 100 |

Acknowledgements

I would like to express my sincere gratitude to those who supported me through this Ph.D. journey. First and foremost, I am deeply indebted to my advisor, Prof. Yiran Chen, for his unwavering guidance, invaluable insights, and constant support throughout my doctoral studies. He encouraged me to chase ambitious goals and guided me through the inevitable challenges. I would also like to express my gratitude to Prof. Hai Li, the co-director of CEI, who granted me the research internship opportunity that ignited my passion for pursuing a Ph.D. in computer engineering. I am grateful to my committee members for their feedback and support: Prof. Daniel Sorin, whose course introduced me to the fascinating world of computer architecture; Prof. Lisa Wills, whose course provided a curated set of classical papers and hands-on experience in accelerator design; Prof. James Morizio, from whom I gained precious knowledge of VLSI design; and Prof. Danyang Zhuo, from whom I learned a fresh perspective on systems and software for AI optimization. I am truly thankful to my collaborators – Dr. Edward T. Hanson, Yitu Wang, Dr. Qilin Zheng, Zhixu Du, and Dr. Zhiyao Xie. Their contributions and insightful feedback enriched this work immensely. I appreciate their dedication and willingness to share their knowledge and expertise. I would also like to thank my lab mates and friends for their emotional support and camaraderie, as we went through the highs and lows of this journey together. Among my friends, I would like to express my special gratitude to Dr. Hsin-Pai Cheng, for his mentorship during my research internship and first year, as well as the in-depth discussion on life and career choices. Finally, I extend my heartfelt gratitude to my family and girlfriend, who accompanied me on the other side of the Pacific Ocean with their unconditional love, encouragement, and unwavering belief in me – their support was the driving force behind my pursuit of this endeavor.

1. Introduction

The advancement of deep learning technology has ushered in a new era of artificial intelligence, unlocking unprecedented performance in a vast range of applications. Deep neural networks (DNNs), the foundational models underpinning deep learning, have demonstrated remarkable power in domains such as computer vision, natural language processing, and recommender systems. These powerful models have revolutionized how we approach complex tasks, from image and speech recognition to language translation and content personalization.

However, the remarkable capabilities of DNNs come at a significant cost. These models typically require millions of parameters and billions of floating-point operations, leading to substantial computational and memory requirements. The large storage and computational overhead associated with DNNs pose a significant challenge, hindering their widespread deployment, especially in resource-constrained environments such as mobile devices, embedded systems, and edge computing platforms.

To address this challenge, researchers in both the algorithm and hardware communities have dedicated significant efforts to accelerate the inference and training of DNN models. On the algorithmic front, various compression methods have been proposed to identify and remove redundancies within the DNNs, thereby reducing their memory footprint and computational complexity. These techniques include pruning, quantization, and low-rank approximations, among others, each offering trade-offs between compression ratio, accuracy, and computational efficiency. In parallel, the hardware community has focused on developing highly optimized software frameworks and specialized acceleration hardware tailored to the unique computational patterns of DNNs. These efforts have yielded optimized libraries as well as dedicated DNN accelerators. These hardware and software solutions aim to maximize the utilization of available computational resources, thereby improving performance and energy efficiency.

Despite these advancements, outstanding challenges persist, limiting the achievable efficiency gains. Recent research has highlighted a significant gap between the theoretical

and practical improvements offered by compression methods, underscoring the need for a more holistic approach that considers hardware and system-level constraints. Moreover, the rapid evolution of DNN architectures, with the introduction of novel operators and paradigms, often nullifies previous optimization efforts, necessitating continuous adaptation and co-design across algorithmic, hardware, and system levels.

In this context, this dissertation endeavors to push the efficiency frontier of DNN models through a comprehensive joint optimization strategy that spans algorithms, hardware architectures, and system designs. The organization of this dissertation is as follows:

Chapter 1 presents an overview of the DNN efficiency issue. We discuss the sources of sparsity and irregularity in DNN models and their impact on system efficiency. An analysis of the necessity of performing joint optimization is also provided to establish the motivation for this dissertation research.

Chapter 2 focuses on algorithm-level optimization. We discuss the potential impact of different DNN compression methods on hardware efficiency. We then propose a hardware-friendly DNN compression method, PENNI [1]. PENNI decomposes original convolutional kernels into common basis kernels and a series of coefficients, with magnitude-based pruning applied to the coefficients. This compressed DNN forms a hardware-friendly structure where the sparsity pattern is shared across input feature map pixels, alleviating sparse pattern processing costs. We then present the ESCALATE algorithm [2], which extends PENNI for the sparse accelerator. We analyze the bottleneck of the compressed model produced by PENNI and reformulate the operator for the efficiency on the customized accelerators.

Chapter 3 focuses on the hardware architecture of a sparse DNN accelerator. We discuss the challenges and opportunities of supporting compressed DNNs on hardware. We then propose a sparse DNN accelerator design, ESCALATE [2]. ESCALATE accelerator utilizes the unique structure of the DNN model compressed by PENNI to boost throughput and reduce the accelerator’s resource consumption.

Chapter 4 focuses on the system design for large-scale DNN training. We discuss the bottlenecks and opportunities in existing recommendation model training systems. We then

propose our system design, NDRec [3]. NDRec offloads the memory-intensive embedding operator to computational storage devices and enables concurrent execution of embedding and other stages.

Chapter 5 summarizes the lessons learned from each section and briefly discusses future research directions.

1.1 Key Terms and Notations

Before delving into the technical details, it is crucial to review the terms and notations employed in this thesis. We use the term Deep Neural Network (**DNN**) to refer to a family of neural network models, including Multi-Layer Perceptron (**MLP**), Convolutional Neural Network (**CNN**), Deep Learning Recommendation Model (**DLRM**), and Transformers. Each DNN model consists of multiple layers. A **layer** is defined as an operator that converts a set of inputs into outputs through a specific mathematical transformation. We use the term **activations** to refer to the input or output of each layer, except for the input to the first layer, which is denoted as the **input sample**.

Specific to the CNN model, we employ **width**, **height**, and **channel** to denote the three dimensions of an activation or input sample. The width and height correspond to the spatial dimensions of the input sample (e.g., the pixel dimensions of an image), while the channel corresponds to the remaining dimension (e.g., the color channels of an RGB image input sample). We denote the activations used for input as the input feature map (**IFM**) and the activations produced by a layer as the output feature map (**OFM**). The term **kernels** refers to 2D slices of the convolution weight, which are applied across the input feature map to produce the output feature map. We use **W**, **H**, and **C** to denote the width, height, and channels of the IFM, respectively. **R** and **S** represent the width and height of the convolutional kernel. The number of output channels is represented as **K**, while the width and height of the OFM are denoted as **X** and **Y**. With these notations, the computation of

one element in the k-th output channel can be represented as:

$$OFM_k^{(w+r/2, h+s/2)} = \sum_{c=1}^C \sum_{r=1}^R \sum_{s=1}^S W^{(k,c,r,s)} IFM^{(c,w+r,h+s)}, \quad (1.1)$$

where $W \in \mathbb{R}^{K \times C \times R \times S}$ is the weight tensor, $IFM \in \mathbb{R}^{C \times W \times H}$ and $OFM \in \mathbb{R}^{K \times X \times Y}$ are the input and output feature maps, respectively.

Specific to the DLRM model, we use **query** to denote the input of the model. The **sparse features** is used to denote the categorical part of the query. We use **sparse index** to refer to one element in the sparse feature to avoid confusion with the term ‘index’ used in memory system design. Embedding Tables (**EMBs**) are the weight of the embedding layer. Each EMB corresponds to one sparse feature. The Embedding Vector (**EV**) denotes one row in the EMB, which corresponds to one category of the corresponding feature. The embedding operation retrieves the EVs from EMB according to the sparse indices and aggregates these EVs into a single vector. The aggregation process is named **pooling** and can be implemented with different operators (e.g., sum, product, weighted sum). We use Embedding Output (**EO**) to denote the vector produced by the embedding operation.

1.2 Sparsity in DNN Models

While deep neural networks (DNNs) are typically designed with dense architectures, a significant amount of inherent sparsity can be exploited within these models. The core computation of DNNs involves numerous multiply-and-accumulate (MAC) operations between weights and activations. If a weight or activation element has a low magnitude, the resulting multiplication produces a small value that has a negligible impact on the accumulation result. By identifying and omitting these low-magnitude weights and activations during computation, the overall computational and storage demands of DNN models can be reduced through pruning. Remarkably, empirical study [4] have demonstrated that DNNs exhibit a high degree of robustness to such pruning techniques, with properly fine-tuned pruned models able to maintain the original accuracy levels.

Figure 1.1 provides a taxonomy of the different forms of sparsity that can be leveraged

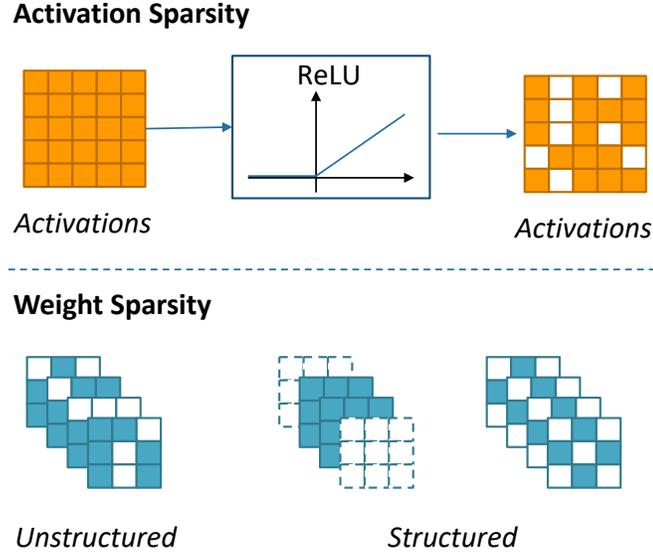


FIGURE 1.1: The taxonomy of sparsity in DNN models.

in DNN models. Sparsity can manifest in the weight parameters, the activations, or a combination of both. **Weight sparsity** is an offline property induced by pruning the pre-trained model weights. The resulting sparsity pattern in the weights is static and fixed for a given model during inference. In contrast, **activation sparsity** is a dynamic characteristic produced by the activation functions within the network at runtime. For instance, the widely used ReLU activation function clamps negative values to zero, inherently introducing sparsity in the activations. We may also gate the activations at runtime according to a predefined threshold. Consequently, the sparsity pattern in activations varies across different input samples.

Of the two forms, weight sparsity is preferred in most DNN compression frameworks due to its static nature and the advantages it confers. After pruning, additional steps such as retraining, fine-tuning, and regularization can be employed to induce sparsity and recover accuracy. While these steps require access to the original training data and incur high computational costs, they need to be performed only once offline. The resulting pruned weights remain static during inference, enabling optimizations in software and hardware to fully utilize the sparsity pattern, thereby improving runtime efficiency. Crucially, the

offline nature of weight pruning allows model developers to derive a set of weights with different compression ratios. This set can then be leveraged during model deployment, where the appropriate weights are selected according to accuracy and latency requirements, potentially even at runtime based on the input sample. Although weight sparsity incurs additional training costs, it offers flexibility to developers and users, making it a more attractive option to pursue. Thus, in this dissertation, we focus on the optimization of the algorithm, hardware, and system surrounding the weight sparsity.

In contrast, activation sparsity imposes significant constraints. Since the sparsity pattern differs for each input sample, pruning must be performed at runtime, introducing additional latency during inference. If the pruning method is complicated, the overhead could potentially offset the benefits of sparsity. Moreover, due to latency constraints, it is impossible to adjust the model weights according to the pruning result, leading to higher accuracy loss compared to weight pruning. Given these rigid constraints, in this dissertation, we simply utilize the activation sparsity induced by the activation function itself, without further investigating the runtime pruning strategy.

Furthermore, sparsity can be categorized as structured or unstructured based on the underlying pattern. **Structured sparsity** adheres to a specific, pre-defined pattern (e.g., four zeros per eight elements), rendering the positions of the zero elements predictable. This predictability can be leveraged for efficient computation and storage. For instance, if the pruning is performed on the channel granularity, the pruned channel can be directly removed from the model, resulting in a compact but dense model. The compressed model can enjoy the benefit of reduced computation and storage without any modifications to the underlying hardware or software.

Structured sparsity can be induced through various techniques, such as pruning with regularization or architectural constraints. Regularization methods like group lasso [5] encourage weights to be pruned in a structured manner, promoting sparsity at the level of filters, channels, or even entire layers. Alternatively, architectural constraints can be imposed by design, forcing specific weight patterns to be zero. For example, we may prune

the entire group of weights if we know that they will be computed in parallel in hardware [6].

In contrast, **unstructured sparsity** imposes no constraints on the sparsity pattern, leading to a random and unpredictable distribution of zero elements. While unstructured sparsity is more challenging to exploit, it typically provides a higher sparsity ratio compared with structured sparsity, given the same accuracy loss tolerance. This is because structured sparsity inherently limits the maximum achievable sparsity due to the imposed patterns.

Exploiting unstructured sparsity often requires more sophisticated techniques, such as sparse encoding or dynamic sparse computation. Sparse encoding involves compressing the non-zero weights and their indices into a compact format, enabling efficient storage and data transfers. Dynamic sparse computation, on the other hand, involves skipping computations involving zero-valued weights and activations at runtime, requiring specialized hardware or software implementations.

The choice between structured and unstructured sparsity depends on various factors, including the target hardware platform, performance requirements, and acceptable trade-offs between sparsity ratio and computational complexity. In some cases, a hybrid approach combining both forms of sparsity may be advantageous, leveraging the benefits of structured sparsity while achieving higher overall sparsity through unstructured pruning.

1.3 Irregularity in DNN Computation

Traditional approaches to accelerating deep neural networks (DNNs) have primarily focused on optimizing the computation efficiency of dense models. Techniques such as maximizing data reuse and minimizing the memory footprint of intermediate results have been explored extensively. The underlying computation in dense DNNs is relatively regular, often mapping well to established problems like matrix multiplications. However, the landscape of DNN models has evolved, introducing inherent irregularities that challenge conventional optimization strategies.

One such source of irregularity is the exploitation of sparsity within DNN models. As

discussed earlier, leveraging sparsity in weights and activations can potentially reduce storage and computational costs. However, realizing these benefits requires skipping computations involving zero values and storing non-zero elements in a compressed format. While these optimizations seem straightforward, they can inadvertently lead to lower hardware utilization and introduce new inefficiencies. Skipping computations with zero operands necessitates identifying pairs of non-zero elements from the inputs. This additional operation requires writing special software kernels on CPU or GPU or building special hardware units on customized accelerators. Both solutions need to implement an operator similar to join or merge-join, which is challenging to parallelize efficiently. Thus, this additional step of non-zero pair identification may become the new bottleneck of the system, offsetting the benefits of sparsity. Furthermore, if the sparsity pattern is unstructured, the number and distribution of non-zero pairs become unpredictable, leading to an imbalanced workload distribution across temporal or spatial dimensions. This imbalance can hinder efficient parallelization and resource utilization on hardware accelerators, resulting in underutilized computational resources and suboptimal performance. Storing weights and activations in a compressed format also presents challenges. Compression typically involves encoding the indices of non-zero elements, requiring decoding during computation. This decoding process often relies on inefficient indirect memory accesses, which can diminish the benefits of caching and locality optimizations in the memory hierarchy.

To make it worse, the sparsity ratio could vary a lot across different layers in the DNN model. For example, as we shall show later in our work, the shallow layer of the DNN model shows a relatively low sparsity while the deeper layer of the DNN model demonstrates a very high sparsity ratio (99% for some models). The discrepancy imposes further challenges to the hardware design. To maximize the efficiency of the whole model, the accelerator needs to be efficient for both high and low sparsity. To optimize for high sparsity, the accelerators typically build smaller compute arrays, more units for identifying the non-zero pairs, and a deeper pipeline. This optimization introduces overhead when the sparsity is relatively low since it reduces the parallelism of the computation part.

Moreover, recent innovations in DNN architectures have introduced novel operators that exhibit inherently irregular computation patterns. A prominent example is the embedding layer found in deep learning recommendation models (DLRMs). In these models, the embedding layer involves accessing a large embedding table (typically hundreds of GBs to TBs in size) based on input indices. These accesses are random and lack spatial locality, effectively nullifying the advantages of traditional caching techniques designed for regular access patterns. Consequently, the embedding layer can become a significant bottleneck, limiting the overall performance of the model.

These irregularities pose significant challenges for traditional hardware architectures and software optimization techniques, which are typically designed with the assumption of regular and predictable computation patterns. Efficiently accelerating sparse and irregular DNNs requires a holistic approach that addresses both the algorithmic and hardware aspects of these models. On the algorithmic front, innovative techniques are needed to restructure and regularize the computation patterns, potentially trading off accuracy for increased regularity and efficiency. From a hardware perspective, novel architectures and specialized accelerators are required to handle the irregularities inherent in sparse and irregular DNNs. These architectures may incorporate dedicated hardware units for efficient sparse computation, irregular memory access patterns, and dynamic load balancing mechanisms. Additionally, co-design approaches that tightly couple hardware and software optimizations can unlock further performance gains.

1.4 The Necessity of Joint Optimization

Previous research efforts typically focus on optimizing one perspective of the system stack, leading to missed opportunities for holistic performance gains. For instance, algorithm development primarily focuses on improving metrics like compression ratio or accuracy, while hardware designers aim for high throughput and energy efficiency without considering the implications of the chosen algorithms. This siloed design paradigm can result in a substantial gap between the theoretical benefits promised by compression methods

and their actual realized gains on real hardware. A striking example of this disconnect is the case of a $25.5\times$ compressed AlexNet model, which achieved only around $3.9\times$ speedup on state-of-the-art sparse accelerators, as demonstrated in ADMM-NN [7]. This substantial performance gap stemmed from the failure to account for hardware constraints during the design of the compression algorithm.

To bridge this divide and unlock the full potential of compression techniques, a holistic approach that jointly optimizes algorithms, hardware, and system-level design is imperative. At the algorithm level, the compression method should be designed with a deep understanding of the target hardware constraints and characteristics. Instead of solely focusing on achieving the highest theoretical compression ratio, the algorithm should aim to maximize the achievable hardware efficiency by carefully considering factors such as memory access patterns, computational complexity, and data dependencies. This hardware-aware algorithm design can lead to more efficient and practical implementations.

At the hardware level, the accelerator design should carefully select the candidate workloads to support based on a thorough analysis of the compression algorithms' computational and memory access patterns. The hardware architecture should be tailored to fully utilize and accelerate the specific operations and data movements exposed by the compression algorithms, rather than adopting a one-size-fits-all approach. Co-design of algorithms and hardware can unlock significant performance and energy efficiency gains.

Moreover, at the system level, the scheduling and resource management mechanisms should be designed to have a deep understanding of the finer-grained details of both the compression algorithms and the underlying hardware accelerators. This level of visibility and awareness can enable more intelligent and flexible scheduling decisions, allowing for better load balancing, resource allocation, and overall system-level optimization. The scheduler should be able to dynamically adapt to the varying computational and memory access patterns of different compression algorithms, ensuring efficient utilization of the available hardware resources.

By adopting this holistic approach, where algorithms, hardware, and systems are jointly

optimized and co-designed, the true potential of compression techniques can be unlocked. This synergistic collaboration between different levels of the design stack can lead to significant performance improvements, energy efficiency gains, and ultimately, a more effective and practical deployment of DNNs in real-world applications.

2. Hardware Friendly DNN Compression Algorithm

Increasing the model size of DNN to improve accuracy also leads to substantial computational and memory requirements during training and inference. While model compression techniques like weight quantization, low-rank approximation, and pruning have been proposed to reduce model size and induce sparsity, current approaches fail to fully leverage the resulting sparsity patterns for efficient hardware implementation. Unstructured pruning methods achieve high sparsity by removing individual unimportant weights, but the irregular distribution of pruned weights makes it challenging to represent the compressed model efficiently in memory. Structured pruning removes entire filters or layers, enabling more predictable sparsity patterns amenable to hardware acceleration. However, simply removing these structures is insufficient to realize the full potential hardware efficiency gains, as conventional hardware still performs unnecessary data transfers and computations on the pruned parameters. To bridge this gap between model compression and realized hardware performance, a holistic approach that co-designs the pruning algorithm with the underlying hardware representation and computational patterns is required. This chapter presents PENNI [1], a framework that decomposes CNN models into compact sets of basis kernels and coefficient matrices, and its extension ESCALATE algorithm [2], a reformulation of PENNI with a focus on the efficiency on sparse accelerators. These two compression algorithms enable high compression ratios through structured pruning while organizing the model parameters and computations in a hardware-friendly manner.

2.1 Background and Related Works

The researchers have proposed a variety of compression techniques to tackle the challenge of computational and storage overhead of DNN models. The error-resilient nature of DNN models enables the optimization to identify the redundant information/structure and approximate the original model with a more compact structure. These compression techniques can be broadly categorized into three main approaches: compact DNN model design, low-rank approximation, and model pruning.

2.1.1 Compact DNN Model Design

This approach focuses on developing resource-efficient model architectures from the ground up, aiming to reduce computational requirements and improve latency. Previous works in this area have explored various architectural innovations to achieve model compactness. Lin et al. [8] introduced the use of global average pooling and 1x1 convolutions, which have been widely adopted in later compact architectures. SqueezeNet [9] utilized these techniques to reduce the number of channels and remove fully-connected layers. A similar idea of using efficient building blocks appeared in InceptionNet [10], where a later version [11] extended the concept by spatially separating the convolutional layers. MobileNet [12] employed depthwise separable convolutions to reduce computation cost by splitting the original convolutional layer channel-wise. Its successor, MobileNet V2 [13], further improved efficiency by adopting residual connections and introducing the inverted bottleneck module. Xie et al. [14] enhanced the expressiveness of depthwise convolutions by allowing limited connectivity within groups, while ShuffleNet [15] adopted grouped convolutions. In addition to these manually designed compact architectures, Neural Architecture Search (NAS) methods aim to automatically find architectures with optimal trade-offs between compactness and performance. Several works [16]–[19] have generated architectures that outperform manually designed ones. These compact DNN model designs have demonstrated promising results in reducing computational complexity and memory footprint, making them well-suited for resource-constrained environments such as mobile and embedded devices.

2.1.2 Low-Rank Approximation

Low-Rank Approximation (LRA) is a powerful compression technique that decomposes the original weight matrices or tensors of a DNN model into a series of low-rank matrices or low-dimensional tensors. This decomposition allows for a more compact representation of the model parameters, reducing the overall memory footprint and computational complexity. One of the earliest works in this area, by Denton et al. [20], utilized Singular Value De-

composition (SVD) to decompose the weight matrices into a product of low-rank matrices. On the other hand, Zhang et al. [21] proposed an approach that takes nonlinear activations into account, minimizing the error of the model’s response while performing the low-rank decomposition. Kim et al. [22] adopted Tucker Decomposition to compress the kernel tensors of convolutional layers, while Lebedev et al. [23] employed canonical polyadic (CP) decomposition for the same purpose. In addition to these decomposition-based methods, there have been approaches that directly train the DNN model with low-rank constraints. Wen et al. [24] and the Centripetal-SGD algorithm [25] are examples of such techniques, where the model is trained to have low-rank weight matrices from the outset. The tensor decomposition methods, such as Tucker and CP decompositions, rely on the selection of appropriate ranks for the decomposed tensors, which can be challenging and require careful tuning. On the other hand, matrix factorization methods have a limited potential for speedup since they do not explicitly consider the redundancies present in the individual weight values. Despite these challenges, LRA methods have demonstrated promising results in compressing DNN models, with the potential to achieve significant reductions in memory footprint and computational complexity.

2.1.3 Model Pruning

The idea of weight pruning dates back to the last century, with Optimal Brain Damage [26] being one of the earliest works in this area. This method proposed pruning weights based on their impact on the loss function. A later work, Optimal Brain Surgeon [27], improved upon this method by replacing the diagonal Hessian matrix with an approximated full covariance matrix. However, due to the enormous size of modern DNNs, these methods incur unacceptable computational costs, making them impractical for large-scale models. Later, Han et al. [4] introduced a pruning approach that compares the magnitude of weights against a threshold and achieves optimal results through iterative pruning and fine-tuning. Building upon this, Guo et al. [28] further improved the sparsity level by maintaining a mask instead of directly pruning the redundant weights. A later work proposes lottery

ticket hypnosis [29], indicating finding the optimal pruned network and training it from random initialization could achieve a higher compression ratio with lower accuracy loss.

Beyond unstructured pruning methods, various structured pruning methodologies have been proposed to ease the translation of sparsity into actual inference speedups. Wen et al. [5] and Yang et al. [30] apply group regularizers during the training process to obtain structured sparsity. Liu et al. [31] utilize l_1 -regularization on the scaling factors of batch normalization layers to identify insignificant channels. ThiNet [32] adopts a data-driven approach to prune channels with the smallest impact on the subsequent layer. In recent works [33]–[36], different criteria have been adopted to rank the importance of filters for pruning. Louizos et al. [37] employ stochastic gates to apply l_0 -regularization to the filters, effectively pruning them. Neural Architecture Search (NAS) methods have also incorporated filter pruning techniques [38], [39].

While structured pruning methods can directly benefit inference efficiency, their pruning granularity limits the overall compression rate or accuracy achievable for CNN models. Unstructured pruning, on the other hand, offers finer-grained control over the pruning process but may require additional techniques to translate the sparsity into actual speedups.

2.2 PENNI Framework [1]

The PENNI framework aims to compress convolutional neural networks (CNNs) by decomposing the convolution filters into a set of basis kernels and a coefficient matrix. The overall process can be divided into four phases. **Filter Decomposition:** In the first phase, the convolution filters of each layer in the CNN are decomposed into a few basis kernels and a corresponding coefficient matrix. This decomposition effectively represents each original filter as a linear combination of the basis kernels, weighted by the coefficients in the matrix. The basis kernels capture the fundamental patterns or building blocks, while the coefficient matrix encodes the contribution of each basis kernel to the original filters. **Retraining with Sparsity Regularization:** After the decomposition, the decomposed network is retrained to recover any potential accuracy loss resulting from the approximation

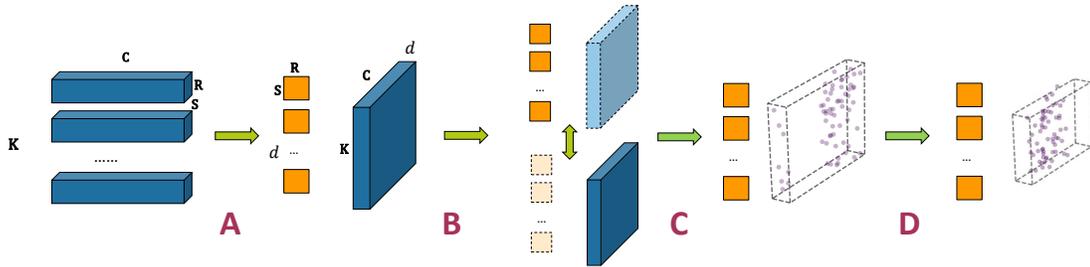


FIGURE 2.1: Four phases in PENNI framework: A. Filter Decomposition; B. Retraining with Sparsity Regularization; C. Coefficient Pruning; D. Model Shrinking.

introduced by the decomposition. During this retraining process, sparsity regularization is applied to the coefficient matrix. **Coefficient Pruning:** The redundant coefficients in the coefficient matrix are pruned based on their magnitude. Coefficients with small values are treated as insignificant and removed. **Model Shrinking:** In the final phase, the model was compacted by identifying the channels corresponding to all-zero weights. This phase finally produces a compact model.

2.2.1 Filter Decomposition

The convolution operation dominates the computation cost of CNN inference. Irregular data access and compute patterns make it extremely difficult to map the operation efficiently onto parallel hardware and further improve inference efficiency. We address this issue by reducing the number of convolution operations and offloading the irregular computation to a sequential and simple pattern.

Previous work [21] on accelerating CNN inference utilizes a low-rank assumption of output feature subspace to represent the original weight matrix with the multiplication of two low-rank matrices, thus reducing the computation required. A low-rank assumption is reasonable in this case because the number of output features is comparable to the dimension of the feature space. Recent work [25] highlights that regularization on convolutional kernels tends to promote their similarity. Building upon this observation, we contend that the low-rank assumption can also be extended to the subspaces of individual convolutional kernels. By embracing this assumption, we approximate the original convolutional layer by sharing

a small set of basis kernels and expressing the original kernels using coefficients.

Decomposition at a kernel granularity is done to obtain an approximated layer. This process applies to a single layer at a time, so the superscript l is omitted for readability. We first reshape the original weight tensor into a 2D matrix $\theta' \in \mathbb{R}^{CK \times RS}$; thus, each kernel can be seen as its row vector $w \in \mathbb{R}^{RS}$. Suppose $\mathbf{U} \subset \mathbb{R}^{RS}$ is a subspace with basis $\mathbf{B} = \{u_1, u_2, \dots, u_d\}$ where $d \leq RS$. The objective of the decomposition process is to find the subspace that minimizes the error between the projected and original vectors, shown in Equation (2.1).

$$\min_{\alpha_w \in \mathbb{R}^d} \sum_{w \in \theta'} \|w - \alpha_w \mathbf{B}^T\|^2. \quad (2.1)$$

$\mathbf{B} = [u_1 \ u_2 \ \dots \ u_d]$ is the basis matrix where each column vector is a basis of the subspace, and α_w is the coefficient vector corresponding to the row vector w . With this decomposition, the i -th channel of the output feature maps can be calculated by:

$$OFM_i^{(l)} = \sigma^{(l)} \left(\left(\sum_{j=1}^C IFM_j^{(l-1)} * (\alpha_{i,j}^{(l)} \mathbf{B}^{(l)T}) + b_j^{(l)} \right) \right), \quad (2.2)$$

where $\alpha_{i,j}^{(l)}$ is the row vector in the coefficient matrix corresponding to the j -th kernel of the i -th filter.

The decomposition problem can be formulated as the best approximation and is perfectly solved using singular value decomposition (SVD). We first obtain $\bar{\theta}'$ by subtracting each row vector with the mean vector, and then compute the covariance matrix $\mathbf{W} = \theta'^T \theta'$. Conducting SVD on \mathbf{W} , and organizing the singular value by their magnitude, we'll have:

$$\mathbf{W} = \mathbf{U} \Sigma \mathbf{V}^T. \quad (2.3)$$

The basis matrix \mathbf{B} is then derived by selecting the first d columns from matrix \mathbf{U} and obtaining the corresponding coefficients by multiplying the θ' by the projection matrix $\mathbf{B}\mathbf{B}^T$. Normally, $RS \ll CK$ and parameter matrices of a pretrained model are dense, so \mathbf{W} is a full rank matrix with the rank RS . Thus, the low-rank approximation on kernel space makes the SVD computation faster than conducting decomposition at the filter granularity.

A singular value may represent the portion of the basis vector contributing to the original vectors; but rather than selecting d based on it, we leave it as a hyper-parameter providing a trade-off between computational cost and model accuracy.

2.2.2 Retraining

Although the discussed filter decomposition scheme gives the best approximation of the original parameters in low-rank subspace, the model accuracy may greatly degrade due to the varying sensitivity of affected weights. Zhang et al. [21] address this issue by considering the non-linear block and minimizing the response error of the layer. However, innate redundancies in the models are not exploited, which limits the compression rate and the speedup. Thus, we incorporate a retraining process for twofold benefits: recovering the model accuracy and exploiting redundancy within the CNN structure through coefficient sparsity regularization. The objective of the retraining phase is to minimize the loss:

$$\mathcal{L}' = \mathcal{L}(\Theta, X', Y') + \gamma \sum_l \sum_i^{CK} \|\alpha_i^{(l)}\|_1, \quad (2.4)$$

where X' and Y' denote the input samples and their corresponding labels in the dataset (the prime is used to distinguish these notations from the symbols we used for spatial dimensions of the output feature maps), the first term is the original loss of the model (i.e., cross-entropy loss), the second term is the sum of the magnitude of the coefficients, and γ is the strength of the sparsity regularization.

When conceptualizing these two parameter sets as distinct layers, it effectively increases the model’s depth, potentially complicating its convergence during training. Thus, in the training process, we generate the reconstructed parameter $\hat{\theta}$ from the basis and coefficients and compute the gradients as the original convolutional layer. The chain rule can then be applied to derive the gradients of the basis and the coefficients from the original convolutional layer’s gradients. Specifically,

$$\frac{\partial \mathcal{L}'}{\partial \mathbf{B}} = \left(\frac{\partial \mathcal{L}'}{\partial \hat{\theta}} \right)^T \mathbf{A}, \quad \frac{\partial \mathcal{L}'}{\partial \mathbf{A}} = \frac{\partial \mathcal{L}'}{\partial \hat{\theta}} \mathbf{B}^T + \gamma, \quad (2.5)$$

where the $\hat{\theta} = \mathbf{A}\mathbf{B}^T$ and $\mathbf{A} \in \mathbb{R}^{CK \times d}$ is the coefficient matrix. Again, we omit the superscript l for readability.

The gradient of the coefficient matrix consists of two terms. The first term pushes the coefficient toward the direction that decreases the error, while the second term coerces the reconstructed kernels to be close to the basis kernels. If we jointly train the basis and coefficients, the coefficients will be updated based on the old basis and vice versa. Joint training makes it very difficult for the model to converge, producing further accuracy drop. We avoid this issue by conducting retraining in an alternating fashion, i.e., freezing the coefficients and training the basis for several epochs and then freezing the basis and training coefficients.

The decomposition approach also offers advantages in terms of sparsity regularization. Employing a non-smooth l_1 -regularizer effectively equals adding a constant to the gradient. This addition tends to dominate the gradient, particularly in later stages of training, leading to instability or failure to converge. However, by regularizing the decomposed filter state, we can circumvent this issue. Considering the gradient from the perspective of the original weights, the regularization constant is predominantly scaled due to its application solely to the coefficients. This scaling factor adjusts the constant proportionally to the diminishing gradients. Despite this adjustment, the constant remains within the same magnitude as the gradient of the loss term. Consequently, this process stabilizes the convergence towards a sparse parameter set. By mitigating the dominance of the regularization constant and aligning its impact with the diminishing gradients, the decomposition strategy effectively enhances the stability and convergence of the training process, facilitating the attainment of a sparse parameter configuration.

2.2.3 Model Shrinking

Retraining the filter-decomposed model with sparsity regularization results in predominantly near-zero coefficients. As shown in [4], we can select a threshold based on the standard deviation of each coefficient matrix and prune all weight values with a magnitude

lower than the threshold. Only a few epochs of coefficient fine-tuning are required to recover the accuracy lost by pruning. A combination of high sparsity level and low accuracy loss can be achieved without any additional iterations.

The sparse coefficients expose redundancies in CNN structures that can be utilized to shrink the model. Model shrinkage begins with reshaping the coefficient matrix $\theta^{(l)}$ into the shape $C \times K \times d$. By selecting the first dimension (i.e., the input channels) and summing the number of the non-zero elements of the remaining two dimensions, we can obtain a vector $p_i^{(l)}$ with C elements. Zeros in $p_i^{(l)}$ indicate that corresponding input channels are redundant since no output channels are connected. Indices of these channels can be represented by the set $P_i^{(l)}$. Selecting the second dimension (i.e., the output channels) and conducting the same procedure, we can get $p_o^{(l)}$ and $P_o^{(l)}$, which indicate redundant output channels. The redundancies in basis kernels can also be derived using the same procedure.

Note that it is possible for redundancies of a layer’s input and output channels not to match. We can exploit this feature by considering the connections between input channels and redundant output channels of the same layer. If some input channels only have connections to redundant output channels, these inputs consequentially become redundant. Thus, we iteratively update the redundancy sets by applying the following steps. First, we take the union of the current layer’s output channels with the next layer’s input channels,

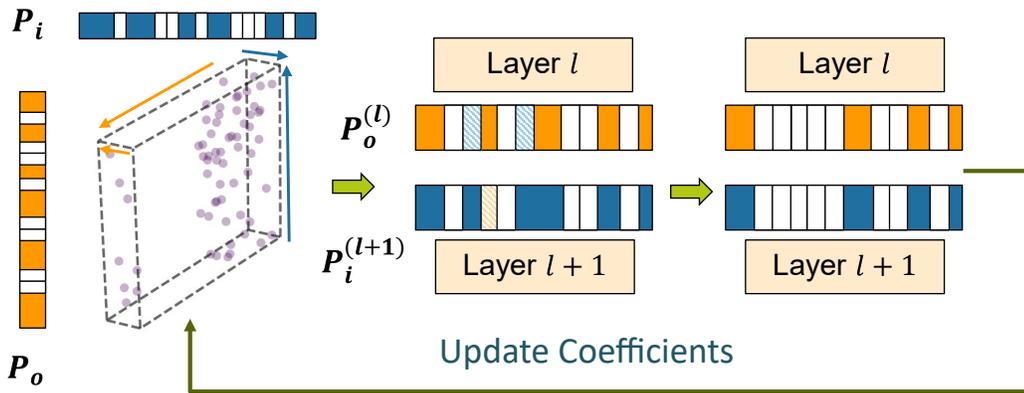


FIGURE 2.2: Model shrinking procedure.

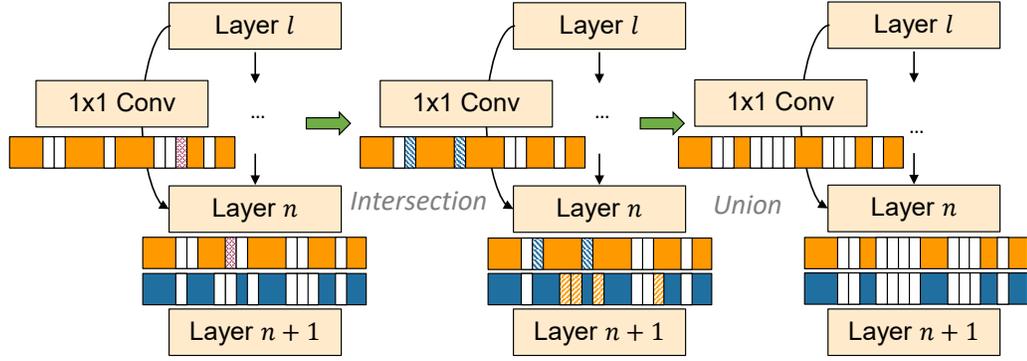
i.e., $P_o^{(l)} \leftarrow P_i^{(l+1)} \leftarrow P_i^{(l)} \cup P_o^{(l+1)}$. Then, we update $\theta^{(l)}$ by setting all corresponding coefficients in P_o and P_i to zero and deriving new redundancy vectors and sets. This procedure is repeated until no modification is made in an iteration. We illustrate the model shrinking procedure in Figure 2.2. The blank items in $P_o^{(l)}$ and $P_i^{(l+1)}$ represent the redundant channels, while the shaded items denote the difference between the two redundant sets.

An issue that may arise with the iterative update procedure for $\theta^{(l)}$ is its application to CNN architectures with skip connections, such as ResNet [40]. This issue primarily stems from the potential inconsistency in dimensions between pruned output feature maps and corresponding skip connections. However, a straightforward solution exists to mitigate this challenge. In cases where the shortcut path incorporates a dimension-matching operation, typically achieved through a 1×1 convolution, we adjust the output channel of the 1×1 convolution and the current layer by considering the intersection of their redundancy sets. This alignment ensures that the dimensions remain consistent across the skip connection. Alternatively, if the shortcut path lacks such dimension-matching operations, we must update the redundancy sets at both the start and end of the skip connection before proceeding with coefficient updates. This preemptive adjustment ensures that any potential discrepancies in dimensions are addressed before updating the coefficients, thereby maintaining the integrity of the model architecture. The model shrinking for models with skip connections are depicted in Figure 2.3. The shaded items represent the difference of the redundant sets in each step. The corresponding items will be eliminated (added) in the intersection (union) step.

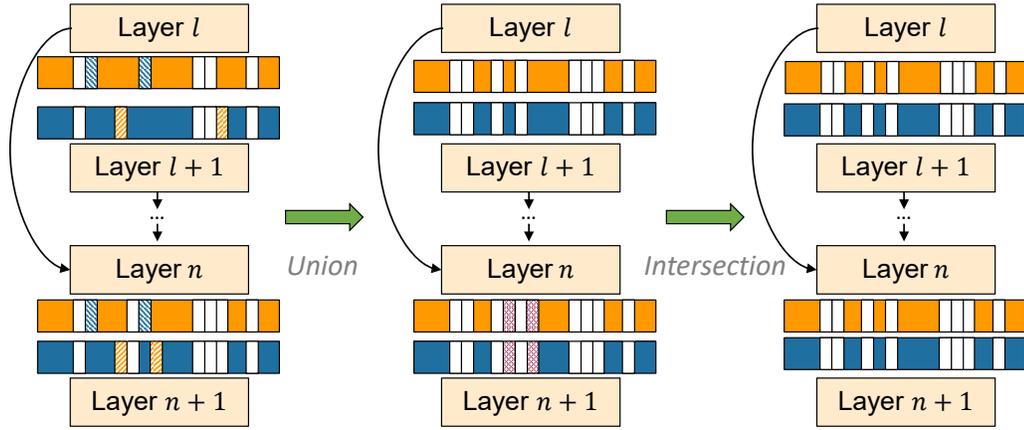
2.2.4 Hardware Benefits

The decisive advantage of PENNI over previous CNN pruning, compression, or filter decomposition methods is its potential for synergistic reduction of memory and computational footprints. PENNI directly leverages filter decomposition by enabling a partition of the convolution step into two distinct stages.

In the initial stage, PENNI conducts channel-by-channel convolutions using each of the



(a) With dimension matching component.



(b) Without dimension matching component.

FIGURE 2.3: Shrinking a model containing skip connections.

d two-dimensional basis kernels. This process results in the generation of Cd intermediate feature maps, akin to the operation of depthwise separable convolution [41]. Here, each branch of the convolution operation duplicates one of the basis kernels across the C input channels. By employing this technique, PENNI effectively reduces the computational load of the convolutional layer, particularly reducing the number of multiply-and-accumulate (MAC) operations, which traditionally represent a significant bottleneck.

The second stage is a weighted sum to produce the convolutional layer’s output feature map. Specifically, Cd intermediate feature maps are multiplied element-wise with the coefficient matrix and then accumulated at the output. As described in Section 2.2.3, the coefficient matrices are incredibly sparse; therefore, we reduce the model’s memory foot-

print and prevent redundant zero-multiply computations by representing the coefficients through a sparse matrix format. Although this stage introduces additional computations that offset the reduction in MACs from the first stage, the overall number of computations is dramatically reduced, thus improving inference latency.

Beyond the aforementioned straightforward benefits of the proposed two-stage convolutional layer scheme, PENNI also offers a unique attribute that can be leveraged for current and future hardware accelerator designs. The deterministic convolutional kernel structure means that the number of basis kernels can be altered to fit nicely with the number of processing elements (PEs) in accelerators such as DaDianNao [42] without forcing the model to conform to the hardware (e.g. reducing layer width). Meanwhile, the weighted sum stage can be computed in a streaming manner, much favored by single-instruction, multiple-data (SIMD) processors. Also, because data access patterns of convolutional layers conventionally require hardware-specific data-reuse algorithms to minimize costly cache evictions, removing interactions of the input channels at the convolution step via depthwise separation alleviates hardware complexity. Lastly, partitioning the convolution step into two stages opens the avenue for further accelerator-based throughput optimizations such as pipelining.

2.2.5 Evaluations

In this section, we demonstrate the effectiveness of the proposed framework. Experiments were held on CIFAR10 [43], and ImageNet [44] datasets. Experiment settings are detailed before comparing compression results between PENNI and both state-of-the-art channel pruning and weight pruning methods. Finally, we conduct an ablation study to show the contribution of each component in the framework.

2.2.5.1 Experiment Settings

CIFAR10. On CIFAR-10, we chose VGG16 [45], ResNet18 and ResNet56 [40] for experimentation. We use ResNet56 to test the performance of compact models. Model training involved the following data preprocessing steps: random flipping, random cropping with 4

pixels padding, and normalization. The VGG16 and ResNet18 models were first pretrained for 100 epochs with a 0.1 initial learning rate; then, the learning rate was multiplied by 0.1 at 50% and 75% epochs, while ResNet56 was pretrained for 250 epochs with the same learning rate scheduling. All pretraining, retraining, and fine-tuning procedures implemented Stochastic Gradient Descent (SGD) as the optimizer with 10^{-4} weight decay, 0.9 momentum, and batch size set to 128. We selected $d = 5$ for the decomposition stage and retrained for 100 epochs with 0.01 initial learning rate and the same scheduling. Regularization strength was set to $\gamma = 10^{-4}$. The interval between training basis and coefficients was set to 5 epochs. The final fine-tuning procedure took 30 epochs with 0.01 initial learning rate and the same scheduling scheme.

ImageNet. On ImageNet, we used AlexNet [46] and ResNet50 for the experiment, incorporating the pretrained models provided by PyTorch [47]. Since AlexNet has different kernel sizes across layers, we selected $d = 64$ and 14 for the first two convolutional layers, and $d = 5$ for the rest 3×3 convolutional layers. For ResNet50, we use 4 sets of parameter settings, with $d = 5, 6$ and regularization strength set to $5e - 5$ and $1e - 4$. The retraining procedure lasted 50 epochs with the same hyper-parameters as CIFAR10 but set batch size to 256 and cosine annealing. For AlexNet, we warmed up with a learning rate of 0.0001 for five epochs; then, the learning rate was set to 0.001 for the remaining 45 epochs. The fine-tuning procedure took 30 epochs with the learning rate set to 0.01 for ResNet50 and 0.0001 for AlexNet.

2.2.5.2 CIFAR10 Result

We selected channel pruning methods PFEC [48], Slimming [31], SFP [33], AOFPP [36], C-SGD [25], FPGM [35] and Group-HoyerSquare [30] for comparison. For the works providing parameter trade-offs, we use results with similar accuracy drops. The results are shown in Table 2.1. ‘Ours-D’ denotes the compression result with only decomposition and retraining phases, while ‘Ours-P’ incorporates all four phases. We only consider the parameters of the convolutional and linear layers, and the FLOP count is taken by calculating the number

Table 2.1: Compression Result on CIFAR10. ‘-’ denotes unavailable data from the original paper.

| Arch | Method | Base | Pruned | Δ_{Acc} | Param. | $R_{Param.}$ | FLOPs | R_{FLOPs} |
|----------|----------|--------|---------------|----------------|---------------|---------------|---------------|---------------|
| VGG16 | Baseline | 93.49% | - | - | 14.71M | - | 314.26M | - |
| | PFEC | 93.25% | 93.40% | -0.15% | 5.4M | 64% | 206M | 34.2% |
| | Slimming | 93.62% | 93.56% | -0.06% | 1.77M | 87.97% | 127M | 43.50% |
| | AOFP | 93.38% | 93.28% | -0.10% | - | - | 77M | 75.27% |
| | Ours-D | 93.49% | 93.14% | -0.35% | 183.4M | 44.44% | 183.4M | 41.65% |
| | Ours-P | 93.49% | 93.12% | -0.37% | 0.135M | 98.33% | 21.19M | 93.26% |
| ResNet18 | Baseline | 93.77% | - | - | 11.16M | - | 555.43M | - |
| | Ours-D | 93.77% | 93.89% | +0.12% | 6.28M | 56.27% | 332.34M | 40.17% |
| | Ours-P | 93.77% | 94.01% | +0.24% | 0.341M | 96.94% | 44.98M | 91.90% |
| ResNet56 | Baseline | 93.57% | - | - | 0.848M | - | 125.49M | - |
| | PFEC | 93.04% | 93.06% | +0.02% | 0.73M | 13.7% | 90.9M | 27.6% |
| | SFP | 93.59% | 93.35% | -0.24% | - | - | 59.4M | 52.67% |
| | C-SGD | 93.39% | 93.44% | +0.05% | - | - | - | 60.85% |
| | FPGM | 93.59% | 93.49% | -0.10% | - | - | 59.4M | 52.67% |
| | Group-HS | 93.14% | 93.45% | +0.31% | - | - | - | 68.43% |
| | Ours-D | 93.57% | 94.00% | +0.43% | 0.471M | 44.46% | 92.80M | 26.15% |
| | Ours-P | 93.57% | 93.38% | -0.19% | 39.37K | 95.36% | 28.98M | 79.40% |

of Multiply-Accumulation (MAC) operations. Based on the computation flow described in Section 2.2.4, we consider that the sparsity of the coefficient matrix can be converted to the reduction in FLOPs. Thus, we ignore the zeros in the coefficient matrices when counting FLOPs. On VGG16, we outperformed channel pruning methods by achieving a reduction over 98% on parameters and 93.26% on FLOPs. Although there is a slightly higher accuracy drop, it is only 0.15% behind AOFP with a 10% extra reduction on FLOPs and 0.42% behind Slimming with an almost double reduction on FLOPs, which is acceptable. Since ResNet18 was originally designed for the ImageNet dataset, no previous work has provided results for comparison. We include it in this paper to show that PENNI is able to shrink over-parameterized models and may improve accuracy. On ResNet56, which is a compact model specially tailored for CIFAR10, we can still prune 94.52% parameters and 76.9% FLOPS with a 0.2% accuracy drop. Our method outperformed previous channel pruning methods by a nearly 20% extra reduction of FLOPs and a 10% extra reduction over the group regularization method. These results underscore the effectiveness and efficiency of PENNI across various network architectures and datasets.

2.2.5.3 ImageNet Results

Table 2.2: Compression Result of AlexNet on ImageNet.

| Method | Top-1 | Top-5 | FLOPs | R_{FLOPs} |
|----------|---------------|---------------|-------------|---------------|
| Baseline | 56.51% | 79.07% | 773M | - |
| AOFP | 56.17% | 79.53% | 492M | 41.33% |
| Ours-D | 55.41% | 78.30% | 573M | 25.88% |
| Ours-P | 55.57% | 78.32% | 232M | 70.04% |

Table 2.3: Compression Result of ResNet50 on ImageNet.

| Method | Top-1 | Top-5 | FLOPs | R_{FLOPs} |
|----------------|---------------|---------------|--------------------|---------------|
| Baseline | 76.13% | 92.86% | 4.09G | - |
| Ours-D | 76.20% | 92.85% | 3.23G | 21.10% |
| ThiNet-70 | 72.02% | 90.67% | 2.58G ¹ | 36.80% |
| Ours-R1 | 73.87% | 91.79% | 220M | 94.73% |
| SFP | 74.61% | 92.06% | 2.38G ¹ | 41.80% |
| C-SGD-50 | 74.54% | 92.09% | 1.81G ¹ | 55.76% |
| Ours-R2 | 74.74% | 92.27% | 527M | 87.12% |
| C-SGD-60 | 74.93% | 92.27% | 2.20G ¹ | 46.24% |
| FPGM-40% | 74.83% | 92.32% | 1.90G ¹ | 53.50% |
| Ours-R3 | 75.00% | 92.21% | 576M | 85.92% |
| AOFP-C1 | 75.63% | 92.69% | 2.58G | 32.88% |
| AOFP-C2 | 75.11% | 92.28% | 1.66G | 56.73% |
| C-SGD-70 | 75.27% | 92.46% | 2.59G ¹ | 36.75% |
| FPGM-30% | 75.59% | 92.63% | 2.36G ¹ | 42.20% |
| Ours-R4 | 75.66% | 92.79% | 768M | 81.23% |

On ImageNet, we chose Slimming, ThiNet [32], SFP, AOFP, C-SGD, and FPGM for comparison. The result of AlexNet compression is shown in Table 2.2. We can prune 70.04% FLOPs with 1% loss on top-1 accuracy. For ResNet50, we observe the 1×1 convolutional layer of the bottleneck block as the coefficient matrix with a 1-D basis and apply regularization to it. Table 2.3 shows the result on ResNet50 compression. We use multiple parameter settings to justify the trade-off between accuracy and compression rate. ‘Ours-D’

¹ Estimated from the data provided in the original paper.

only involves decomposition and retraining step with $d = 5$, while ‘R1’ and ‘R2’ incorporate pruning and shrinking phases with regularization strength set to $1e - 4$ and $5 - e5$. ‘R3’ and ‘R4’ has the same parameter apart from setting $d = 6$ in the decomposition phase. The results show that the decomposition step can reduce more than 20% of the FLOPs with no accuracy drop. With the pruning and shrinking procedures, 94.73% of the FLOPS can be pruned with 2.4% top-1 accuracy loss. When we relax the regularization, we can still prune 81.23% of the FLOPs with only 0.5% accuracy loss. The FLOPs reduction is nearly two times the reduction of previous channel pruning methods. An even larger compression rate can be achieved by combining the 1×1 convolutional layer with the coefficient matrices.

2.2.5.4 Learning Curves

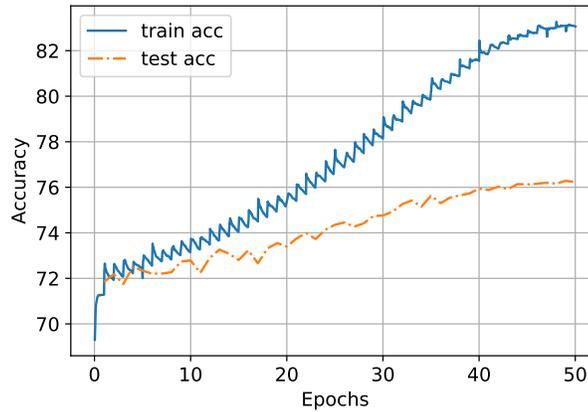
Figure 2.4 depicts the change in training and validation accuracy during the retraining and fine-tuning phases of the ResNet-50 model on the ImageNet dataset. We set $d = 6$ and regularization strength $\lambda = 5e - 5$. The results demonstrate that alternative training between the basis and the coefficient, together with using the approximated weight matrix during training, is effective in guaranteeing that the training of the compressed model progresses regularly. The learning curve shows a similar trend with a regular training process and demonstrates that the training process successfully converges.

2.2.5.5 Inference Acceleration

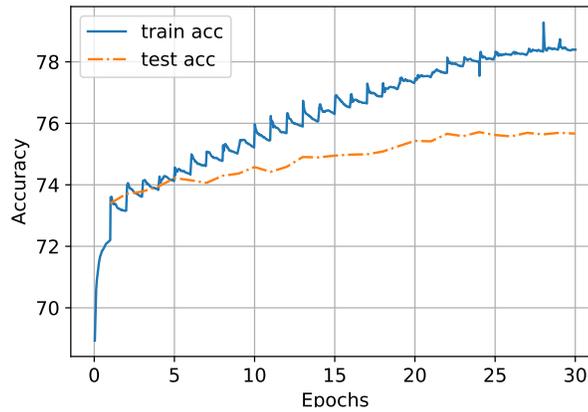
Table 2.4: Measured inference performance of VGG16-CIFAR10 on different devices.

| Device | Variation | Latency(ms) | Memory(MB) |
|--------|-----------|-------------|------------|
| CPU | Baseline | 12.9 | 137 |
| | PENNI | 5.96 | 77.6 |
| GPU | Baseline | 10.8 | 487 |
| | PENNI | 7.26 | 424 |

We used Intel Xeon Gold 6136 to test the inference performance for CPU platform and NVIDIA Titan X for the GPU platform. For software, we used PyTorch 1.4 [49] to implement the inference test. Batch size was set to 128 (1) for inference testing on the GPU



(a) The learning curve of the retraining phase.



(b) The learning curve of the fine-tuning phase.

FIGURE 2.4: The change in training and validation accuracy. The model is ResNet-50 with parameter setting R4.

(CPU). GPU inference batch size is higher than CPU to increase utilization and emphasize the latency impact of our method on the highly parallel platform. We indicate these settings as latency and peak memory consumption values vary across platforms or library versions.

Table 2.4 displays inference latencies and memory consumption recorded for the baseline and PENNI framework. As mentioned in 2.2.4, one of PENNI’s defining strengths is its impact on computational and memory footprints. Results shown in Table 2.4 reveal a $1.5\times$ ($2.2\times$) reduction in measured inference latency on the GPU (CPU). Peak memory consumption also benefited from a $1.1\times$ ($1.8\times$) reduction. It is important to note that these

metrics were taken without applying the convolution computation reorganization described in 2.2.4; this is done intentionally to reveal the effectiveness of our model shrinkage with zero changes to the hardware and inference-time computation. The reduction in memory is a straightforward consequence of the decomposition and shrinking stages of the PENNI framework. Although our method is successful at dramatically decreasing model size in memory, intermediate feature maps seem to dominate on-device memory consumption, especially with a batch size of 128 on the GPU.

2.2.5.6 Subspace Dimension

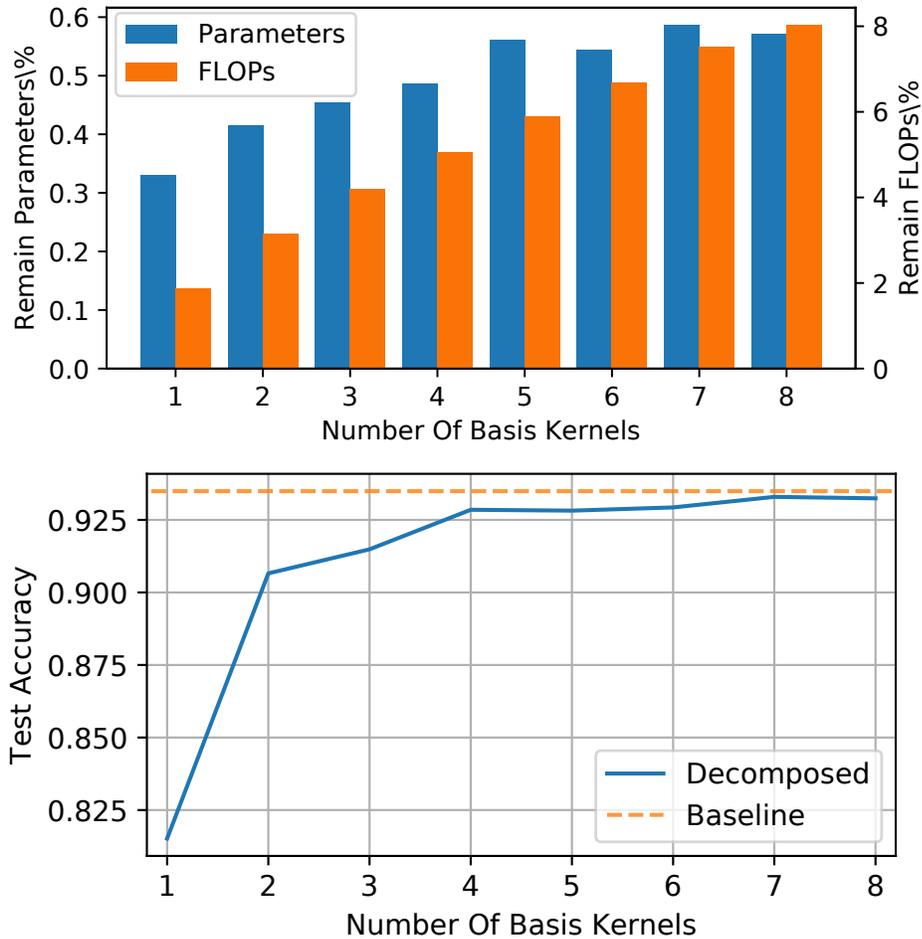
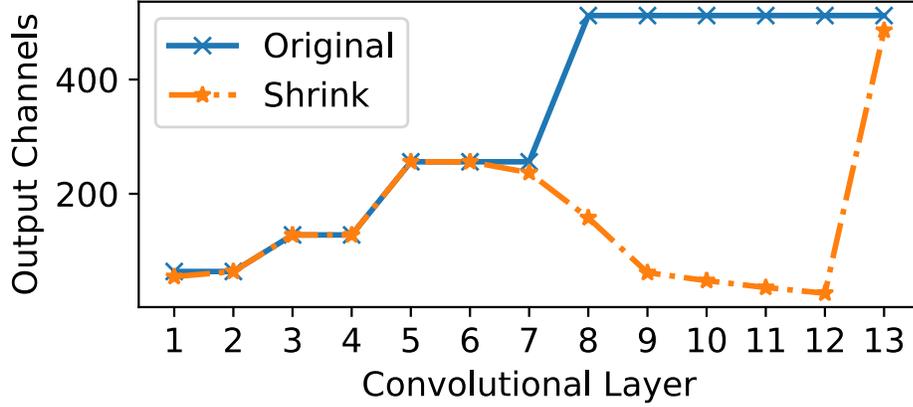
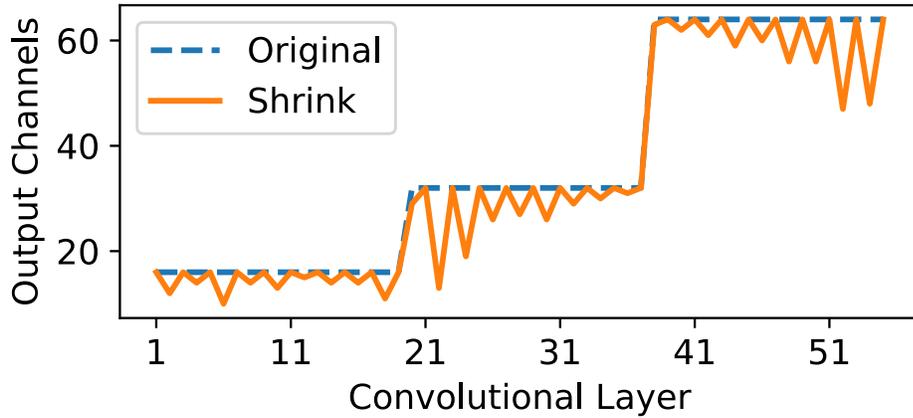


FIGURE 2.5: Test accuracy, parameters, and computation reduction with different numbers of basis kernel d .



(a) VGG16-CIFAR10



(b) ResNet56

FIGURE 2.6: Layer width after the model shrinking.

To justify the selection of the parameter d , we conduct an experiment with different decompose dimensions. We used the same VGG16 baseline model and hyper-parameters as Section 2.2.5.2. The result shown in Figure 2.5 indicates that the remaining FLOPs scales linearly with the number of basis kernels. This is expected since the number of convolutional operations is determined by d . The parameters scale linearly before 6-D basis and have minor differences with the increasing dimension. This is because even though more basis vector requires more coefficients, it also adds flexibility and thus leads to sparser coefficients. The test accuracy reveals the same trend; with $d \geq 4$, a minor improvement in the accuracy can be brought by increasing d . Thus, we select $d = 5$ for the balance between parameter and FLOPs reduction and accuracy drops.

2.2.5.7 Model Shrinking

We show the effectiveness of model shrinking by comparing layer widths. As shown in Figure 2.6, on VGG16, the model shrinking procedure effectively removes redundant channels in the second half of all layers. Meanwhile, on ResNet56, the shrinking is limited by the dimension matching requirement of the skip connections. The oscillation pattern of the layer width indicates that redundancies of the inner-block layer can be effectively exploited. The result also reveals the importance of different layers in the DNN model. In VGG16, the early layers correspond to high-level features. Thus, most channels are useful, and it is hard to remove the channel directly. Similarly, in ResNet56, the layers correspond to downsampling, which is hard to prune since it will cause information loss. This result also reveals that ResNet56 is a more compact model compared with VGG16, with few redundant channels to prune. These findings underscore the potential of PENNI to enhance unmodified inference software and hardware by leveraging structural redundancies.

Table 2.5: Detailed Structure of the compressed VGG16-CIFAR10

| Name | Size | Before shrinking | | | After shrinking | | |
|---------|------|------------------|----------------------------------|---------|-----------------|----------------------------------|---------|
| | | Width | Coefficients (Non-zero/Total) | Spar./% | Width | Coefficients (Non-zero/Total) | Spar./% |
| conv1_1 | 3 | 64 | 196/960 | 79.58 | 55 | 196/825 | 76.24 |
| conv1_2 | 3 | 64 | 2313/20480 | 88.71 | 64 | 2238/17600 | 87.28 |
| conv2_1 | 3 | 128 | 5862/40960 | 85.69 | 128 | 5862/40960 | 85.68 |
| conv2_2 | 3 | 128 | 12052/81920 | 85.29 | 128 | 12052/81920 | 85.28 |
| conv3_1 | 3 | 256 | 23169/163840 | 85.86 | 256 | 23169/163840 | 85.85 |
| conv3_2 | 3 | 256 | 36870/327680 | 88.75 | 256 | 36870/327680 | 88.74 |
| conv3_3 | 3 | 256 | 27719/327680 | 91.54 | 237 | 27716/303360 | 90.86 |
| conv4_1 | 3 | 512 | 15688/65536 | 97.61 | 158 | 15665/187230 | 91.63 |
| conv4_2 | 3 | 512 | 4534/1310720 | 99.65 | 62 | 4530/48980 | 90.75 |
| conv4_3 | 3 | 512 | 2668/1310720 | 99.79 | 48 | 2666/14880 | 82.08 |
| conv5_1 | 3 | 512 | 1318/1310720 | 99.89 | 36 | 1315/8640 | 84.78 |
| conv5_2 | 3 | 512 | 874/1310720 | 99.93 | 26 | 874/4680 | 81.32 |
| conv5_3 | 3 | 512 | 2554/1310720 | 99.80 | 486 | 2554/63180 | 95.95 |
| Total | | | 135817/8172480 | 98.34 | | 135707/1263775 | 89.26 |

2.2.5.8 Layerwise Analysis

We present the layerwise breakdown of the compressed VGG16-CIFAR10 model in Table 2.5. The result reveals the trend of increasing sparsity ratio from the shallow layers to the deeper layers. This observation aligns with the trend shown in model shrinking results. In addition, by comparing the results of before and after shrinking, we can also find that the sparsity distribution across different layers after shrinking is more balanced, avoiding an extremely high sparsity ratio of over 99%. As we discussed before, it is difficult for a sparse accelerator to be efficient for both high and low sparsity ratios since they demonstrate different requirements for the hardware components. Thus, these results indicate model shrinking could also help improve the efficiency of the compressed model on sparse accelerators.

2.3 ESCALATE Algorithm [2]

In Section 2.2.4, we provided a brief discussion on how PENNI could potentially benefit the hardware implementation and evaluate its hardware efficiency on general-purpose platforms, including CPU and GPU. However, we did not make an assumption on the underlying hardware paradigm in this discussion. When trying to implement the PENNI algorithm on the typical accelerator design paradigms like MAC arrays, there are more factors to consider. For example, PENNI assumed a 32-bit floating point format when evaluating the impact on accuracy, which is too expensive for the edge settings.

To enable efficient mapping of the decomposed CNN models to sparse accelerator architectures, we expand the PENNI framework into the ESCALATE algorithm. We begin by analyzing the computational bottleneck inherent in the original PENNI algorithm. Then, we propose a reformulated decomposition algorithm in ESCALATE that alleviates this bottleneck by leveraging the distributive property of convolution to create an efficient computation flow. Furthermore, ESCALATE introduces hybrid quantization, which judiciously assigns precision levels based on the reuse characteristics of different weight components. Notably, ESCALATE extends the applicability of kernel decomposition to unify the com-

putation of regular convolution and depth-wise separable convolution (DSC) under a single framework. This will guarantee the efficiency of the accelerator design across both convolution variants.

2.3.1 Computation Reorganization

PENNI is a CNN compression method that leverages kernel decomposition to reduce the computational and memory requirements of convolutional neural networks. As discussed in Section 2.2.4, this decomposition scheme separates the convolution operation into two distinct stages: shared kernel convolution and weighted accumulation. The resulting weight components are imbalanced, with one component being significantly smaller than the other. Using the notations defined in Section 1.1, and letting d represent the number of basis kernels, the coefficients can be reshaped into a $K \times C \times d$ tensor. With this representation, the k -th output feature map (OFM) can be computed as follows:

$$OFM_k = \sum_{c=0}^{C-1} \sum_{i=0}^{d-1} C_e^{(k,c,i)} IFM_c * B_i, \quad (2.6)$$

where IFM_c is the c -th input feature map, B_i is the i -th basis kernel, and $*$ denotes the 2D convolution operation. The shared kernel convolution (i.e., the inner summation) is conducted in a depthwise fashion, without any reduction across input channels, generating d times more feature maps than the input. The weighted accumulation (i.e., the outer summation) of these intermediate feature maps becomes the computational bottleneck. While kernel decomposition can achieve a high compression ratio for CNN models, translating this compression ratio into actual speedup is challenging. From Equation (2.6), we can observe that each pixel of the produced feature maps is only reused across output channels. To maximize input reuse, we would either need to frequently read and write the output channels, or build a large buffer to hold all intermediate feature maps. Both of these designs are energy-intensive and may negate the benefits of the compression. Furthermore, although the coefficients are highly sparse, the irregularity of the sparsity pattern makes it difficult to skip the zeros without degrading the parallelism of the computation. This irregularity

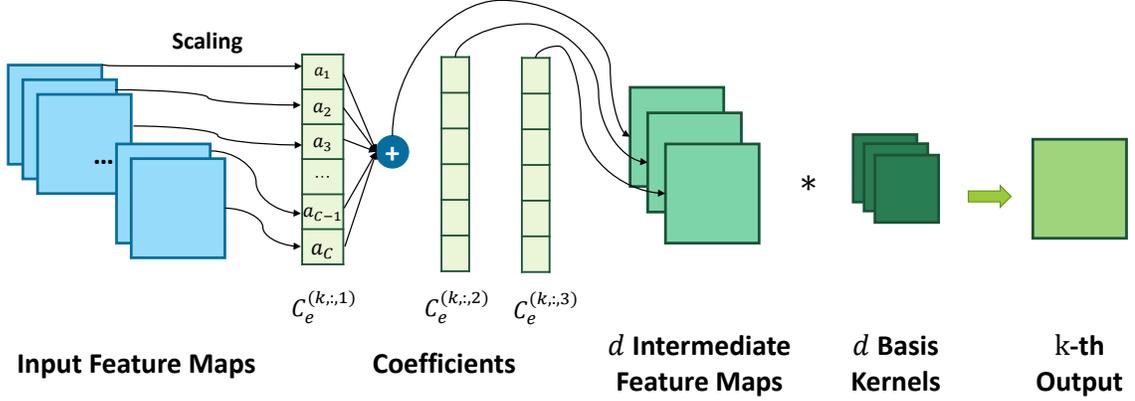


FIGURE 2.7: The computation process of one output feature map of the reorganized decomposed convolution. For simplicity, we assume $d = 3$ here.

poses an additional challenge in realizing the full potential of the compression scheme.

Based on the analysis presented above, more efficient computation can be achieved by pursuing the following goals: (1) reducing the total number of input feature maps; (2) reducing the number of input feature maps related to each output channel; and (3) increasing the reuse possibilities of each input feature map in the weighted accumulation stage. These desirable objectives can be realized through a simple observation: since the convolution operator follows the distributive property, the two stages of the computation – shared kernel convolution and weighted accumulation – are interchangeable. Specifically, we can exchange the order of the summations in Equation (2.6). With such a reorganization, we first compute the weighted accumulation of the input feature maps, and then conduct the convolution on the accumulated feature maps. The same output channel can be computed by

$$OFM_k = \sum_{i=0}^{d-1} \left(\sum_{c=0}^{C-1} C_e^{(k,c,i)} IFM_c \right) * B_i. \quad (2.7)$$

The reorganized convolution is illustrated in Figure 2.7. With the new computation order, we only need to compute the weighted accumulation of C input feature maps. Each input feature map can be reused for Kd times, and each basis kernel can be reused for K times. Since the number of output channels (K) is typically larger than or equal to the

number of input channels (C), this reorganization can explore more reuse opportunities and reduce the buffer size and data movements required. Furthermore, converting the convolution from the depthwise-like convolution into a normal convolution with d channels also improves parallelism and data reuse opportunities. By performing the weighted accumulation first, the subsequent convolution operation involves fewer input channels (d instead of C), enabling more efficient use of computational resources and better data reuse. More in-depth discussions about the benefits brought by this reorganization of the computation can be found in Chapter 3.

2.3.2 Hybrid Quantization

Quantization is a crucial step for deploying deep neural networks (DNNs) on edge devices. Empirical results [50] show that using 8-bit precision incurs minor accuracy loss while greatly boosting inference efficiency. In PENNI, we assumed a 32-bit precision for evaluation. ESCALATE algorithm takes the impact of quantization into consideration. In kernel decomposition, the weight sparsity primarily exists in the coefficients, as they constitute the main part of the parameters. We observe that the coefficients are unique to each pair of input-output channel, while the basis kernels are reused by all output channels. Selecting the same quantization precision for coefficients and basis kernels would be suboptimal – high precision is redundant for coefficients, while low precision for basis kernels would severely affect accuracy. Based on this observation, we propose a hybrid quantization scheme to maximize the benefits of both accuracy and compression ratio. The basic idea is to use high precision for the frequently reused basis kernels, while using low precision for the coefficients. Specifically, we quantize the basis kernels to 8 bits and the coefficients to ternary values. We choose to quantize the basis kernels to 8 bits and the coefficients to ternary values. Directly quantizing the entire coefficients tensor into ternary values would cause a severe accuracy drop. We observe that, for a given output channel k , only the $C_e^{(k, :, :)}$ slice is involved in the computation. Therefore, we apply *filter-wise* quantization, which allows different positive and negative scaling factors for each output

channel slice of the coefficients tensor. The output feature maps can then be re-quantized to match the range of each output channel.

To obtain the ternarized value of the coefficients, we adopt a quantization-aware training scheme. We use a similar method as described in [51]. Specifically, we store the full precision coefficients during the quantization. During the forward pass, we select a threshold for each coefficient slice corresponding to one output channel. The quantized value of the k -th slice is obtained by

$$\tilde{C}_e^{(k,:::)} = \begin{cases} w_k^{pos}, & C_e^{(k,:::)} > t \cdot \max(|C_e^{(k,:::)}|) \\ 0, & |C_e^{(k,:::)}| \leq t \cdot \max(|C_e^{(k,:::)}|) \\ -w_k^{neg}, & C_e^{(k,:::)} < -t \cdot \max(|C_e^{(k,:::)}|) \end{cases}, \quad (2.8)$$

where $\max(|C_e^{(k,:::)}|)$ is the maximum magnitude of the k -th coefficient slice, t is a hyper-parameter controlling the threshold, and w_k^{pos} and w_k^{neg} are the scaling factors for positive and negative values, respectively. We use different scaling factors for *each slice* of the coefficient matrix C_e , i.e., each accumulation operation, and obtain the scaling factor through training. We use the gradient of quantized coefficients to update the full precision parameters as well as the scaling factors. To simplify the hardware design, we divide the negative scaling factor by the positive one and quantize the quotient into 2 bits. During inference, we can attach the coefficient as a sign bit to each activation, and shift the negative one by the quotient. With these optimizations, we completely remove the multiplication in the first stage and enable arbitrary reordering of the activations without worrying about the relative order of the activations and weights. We claim those benefits in the hardware accelerator design developed in Chapter 3. Since the first layer of CNNs usually only has a very small number of channels (e.g., 3 channels for color image-related tasks), quantizing the first layer will incur a severe loss of information. Moreover, since d is normally larger than the number of the input channels in the first layer, applying decomposition will not bring any benefit to improve the computational efficiency. Thus, we do not apply compression to the first layer.

2.3.3 Decomposing the Compact Model

Depthwise separable convolution (DSC) splits the normal convolution into two steps: depthwise convolution and pointwise convolution. The depthwise convolution (DW) assigns an exclusive kernel to each input channel and eliminates the cross-channel interactions to reduce the computational cost. Since the input feature maps are not shared by different kernels in the depthwise stage, DW leads to under-utilization of PEs on the accelerators optimized for normal convolution. The lack of input reuse also results in a lower computation-to-memory ratio and the demand for a higher on-chip buffer bandwidth. To efficiently support both DSC and normal convolution, we unify the computation flow of both types of convolution with ESCALATE algorithm. We apply the same decomposition scheme to DSC and use the same decomposed form as described in Equation (2.7). With the unified computation flow, our design can efficiently support both types of convolution. With the notations defined before, the weights of DSC can be represented by the kernels in the depthwise convolution $W_{DW} \in \mathbb{R}^{C \times RS}$ and the coefficients in pointwise convolution $W_{PW} \in \mathbb{R}^{C \times K}$. We can decompose the weights of the depthwise convolution as $W_{DW} = C'_e B$ where $C'_e \in \mathbb{R}^{C \times d}$ and $B \in \mathbb{R}^{d \times RS}$. Then, we can combine W_{PW} and C'_e into the coefficient matrix by computing the Hadamard product of each column of C'_e and W_{PW} , which can be represented as:

$$C_e^{(c,k,i)} = W_{PW}^{(c,k)} C'_e{}^{(c,i)}. \quad (2.9)$$

Thus, we have a unified representation with Section 2.3.1. Although this decomposition increases the number of convolution operations by k times, the convolution kernels are shared across input channels. This kernel sharing allows us to support both types of convolution on the same architecture efficiently.

2.3.4 Evaluation

To evaluate the effectiveness of the ESCALATE algorithm, we test a variety of representative CNN models on both CIFAR-10 [43] and ImageNet [44] datasets. VGG16 [45], which is known to be redundant and easy to compress, is used as a sanity check for the

proposed framework. We select ResNet [40] as the target model for evaluation, including two variants for CIFAR-10: a shallow ResNet18 and a deep ResNet152. For ImageNet, we use ResNet50, which is widely evaluated by previous compression works. We also select MobileNet [12] and MobileNetV2 [13] to evaluate the effectiveness of our framework on compact models.

2.3.4.1 Experiment Settings

On CIFAR-10, baseline models are trained for 350 epochs using Nesterov accelerated SGD optimizer with 0.9 momentum and 0.0001 weight decay. The learning rate is set to 0.1 initially and multiplied by 0.1 at the 50% and 75% epochs. For ESCALATE, we select $d = 6$ for decomposition. The retraining process lasts for 300 epochs using ADAM [52] optimizer with 0.001 initial learning rate and the same decay policy. We set $t = 0.05$ in the ternary quantization process of coefficients. On ImageNet, we use the pre-trained model provided by PyTorch [53] as the ResNet50 baseline and train the baseline model of MobileNet with a 0.1 initial learning rate. The retraining process lasts for 60 epochs with the ADAM optimizer with a 0.0001 initial learning rate. Other parameters are the same as those used in CIFAR-10 training.

2.3.4.2 Analysis

The compression results of the ESCALATE algorithm are summarized in Table 2.6. We only show the result of convolutional layers since the ESCALATE algorithm only processes convolutional layers. Since other layers take up a small portion of the overall computation cost, they have a minor impact on the overall performance. We assume 32-bit floating-point precision for the baseline models. For ESCALATE, we use 8bit for the first convolutional layer and all the basis kernels and use the SparseMap encoding described in [54] for the coefficients. We compare our result with the quantization method STQ [55], the structured pruning method ResRep [56], the non-structured pruning method STR [57], and the joint pruning and quantization method presented in ADMM-NN-S [58]. Although ADMM-NN-S presents the results of both structured and non-structured pruning methods, its structured

Table 2.6: Compression result of ESCALATE algorithm.

| Model (Method) | Top-1 (%) | CONV (MB) | Compression Ratio (\times) | Sparsity (%) | Pruning Ratio ² (%) |
|------------------------|--------------|--------------------|-----------------------------------|-----------------|-----------------------------------|
| CIFAR-10 | | | | | |
| VGG16 | 93.49 | 56.12 | - | - | - |
| STQ | 92.38 | 2.21 | 25.10 | N/A | N/A |
| ADMM-NN-S | 93.10 | 0.54 | 109 | | 98.3 |
| Ours | 92.74 | 0.71 | 79.04 | 89.24 | 96.1 |
| ResNet18 | 93.79 | 42.58 | - | - | - |
| ADMM-NN-S | 93.3 | 0.33 | 135 | | 98.6 |
| Ours | 93.63 | 0.4 | 106.45 | 97.4 | 98.21 |
| ResNet152 | 95.36 | 221.19 | - | - | - |
| Naive_L1 | 94.12 | 20.45 ³ | 13.31 | | 92.49 |
| Ours | 93.86 | 0.68 | 325.27 | 99.2 | 99.4 |
| MobileNetV2 | 94.09 | 8.40 | - | - | - |
| ADMM-NN-S ⁴ | 94.90 | 0.54 | 18.8 | | 83.6 |
| Ours | 93.32 | 0.73 | 11.51 | 96.98 | 91.86 |
| ImageNet | | | | | |
| ResNet50 | 76.25 | 78.03 | - | - | - |
| STR | 74.31 | 7.623 | 10.24 | | 90.23 |
| ResRep | 75.3 | 48.16 ⁵ | 1.62 | N/A | 62.1 |
| Ours | 73.89 | 7.17 | 10.92 | 88.22 | 92.16 |
| MobileNet | 70.10 | 26.94 | - | - | - |
| STR | 68.35 | 6.653 | 4.05 | | 75.28 |
| ResRep | 68.02 | 13.815 | 1.95 | N/A | 73.91 |
| Ours | 67.89 | 3.02 | 8.92 | 67.6 | 63.9 |

pruning is a fine-grained variant; the hardware still needs to be specialized to skip the pruned columns in the filters. Since non-structured pruning shows a higher pruning ratio, we selected it as the baseline model for both algorithm and hardware evaluation. For structured pruning methods, we ignore the sparse encoding overhead and assume the parameters are represented using 32-bit floating point format. The proposed ESCALATE algorithm approaches the non-structured baselines in terms of both compression ratio and accuracy.

On CIFAR-10, we achieve a compression ratio ranging from $11\times$ to $325\times$. ESCA-

² Relative to the original weights before decomposition.

³ Does not include indices.

⁴ Trained with Mixup augmentation.

⁵ Estimated from FLOPs reduction reported in the paper.

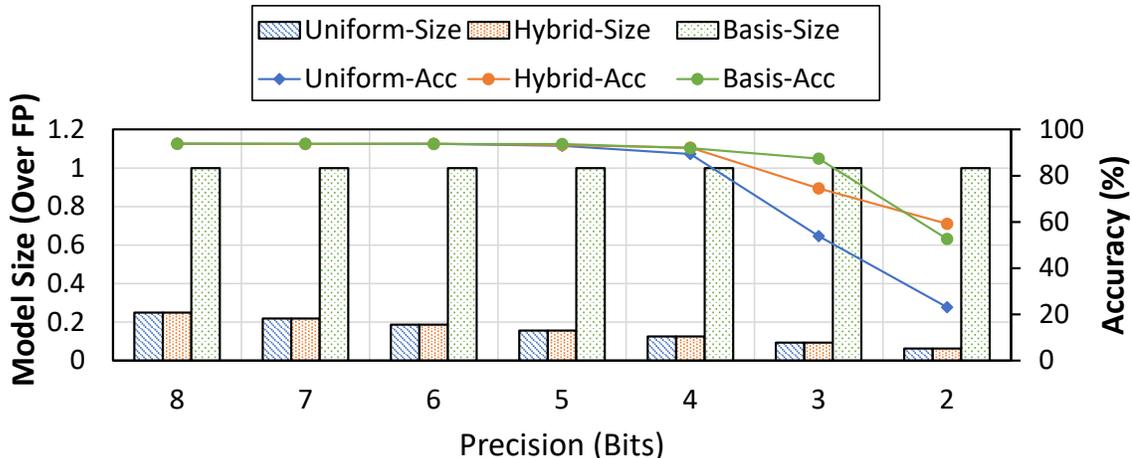


FIGURE 2.8: The comparison of model size and accuracy with uniform, hybrid and basis-only quantization.

LATE algorithm reaches a similar compression ratio on all CIFAR-10 models with the non-structured ADMM-NN-S. The encoding overhead of SparseMap results in the gap. ESCALATE is also able to efficiently explore the redundancy in a large model like ResNet152 and prune 99.2% of parameters, compressing the original model by $325\times$. Some downsampling layers in ResNet152, especially those in the last three residual blocks, are completely pruned. These results indicate that the ESCALATE algorithm can effectively identify and eliminate the redundancy in the CNN models. For the compact MobileNetV2 model, we can also achieve over $11\times$ compression ratio. ESCALATE incurs a 1.5% accuracy loss on ResNet152 and less than 0.8% accuracy loss on the remaining three models compared to the uncompressed models. On ImageNet, we achieve a similar sparsity with the non-structured pruning method STR with less than 0.5% accuracy gap. For MobileNet, we maintain the accuracy at 67.89% while compressing the original model into 3.02MB.

We also show the advantage of hybrid quantization by performing post-training quantization on the decomposed ResNet18 model. Uniform quantization policy enforces the same precision on both basis and coefficients, while hybrid quantization keeps the basis in 8 bits and only further quantizes the coefficients. The result is shown in Figure 2.8. We omit coefficient-only quantization in the figure since it shows identical behavior with the

hybrid. Since the basis kernels only occupy a small portion of the parameters while being frequently reused in computation, keeping the basis kernels in high precision effectively maintains model accuracy. These results show that hybrid quantization can achieve almost the same compression ratio as uniform quantization while maintaining accuracy.

2.4 Conclusion

This chapter introduces the PENNI framework and its extension ESCALATE algorithm for enabling hardware-friendly compression of convolutional neural networks (CNNs). PENNI aims to improve inference latency without modifying the inference algorithms or hardware, achieving speedup by shrinking the model size and translating the induced structured sparsity to practical acceleration. The core idea is to leverage a low-rank assumption to decompose CNN filters into basis kernels and coefficient matrices, which are then pruned to increase sparsity. We employ a novel alternating fine-tuning method to further enhance model performance while maintaining high sparsity levels. The evaluation highlights that PENNI is able to prune 97% parameters and 92% FLOPs on ResNet18 CIFAR10 with no accuracy loss, and achieve a 44% reduction in run-time memory consumption and a 53% reduction in inference latency on general-purpose computing platforms.

The unique decomposed structure generated by PENNI creates opportunities for reorganizing convolution computation in a hardware-efficient manner, directly benefiting modern DNN accelerators. Building upon PENNI, the ESCALATE algorithm further exploits this potential for sparse accelerator design. It reformulates the decomposed convolution to eliminate computational bottlenecks and introduces hybrid quantization to exploit the discrepancy in parameter reuse frequencies. The evaluation demonstrates up to $325\times$ and $11\times$ compression ratio for models on CIFAR-10 and ImageNet, respectively. Building upon the decomposed form, ESCALATE provides the potential of an efficient sparse accelerator design, which will be discussed in the following chapter.

3. Sparse DNN Accelerator with Kernel Decomposition¹

In the previous chapter, we discussed hardware-aware compression methods and evaluated their effectiveness on general-purpose platforms like CPUs and GPUs. However, these platforms cannot fully utilize the sparsity in the coefficients, leaving a significant potential for further optimization. In this chapter, we shift our focus to the hardware level and explore how to design specialized accelerators to support the compressed DNN models efficiently.

Researchers have been trying to build sparse-aware accelerators to utilize the sparsity in the compressed DNN model. However, their efficiency gains are fundamentally limited by the irregularity of pruned sparse patterns. As an alternative solution, structured pruning was introduced to produce a predictable sparsity and simplify the sparse processing unit design. However, compared with unstructured pruning, this approach constrains the achievable compression ratios.

The PENNI framework and its extension, the ESCALATE algorithm, provide a foundation for the sparse accelerator design to address the challenges of accelerating the compressed DNN model. The compression performed by PENNI decouples the computation into two stages - one focused on weighted accumulation with sparse coefficients and the other on dense convolution with a common set of basis kernels. This decoupling isolates the sparsity processing from the critical inference path, enabling high compression ratios without strict constraints on the sparsity patterns. The computation reorganization and the hybrid quantization in the ESCALATE algorithm reformulate the original PENNI and allow a highly efficient sparse accelerator design. In this chapter, we introduce ESCALATE accelerator [2], a sparse CNN accelerator that leverages the kernel decomposition.

The ESCALATE framework, including the ESCALATE algorithm discussed in Section 2.3 and the accelerator to be introduced in this chapter, co-designs the compression algorithm and the underlying hardware architecture in a coherent manner. ESCALATE unlocks opportunities for substantial performance and efficiency gains during sparse model inference

¹ This chapter is primarily based on: ESCALATE: Boosting the Efficiency of Sparse CNN Accelerator with Kernel Decomposition. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '21)*. © 2021 Association for Computing Machinery. <http://dx.doi.org/10.1145/3466752.3480043>

that surpasses what can be achieved by traditional sparse accelerator designs.

3.1 Background and Related Works

The computation of deep neural networks (DNNs) exhibits massive parallelism and regularity, providing opportunities for designing specialized accelerators that capture unique optimization opportunities to boost performance and energy efficiency. Early works like the Diannao series [42], [59] tailored the general pipeline and designed neural function units to exploit this parallelism. Eyeriss [60] proposed a row-stationary dataflow to explore data reuse opportunities.

As DNNs became more computationally intensive, sparsity emerged as a key focus area for accelerator design. By avoiding storing and processing pruned parameters, sparse accelerators could benefit from model compression. Cambricon-X [61] skipped zero weights, while Cnvlutin [62] skipped zeros in input activations. SCNN [63] and SparTen [54] exploited two-sided weight and activation sparsity. ReCom [64] and ReGAN [65] using processing-in-memory architecture to support sparsity in DNN. Bit-serial accelerators [66]–[68] reduced computation by removing ineffectual bits during multiplication, though they did not reduce data transfer costs.

The redundancy in CNNs arises from two sources - sparse activations produced by the ReLU function, and sparse weights from pruning unimportant values. On average, this introduces a reduction of 2-5 \times model size and 4-20 \times computational cost. However, leveraging sparsity poses challenges as the accelerator needs to identify and dispatch non-zero weight-activation pairs correctly to processing elements (PEs) amid irregular distributions.

Current sparse accelerators like Cambricon-X, Cnvlutin, SCNN, and SparTen propose various mechanisms but incur considerable hardware and energy overheads that offset the benefits of sparsity [58]. This challenge can be mitigated by algorithm-hardware co-design approaches that perform pruning while considering hardware efficiency. For example, Cambricon-S [69] uses coarse-grained pruning, ADMM-NN [7] jointly prunes and quantizes layers, while PatDNN [70] and ADMM-NN-S [58] enforce specific sparsity patterns during

pruning for more efficient processing. Another series of works attempts to build accelerators to support the basic sparse operators. For example, ExTensor [71] targets sparse tensor operations, while SIGMA [72] is designed for sparse GEMM, mapping CNNs. Flexagon [73] makes an analysis of different sparse matrix multiplication patterns and derives designs for different patterns. Highlights [74] extract the fiber tree structure to support finer-grained operations. Overall, while sparsity is a promising approach, the co-design of algorithms and hardware is key to fully unlocking its benefits for efficient DNN acceleration.

3.2 Motivation

ESCALATE is motivated by the drawbacks of deploying the CNN model compressed by the existing non-structured and structured pruning methods. Non-structured (or element-wise) pruning can achieve the highest compression ratio on CNNs among different types of compression methods. However, the irregularity of the sparsity pattern leads to a huge gap between the algorithm-level computation reduction and the actual hardware speedups. For example, Eyeriss v2 [75] only achieves $1.2\times$ speedup on sparse MobileNet. ADMM-NN [7] demonstrates only $3.9\times$ speedup for the pruned convolutional layers in AlexNet with a $25.5\times$ pruning ratio. In essence, non-structured pruning leads to inefficient and expensive hardware implementations, which largely offset the benefits of the large compression ratio. On the other side, structured pruning maintains the regularity of weights so that the pruning ratio can be almost fully converted to the speedup. However, the structured constraint limits its pruning ratio. For example, structured pruning on ResNet50 only achieves $2.64\times$ pruning ratio [56] while non-structured pruning could reach $17.4\times$ [7]. In summary, none of the two types of pruning methods sufficiently translate the elimination of redundancy in CNNs to higher inference performance.

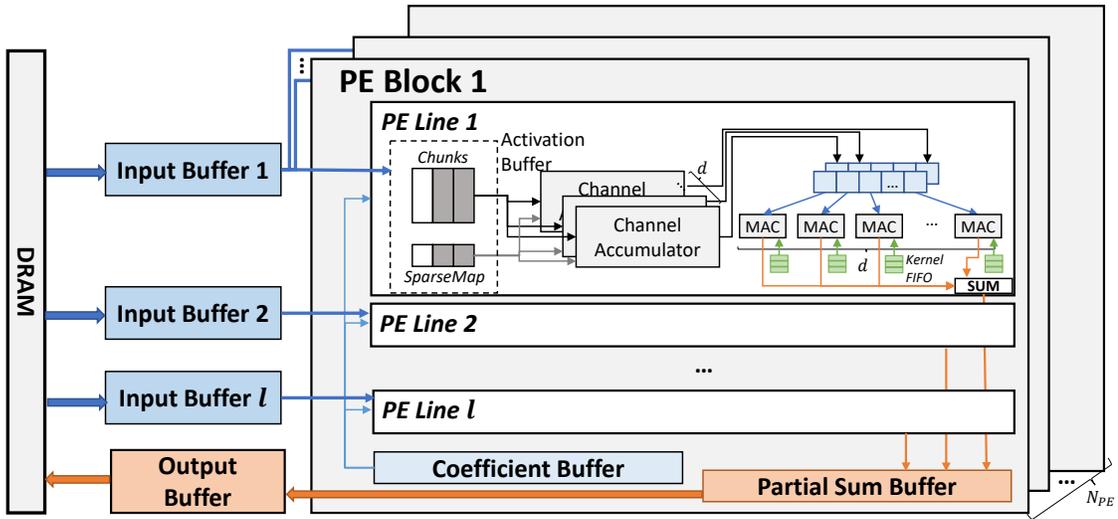
To tackle the challenge of efficient execution of compressed deep neural network models, we decided to explore the potential of the decomposed weight structure produced by the PENNI compression framework. The kernel decomposition performed by PENNI converts the original convolution operation into an alternative form, known as decomposed convo-

lution. As discussed in Section 2.2.4 in the previous chapter, kernel decomposition creates two imbalanced parts to approximate the original weight tensor, naturally forming two computation stages. The stage involving the coefficient matrix requires only scalar matrix product and reduction operations. This stage opens the opportunity to utilize the sparsity in the coefficients in a structured manner. Specifically, we can skip the entire feature map corresponding to zero coefficients directly. Compared to non-structured pruning techniques, kernel decomposition alleviates the irregularity in the sparsity-aware computation and provides better hardware efficiency. Unlike structured pruning methods, which enforce a specific sparsity pattern during pruning, our approach does not require such constraints on the coefficient matrix, allowing us to achieve higher pruning ratios.

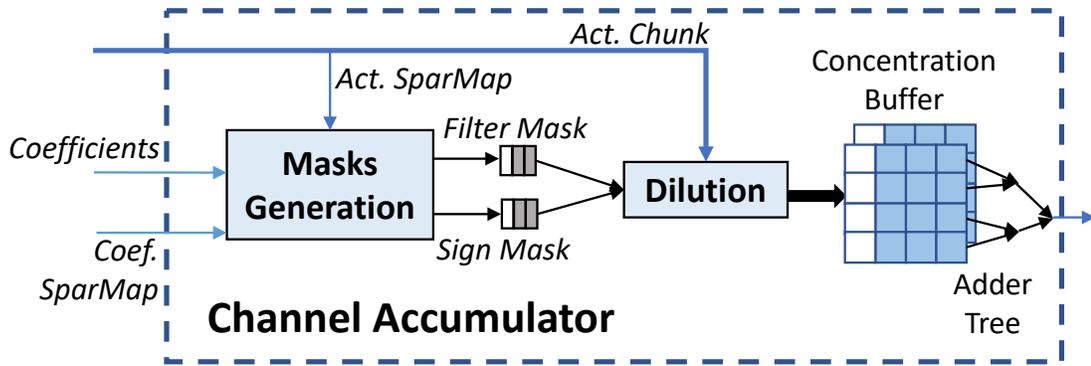
Motivated by the potential of obtaining both high pruning ratios and hardware efficiency, we built an algorithm-hardware co-design framework, including the ESCALATE algorithm discussed in the previous chapter and the ESCALATE accelerator introduced in this chapter. By co-designing the compression algorithm and the underlying hardware architecture, we can optimize the entire inference pipeline, leveraging the structured sparsity introduced by kernel decomposition to maximize performance and energy efficiency. The decomposed convolution operation enables a hardware-friendly execution pattern, where the sparse coefficient matrix can be processed efficiently by skipping zero values in a structured manner. This structured sparsity processing alleviates the irregularity and control overhead associated with traditional non-structured pruning techniques, enabling more efficient utilization of hardware resources. Moreover, the lack of constraints on the sparsity pattern of the coefficient matrix allows for higher compression ratios compared to structured pruning methods, further reducing the computational workload and memory requirements of the compressed model.

3.3 ESCALATE Architecture Overview

With the reformulated computation enabled by the ESCALATE algorithm, it is natural to build hardware components corresponding to the two stages of the decomposed convo-



(a) The microarchitecture design of *ESCALATE*. d corresponds to the hyperparameter in the decomposed convolution, while l and N_{PE} is the design space parameter.



(b) The design of channel accumulator.

FIGURE 3.1: The over view of *ESCALATE* accelerator.

lution: weighted accumulation and basis kernel convolution. We illustrate the ESCALATE accelerator design in Figure 3.1. ESCALATE presents a hierarchical PE design by splitting each PE into slices (lines). Each PE slice has two parts, corresponding to the two stages of the decomposed convolution. The first part, namely the channel accumulator (CA), utilizes the Dilution-Concentration mechanism to efficiently eliminate ineffectual computations on the sparse coefficient matrix during the weighted accumulation stage. The second part consists of a row of multiply-accumulate (MAC) units, with each MAC having a small

FIFO to hold one basis kernel. The basis kernels are loaded into the FIFOs before the computation begins and remain resident, actively exploiting the channel parallelism of the intermediate feature maps and the reuse of the basis kernels. To process multiple rows of input feature maps in parallel, ESCALATE employs separate buffers to store each slice of the input feature maps. The PE slices at the same position across different PE blocks are connected to the same input buffer. Since the coefficients are unique to the computation of each PE block, individual coefficient buffers are built within each PE block. This hierarchical design effectively maps the two stages of the decomposed convolution onto specialized hardware components, enabling efficient execution of the reformulated computation while leveraging the structured sparsity and parallelism opportunities exposed by the ESCALATE algorithm.

ESCALATE makes three contributions. First, the ESCALATE accelerator features a novel Basis-First dataflow to exploit the unique parallelism and data reuse opportunities brought by kernel decomposition. Second, we propose a Dilution-Concentration scheme to efficiently eliminate ineffectual computation with bit gather. Lastly, ESCALATE provides an efficient input buffer design to support asynchronously running PE slices to alleviate the impact of workload imbalance.

3.4 Basis-First Dataflow

Existing dataflows are designed for regular convolution. Those dataflows do not fit the two-stage computation of the decomposition convolution. We propose the *Basis-First (BF)* dataflow for decomposed convolution. Our objectives are to maximize the reuse of each batch of inputs, reduce the global buffer accesses, and avoid stalling the MACs. The BF dataflow is illustrated in Figure 3.2.

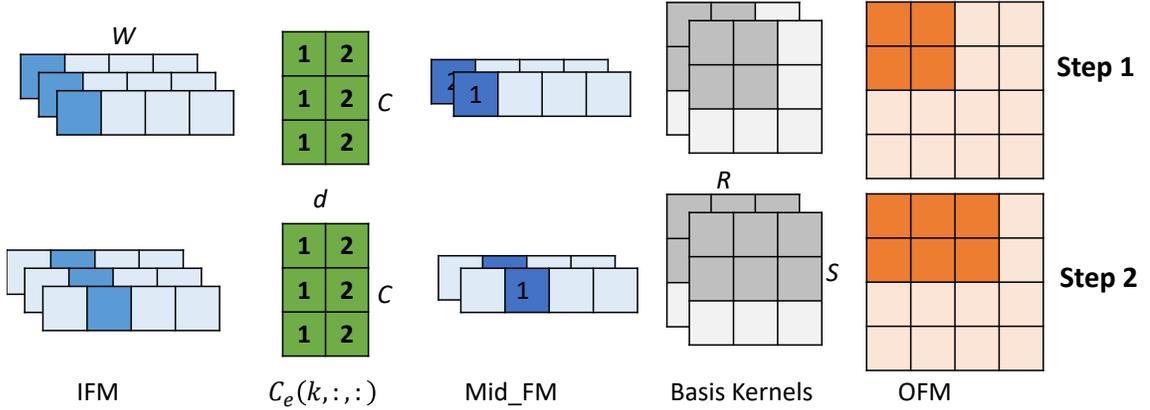
BF dataflow confines the computation of one output channel to one PE block so that we can fully utilize the distributed coefficient buffer and avoid cross-PE communication. We spatially map one row of the output feature map to one PE slice with the offset of the total number of PE slices. Inside each PE slice, we process one position of all input

```

for(k=blk_id; k<K; k+=Npe)           //Each PE Block
  for(y=line_id; y<H; y+=1)         //Each PE Slice
    for(x=0; x<W; x++)
      for(i=ca_id; i<d; i++)         // Each CA-MAC pair
        for(c=0; c<C; c++)         //Channel Accumulator
          mid[i][x][y] += I[c][x][y] * Ce[k][c][i]
        for(r=0; r<R; r++)         //MAC
          for(s=0; s<S; s++)
            O[k][x-R/2+r][y-S/2+s] += mid[i][x][y] * B[i][r][s]

```

(a) The nested-loop expression. N_{pe} and l are design space parameters corresponding to the number of PE blocks and the number of slices per block.



(b) The illustration of BF dataflow in one PE slice of the k -th PE Block. The number represents the index of the CA-MAC pair. We assume $d = 2, C = 3$, and the current PE slice is the first one. We assume a unit-stride convolution for simplicity.

FIGURE 3.2: The Basis-First Dataflow.

feature maps at one time and spatially map each intermediate channel (i.e., index i) to each CA-MAC pair. Based on the same observation with [63], we multiply each element of the intermediate feature maps by all weights in the corresponding kernel (partial weights for the elements on the edge). The product is sent to the partial sum buffer and accumulated to the corresponding output position through the read-modify-write operation. Since the output accumulation is not at the critical path of the processing and is less sensitive to the latency, we do not attempt to reduce bank conflicts at the partial sum buffer with additional optimizations.

The ESCALATE accelerator also supports regular convolution layers that cannot be compressed effectively (e.g., the first convolutional layer). For these layers, ESCALATE

employs an input stationary dataflow to align with the objective of maximizing input reuse, which is also a key goal for decomposed convolution. The channel accumulator is bypassed, and only the MAC units are utilized for regular convolution layers. For the fully connected layer, we convert it into 1×1 convolution with 1×1 input feature maps to maintain a unified mapping. We do not build separate units to support the sparsity in those layers because: 1) they only take up a small portion of the overall computation ($< 5\%$); 2) those layers usually have a low sparsity ratio, and the overhead of supporting sparsity for those layers can outweigh the computation reduction. For example, as shown in [7], the pruning ratio of the first layer is only $1.2\text{-}1.6\times$, and the sparse weight even causes performance degradation compared with its dense version.

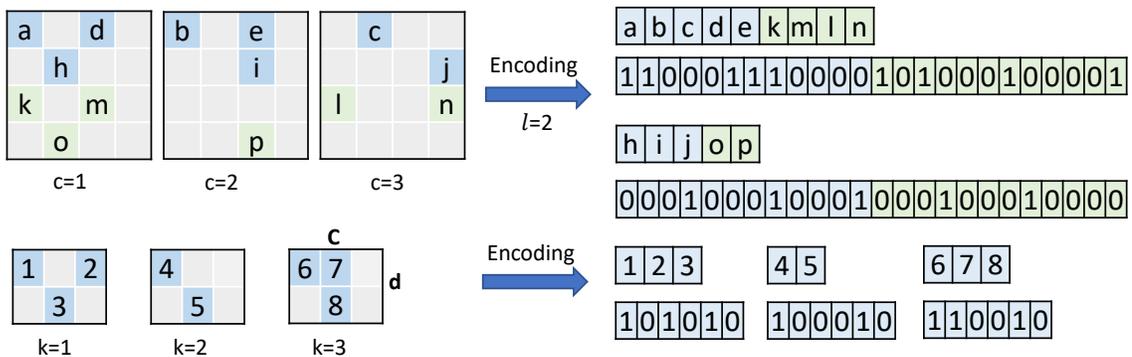
3.5 Dilution-Concentration

3.5.1 Sparse Encoding

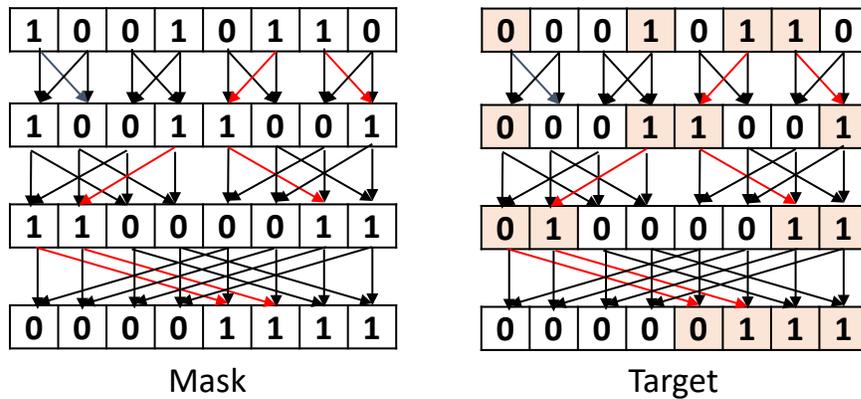
The sparsity in weights and activations can reduce storage and bandwidth consumption. However, we need a proper encoding scheme to index the non-zero values. Previous works [61], [63] adopt the Compressed Sparse Row (CSR) or Compressed Sparse Column (CSC), which uses separate arrays to store the row/column index of the non-zero element. As discussed in [54], CSR and CSC are not efficient when the sparsity ratio is relatively low. Making it worse, the cost of storing one index is much higher than storing multiple ternary values, making the indexing overhead outweigh the benefits brought by sparsity.

We adopt the same SparseMap encoding of [54]. As shown in Figure 3.3(a), we slice the input feature map along the W dimension (i.e., row dimension) and store the rows of stride l in a contiguous array, where l is the number of concurrent accumulations in the first stage. The coefficients are sliced along the K dimension and stored in separate arrays for each output channel. We store a separate bit mask for each array, indicating whether each position has non-zero values. Both activations and coefficients are stored in C -order to match the computation in the first stage of the decomposed convolution. To maintain the space efficiency under high sparsity situations, we also introduce a 2-level SparseMap

encoding. We split the sparse map into 16-bit chunks and used one bit per chunk to indicate whether the chunk is all zero. The all-zero chunks are not stored. Apart from the space efficiency, SparseMap encoding also simplifies the processing of the compressed arrays. We are able to match the index of non-zero elements in activation and weight arrays with bit-wise AND and bit-gather operations, which have a low energy cost. Moreover, SparseMap allows us to fully utilize the on-chip bandwidth since we can implicitly extract the number of processed elements through the index-matching process. We can insert a barrier indicating moving to a new position, so we do not need to split and pad the input as fixed-length chunks or have varied access lengths.



(a) The SparseMap encoding of the activations and coefficients. we assume $C = 3, d = 2, K = 3$ for this example.



(b) Bit gather operation with an inverted butterfly network.

FIGURE 3.3: The illustration of sparse encoding and bit-gather operation.

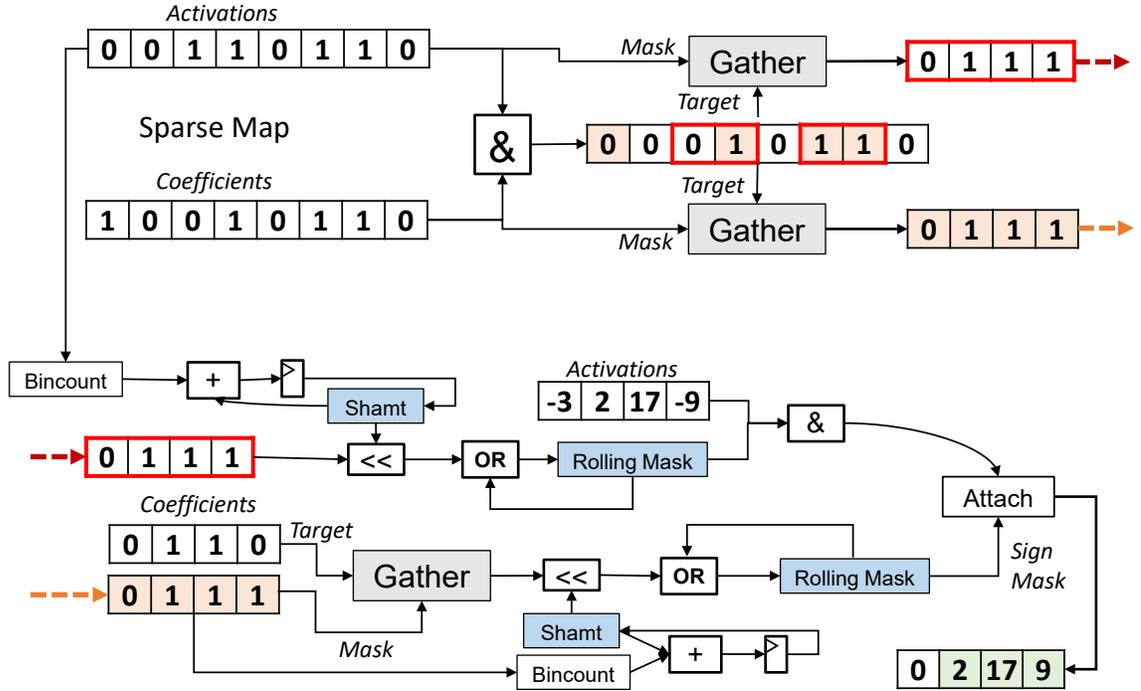


FIGURE 3.4: Dilution process. For simplicity, we omit the higher 4 bits during the process in the figure.

3.5.2 Dilution

The purpose of the dilution process in the channel accumulator is to match the input chunk of activations with the sparse coefficients and filter out the activations corresponding to the zero coefficients. This filtering operation is crucial for exploiting the sparsity introduced by the kernel decomposition, enabling the elimination of ineffectual computations and improving overall efficiency. Since the activations are kept at high precision (e.g., 8 or 16 bits), the cost of shuffling or moving these activation values can be relatively high compared to the computational operations.

To mitigate this cost, the dilution process keeps the “holes” or vacant positions in the filtered chunks, leaving them for the subsequent concentration step. This approach avoids unnecessary data movement and ensures that the activations remain in their original order, simplifying the concentration process. Given that the coefficients are quantized to ternary values (-1, 0, 1), the dilution process only needs to generate two masks: one for filtering

out the activations corresponding to zero coefficients, and another for handling the sign of the non-zero coefficients. The generation of these two masks can be efficiently implemented using bit gather operations, a technique that has been well-studied in previous literature [76]. In the bit gather operation, a mask is used to indicate the valid elements. The operation collects all bits corresponding to the valid bits of the mask and places them together on the same side of the array, while preserving their original order. This operation can be performed efficiently using specialized hardware instructions or optimized software implementations.

We show an example of bit gather implementation with an inverse butterfly network of $\log_2(n)$ steps in Figure 3.3(b). With the bit gather operator, we depict the dilution process in Figure 3.4. Specifically, we calculate the intersection of input activations and coefficients by performing a bit-wise AND operation. The *filtering mask* is generated by gathering the intersection with the activation sparse map as the mask. To obtain the sign mask, we first generate a *coefficient mask* by gathering the intersection with the sparse map of coefficients. The *coefficient mask* indicates whether the corresponding non-zero value of coefficients appears in the intersection. Then, we gather the non-zero value (i.e., the sign bit of ternary coefficients) with the *coefficient mask* and generate the *sign mask*. With the

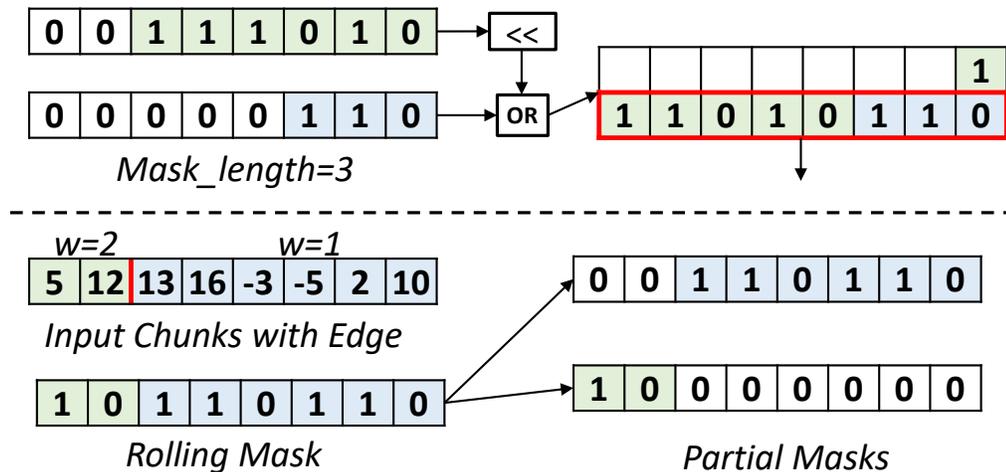


FIGURE 3.5: Rolling mask and implicit barrier.

filtering mask and *sign mask*, we can filter the input chunks and change the sign of each filtered activation. The sign bit is also attached to each activation for the concentration step to apply proper scaling. Since the sparse map is transferred and stored separately from the non-zero values, we can start the generation of the masks ahead of the reading of the chunks. The whole process can be pipelined to satisfy the timing constraints without impacting the throughput.

Since the distribution of non-zero elements could vary between activations and coefficients, the size of the mask generated through one pass might not be able to cover the current input chunk. To address this issue, ESCALATE employs a rolling mask mechanism, as shown in Figure 3.5. The newly generated masks are left-shifted by the length of the current rolling mask and attached to the rolling mask through a bit-wise OR operation. Once the size of the rolling mask is large enough to cover the current chunk, the corresponding part is evicted from the buffer and used for filtering. This rolling mask scheme also supports the implicit barrier between consecutive input positions. A counter is maintained for the rolling mask. If the count indicates that all elements corresponding to the current position have been processed, the current mask is broken into two partial masks, which are separately applied to the current input chunk. In other words, the rolling mask creates a barrier to separate the activations corresponding to different positions. With this barrier, ESCALATE can avoid the expensive element rotation operation and fully utilize the bus bandwidth, at the cost of two partially utilized cycles.

3.5.3 Concentration

With the "diluted" and signed chunks of activations produced by the dilution process, the concentration process is relatively simple. As shown in Figure 3.6, we collect the chunks produced by the dilution process, folding each of the chunks into multiple columns, and trying to fill all zeros with the column-wise "look-ahead" and "look-aside" approach. This process is equivalent to reordering the activations. Since we have attached the coefficients as the sign bit, we can permute the activations in arbitrary order without affecting the

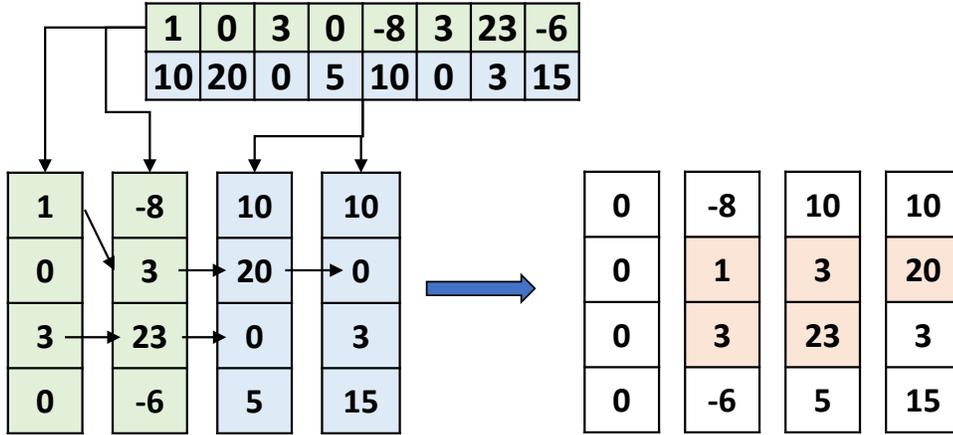


FIGURE 3.6: Concentration through look-ahead and look-aside.

correctness of the computation. ESCALATE employs double buffers in this concentration step. When one buffer is used to feed the activations to the reduction tree for accumulation, we collect and concentrate the processed chunks in the other buffer. The functionalities of the two buffers swap when the adder tree has processed all non-zero chunks. The concentration process is performed in both buffers until the buffer is full or there's no more possible movement of the elements to eliminate zeros. The barrier introduced by the rolling mask mechanism results in an immediate flush of the buffer, guaranteeing that the chunks from different positions will not get mixed up during the concentration process.

The Dilution-Concentration mechanism is implemented in the Channel Accumulator Unit (CA), which is depicted in Figure 3.1(b). The CA unit is responsible for performing the weighted accumulation stage of the decomposed convolution, leveraging the sparsity introduced by the ESCALATE algorithm. Since the SparseMaps, which encode the sparsity pattern of the coefficients, are stored and processed separately from the input chunks, ESCALATE can start the mask generation process in advance and overlap it with other parts of the computation. By decoupling the mask generation from the main datapath and executing it in parallel, ESCALATE can effectively hide the latency associated with generating the filtering and sign masks required for the dilution process. This parallel execution ensures that the mask generation does not become a bottleneck, allowing the

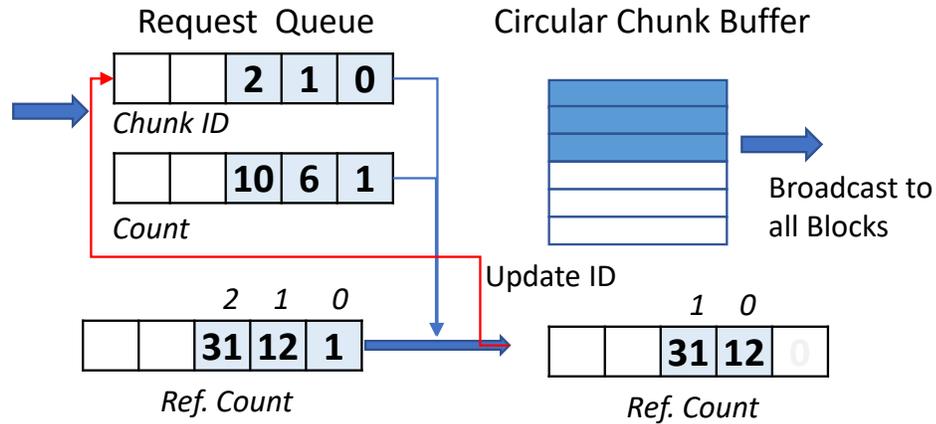


FIGURE 3.7: The design of input buffer. Each input buffer is implemented as a circular queue and uses the reference count to evict the finished chunks.

dilution and concentration processes to proceed without unnecessary stalls.

3.6 Buffer Design

Unlike previous sparse accelerators, ESCALATE does not require a reduction across PE slices. While the load imbalance resulting from the lack of reduction can be mitigated by asynchronously running PE slices, this approach could potentially reduce the reuse opportunities of the input feature maps, leading to higher costs associated with data movement and memory accesses. To address this issue while maintaining efficient execution, ESCALATE introduces an altered input buffer design. Instead of employing a large, unified input buffer, the design breaks the input buffer into multiple individual buffers. Each of these individual buffers is connected to the PE slices at the same position across all PE blocks, as these PE slices process the same lines of the input feature maps. Within each individual buffer, the non-zero elements of the input feature maps are organized into chunks. These chunks are accessed strictly in order, and as such, they are stored in a circular queue data structure. Registers are used to maintain the head and tail pointers to this circular queue. For each chunk, the buffer maintains a chunk ID as well as a counter that tracks the number of PE slices that have not yet processed this particular chunk. Buffer access requests from the PE slices are collected through an H-Tree interconnect, with each PE slice connected to

Table 3.1: Configurations of ESCALATE and baselines.

| ESCALATE | | | |
|--|--------|-------------|-------------------|
| d | 6 | Input Buf. | 8KB |
| N_{PE} | 32 | Coef. Buf. | 512Bytes |
| l | 5 | Output Buf. | 4KB |
| Input Bus | 16Byte | Psum Buf. | 2KB |
| Precision | 8bit | Act. Buf. | 16Byte \times 4 |
| Other Baselines | | | |
| Proportional scaling of on-chip SRAM buffer. | | | |
| 1024 8-bit multipliers | | | |

the H-Tree. The nodes of the H-Tree act as arbitrators, calculating the number of PE slices to which the current winning requests could be cast. Additionally, a separate request queue within each buffer stores all outstanding access requests. When a request is granted, the corresponding chunk, along with its ID matching the head of the request queue, is broadcast to all PE slices. The counter associated with the accessed chunk is then updated by subtracting the count in the current request. If the counter is decreased to zero, indicating that all PE slices have processed the chunk, the corresponding chunk is evicted from the circular queue. A signal is then sent to all slices to update the chunk ID, and the chunk ID in the request queue is also updated accordingly. To efficiently utilize the available capacity of the input buffers, ESCALATE employs a greedy policy in each arbitrator node of the H-Tree. This policy prioritizes requests for earlier chunks, ensuring that the chunks are processed in a timely manner and reducing the likelihood of stalls due to unavailable data. The design of the input buffer is illustrated in Figure 3.7.

3.7 Evaluation

To evaluate the effectiveness of the ESCALATE framework, we test a variety of representative CNN models on both CIFAR-10 [43] and ImageNet [44] datasets. We use the same model selection described in Section 2.3.4 and use the compressed model produced by the ESCALATE algorithm for the evaluation of the accelerator design.

Table 3.2: Unit energy cost per 8-bit integer operation extracted from a commercial 65nm technology.

| | DRAM | MAC | Multiply | Add |
|----------------------|------|-------|----------|-------|
| Energy(pJ/8-bit Int) | 100 | 0.407 | 0.186 | 0.036 |

3.7.1 Experiment Settings

To estimate power and area, we implement the RTL design of ESCALATE and synthesize it using Synopsys Design Compiler with TSMC 65nm library under the typical corner, 1V, and 25°C. The design achieved 800MHz frequency. The power and area estimated from the synthesized result is shown in Table 3.3. We then implement a cycle-accurate simulator and verify it against the RTL implementation. The simulator generates SRAM and DRAM access traces. For SRAM, we use CACTI 7.0 [77] to estimate the power consumption. For DRAM, we simulate the trace using ramulator [78] and extract the energy consumption from the command trace with DRAMPower [79]. We only evaluate the convolutional layers since ESCALATE does not process other types of layers, and the SCNN baseline only supports convolutional layers.

For performance baseline, we select a DNN accelerator optimized for dense networks, Eyeriss [60], and two sparse CNN accelerators, SCNN [63] and SparTen [54]. We use TimeLoop [80] to simulate Eyeriss, and use DNNSim [81] to simulate SCNN, respectively. For SparTen, we implement a cycle-accurate simulator and verify it against results reported by the original paper. CACTI is used to estimate the area and energy consumption of on-chip buffers. We extract the energy consumption of unit operations under the same 65nm technology node, as shown in Table 3.2, to estimate the energy cost of other hardware components.

Table 3.3: Power and Area estimation of PE Block(65nm)

| Component | Area(mm ²) | Power(mW) |
|-------------------|------------------------|-----------|
| Activation Buffer | 0.0098 | 5.44 |
| MAC Row | 0.0159 | 7.79 |
| Dilution | 0.0450 | 17.77 |
| Concentration | 0.0906 | 46.74 |
| Coef.&Psum Buffer | 0.0538 | 8.33 |
| Total | 0.2150 | 86.07 |

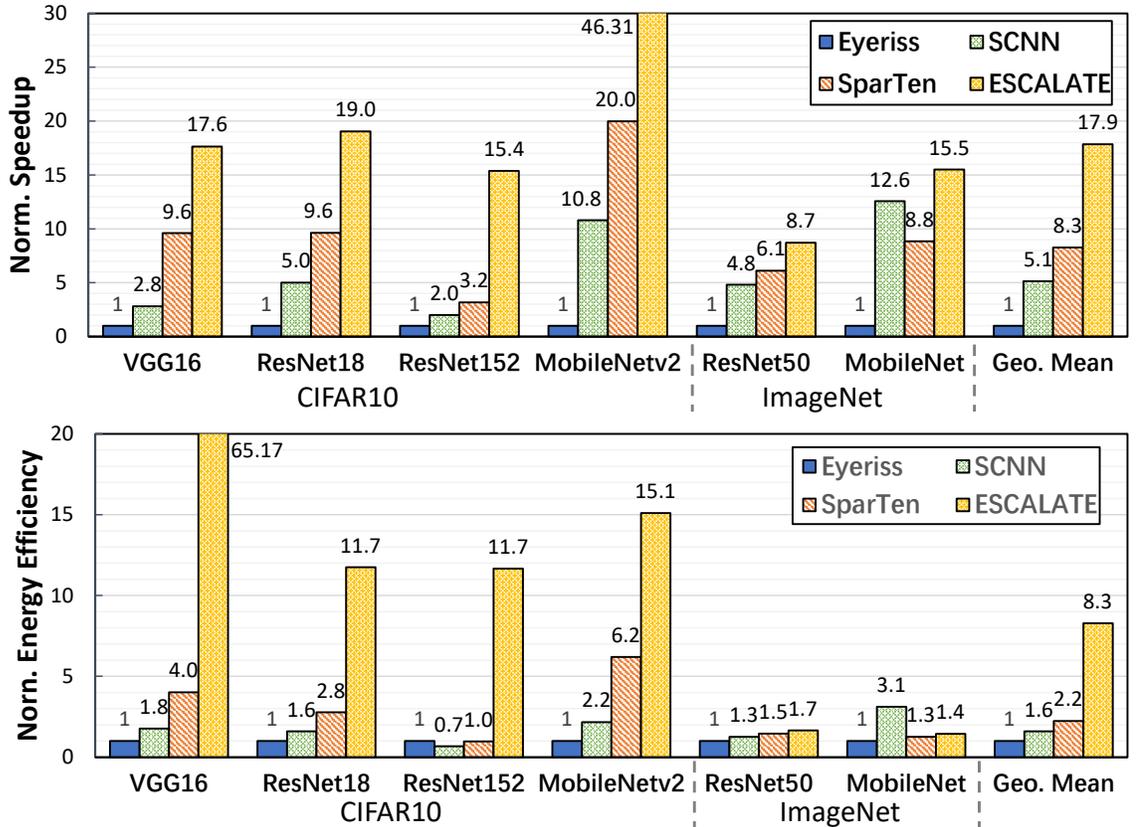


FIGURE 3.8: The normalized speedup and energy efficiency (*over Eyeriss*) of ESCALATE and baseline accelerators.

According to [82], the energy cost of DRAM accesses can be approximated as 100pJ per 8 bits. The configuration of all baseline designs is adjusted to have the same number of multipliers. For baseline accelerators, we use the model checkpoints from ADMM-NN-S [58] for all CIFAR-10 models except ResNet152 and the checkpoints from STR [57] for all ImageNet models. Since no previous work provides a checkpoint for ResNet152, we use the naïve magnitude-based pruning method with $l1$ -regularization to build a sparse model. The sparsity of the models used for baseline accelerators can be found in Table 2.6. Since the result is also related to the activation sparsity, the result may vary with different input samples. We randomly generate 10 input samples and present the average speedup and energy consumption. We list the configurations of ESCALATE accelerator and baseline accelerators in Table 3.1.

3.7.2 Main Result

Figure 3.8 showcases the normalized speedup and energy efficiency of all accelerators over the Eyeriss baseline. Compared with the baseline accelerators, ESCALATE achieves the best performance across all evaluated models. Compared with Eyeriss, which does not exploit sparsity, ESCALATE achieves a speedup ranging from $8.7\times$ to $46.31\times$. This significant speedup is attributed to the computation reduction brought by both sparsity exploitation and the decomposed form of convolution.

Since the second stage of the decomposed convolution is dense, the upper limit for the speedup of each layer is determined by the layer shape, specifically C/d . For CIFAR-10 models, the high sparsity ($>90\%$) enables the channel accumulator to produce intermediate feature maps in time. Thus, if there are no idle MAC cycles, the speedup is bounded by the layer shape. For ImageNet models, where the sparsity is relatively low, the channel accumulator requires more cycles to produce an intermediate element than a MAC needs to consume, limiting the performance due to idle MAC cycles (discussed in Section 3.8.2). Compared with sparse baselines, ESCALATE outperforms SCNN by $3.5\times$ and SparTen by $2.16\times$ on average.

Regarding energy efficiency, the results differ based on the CNN model type. For CIFAR-10 models, where input feature maps are relatively small, the off-chip access of weights dominates energy consumption. The coefficient compression in ESCALATE enables retaining most weights on-chip during computation, almost eliminating expensive DRAM accesses for weights and resulting in over $10\times$ improvement in energy efficiency. For ImageNet models, the movement of input feature maps dominates DRAM traffic. ESCALATE has similar energy consumption to SparTen due to the same channel-first order used to process input feature maps. The large activation buffer in SCNN effectively reduces this cost, resulting in up to $3.1\times$ energy efficiency improvement over ESCALATE. On average, ESCALATE achieves an $8.3\times$ energy efficiency improvement over Eyeriss, $5.19\times$ over SCNN, and $3.78\times$ over SparTen.

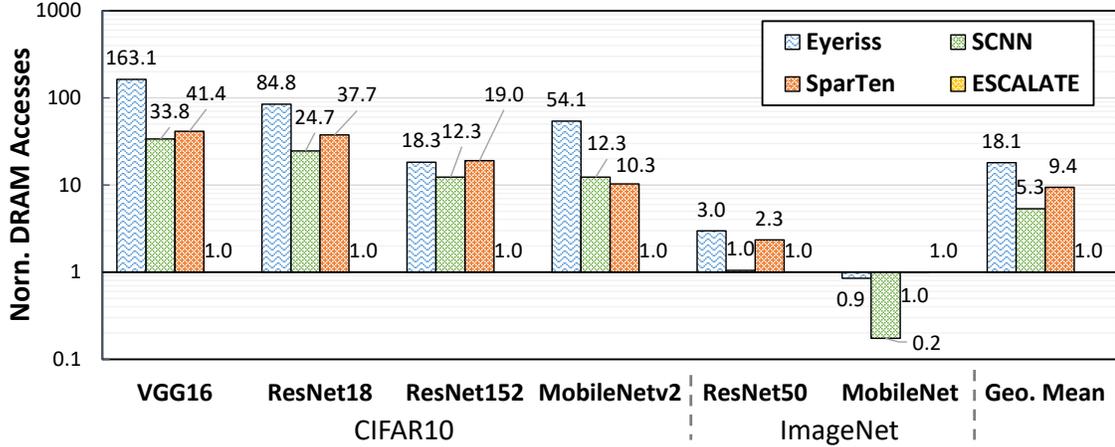


FIGURE 3.9: The normalized DRAM accesses (*relative to ESCALATE*) of baseline accelerators on all evaluated models.

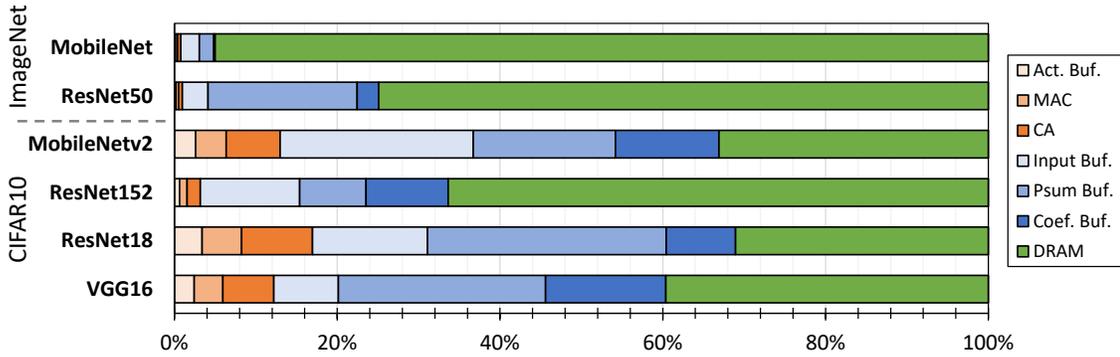


FIGURE 3.10: The inference energy breakdown of all evaluated models. We omit the output buffer since its energy consumption is negligible.

Figure 3.9 illustrates the normalized number of DRAM accesses over ESCALATE. As mentioned above, for ImageNet models, DRAM accesses for input feature maps dominate energy consumption, and ESCALATE requires similar or larger numbers of DRAM accesses compared to the baselines. On CIFAR-10, ESCALATE effectively reduces expensive DRAM accesses, with the reduction relative to SCNN resulting from eliminated off-chip weight accesses. Compared with SparTen, ESCALATE’s input buffer design exhaustively exploits input reuse without enforcing a large-scale synchronization barrier. On average, ESCALATE reduces DRAM accesses by $18.1\times$ over Eyeriss, $5.3\times$ over SCNN, and $9.4\times$ over SparTen, respectively.

Figure 3.10 shows the energy breakdown of ESCALATE accelerator on all evaluated models. Apart from the DRAM accesses we have discussed before, the breakdown also reveals a divergence in the energy consumption of buffers. For shallow models like ResNet18 or VGG16, the partial sum buffer dominates the buffer energy consumption; this is because the output feature maps are kept dense in the partial sum buffer and require frequent read-modify-write operations. For deeper models like ResNet152, the cost of reading input activations outweighs the partial sum accumulation. The large number of 1×1 convolutional layers amortizes the read-modify-write cost of other types of layers. Moreover, the ResNet152 model features more ‘late layers’, which have a large number of input channels but a relatively small input size. This observation also applies to MobileNetv2 result since MobileNetv2 quickly downscales the size of feature maps in early layers that have a small number of channels. In summary, the reduction in DRAM accesses, especially off-chip weight accesses, is the main reason for the improved energy efficiency of ESCALATE.

3.7.3 Layer-wise Analysis

The computational efficiency of the ESCALATE accelerator is affected by the layer width and the size of the input feature maps. To evaluate the impacts of these factors, a layer-wise analysis is performed. Figure 3.11 shows the layer-wise normalized speedup of ESCALATE and all baseline sparse accelerators in ResNet-18. The layer shape of each residual block is marked, and the sparsity of each layer is presented. For the first layer, ESCALATE is slower than the dense baseline for two reasons: (1) The computation is directly mapped into the MAC row without skipping zero activations or coefficients. (2) The fallback input stationary dataflow is not as efficient as the row stationary dataflow in Eyeriss.

Since the first layer contributes only a small portion of the overall computational cost, ESCALATE does not further optimize for this layer to avoid increasing design complexity and degrading the efficiency of other layers. For the other layers, a clear boundary can be observed between SCNN and SparTen. SCNN effectively utilizes spatial parallelism in early

layers, while SparTen exploits channel parallelism in late layers. ESCALATE presents a similar layer-wise speedup pattern to SparTen since both designs use input channels as the innermost loop of the dataflow. Compared with SparTen, the two overlapping computation stages of ESCALATE further boost efficiency. As mentioned earlier, due to the high sparsity in CIFAR-10 models, the performance is bounded by layer shapes. Within the first three blocks of layers, ESCALATE almost reaches the C/d limit of speedup, where C is the number of input channels and d is the number of basis kernels. For the last block, the speedup varies across layers. Since the input feature maps of these layers are very small, most intermediate results require less than RS cycles in the MAC row, leaving the MACs running idle.

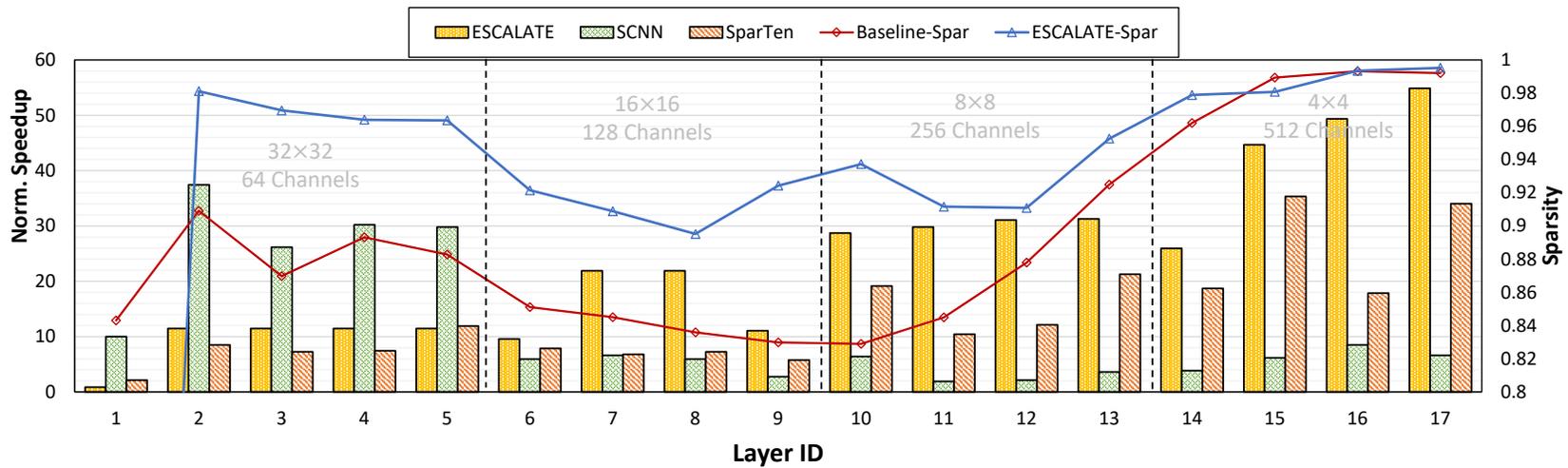


FIGURE 3.11: Layerwise sparsity and speedup over dense accelerator (Eyeriss) in ResNet-18 model.

3.8 Discussion

3.8.1 Design Trade-off

ESCALATE provides a trade-off between accuracy and latency/energy by adjusting the number of basis kernels d in the decomposition step. With a different d , we can adjust l to maintain the same number of MACs and resource consumption. Increasing the number of basis kernels can effectively increase the model’s accuracy. However, it also reduces the number of PE slices per block, leading to a reduction in row parallelism. We show the trade-off with both ResNet18 and ResNet50 models in Figure 3.12. For $d = 7$, we achieve 93.83% accuracy on ResNet18 and 74.09% on ResNet50. When increasing d , we have a smaller l under the constraints of maintaining the number of MACs, thus reducing the row parallelism and increasing the latency. The change in l also affects the number of input buffers. A larger l requires more input buffers but reduces the cost of off-chip DRAM accesses. The energy consumption of other components shows negligible change with the number of basis kernels.

3.8.2 Overhead Analysis

In ESCALATE design, both CA and MAC can be idle. If CA cannot produce a new intermediate element before MAC finishes the current computation, the MAC has to stall.

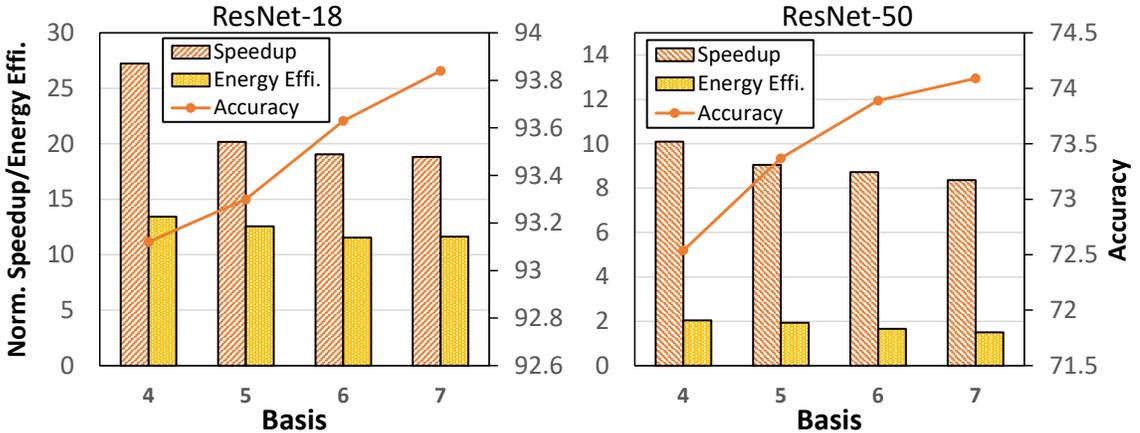


FIGURE 3.12: The accuracy and latency/energy trade-off with different numbers of basis kernels (d).

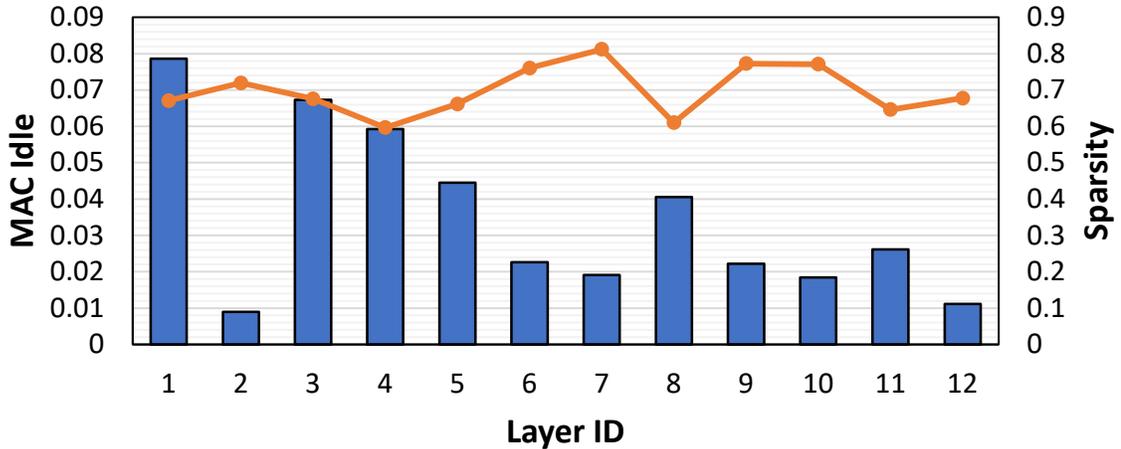


FIGURE 3.13: The MAC idle cycles and sparsity of each layer in MobileNet.

Conversely, if MAC requires more cycles to consume one element (e.g., a large kernel), the CA has to stall. We only consider the idle MACs as overhead since the idle CA can still process masks or perform concentration. As we mentioned before, in CIFAR-10 models, we did not observe significant idle MACs ($< 0.05\%$ of overall cycles) thanks to the high sparsity ratio, while the issue of idle MACs limits the speedup of ImageNet models. We show the layer-wise portion of MAC idle cycles in Figure 3.13. The portion of idle cycles is determined by both the sparsity ratio and the distribution of non-zero elements. CA requires more cycles to process the computation corresponding to denser coefficients, leading to more idle cycles of MAC.

3.8.3 Modern Compact CNNs

Through our experiments, we observed that the variation in layer dimensions could complicate the accelerator design. For sparse accelerators, it's difficult to be efficient under *all* types of layer configurations. Modern compact CNNs, like EfficientNet [19], usually include a rich set of layer variants, making it hard to be efficiently deployed to a sparse-aware accelerator. We also noticed that, in our experiments, sparse VGG16 is $1.5\times$ faster than sparse MobileNetv2, while they achieve a similar accuracy (0.5% difference). Modern compact models are mostly designed for general-purpose processors on the edge. Since these

platforms cannot efficiently support processing sparse data, the compact models do not include the performance of the sparse version into their design considerations. They are not suitable for sparse-aware accelerators and might even be outperformed by the sparse version of large and redundant models. This situation motivates us to explore the possibilities of jointly designing sparse-aware accelerator and hardware-aware CNN models in future works.

3.9 Conclusion

In this chapter, we present the ESCALATE accelerator, a sparse DNN accelerator that supports the decomposed convolution produced by PENNI. We propose a ‘Basis-First’ dataflow and corresponding microarchitecture design to support the PENNI-compressed CNN model. The evaluation shows that the ESCALATE accelerator outperforms previous sparse CNN accelerator designs with up to $2.16\times$ reduction in latency and $3.77\times$ improvement in energy efficiency. By combining the ESCALATE algorithm and the ESCALATE accelerator into a cohesive algorithm-hardware co-design approach, we unlock significant performance and efficiency benefits that surpass what can be achieved by treating the compression algorithm and hardware architecture as separate components. The ESCALATE framework demonstrates the importance of co-designing the algorithm and hardware in a coherent manner, enabling the synergistic optimization of the entire inference pipeline. This co-design approach is particularly effective for executing compressed DNN models on resource-constrained edge devices, where both high computational performance and energy efficiency are critical.

4. Near-Data Processing System for Large-Scale DNN-based Recommendation Model Training¹

In previous chapters, we discuss the inference acceleration for sparse DNNs. Since the inference scenario is relatively simple and typically deployed at the edge (e.g., mobile phone, IoT devices, etc.), a dedicated accelerator design could fulfill the requirement and it does not involve the interaction between different components in the system. In this chapter, we shift focus to the system level and discuss how to build a holistic system to support large-scale training workloads.

Deep learning-based recommendation models are widely used in a variety of services. DLRM, one of the most popular models, uses embedding layers to process the categorical features. The parameters of the embedding layer could consume terabytes of memory, while the pattern of parameter access is random and extremely sparse. The massive embedding layer necessitates a large amount of GPU memory to store embedding tables during training, resulting in a large number of underutilized GPUs given the growing size of the model and data. Furthermore, the irregular access to a large number of embedding vectors and the corresponding embedding operations demand a high memory bandwidth. In this chapter, we introduce NDRec [3], a near-data processing system for large-scale DNN-based recommendation model training to tackle this challenge.

4.1 Background and Related Works

4.1.1 DNN-based Recommendation Model

We focus on DLRM [83], a representative DNN-based recommendation model that has been widely used in the industry. Figure 4.1 shows the general model architecture of DLRM, which takes both dense and sparse features as inputs. Dense features are numerical values, such as length of time. Sparse features are categorical values, such as genres of movies. Dense features are processed by a bottom MLP, while sparse features are transformed into

¹ This chapter is primarily based on: NDRec: A Near-Data Processing System for Training Large-Scale Recommendation Models. in *IEEE Transactions on Computers, Early Access*. © 2024 IEEE. <http://dx.doi.org/10.1109/TC.2024.3365939>

dense vectors by an embedding layer. The outputs of the bottom MLP and the embedding layer are combined by a feature interaction operation and fed to a top MLP to generate the prediction output. The inference process of DLRM handles single or small batches of dynamic input samples, with low latency as the main optimization goal. The training process of DLRM processes large batches of input samples from a static dataset, with high throughput as the main optimization goal. The training process is typically performed on GPU or other powerful accelerators.

The embedding layer consists of multiple EMBs, each corresponding to one sparse feature. Each row of an EMB, i.e., an embedding vector, represents one category, and each sparse feature contains one or more indices that refer to the rows of the EMB. In the forward pass of the embedding layer, we fetch the *embedding vectors* (EVs) from each EMB according to the indices in the corresponding sparse feature and aggregate these vectors. Most recommendation models use summation as the aggregation operator. To avoid confusion with the embedding vector, we use *embedding output* (EO) to denote the aggregated result from one embedding table. These terms are also illustrated in Figure 4.1.

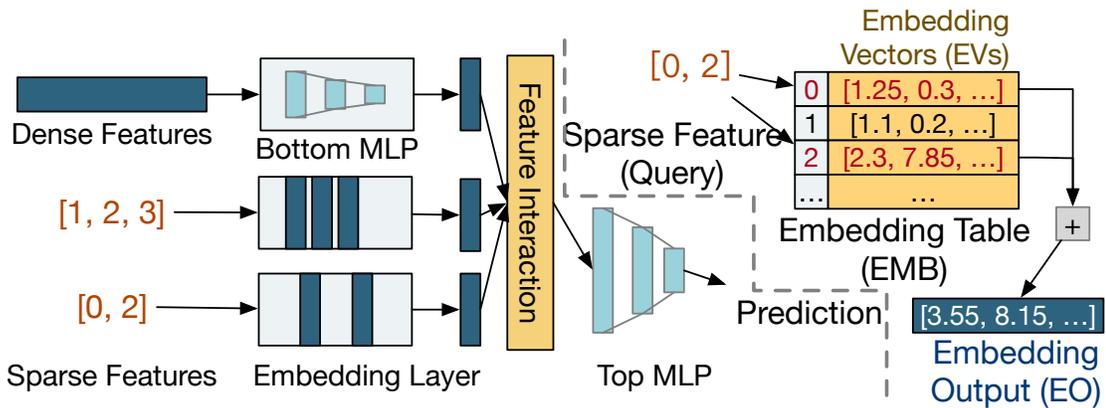


FIGURE 4.1: (left) The general architecture of DLRM. (right) An example of the embedding operation.

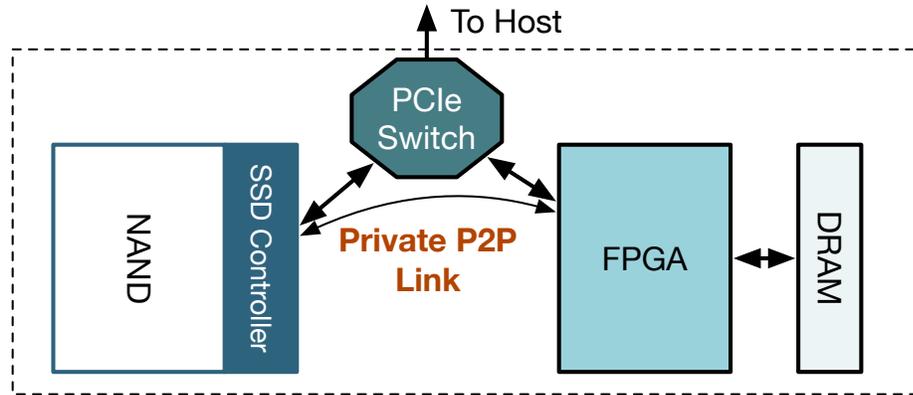


FIGURE 4.2: SmartSSD Architecture

4.1.2 Computational Storage

Moving large amounts of data from storage to host main memory for data-intensive application processing may lead to bottlenecks resulting from limited I/O bandwidth. Computational storage devices (CSDs) are introduced to address this issue by allowing the processing of the data in place. Storage vendors have proposed two paradigms of CSDs: (1) integrating the processing units into the SSD controllers [84]; (2) building a separate processing unit near the SSD with a private P2P link [85]. We select the latter type of CSD design, specifically the SmartSSD, for its flexibility in customizing the hardware. SmartSSD builds a separate FPGA chip alongside the SSD device and provides a private PCIe 3.0 \times 4 link between the two parts, as shown in Fig. 4.2. The FPGA has a dedicated 4GB DRAM for a customized accelerator design. The functionalities related to SSD—interface handling, wear leveling, error correction, etc—are implemented in the SSD controller. The data movement between SSD and FPGA’s DRAM can be invoked with a command by the host program, but the actual transfer bypasses the host [86].

4.1.3 Coherent Interconnect

Conventional system interconnects, for example, PCI-express (PCIe), were designed for IO devices. Although such interconnects are able to provide high bandwidth, they are

inefficient for fine-grained transfers. Moreover, handling the coherence and synchronization with the software will incur extra overhead. Novel coherent system interconnects, e.g., Compute eXpress Link (CXL) [87], are proposed to address this issue. CXL utilizes the physical layer of PCIe and provides *CXL.io*, *CXL.mem*, and *CXL.cache* protocols for devices. *CXL.io* protocol is an enhancement of the PCIe protocol, while the *CXL.cache* and *CXL.mem* protocols have smaller packetization overhead and a higher priority, which creates low access latency. Three types of devices are defined in the CXL specification. The type-1 device only has *CXL.cache* protocol and hides its device-attached memory. Type-2 is the accelerator device with both *CXL.cache* and *CXL.mem* protocols. Type-3 devices have *CXL.mem* protocol and can be used as a memory expansion of the system.

Type-2 and Type-3 devices can (selectively) map their attached memory (e.g., the GPU memory) as host-managed device memory (HDM). HDM is mapped to the CPU’s host physical address (HPA) space and is directly accessible to the host as cacheable memory. The coherency of HDM can either be managed by the host (HDM-H) or the device (HDM-D or HDM-DB). HDM-DB utilizes back invalidation channels to enable direct snooping by the device to the host. Device-coherent HDM has a bias-based coherency model. The cacheline can be host-biased or device-biased, whose coherency is resolved at the root complex or the device coherence engine. The bias mode can be either explicitly controlled by software or managed by hardware. The device-biased mode saves unnecessary traffic to/from the host for internal memory access in Type-2 devices. Type-2 devices are also able to access any cacheable memory within the system address space with hardware coherence. We adopt the latest CXL 3.0 standard [88] in our design since it supports multiple Type-2 devices per root port, back invalidation-based device coherent HDM, and multi-level switching.

4.1.4 CXL-enabled SSD

The integration of CXL introduces an enticing prospect for leveraging SSDs as an extension of the main memory. The substantial storage capacity and cost-effectiveness of SSDs offer a promising solution to address the challenges posed by memory-intensive ap-

plications. A prior proposal [89] outlines a practical approach to developing CXL-enabled SSD devices by enhancing the SSD controller. For the SSD to function as a Type-3 device, it must provide byte addressability and, at a minimum, support access at the cacheline granularity. This requirement contrasts with the page-grained access inherent in NAND flash, potentially leading to significant read/write amplification issues. Additionally, the extended access latency and finite endurance of SSDs pose formidable obstacles in constructing such devices. A recent investigation [90] delves into design options aimed at overcoming these challenges. The study demonstrates that, through the implementation of caching, prefetching, and software assistance, it is feasible to construct CXL-enabled SSDs with commendable performance. Notably, industry vendors are actively exploring CXL-enabled SSDs and have showcased prototypes of such designs [91].

4.1.5 Accelerating DLRM Training

As a broadly adopted model in the industry, there have been numerous attempts to accelerate DLRM training. Many researchers have proposed customized DLRM accelerators to reduce query latency and improve throughput. Most prior works adopt a near-memory processing (NMP) approach, where computations are performed near the memory to alleviate bandwidth bottlenecks.

TensorDIMM [92] was the first to employ a DIMM-based NMP unit in a disaggregated GPU memory system, overcoming memory bandwidth bottlenecks for embedding operations. Centaur [93] focuses on CPU-centric systems and proposes a chiplet-based CPU+FPGA solution. RecNMP [94] introduces a lightweight near-memory architecture with customized instructions to accelerate memory-bound embedding operations, further implemented in the versatile FPGA-enabled NMP platform AxDIMM [95].

To reduce the high infrastructure costs of large and fast DRAM-based memories, RecSSD [84] utilizes Flash-based SSDs with larger capacity for industry-scale recommendation models. RecSSD is implemented in FTL firmware and compatible with existing NVMe protocols, requiring no hardware changes. RM-SSD [96] goes further by building an MLP

engine to address the computation challenge for MLP-dominated models.

These accelerators focus on the inference task and leverage knowledge such as the potential distribution of accesses to address bottlenecks. An alternative solution is to optimize the memory hierarchy and/or memory access pattern without designing new hardware. In this approach, the embedding table is stored in main memory and/or storage, and only necessary embedding vectors are retrieved during each iteration.

cDLRM [97], ScratchPipe [98], and BagPipe [99] use a two-level storage system (main memory and GPU memory), prefetching data to GPU memory with a caching strategy. AIBox [100] and RecShard [101] also include storage in their tiered systems. Apart from caching strategies, these solutions leverage statistical features of embedding vectors to determine data placement.

4.2 Motivation

DNN-based recommendation models have increasing memory capacity and bandwidth requirements, but their computation demand does not scale proportionally. Based on the representative model configurations from Meta [102], we observe that: 1). The computation demand of the embedding layer is determined by the embedding dimensions (i.e., the number of EVs to reduce) and the number of tables, not by the number of parameters. 2). The models with more computation demand in the embedding layer (i.e., with more embedding tables) also have more and wider MLP layers, which are the dominant operators. These observations suggest that the embedding layer can be offloaded from GPU without creating a new bottleneck.

SmartSSD is a potential candidate for offloading the embedding, as it provides large and cheap capacity with SSD. However, the limited memory bandwidth and long access latency of SSD pose challenges for implementing embedding. We explore opportunities in the overall training process to overcome these challenges.

Opportunity 1: Use the knowledge of access pattern to embedding table for scheduling.

Unlike in inference, we can know the exact inputs for future iterations during training.

This information can help us schedule the data movement among SSD, DRAM Cache, and on-chip buffer. Furthermore, previous works [96], [101] have shown that the embedding vector access distribution is skewed. A small subset of embedding vectors accounts for a large fraction of the total accesses. We can leverage this statistical feature in the caching strategy design to minimize access to SSD.

Opportunity 2: Relax the dependency to reduce the bandwidth demand. For each training iteration, the embedding operation has a read-after-write (RAW) dependency with the weight update process of the previous iteration. The subsequent forward process also depends on the embedding operation (RAW). This means that even if we offload the embedding operation to other devices, we still have to wait for the previous training iteration to finish before the embedding operation can start and produce the result for GPU. The embedding operation cannot run in parallel with other processes on GPUs. These dependencies drive the high memory bandwidth demand. We found it is possible to break the embedding operation into two stages. In the first stage, we use the old embedding vectors to compute an intermediate embedding output. This stage is memory-intensive but can start earlier in the previous iteration. In the second stage, we correct the intermediate result when the gradients are available. This relaxation preserves the ‘illusion’ of respecting RAW dependencies (i.e., the optimizer behavior is unchanged) and enables concurrent execution between GPU and SmartSSD. The concurrent execution can hide the latency of the embedding operation, thus lowering the bandwidth requirement for the same target training throughput.

Opportunity 3: Trade the cheap GPU computation for expensive memory bandwidth. The aforementioned correction to the embedding output can be represented as a matrix multiplication between the gradients and the indicator matrix of the overlapped embedding vectors between two iterations. The batched matrix multiplication is a relatively simple and well-optimized task for GPU. Since the cost of a compute-saturated task is lower than the memory-saturated one [103], we argue that it is worthwhile to trade the extra computation on GPU for significantly reducing the bandwidth demand of embedding operation

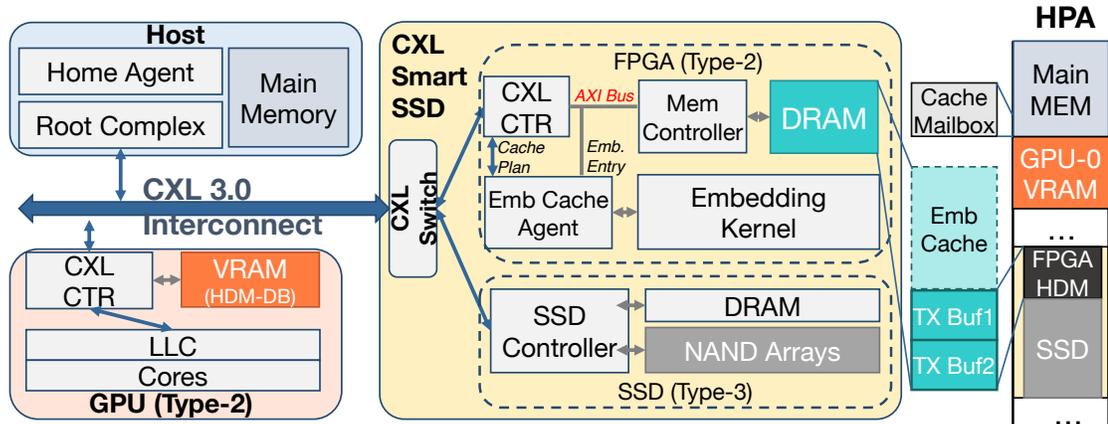


FIGURE 4.3: The architecture of the NDRec system and the allocation of the host physical Address Space (HPA).

on SmartSSD. Moreover, the extra matrix multiplication on GPU has a negligible overhead compared with the MLPs.

4.3 NDRec System

4.3.1 System Architecture

The proposed design is shown in Figure 4.3. The blue arrow indicates coherent traffic in the CXL subsystem, while the gray arrows show internal traffic. ‘HDM-DB’ stands for host-managed memory with device coherent using back-invalidation. NDRec system consists of host processors with CXL root complex, CXL-enabled GPUs, and CXL-enabled SmartSSDs. A CXL-enabled GPU is a Type-2 device that supports io, cache, and memory protocols. Meanwhile, a CXL controller replaces the original PCIe controller and handles the protocol traffic on the GPU. The CXL controller has a device coherence engine that resolves accesses from the CXL.mem protocol or peer caches and forwards cache coherence messages to the last-level cache. GPU memory (VRAM in the figure) is mapped as HDM-DB, which uses back invalidation channels to maintain coherency with the host. This is because the access latency from GPU’s LLC to GPU memory is critical for the performance of the kernels running on GPU. We assume that hardware manages the bias mode autonomously, as suggested by the spec [88].

Because there is yet to be any CXL-enabled SmartSSD on the market, we extrapolate its design based on existing architecture. The device consists of a CXL switch, an SSD with CXL interface as a Type-3 device, and an FPGA as a Type-2 device. For the SSD, we follow a previous proposal [89] where the SSD controller directly supports memory protocol and byte addressability. We use non-deterministic (ND) and bufferable (BF) attributes for all requests to the SSD for performance reasons, i.e., improving asynchrony and overlap of compute and multiple in-flight requests. The original controller functions, such as wear leveling, error corrections, etc., remain unchanged. Load/store requests from the memory protocol are translated into read and write requests and sent to the controller. The control and management commands can be issued through the io protocol. From the system perspective, the SSD can now be accessed with load/store requests as regular memory. For the FPGA, we replace the original PCIe endpoint with a CXL controller, similar to the CXL-enabled GPU. The CXL controller also has a coherence engine for resolving the request to the FPGA memory. We map part of the FPGA memory as HDM-DB, which will be used as transaction buffers (Section 4.3.5). Software controls the bias mode of this region. The coherence engine handles the external and internal access to the transaction buffers to maintain coherency. The rest of the FPGA memory is used as the embedding cache region and reserved as private device-attached memory (PDM).

The remaining resources of the FPGA are reserved for customized logic, where we build an embedding cache agent and an embedding kernel. The embedding cache agent cooperates with the host cache planning process to manage the embedding cache (Section 4.4.1). A hardware kernel performs the forward and backward computation of the embedding operation (Section 4.4.2). Meanwhile, the embedding kernel sends read/write requests of the EVs to the embedding cache agent, in which the counter and statistics of the EV are updated, and the request is fulfilled by forwarding the request to the AXI bus. Access to EVs is non-coherent since EVs reside in the reserved PDM region (embedding cache region). Meanwhile, the embedding cache agent pulls cache plans from the cache mailbox region in the host memory and executes these plans by sending read and write requests to the CXL

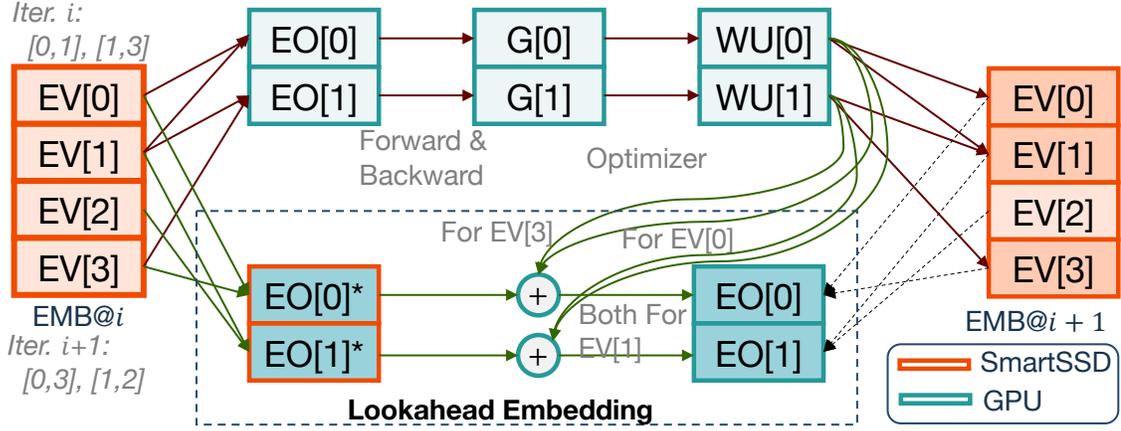


FIGURE 4.4: The illustration of lookahead embedding assuming there are two samples per batch.

controller. The EVs in the embedding cache can be viewed as coherent snapshots of the EVs stored in SSD. Load/eviction operations are implemented with memory copy.

All aforementioned memories, except for the embedding cache region and the SSD’s DRAM, are mapped into the host physical address space. The SSD controller uses the DRAM to run the firmware and buffer data, and it is transparent to the system. The system can have multiple GPUs and/or SmartSSDs, which can be connected to the root port directly or through multiple levels of switching. In this work, we study the configuration with up to two levels of switching and a simple tree topology. We avoid using more advanced fabric management features in CXL 3.0 due to a lack of performance studies to cross-validate against.

4.3.2 Lookahead Embedding

As we mentioned in section 4.2, the RAW dependencies of the embedding layer prevent concurrent execution without speculation. This restriction imposes a tight latency constraint on the memory-intensive embedding operation, thus requiring high memory bandwidth. By offloading the embedding operation, we also hope to perform the embedding operation concurrently with other operations running on GPU and relax the bandwidth requirement.

Assuming the recommendation model uses the summation as the pooling operation, the computation of the embedding output can be represented as,

$$v_{i,j}^t = \sum_{k=1}^{N_{i,j}^t} E_j^t[O_{i,j}[k]], \quad (4.1)$$

where the subscripts i , j , and t represent the j -th sparse feature of the i -th sample in the t -th iteration; E_j is the j -th EMB, $O_{i,j}$ is the array of indices of the current query, and $N_{i,j}$ is the number of those indices. The backward process is done by aggregating the gradient vectors from the samples that include the EV in the forward pass,

$$\frac{\partial L}{\partial E_j^t[k]} = \sum_i \frac{\partial L}{\partial v_{i,j}^t} \mathbb{I}_{k \in O_{i,j}^t}, \quad (4.2)$$

where $\mathbb{I}_{k \in O_{i,j}^t}$ is an indicator function.

Thus, we may decompose the process of computing EVs in Equation (4.1) into two stages, embedding lookup with the old embedding table and correction with the new gradients. Specifically, if with SGD optimizer, the EV in Equation (4.1) can be calculated as,

$$v_{i,j}^t = \sum_{k=1}^{N_{i,j}^t} E_j^{t-1}[O_{i,j}^t[k]] - \eta \sum_{p \in O_{i,j}^t} \sum_q^{N_{batch}} \frac{\partial L}{\partial v_{q,j}^{t-1}} \mathbb{I}_{p \in O_{q,j}^{t-1}}, \quad (4.3)$$

where η is the learning rate, N_{batch} is the batch size, p covers all indices related to the i -th sample of the current iteration, and q captures the sample indices that include the entry p in the previous iteration. The second term indicates the difference between the embedding vector calculated with the old embedding table and the updated embedding table. It can also be expressed in the vector form,

$$M_i \frac{\partial L}{\partial v_{:,j}^{t-1}}, M_i = \left[\sum_p \mathbb{I}_{k \in O_{i,j}^{t-1}}, \dots, \sum_p \mathbb{I}_{k \in O_{N_{batch},j}^{t-1}} \right]^T. \quad (4.4)$$

Since the indices of each batch are determined for each training run, the vector M_i can be generated through the preprocessing on the CPU. Thus, we can compute the first term in

Equation (4.3) on SmartSSD concurrently with the training process running on the GPU. The result is sent to GPU. Then, the correct EV can be obtained by performing a matrix multiplication on GPU according to Equation (4.3).

We name this scheme as *lookahead embedding*. Figure 4.4 depicts the lookahead embedding process. ‘G’ and ‘WU’ in the figure denote gradients and weight updates, respectively. ‘EMB@ i ’ stands for the embedding table at the start of the i -th iteration. We omit the lookahead embedding for the i -th iteration for readability. Lookahead embedding enables the concurrent execution of GPU and the SmartSSD. Figure 4.5 compares the training timelines with and without the lookahead embedding scheme. ‘WU’ in the figure stands for weight update. For example, when the GPU is processing the forward/backward for the n -th iteration, the backward process updates the embedding vectors with the gradients generated at the end of the $(n-1)$ -th iteration. Meanwhile, the forward process speculatively calculates the embedding output for the $(n+1)$ -th iteration with the forwarded embedding vectors updated by the backward process of the $(n-1)$ -th iteration. We will discuss the forwarding in Section 4.4.2. In other words, the lookahead embedding removes the RAW dependency between the forward process of T_{n+1} and the backward process of T_n . Instead, the forward process of T_{n+1} relies on the output of the backward process of T_{n-1} .

Although we only discussed the model with summation as the pooling operator, as previous works did [92], [94], [95], [101], the proposed lookahead embedding also applies to other linear operators (e.g., mean, weighted sum, etc.). We also noticed sum and mean are the only pooling operators supported by a majority of recommendation model frameworks [104], [105] or adopted by the published model configurations [102], [106]. The decomposition made by lookahead embedding does not make any approximation in embedding lookup. The EOs used by the forward process are the accurate ones computed in two stages. The EV update is delayed by one iteration but still follows the original computation formula. Thus, lookahead embedding does not alter the behavior of the training process and has no impact to the accuracy.

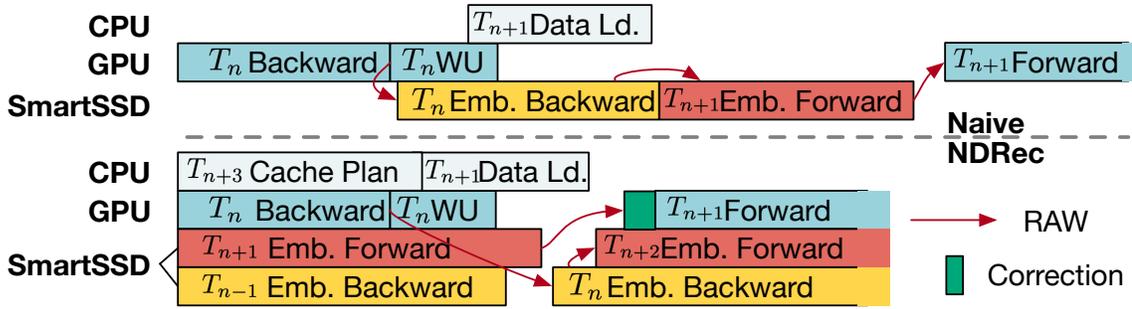


FIGURE 4.5: The illustration of the training timeline of naïve offloading and NDRRec.

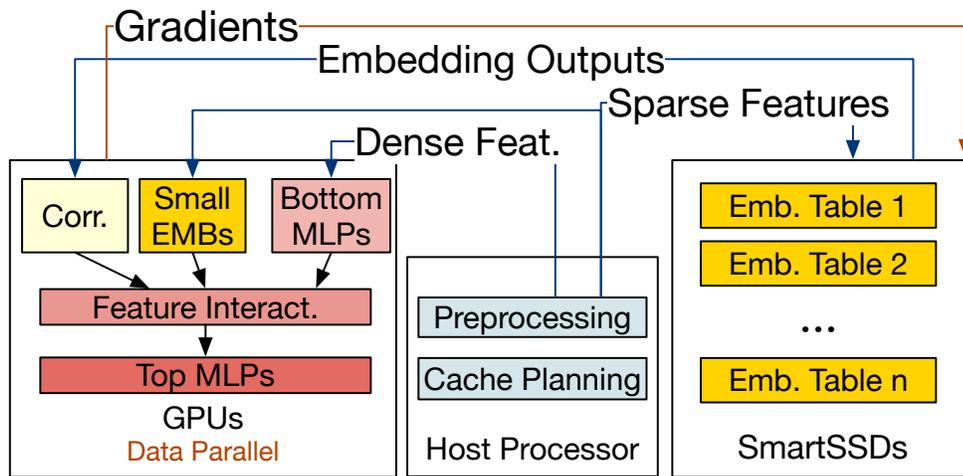


FIGURE 4.6: The task coordination in NDRRec system.

4.3.3 Task Coordination

The major objective of NDRRec is to offload the embedding operation to computational storage devices and enjoy the large, cheap capacity provided by SSD. Unlike the original sequential execution flow of the model, offloading decomposes the recommendation model training into multiple smaller tasks that can run concurrently. Apart from the computation of MLP layers remaining in GPU and the embedding layer offloaded to the SmartSSDs, we also need to run the preprocessing process to feed the input features and one or more cache planning processes (depending on the number of the SmartSSD devices) to schedule

the data movement.

We illustrate the task coordination of the NDRec system in Figure 4.6. The GPUs process the compute-intensive part of the recommendation model (i.e., MLPs) with the data parallel strategy. Apart from the MLP layers, we also place small embedding tables on the GPU devices. The small embedding tables show the higher operational intensity (if we assume the embedding vectors can fit into the L2 cache) according to the pigeonhole principle, i.e., if the total number of embedding vectors is much smaller than the number of accesses per iteration, some embedding vectors must be accessed multiple times. The parameters of the rest of the embedding tables are stored in the SSD, and the FPGA handles the computation of the embedding lookup in each SmartSSD device. The bandwidth requirement determines the placement of embedding tables. We will provide quantitative analysis on the embedding table placement in Section 4.3.4.

The FPGA DRAM of each SmartSSD device is split into multiple regions, as shown in Figure 4.3. Two regions work as transaction (TX) buffers to hold the EOs of the current iteration (old EO buffer) or store the result of the lookahead embedding (new EO buffer) for the next iteration. The total size of embedding outputs determines the size of the TX buffer region. If we assume TX buffer 1 is used as the old EO buffer and TX buffer 2 is used as the new EO buffer, GPU will fetch from the TX Buffer 1, while the embedding kernel will write the result of lookahead embedding into the TX Buffer 2. The two buffer will switch their roles at the beginning of each iteration. At the beginning of the next iteration, TX buffer 1 becomes the new EO buffer, and the newly generated EOs will overwrite the data. TX buffer 2 becomes the old EO buffer and keeps its data for GPU to fetch. The embedding cache region occupies the remaining capacity of the memory. Currently, we have not partitioned the embedding table further and assigned them to different SSDs since cross-device access incurs high overhead.

During one training iteration, the data loader and preprocessing process load the raw input data from storage or main memory. The dense features and sparse features for small embedding tables are sent to the GPU, while the sparse features for large embedding

tables are sent to the SmartSSDs. The forward/backward computation of MLP layers on GPU and forward/backward computation of the embedding layers on SmartSSDs run concurrently, as shown in Figure 4.5. Apart from the computations, we have one cache planning process running on the CPU for each SmartSSD. The cache planning process maintains the metadata of the entries in the embedding cache, schedules the data movement, and writes the movement schedule to the cache mailbox. Before starting a new iteration, the system will perform a synchronization to guarantee all queuing data movement commands have been committed. We will discuss synchronization details in Section 4.3.5.

4.3.4 Embedding Table Placement

GPU or SSD. As aforementioned, the small embedding tables typically show higher operational intensity (FLOPs/DRAM Byte), which indicates assigning them to GPU is a better solution. We adopt a simple rule in determining the threshold on the placement decision—the operational intensity should be large enough to avoid the throughput being bounded by the memory bandwidth.

We proceed to compute the average operational intensity (AOI), where AOI is defined as the ratio of the number of floating-point operations (FLOPs) needed for embedding lookup

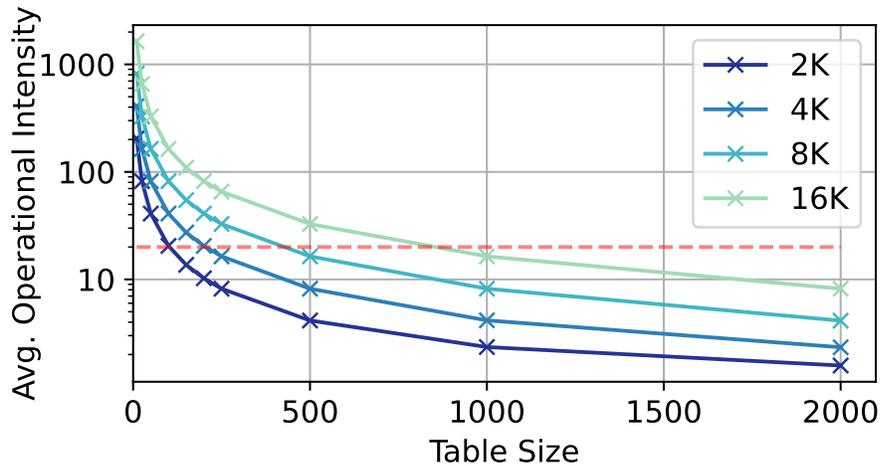


FIGURE 4.7: The operational intensity (FLOPS/DRAM byte) with different table sizes and batch sizes.

per batch to the table size. This metric serves as a proxy for estimating the position on the roofline analysis, helping discern whether the embedding lookup for a particular table is constrained by memory bandwidth or computation. Through a comparative analysis with the GPU roofline, we employ this heuristic to ascertain the optimal placement. The analysis results on our experiment platform are shown in Figure 4.7. We use the RTX A5000 for this experiment. The dashed red line indicates the operational intensity corresponding to the intersection of the bandwidth ceiling and peak performance ceiling on the roofline mode. We use this heuristic for the placement of the embedding table in the following experiments.

Among SSD. The key metric for determining the table placement across SSDs is the achieved aggregated bandwidth. As mentioned in [101], the bandwidth requirement of EMBs is mainly related to the pooling size, i.e., how many embedding vectors are involved in each embedding lookup. Since we focus on maximizing the bandwidth between DRAM and FPGA and most of the embedding vectors can stay on the on-chip buffer within one training iteration, we use the number of unique embedding vectors per iteration rather than the pooling size as the proxy of the bandwidth requirement. Thus, to derive an EMB placement plan, we sample 1% of the training data (as suggested in [101]) and calculate the average number of unique EVs per iteration. Then, we sort this number and assign a pair of tables from both ends of the sorted list to the SSDs in a round-robin fashion.

4.3.5 Communication

NDRec system performs a synchronization at the end of each iteration to guarantee resolution of the five key operations or transactions. Specifically: 1) the first stage of the lookahead embedding has finished on SmartSSD, and the result has been written into one of the TX buffers; 2) the backward process on GPU has finished, and gradients of embedding tables have been produced; 3) the backward process on FPGA has finished, and all updated are committed to the FPGA DRAM; 4) the embedding cache agent has committed all data movement schedules; 5) the cache planning process running on CPU has written the new data movement schedule to the mailbox and sent the pointer of the

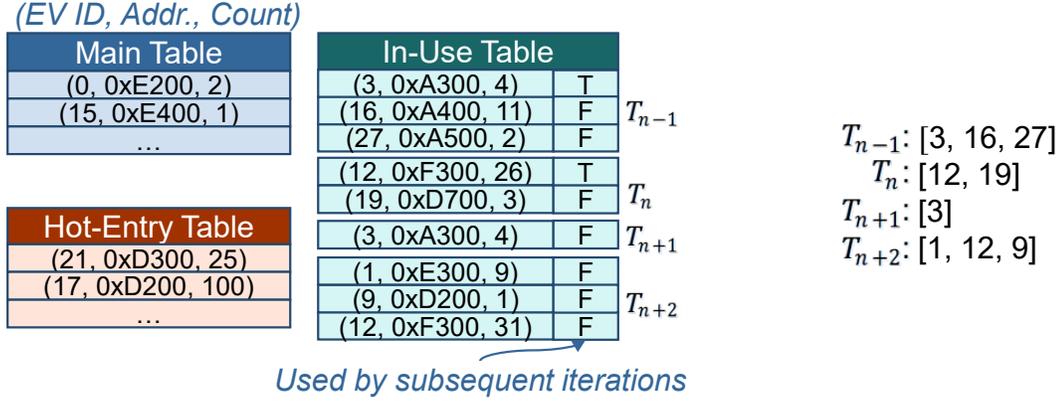
mailbox to the embedding cache agent on FPGA. Afterwards, a new iteration for training can commence.

At the beginning of each iteration, a copy of the EMB gradients is created on the GPU, and the corresponding address is passed to the embedding kernel. Then, we issue the command to FPGA and swap the role of the TX buffers. After swapping, the old EO buffer (which contains the EOs for this iteration) is configured as host-biased and ready to be fetched by the GPU, while the new EO buffer (which contains stale EOs) is configured as device-biased and ready to receive the new result generated by the embedding kernel. The write to the new EO buffer will issue a back invalidation request, flushing the cache line in all peer caches (i.e., GPU caches), and avoiding the stale data used by GPU in the future. GPU then fetches the result from the TX buffer and performs the training process, while the embedding kernel loads the gradients directly from GPU memory to start the backward process. The embedding cache agent pulls the data movement schedule from the mailbox region in host memory and executes the schedule to write the EVs in DRAM back to SSD and load new EVs. Since the embedding cache is software-managed, we need to perform a memory copy rather than directly issue a cacheable load request to avoid data consistency issues. The memory copy also helps avoid the impact of unpredictable SSD access latency during forward/backward.

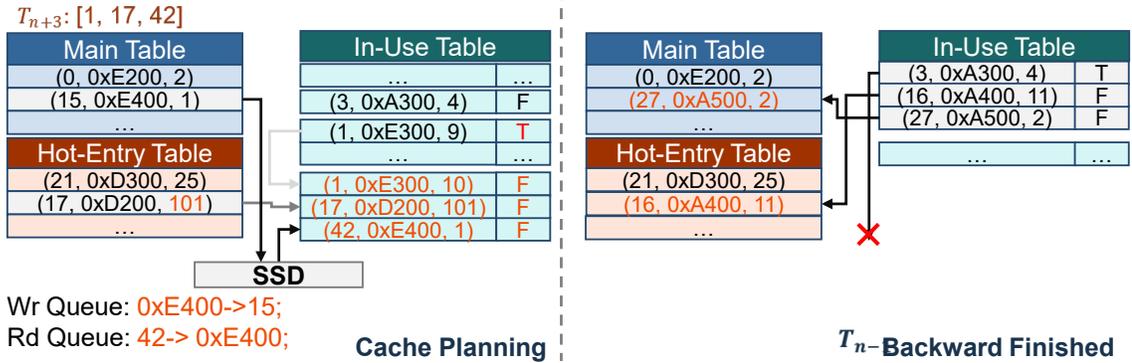
4.4 Near-Storage Embedding

4.4.1 Embedding Cache Design

As discussed in previous works [96], [101], accesses to embedding tables show a power-law distribution. Thus, we can build an embedding cache to capture this access pattern, reuse the frequently used embedding vectors, and reduce the expensive SSD access. Previous designs [97], [99] utilize the information of upcoming iterations to build a software-managed lookahead cache and preload the necessary embedding vectors to the cache. On the CPU-GPU platform, the cache management process running on the CPU needs to not only *plan* for the data movement but also *collect* and *exchange* these data from the main memory and



(a) The metadata structure of the embedding cache.



(b) An example of the management process of the embedding cache. The changes in each step are highlighted. We only show the manipulation of the metadata, and the data movement is executed only when the queues are submitted.

FIGURE 4.8: An example of the software-managed cache.

GPU memory. Both cDLRM [97] and ScratchPipe [98] show the overhead of these processes running on the CPU could dominate the latency, especially when we have multiple caches to manage (i.e., with multiple GPUs). For SmartSSD, only the *plan* phase needs to be done by the CPU. The data movement is then handled by the embedding cache agent on SmartSSD, runs asynchronously with the computational kernel, and does not require the involvement of the CPU. Thus, we can overlap the planning and exchange phases and have a higher tolerance for the search cost.

We built an embedding cache in the FPGA DRAM on each SmartSSD. The cache design is based on two principles: 1) trade the manipulation of metadata for the movement

of embedding vectors; 2) Use the access pattern’s statistical feature to reduce SSD access. Moreover, unlike traditional cache, we need to prefetch all necessary embedding vectors *before* the iteration and guarantee the hits during the forward and backward process of the embedding. Thus, we design the embedding cache as a software-managed fully-associated cache, with each cache block representing an embedding vector. The cache blocks are stored in FPGA DRAM, while all metadata is stored in the main memory and manipulated by the cache planning process on the CPU. The metadata is maintained as a table of (EV ID, Address) pairs. We also maintain an access counter for each embedding vector presented in the cache to evaluate the access frequency of the EV. Apart from the main table that contains the EV-address pairs, we also have a hot-entry table and an in-use table to utilize the statistical information of the EV access. The main table, hot-entry table, and in-use table are exclusive to each other, i.e., each EV can only be present in one of these three tables. The hot-entry table contains frequently accessed EVs. The EV will be promoted to the hot-entry table if the counter surpasses a certain threshold. The EVs in the hot-entry table will not be evicted. The in-use table tracks EVs used by the process running on the SmartSSD (i.e., four sections for T_{n-1}, T_n, T_{n+1} , and T_{n+2}). The entry in the in-use table has an extra reuse bit indicating whether they will be used by the subsequent iterations.

We illustrate the embedding cache in Figure 4.8. The cache planning process, which runs on the CPU, generates a data movement plan and writes it to the mailbox region in the main memory. The mailbox is implemented as a circular buffer, where the plan for the next iteration is appended to the end of the buffer. Upon reading the plan from the mailbox, the embedding cache agent on the FPGA executes the plan. If we assume GPU is running forward and backward processes for the n -th iteration, as shown in Figure 4.5, the forward process on SmartSSD computes for the $(n + 1)$ -th iteration. Meanwhile, the embedding cache agent executes the data movement plan to collect the EVs required by the $(n + 2)$ -th iteration, while the cache planning process on the CPU generates the plan for the $(n + 3)$ -th iteration. The cache planning process will run two iterations ahead of the current forward process running on the SmartSSD so that data movement for the next

iteration can overlap with cache planning.

In the cache planning phase, we first create a section in the in-use table to collect all EVs to be used by the next iteration (i.e., T_{n+3}). The metadata of these EVs can either be from the in-use table, that main or hot-entry table, or a new entry created for a load from SSD. If the EV is also used in the previous iterations (e.g., EV 1 in the figure), we directly get the metadata from the corresponding section in the in-use table and update the reuse bit, indicating the EV should not be returned to the main or hot-entry table. For the rest of the EVs, we first search in the hot-entry table, then the main table for the entry. The hit will remove the EV from the main or hot-entry table and add it to the new section of the in-use table. If the EV cannot be found in both tables (e.g., EV 42 in the figure), we generate an SSD access command and randomly select one EV (e.g., EV 15 in the figure) in the main table for eviction. A new entry will be created in the in-use table for this EV. The SSD access and eviction commands are sent to two priority queues, where a coalescer will check whether the command can be merged to improve the transfer efficiency. Finally, at the synchronization point, the plan generated from the two queues will be written into the mailbox. The plan guarantees that the write command (i.e., eviction) will be first executed to avoid the dirty entries being overlapped by the new entries. Meanwhile, we will also flush the table for T_{n-1} by returning the EVs not used by later iterations back to the main table.

When there is a hit, the counter related to the corresponding EV will be updated. We will compare the updated counter with a predefined threshold and promote the entry to the hot-entry table when the entry is flushed from the in-use table if it is larger than the threshold. When the hot-entry table is full (i.e., larger than a predefined size), the tail entry will be evicted back to the main table from the hot-entry table. We will discuss the selection of the parameters for the hot-entry table in Section 4.5.3.

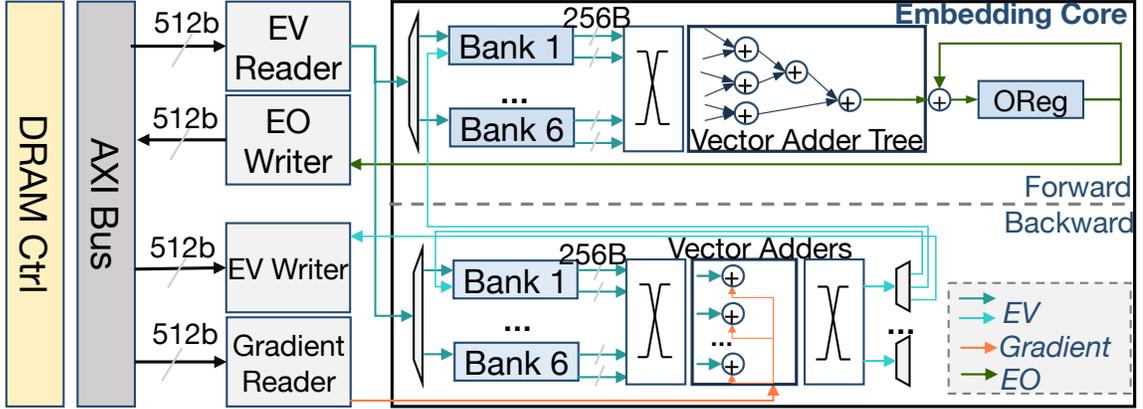


FIGURE 4.9: An example of the embedding kernel configuration with the embedding dimension of 64.

4.4.2 Embedding Kernel

The embedding kernel design aims to maximize the reuse of the embedding vectors presented on-chip and reduce off-chip access. We utilize the overlap of embedding vectors between consecutive iterations to achieve this goal. If we take the timeline presented in Figure 4.5 for example, when we start the T_{n+2} forward process, the EVs used by the forward process of T_{n+1} and not used by the backward process of T_n are presented in the on-chip buffer and does not need to be reloaded from the embedding cache. Similarly, the backward process of T_n can reuse the EVs from the backward process of T_{n-1} . Meanwhile, the EVs updated by the backward process of T_n can be forwarded to the forward process of T_{n+2} to run these two processes concurrently.

The architecture of the proposed embedding kernel is depicted in Figure 4.9. We used a banked buffer design where each buffer bank corresponds to the number of dual-port BRAM blocks (32-bit per port [107]) that can provide the width of two embedding vectors. The bank size depends on the embedding dimension. For example, if the dimension is 64, each buffer bank of the embedding table contains 64 BRAM blocks or 32 URAM blocks. We can build 12 such buffer banks given the available memory resources for the kernel and memory controller on SmartSSD [108]. We then evenly split the buffer banks and used

them for the forward and backward process, respectively. For the forward process, the EVs read from the buffer are then sent to a crossbar and a vector adders tree to generate the partial embedding output. The output is then accumulated in the output register and sent to the result writer when all EVs are reduced. For the backward process, one gradient vector (processed by the optimizer) is loaded from the gradient reader per cycle. All EVs related to the gradient vector are read from the buffer, updated, and then written back to the buffer. When all gradients related to the EV have been applied, the updated EV will be forwarded to the buffer for the forward computation or written back to memory if it is not used in subsequent iterations. The assignment of the EVs to buffer banks is done during the preprocessing phase. We assign the EVs in a round-robin manner and swap the assignment with other EVs in the same iteration if there is a bank conflict. When the forward/backward process is done for one iteration, the buffer will hold EVs for the next iteration to reuse them.

The kernel is connected to the memory controller through the AXI bus. We build two reader units and two writer units. The EV reader receives the requests from the controller and initiates the request to the memory bus. Since the length of each EV could be larger than the width of the memory port, the reader will send a burst request, compose the response data into one vector, and send it back to the block. The gradient reader sends requests to load the gradient from GPU memory and sends it to the vector adder for the backward process. The EO writer stores the embedding output back to memory. Meanwhile, the EO writer updates the embedding table entries in the embedding cache. The readers/writers are connected to the AXI bus through 512-bit ports to match the width of the memory controller.

4.5 Evaluation

4.5.1 Experiment Settings

Platform. Due to the lack of commercially available CXL 3.0 systems or official support for P2P access between GPU and SmartSSD, we are not able to fully evaluate NDRec on

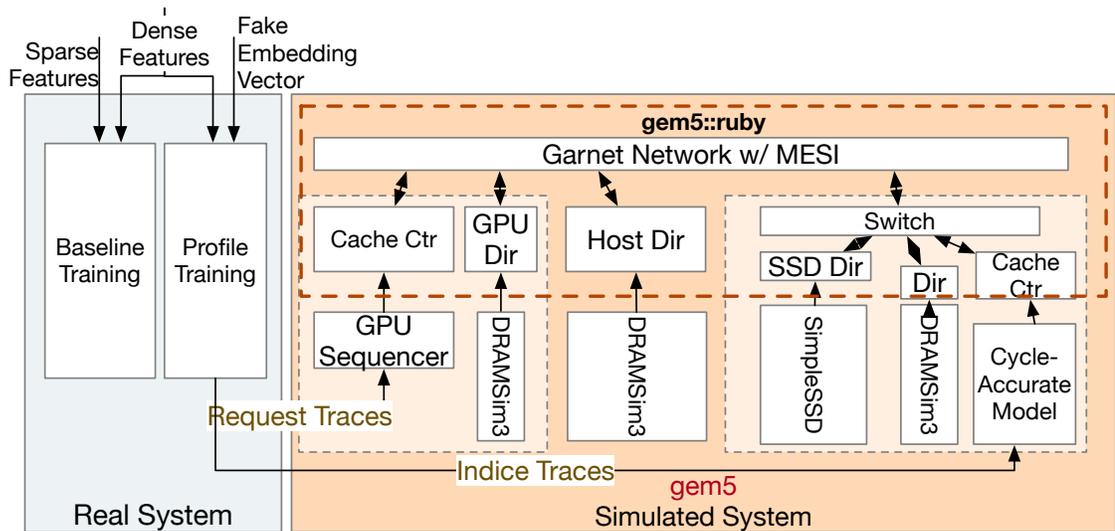


FIGURE 4.10: The simulation framework.

a physical system. Instead, we design our evaluation setting based on RecNMP [94]. We use TorchRec [104] to run the training pipeline on the real GPU-based system and replace the embedding operation with a predefined tensor. The traces of the training process, containing the detailed latency of each event during training, are then exported using profiling tools. Then, we feed these traces into our simulation framework. The experiment methodology is depicted in Figure 4.10. We performed our study with gem5 [109]. The CXL is simulated using the Ruby memory system and Garnet network model [110] with the port, link, and switch latencies shown in [111]. We modified the MESI protocol provided in Ruby to support CXL. We downgrade the link bandwidth to make a fair comparison with the PCIe 4.0 bus in the baseline system. The proposed embedding kernel design is implemented with Xilinx HLS and verified on the SmartSSD device. Then, we built a cycle-accurate simulation model for embedding kernel running on SmartSSD according to the implementation result. DRAMSim3 [112] and MQSim-CXL [90] are used to simulate the DRAM and SSD with a CXL interface in the SmartSSD device, respectively. Table 4.1

lists key hardware setups and parameters we used for evaluation. We use 200 as the hot-entry threshold and 2048 as the hot-entry table size in the embedding cache. The choice of these parameters will be discussed in Section 4.5.3. We extract the traces of 100 iterations in the middle of one epoch for simulation and report the average latency.

Benchmarks. We use two publicly available datasets, Kaggle Display Advertising Challenge (DAC) [113], and Criteo Terabyte [114]. Both datasets have 13 dense features and 26 sparse features. The Kaggle DAC and Criteo Terabyte have 39.3M and 645M training samples, respectively. We also create two synthetic datasets that follow the distribution of Terabyte. We create two models for the random datasets to evaluate the configuration with higher computation demand for embedding (Random-XL, pooling size 40) and the MLPs (Random-MLP, pooling size 10), respectively. The details on model configurations are listed in Table 4.2.

Baselines. We select two optimized out-of-core training implementations provided by the TorchRec framework, UVM and UVM-Caching, and a look-ahead caching solution cDLRM [97] as our baselines. We compare NDRec with a near-memory processing solution based on the design of RecNMP [94]. We follow the design of RecNMP-opt, implement the performance model following the description in the paper, and connect the NMP-enhanced memory modules with the system as a type-3 device in CXL (NMP-CXL). We follow the parameters in the original papers with 2 DIMMs \times 4 Ranks configuration and only increase

Table 4.1: Hardware Setup for the evaluation

| Server | |
|-------------------|-----------------------------------|
| CPU | AMD EPYC 7352 \times 2 @ 2.8GHz |
| GPU | NVIDIA RTX A5000 24GB \times 4 |
| DRAM | 16 \times DDR4 3200 64GB DIMM |
| Mem Ctrl. | 8 Channels/Node \times 2 Nodes |
| Simulation | |
| DRAM | DDR4 2400 8Gb \times 4 |
| SSD | Samsung 983 DCT 3.84 TB |
| Interconnect | CXL 3.0 \times 16@16GT/s/lane |
| Kernel Freq. | 250MHz |
| # of SmartSSDs | 4 |

Table 4.2: Model Configurations

| Model | # MLPs | MLP Size | Dim | Tables | EMB Size |
|-----------------|--------|------------|-----|--------|-----------|
| | | (Mean/Max) | | | |
| Kaggle DAC | 7 | 231(512) | 16 | 26 | 2.06 GB |
| Criteo Terabyte | 8 | 465(1024) | 128 | 26 | 91.10 GB |
| Random-XL | 7 | 460(1024) | 128 | 52 | 538.90 GB |
| Random-MLP | 20 | 1590(5120) | 64 | 16 | 75.90 GB |

the number of channels to 16 to have sufficient capacity to hold the model. We did not compare with the more recent work RecShard [101] since the publicly available datasets are too small for RecShard, as mentioned by the authors. The randomly generated dataset may make an unfair comparison since RecShard Relies on the statistical features of the embedding tables. The UVM strategy uses the unified virtual memory feature provided by the GPU and stores the embedding tables in DRAM, while UVM-Caching adopts an LRU cache in EV granularity to manage the movement of embedding vectors. We use the codebase published by the authors [115] with the recommended parameters discussed in the original paper to reproduce the result of cDLRM on our server. For the Kaggle DAC model whose parameters are able to fit into the GPU memory, we also evaluate the naïve table-wise sharding strategy.

4.5.2 End-to-End Results

Figure 4.11 presents the normalized speedup of NDRec over baseline methods. For the small model, Kaggle DAC, we achieve up to $2.6\times$ speedup over the UVM method. The results also show that the increasing batch size diminishes the advantage of NDRec. This trend originates from the small model size, which makes most embedding vectors reside in GPU memory, even with the simplest UVM strategy in the later iterations. The negligible gap between UVM and caching can also prove this explanation. For the 8K batch size, static (UVM) and look-ahead (cDLRM) caching introduce overhead over UVM and lead to performance degradation. NDRec is slower than NMP-CXL, and the gap deepens as the batch size increases. This is due to the small size of Kaggle DAC. The NMP solution benefits from its high internal bandwidth. As for Criteo Terabyte, we achieve up to $4.33\times$,

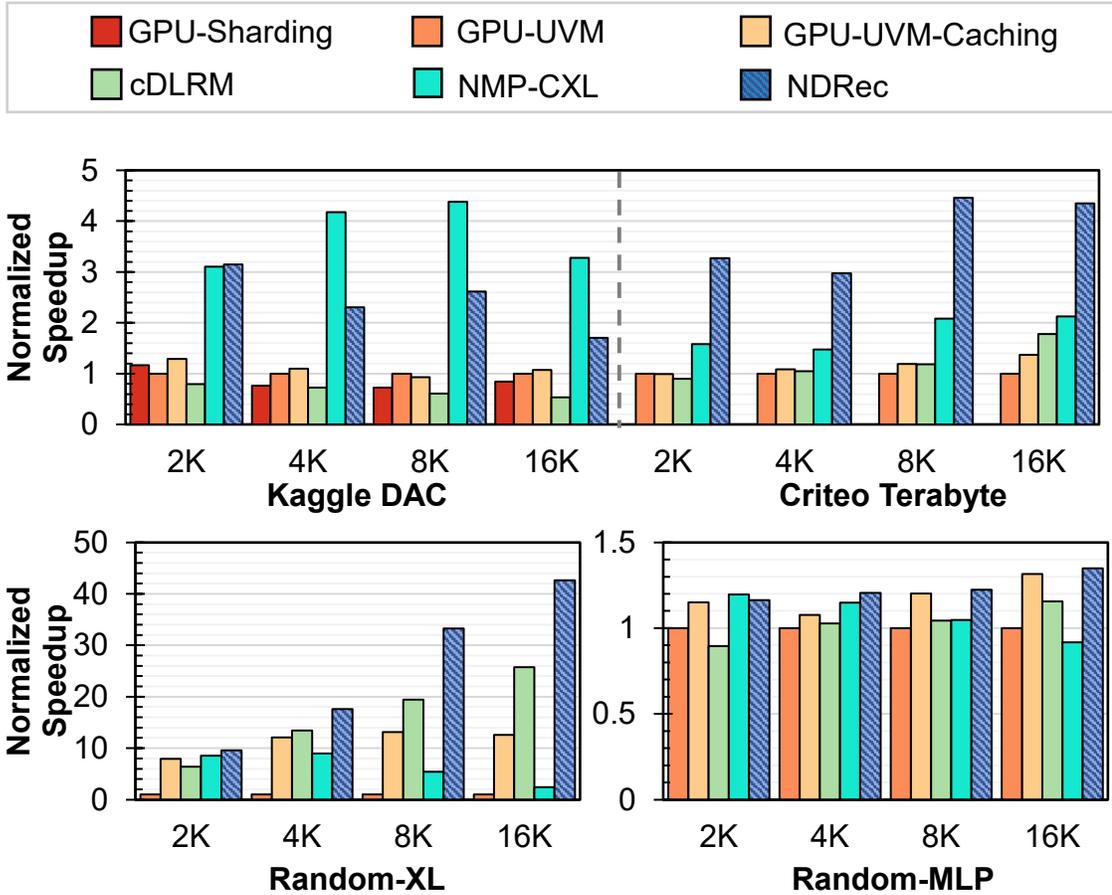


FIGURE 4.11: Normalized speedup of NDRec and baseline approaches over UVM-based GPU training on 4 GPUs.

3.97 \times , and 2.13 \times speedup over UVM, cDLRM, and NMP-CXL, respectively. A larger batch size helps NDRec explore the reuse of embedding vectors in the embedding cache and on-chip buffer. For the NMP solution, the impact of limited rank cache capacity and the interference of weight updates outweigh the benefit of the larger bandwidth. These two real-world benchmarks demonstrate the advantage of the NDRec system. For small models, even if the whole embedding table can fit into the GPU memory, the concurrent execution of embedding lookup and the rest of the model still achieve noticeable speedup. Furthermore, in a multi-GPU environment, NDRec mitigates the need for additional synchronization resulting from all-to-all communications, a common challenge associated with sharding. The overhead of synchronization can be particularly significant in the training of smaller

models. For large models, the embedding table exceeds the size of GPU memory. NDRec can eliminate the frequent data movement between main memory and GPU through the system interconnect with limited bandwidth.

The two synthetic benchmarks, Random-XL and Random-MLP, illustrate two extreme cases where the computation of embedding or MLP dominates the training time. When the embedding layer dominates the overall cost, all strategies except for the NMP-CXL show over $8\times$ speedup over the UVM baseline. NDRec further shows up to $3.38\times$ and $1.71\times$ speedup over UVM caching and cDLRM, respectively. The increased number of embedding tables and a larger pooling size contribute to the opportunities for exploring embedding vector reuse. We also noticed that the speedup of the NMP solution decreases as the batch size increases. This is due to the limited rank cache size in the NMP solution, causing a low hit rate. The Random-MLP, however, achieves up to $1.35\times$ speedup with different approaches. Since the MLP computation is the dominating factor in the training latency, the data movement cost can be hidden by the computation. These two synthetic benchmarks prove the effectiveness of NDRec in addressing the overhead brought by the embedding operation. Since the synthetic benchmarks only reflect the extreme cases, we will only use the Criteo Terabyte for the subsequent studies.

4.5.3 Embedding Cache

We evaluate the effectiveness of the embedding cache by examining the hit rate, cache planning cost, and the number of accesses to SSD. The result is shown in Figure 4.12. We separately show the hit rate in the main and hot-entry tables. For different batch sizes, the hit rate in the hot-entry table remains constant since the hit rate for the hot-entry table is determined by the statistical feature of the dataset. Meanwhile, the hit rate in the main table increases with the batch size. Overall, we achieved over 90% cache hit rate on all settings and even over 95% on the 16K batch size. Since the software-managed cache is fully associated, we can fully utilize the capacity of the embedding cache. The search cost increases with the batch size as there are more embedding vectors to search. However,

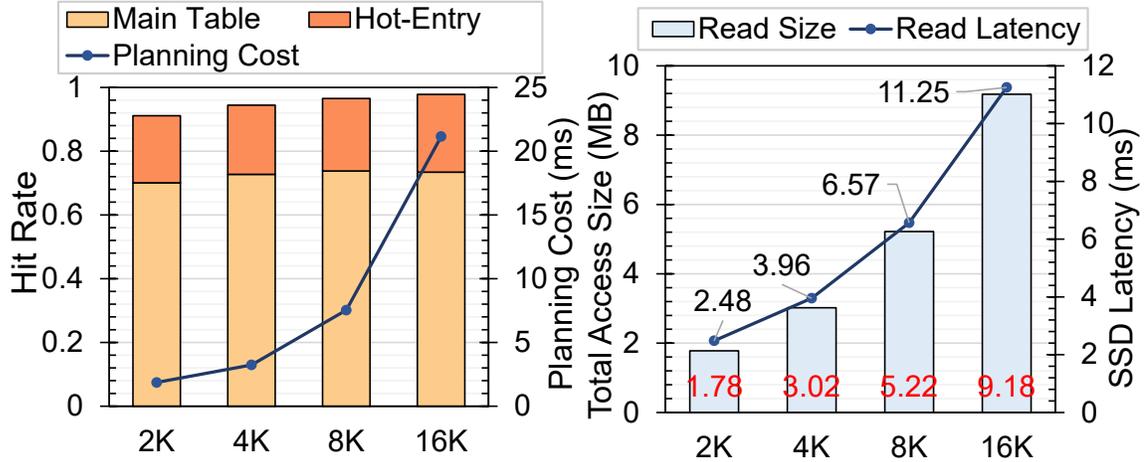


FIGURE 4.12: (left) Hit rate and cache planning overhead; (right) Average data movement size and latency to SSD per iteration.

the overall search cost can still be overlapped by the computation latency. Regarding the SSD access, the embedding cache significantly reduces the read and write from/to SSD. The accesses to SSD are a mix of approximately 50% read and 50% write for EV loading and eviction. Even with the 16K batch size, there is only 9.18MB traffic between SSD and embedding cache with less than 12ms transfer latency per device. Since only the evicted entries are updated to SSD, the embedding cache significantly reduces the write to SSD, avoiding wearing out the SSD.

We further investigate the design choices for the hot-entry table, specifically, the selection of hot-entry threshold and hot-entry table size. The result is shown in Figure 4.13. A small threshold could promote a less frequently accessed EV to the hot-entry table and prevent it from being evicted. A small hot-entry table size might lose the opportunity to capture the frequently accessed EVs. We also notice that, given the large capacity of the embedding cache and fully associated organization, the hit rate reaches a plateau when both parameters pass a certain value. Thus, we select 200 as the hot-entry threshold and 2048 as the hot-entry table size.

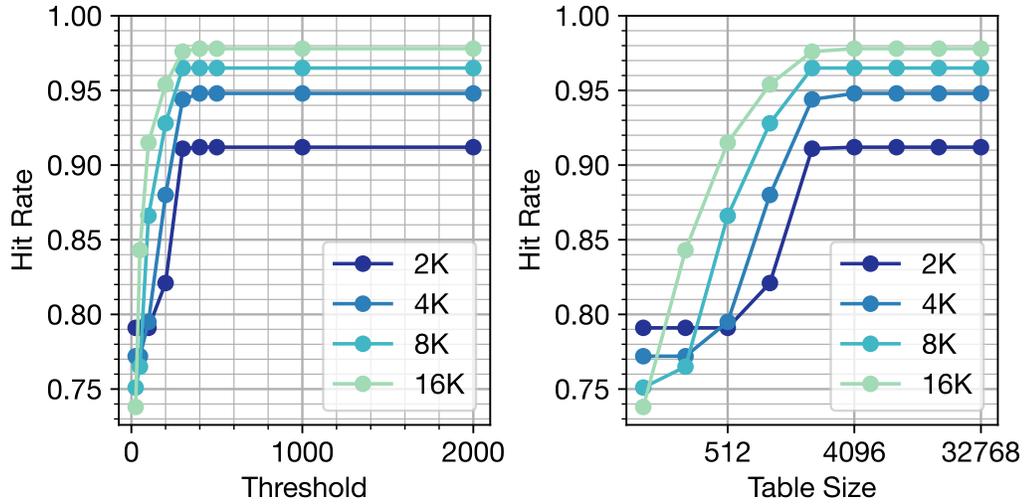


FIGURE 4.13: The DRAM cache hit rate with different hot-entry thresholds (*left*) and hot-entry table sizes (*right*).

4.6 Discussion

4.6.1 Bandwidth

We present the achieved aggregated bandwidth between the embedding cache and FPGA on SmartSSDs under different configurations in Figure 4.14. The theoretical maximum bandwidth of the memory on a single SmartSSD device is 19.2GB/s. The achieved band-

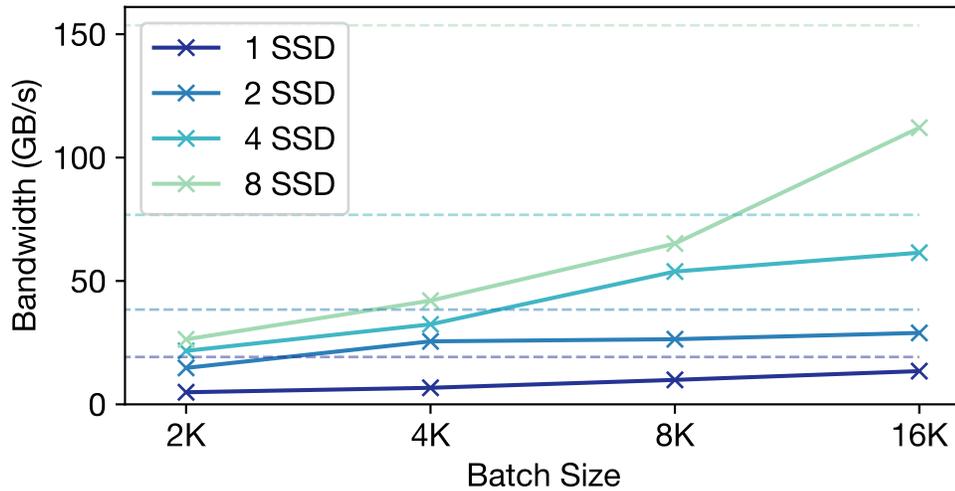


FIGURE 4.14: Achieved aggregated bandwidth with different configurations. The dashed lines illustrate the theoretical maximum bandwidth.

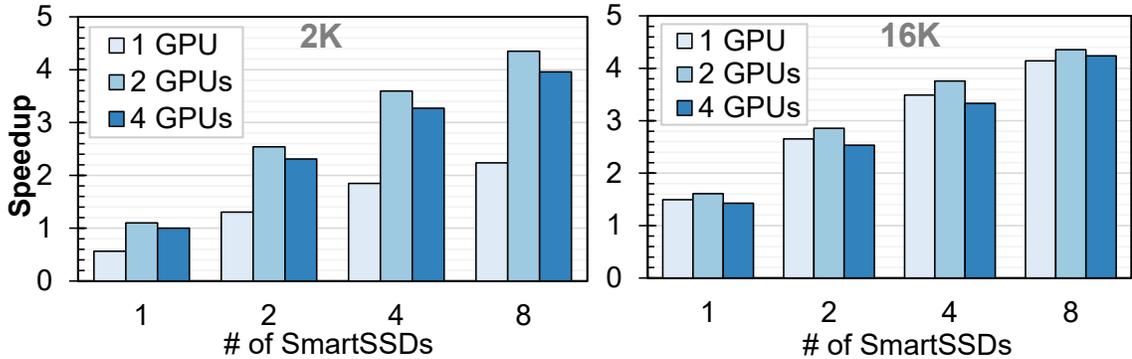


FIGURE 4.15: Average speedup over GPU-UVM with 2K and 16K batch size and various combinations of GPUs and SmartSSDs.

width is affected by multiple factors, including the efficiency of the memory controller, the size of EV loading required for each iteration, and the number of reused EVs in the on-chip buffer. A large batch size requires loading more EVs from the embedding cache, thus making it easier for the reader to schedule the memory request. The peak bandwidth approaches the efficiency of burst read as listed in [116]. For a small batch size, fewer EVs are required for each iteration. The access pattern is more random and reduces the efficiency of the memory controller. We notice that, for the 8K batch size, the 4 SSDs configuration achieves a similar bandwidth to that of the 8 SSDs configuration, indicating the bandwidth is not a limiting factor here. We can also tell that, for the configuration with less than 4 SmartSSDs, the achieved bandwidth with 16K batch size is limited by the device bandwidth. For further scaling up of the training batch size, the limited memory bandwidth could be the bottleneck for the performance. However, the configurations mentioned in [102] use the 16K batch size at most (2048 local batch size with 8 GPUs per node). Eight or more SmartSSDs should be able to serve one training node.

4.6.2 Scalability

We evaluate the scalability of the proposed NDRec system by scaling the number of GPUs and SmartSSDs. For GPU, we use weak scaling, i.e., the overall batch size remains constant so that we can maintain a similar amount of workload on SmartSSDs. Results are

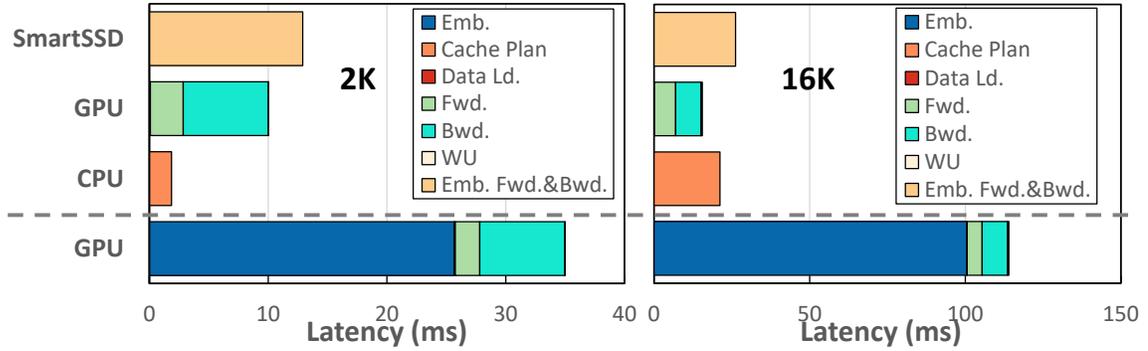


FIGURE 4.16: The execution time breakdown of NDRRec and GPU-UVM.

depicted in Figure 4.15. When we have more SmartSSDs, the speedup ratio scales accordingly, showing the extra bandwidth/computing power provided by additional SmartSSDs can be efficiently utilized. We also noticed that when the number of SmartSSDs is smaller than or equal to 4, the speedup ratio scales linearly, indicating the overall performance is limited by the bandwidth or computing power of the SmartSSDs. Combined with the bandwidth analysis shown in Figure 4.14, we can conclude that 4 SmartSSDs are sufficient to support the 8K batch size while 8 SmartSSDs are necessary for the 16K batch size. As for the GPUs, we can tell that for the 2K batch size, the GPUs form the bottleneck. Thus, increasing the number of GPUs from 1 to 2 shows nearly $2\times$ improvement. For the 16K batch size, an additional GPU brings marginal benefit. In both cases, further increasing the number of GPUs to 4 narrows the gap over the baseline, indicating the baseline benefits from the extra GPUs while NDRRec does not since the computation on GPU is no longer the bottleneck of the training.

4.6.3 Latency Breakdown

We present a latency breakdown of NDRRec and GPU-UVM in Fig. 4.16. The result illustrates that the embedding operation dominates the training latency due to frequent data movement between the GPU and main memory. NDRRec addresses this bottleneck by overlapping the embedding and other stages and avoiding expensive data movement.

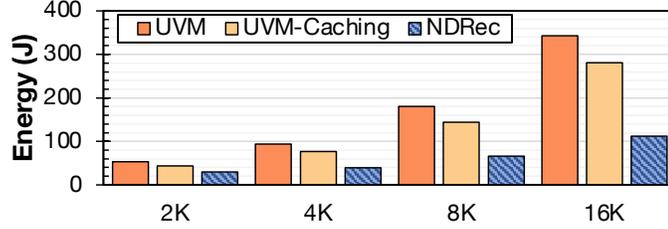


FIGURE 4.17: The average energy consumption per iteration.

4.6.4 Energy Consumption

We compare the energy consumption of NDRac per training iteration with the baselines. To estimate the energy consumption of the simulated part of NDRac, we use the measured worst-case dynamic power of the embedding kernel running on SmartSSD to obtain a pessimistic number for comparison. The result is illustrated in Figure 4.17. NDRac can utilize the customized FPGA kernel and reduce the burden of data movement and GPU. On average, NDRac reduces the energy consumption by 54.9% and 43.8% over the UVM and UVM-Caching systems, respectively.

4.6.5 SSD Endurance

SSD devices have limited endurance due to the wear out of the flash cells. The endurance issue could potentially increase the total cost of operation (TCO). The DRAM cache design is able to alleviate this issue when we use SSD as backing storage. Instead of writing every updated EV back to SSD, we stored them in the DRAM cache and only write back to SSD when the EV was evicted from the cache. A relatively high reuse ratio (as indicated by the hit rate) could reduce the number of writebacks. As shown in Figure 4.12, on average, we need to write 4.59MB (50% of the total 9.18MB SSD access) to SSD per iteration (with 4 SSDs, 16K batch size). Considering 650M training samples in the Terabyte dataset, we need to write 182GB to SSD per training epoch. For an optimistic estimation, one SSD device could support 38.5K training epochs. We envision this issue to be addressed with novel flash technologies (e.g., Z-SSD [117]).

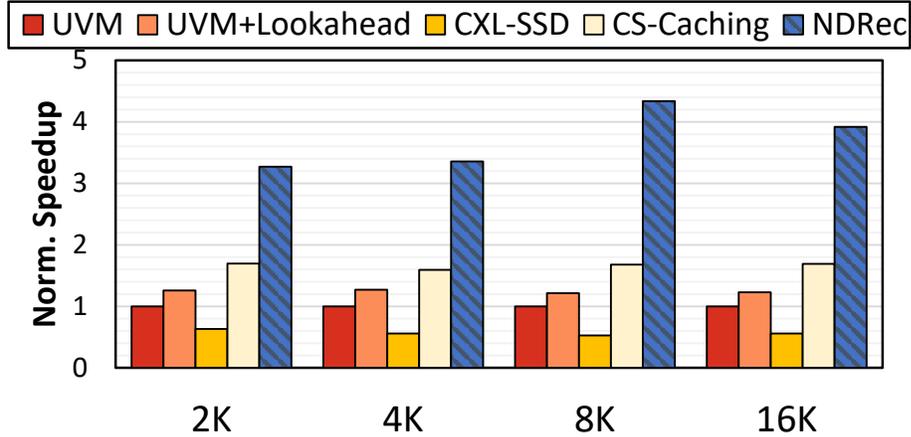


FIGURE 4.18: The comparison with alternative solutions. ‘CS-Caching’ stands for the use of a caching strategy like cDLRM directly with CXL-SSD.

4.7 Ablation Study

4.7.1 Alternative Architecture

We compare the proposed design with three possible alternative solutions, including applying lookahead embedding to the UVM baseline, directly accessing from CXL SSD, and using the caching strategy similar to cDLRM between GPU and CXL SSD. The result is depicted in Figure 4.18. Applying lookahead embedding to the UVM baseline brings around $1.2\times$ speedup across all batch size configurations. Lookahead embedding alleviates the bandwidth requirement, but data movement is still the major bottleneck. CXL-SSD allows cacheable access to the embedding table. However, the limited size of the GPU cache cannot effectively capture the locality of the access to the embedding table. Moreover, since the CXL interface does not alter the internal architecture of SSD, the access latency, especially the long tail latency, leads to over 50% slowdown compared with the baseline. A caching strategy brings over $1.5\times$ speedup since the access to SSD can be scheduled ahead and overlapped by the computation. NDRec shows advantages over all alternative solutions since we offload both computation and data movement to the SmartSSD.

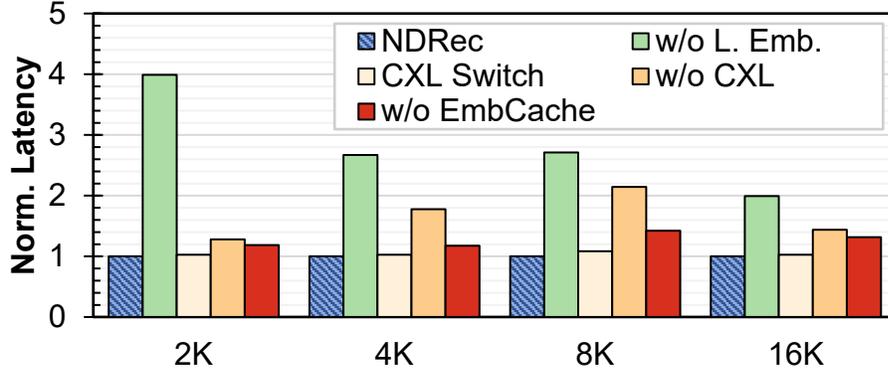


FIGURE 4.19: The analysis of the contribution of each component.

4.7.2 Component Contribution

We perform an ablation study to evaluate the contribution of individual components in NDRec. The result is depicted in Figure 4.19. We remove lookahead embedding, CXL, and embedding cache separately in our evaluated system. For the system without lookahead embedding, the training tasks are scheduled as shown in Figure 4.5 ‘Naïve’. We see up to $4\times$ slowdown since the fully sequential training timeline significantly increases the bandwidth requirement of the embedding operation. The gap narrows to $2\times$ when we increase the batch size to 16K, as the cache design alleviates the bandwidth issue. For the system without CXL, we use DMA to communicate between GPU and CPU and refer to [118] to verify the achieved bandwidth in the simulator. To efficiently use DMA, we only initiate the data transfer when the embedding forward process is finished, which forces an additional synchronization barrier. The lack of CXL can cause up to $2.5\times$ increment in latency due to the inefficient communication for the small amount of data. We also evaluate the case where two-level switching is required for SmartSSD to reach the host. Negligible overhead can be observed. For the system without embedding cache, we employ a simple static caching strategy where 3% of embedding parameters are cached in FPGA DRAM and randomly evict entries to load the necessary embedding vector for the new iteration. It increases the latency by 20% for 2K and 4K batch sizes, while up to 41% increment in latency can be observed for larger batch sizes. The simple static cache cannot effectively exploit the reuse

opportunities.

4.8 Conclusion

In this chapter, we present NDRec, a near-data processing system for large-scale recommendation model training. We offload the embedding operation to SmartSSDs to enable the training of large recommendation models and increase the utilization of GPUs. We then propose the lookahead embedding scheme to enable concurrent execution between GPU and CSDs. A software-managed DRAM cache and a customized FPGA kernel are designed to maximize the performance of the embedding operation. The evaluation result shows that NDRec could achieve up to $4.33\times$ and $3.97\times$ speedup over heterogeneous CPU-GPU platform and GPU caching, respectively. NDRec eliminates the capacity bottleneck and could enable the adoption of larger-scale recommendation models.

5. Conclusion

The extensive use of Deep Neural Networks (DNN) in various applications has led to a continuous effort to optimize their training, serving, and deployment. As the DNN model keeps evolving, it brings new challenges to algorithm researchers, computer architects, and system designers. This dissertation focuses on joint optimization for the DNN model with two examples that represent two ends of the spectrum: the Convolutional Neural Network (CNN) model for edge inference and the Recommendation Model for large-scale training in data centers.

The core of joint optimization at multiple levels in the system stack is to consider the constraints and opportunities imposed by other levels. Unlike optimization methods that focus on a single level, joint optimization methods use end-to-end metrics, such as deployment accuracy and runtime speedup, as the optimization targets. Methods that focus on a single level may find an optimal point in the target level but lead to suboptimal performance on another level. For example, unstructured pruning may not outperform structured pruning in terms of speedup, even though it has a higher sparsity ratio. In contrast, the joint optimization method provides an optimal trade-off or Pareto Frontier among the target metrics.

The study on the joint optimization methodology is not only focused on finding this trade-off from the existing system or selecting a point on this frontier, but also on how to push this frontier, or in other words, find a better trade-off. This process involves developing new algorithms, hardware, or system design. On the algorithm level, the hardware-aware or system-aware methods are desired. The compression algorithm, new DNN model architecture, or novel inference/training paradigm should actively take the implementation perspective into consideration. On the hardware level, it is important to carefully balance compatibility and specialty. The hardware design should select a proper set of characteristics from the algorithms to support to maximize efficiency and compatibility. On the system level, achieving joint optimization requires integrating algorithms and hardware designs seamlessly into a cohesive system architecture. System-level optimizations aim to

leverage the strengths of both algorithmic and hardware innovations while mitigating their respective limitations.

We have discussed a variety of trade-offs in this dissertation, including the trade-off among accuracy, latency, throughput, and energy consumption. However, there is an important trade-off is not touched, the generalizability and efficiency. The co-design methodology itself sacrifices generalizability for efficiency since the hardware/system design is tailored for a selected set of workloads. Given the cost of chip production, the co-design only makes sense when the workload is broadly adopted. DNN itself used to be a single workload for the designers of the neural processing unit (NPU). However, as the algorithm rapidly evolves, we see drastic differences in DNN models, for example, the operators in CNN differ a lot from that of transformers in terms of data flow, memory access pattern, and computation requirements. Thus, it is difficult to design a single accelerator architecture or system to efficiently support all types of DNN. The situation might even worsen with the continuous development of the DNN model design.

Thus, we envision that future research on joint optimization should take the generalizability into the metric. One possible path is to develop typical operators or computation patterns and optimize the hardware to support these patterns with the consideration of reconfigurability. The co-design for a specific model is performed by adjusting the scheduling/coordination strategy at the system level. The NDRec [3] we discussed in this dissertation is an initial attempt. We extract the typical operators and implement them on FPGA, then adjust the task coordination and system schedule to optimize the overall efficiency. However, more work needs to be done as the NDRec may not easily generalize for future recommendation models.

Bibliography

- [1] S. Li, E. Hanson, H. Li, and Y. Chen, “PENNI: Pruned Kernel Sharing for Efficient CNN Inference,” in *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event*, ser. Proceedings of Machine Learning Research, vol. 119, PMLR, 2020, pp. 5863–5873.
- [2] S. Li, E. Hanson, X. Qian, H. H. Li, and Y. Chen, “ESCALATE: Boosting the Efficiency of Sparse CNN Accelerator with Kernel Decomposition,” in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021, pp. 992–1004.
- [3] S. Li, Y. Wang, E. Hanson, A. Chang, Y. Ki, H. Li, and Y. Chen, “NDRec: A Near-Data Processing System for Training Large-Scale Recommendation Models,” *IEEE Transactions on Computers*, no. 01, pp. 1–14, Feb. 5555, ISSN: 1557-9956. DOI: 10.1109/TC.2024.3365939.
- [4] S. Han, H. Mao, and W. J. Dally, “Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding,” *International Conference on Learning Representations (ICLR)*, 2016.
- [5] W. Wen, C. Wu, Y. Wang, Y. Chen, and H. Li, “Learning Structured Sparsity in Deep Neural Networks,” in *Proceedings of the 30th International Conference on Neural Information Processing Systems*, ser. NIPS’16, Barcelona, Spain, 2016, pp. 2082–2090, ISBN: 9781510838819.
- [6] E. Hanson, S. Li, H. Li, and Y. Chen, “Cascading Structured Pruning: Enabling High Data Reuse for Sparse DNN Accelerators,” in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ser. ISCA ’22, New York, New York, 2022, pp. 522–535, ISBN: 9781450386104. DOI: 10.1145/3470496.3527419. [Online]. Available: <https://doi.org/10.1145/3470496.3527419>.
- [7] A. Ren, T. Zhang, S. Ye, J. Li, W. Xu, X. Qian, X. Lin, and Y. Wang, “ADMM-NN: An Algorithm-Hardware Co-Design Framework of DNNs Using Alternating Direction Methods of Multipliers,” in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS ’19, Providence, RI, USA, 2019, pp. 925–938, ISBN: 9781450362405. DOI: 10.1145/3297858.3304076.
- [8] M. Lin, Q. Chen, and S. Yan, “Network in Network,” *arXiv preprint arXiv:1312.4400*, 2013.
- [9] 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 arXiv:1602.07360*, 2016.

- [10] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, “Going deeper with convolutions,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 1–9.
- [11] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, “Rethinking the Inception Architecture for Computer Vision,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 2818–2826.
- [12] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, “MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications,” *arXiv preprint arXiv:1704.04861*, 2017.
- [13] 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 (CVPR)*, Jun. 2018.
- [14] 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.
- [15] 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.
- [16] M. Tan, B. Chen, R. Pang, V. Vasudevan, M. Sandler, A. Howard, and Q. V. Le, “MnasNet: Platform-Aware Neural Architecture Search for Mobile,” in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2019, pp. 2820–2828.
- [17] H. Cai, L. Zhu, and S. Han, “ProxylessNAS: Direct Neural Architecture Search on Target Task and Hardware,” in *International Conference on Learning Representations*, 2019. [Online]. Available: <https://openreview.net/forum?id=Hy1VB3AqYm>.
- [18] H. Liu, K. Simonyan, and Y. Yang, “DARTS: Differentiable Architecture Search,” in *International Conference on Learning Representations*, 2019. [Online]. Available: <https://openreview.net/forum?id=S1eYHoC5FX>.
- [19] M. Tan and Q. V. Le, “EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks,” in *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*, ser. Proceedings of Machine Learning Research, vol. 97, PMLR, 2019, pp. 6105–6114.
- [20] E. L. Denton, W. Zaremba, J. Bruna, Y. LeCun, and R. Fergus, “Exploiting Linear Structure Within Convolutional Networks for Efficient Evaluation,” in *Advances in neural information processing systems*, 2014, pp. 1269–1277.

- [21] 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.
- [22] Y. 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,” in *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, 2016.
- [23] V. Lebedev, Y. Ganin, M. Rakhuba, I. V. Oseledets, and V. S. Lempitsky, “Speeding-up Convolutional Neural Networks Using Fine-tuned CP-Decomposition,” in *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.
- [24] W. Wen, C. Xu, C. Wu, Y. Wang, Y. Chen, and H. Li, “Coordinating Filters for Faster Deep Neural Networks,” in *The IEEE International Conference on Computer Vision (ICCV)*, Oct. 2017.
- [25] X. Ding, G. Ding, Y. Guo, and J. Han, “Centripetal SGD for Pruning Very Deep Convolutional Networks with Complicated Structure,” in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2019, pp. 4943–4953.
- [26] Y. LeCun, J. S. Denker, and S. A. Solla, “Optimal Brain Damage,” in *Advances in neural information processing systems*, 1990, pp. 598–605.
- [27] B. Hassibi and D. G. Stork, “Second order derivatives for network pruning: Optimal Brain Surgeon,” in *Advances in neural information processing systems*, 1993, pp. 164–171.
- [28] Y. Guo, A. Yao, and Y. Chen, “Dynamic Network Surgery for Efficient DNNs,” in *Advances in neural information processing systems*, 2016, pp. 1379–1387.
- [29] J. Frankle and M. Carbin, “The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks,” in *International Conference on Learning Representations*, 2018.
- [30] H. Yang, W. Wen, and H. Li, “Deephoyer: Learning sparser neural network with differentiable scale-invariant sparsity measures,” in *International Conference on Learning Representations*, 2019.
- [31] 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.
- [32] J.-H. Luo, J. Wu, and W. Lin, “ThiNet: A Filter Level Pruning Method for Deep Neural Network Compression,” in *Proceedings of the IEEE international conference on computer vision*, 2017, pp. 5058–5066.

- [33] Y. He, G. Kang, X. Dong, Y. Fu, and Y. Yang, “Soft Filter Pruning for Accelerating Deep Convolutional Neural Networks,” in *Proceedings of the 27th International Joint Conference on Artificial Intelligence*, ser. IJCAI’18, Stockholm, Sweden, 2018, pp. 2234–2240, ISBN: 9780999241127.
- [34] D. Zhang, H. Wang, M. Figueiredo, and L. Balzano, “Learning to Share: Simultaneous Parameter Tying and Sparsification in Deep Learning,” in *International Conference on Learning Representations*, 2018. [Online]. Available: <https://openreview.net/forum?id=rypT3fb0b>.
- [35] Y. He, P. Liu, Z. Wang, Z. Hu, and Y. Yang, “Filter Pruning via Geometric Median for Deep Convolutional Neural Networks Acceleration,” in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2019, pp. 4340–4349.
- [36] X. Ding, G. Ding, Y. Guo, J. Han, and C. Yan, “Approximated Oracle Filter Pruning for Destructive CNN Width Optimization,” in *International Conference on Machine Learning*, 2019, pp. 1607–1616.
- [37] C. Louizos, M. Welling, and D. P. Kingma, “Learning Sparse Neural Networks through L_0 Regularization,” in *International Conference on Learning Representations*, 2018.
- [38] Y. He, J. Lin, Z. Liu, H. Wang, L.-J. Li, and S. Han, “AMC: AutoML for Model Compression and Acceleration on Mobile Devices,” in *Proceedings of the European Conference on Computer Vision (ECCV)*, 2018, pp. 784–800.
- [39] Z. Liu, H. Mu, X. Zhang, Z. Guo, X. Yang, K.-T. Cheng, and J. Sun, “MetaPruning: Meta Learning for Automatic Neural Network Channel Pruning,” in *Proceedings of the IEEE International Conference on Computer Vision*, 2019, pp. 3296–3305.
- [40] 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.
- [41] F. Chollet, “Xception: Deep Learning with Depthwise Separable Convolutions,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 1251–1258.
- [42] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, *et al.*, “DaDianNao: A Machine-Learning Supercomputer,” in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, IEEE, 2014, pp. 609–622.
- [43] A. Krizhevsky, *Learning Multiple Layers of Features from Tiny Images*, 2009.

- [44] 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.
- [45] K. Simonyan and A. Zisserman, “Very Deep Convolutional Networks for Large-Scale Image Recognition,” in *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.
- [46] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “Imagenet classification with deep convolutional neural networks,” in *Advances in neural information processing systems*, 2012, pp. 1097–1105.
- [47] PyTorch. “Torchvision Models.” (2019), [Online]. Available: <https://github.com/pytorch/vision/tree/master/torchvision/models> (visited on 01/15/2020).
- [48] H. Li, A. Kadav, I. Durdanovic, H. Samet, and H. P. Graf, “Pruning Filters for Efficient ConvNets,” *arXiv preprint arXiv:1608.08710*, 2016.
- [49] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, *et al.*, “PyTorch: An Imperative Style, High-Performance Deep Learning Library,” in *Advances in Neural Information Processing Systems*, 2019, pp. 8024–8035.
- [50] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko, “Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 2704–2713.
- [51] C. Zhu, S. Han, H. Mao, and W. J. Dally, “Trained Ternary Quantization,” in *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*, 2017.
- [52] D. P. Kingma and J. Ba, “Adam: A Method for Stochastic Optimization,” in *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.
- [53] PyTorch. “torchvision.models – PyTorch 1.7.0 documentation.” (2020), [Online]. Available: <https://pytorch.org/docs/1.7.0/torchvision/models.html> (visited on 11/17/2020).
- [54] A. Gondimalla, N. Chesnut, M. Thottethodi, and T. Vijaykumar, “SparTen: A Sparse Tensor Accelerator for Convolutional Neural Networks,” in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 151–165.

- [55] G. Morin, R. Razani, V. P. Nia, and E. Sari, “Smart Ternary Quantization,” *arXiv preprint arXiv:1909.12205*, 2019.
- [56] X. Ding, T. Hao, J. Tan, J. Liu, J. Han, Y. Guo, and G. Ding, “ResRep: Lossless CNN Pruning via Decoupling Remembering and Forgetting,” in *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, Oct. 2021, pp. 4510–4520.
- [57] A. Kusupati, V. Ramanujan, R. Somani, M. Wortsman, P. Jain, S. Kakade, and A. Farhadi, “Soft Threshold Weight Reparameterization for Learnable Sparsity,” in *International Conference on Machine Learning*, PMLR, 2020, pp. 5544–5555.
- [58] X. Ma, S. Lin, S. Ye, Z. He, L. Zhang, G. Yuan, S. H. Tan, Z. Li, D. Fan, X. Qian, X. Lin, K. Ma, and Y. Wang, “Non-Structured DNN Weight Pruning—Is It Beneficial in Any Platform?” *IEEE Transactions on Neural Networks and Learning Systems*, vol. 33, no. 9, pp. 4930–4944, 2022. DOI: 10.1109/TNNLS.2021.3063265.
- [59] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, “DianNao: A Small-Footprint High-Throughput Accelerator for Ubiquitous Machine-Learning,” in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS ’14, Salt Lake City, Utah, USA, 2014, pp. 269–284, ISBN: 9781450323055. DOI: 10.1145/2541940.2541967.
- [60] 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.
- [61] S. Zhang, Z. Du, L. Zhang, H. Lan, S. Liu, L. Li, Q. Guo, T. Chen, and Y. Chen, “Cambricon-X: An accelerator for sparse neural networks,” in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016, pp. 1–12. DOI: 10.1109/MICRO.2016.7783723.
- [62] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos, “Cnvlutin: Ineffectual-Neuron-Free Deep Neural Network Computing,” in *Proceedings of the 43rd International Symposium on Computer Architecture*, ser. ISCA ’16, Seoul, Republic of Korea, 2016, pp. 1–13, ISBN: 9781467389471. DOI: 10.1109/ISCA.2016.11.
- [63] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally, “SCNN: An Accelerator for Compressed-Sparse Convolutional Neural Networks,” in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ser. ISCA ’17, Toronto, ON, Canada, 2017, pp. 27–40, ISBN: 9781450348928. DOI: 10.1145/3079856.3080254.
- [64] H. Ji, L. Song, L. Jiang, H. Li, and Y. Chen, “ReCom: An efficient resistive accelerator for compressed deep neural networks,” in *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2018, pp. 237–240. DOI: 10.23919/DATE.2018.8342009.

- [65] F. Chen, L. Song, and Y. Chen, “ReGAN: A pipelined ReRAM-based accelerator for generative adversarial networks,” in *2018 23rd Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2018, pp. 178–183. DOI: 10.1109/ASPAC.2018.8297302.
- [66] J. Albericio, A. Delmás, P. Judd, S. Sharify, G. O’Leary, R. Genov, and A. Moshovos, “Bit-Pragmatic Deep Neural Network Computing,” in *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2017, pp. 382–394.
- [67] A. Delmas Lascorz, P. Judd, D. M. Stuart, Z. Poulos, M. Mahmoud, S. Sharify, M. Nikolic, K. Siu, and A. Moshovos, “Bit-Tactical: A Software/Hardware Approach to Exploiting Value and Bit Sparsity in Neural Networks,” ser. ASPLOS ’19, Providence, RI, USA, 2019, pp. 749–763, ISBN: 9781450362405. DOI: 10.1145/3297858.3304041.
- [68] S. Sharify, A. D. Lascorz, M. Mahmoud, M. Nikolic, K. Siu, D. M. Stuart, Z. Poulos, and A. Moshovos, “Laconic deep learning inference acceleration,” in *Proceedings of the 46th International Symposium on Computer Architecture*, ser. ISCA ’19, Phoenix, Arizona, 2019, pp. 304–317, ISBN: 9781450366694. DOI: 10.1145/3307650.3322255.
- [69] X. Zhou, Z. Du, Q. Guo, S. Liu, C. Liu, C. Wang, X. Zhou, L. Li, T. Chen, and Y. Chen, “Cambricon-S: Addressing Irregularity in Sparse Neural Networks through A Cooperative Software/Hardware Approach,” in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018, pp. 15–28. DOI: 10.1109/MICRO.2018.00011.
- [70] W. Niu, X. Ma, S. Lin, S. Wang, X. Qian, X. Lin, Y. Wang, and B. Ren, “PatDNN: Achieving Real-Time DNN Execution on Mobile Devices with Pattern-Based Weight Pruning,” in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS ’20, Lausanne, Switzerland, 2020, pp. 907–922, ISBN: 9781450371025. DOI: 10.1145/3373376.3378534.
- [71] K. Hegde, H. Asghari-Moghaddam, M. Pellauer, N. Crago, A. Jaleel, E. Solomonik, J. Emer, and C. W. Fletcher, “ExTensor: An Accelerator for Sparse Tensor Algebra,” in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO ’52, Columbus, OH, USA, 2019, pp. 319–333, ISBN: 9781450369381. DOI: 10.1145/3352460.3358275.
- [72] E. Qin, A. Samajdar, H. Kwon, V. Nadella, S. Srinivasan, D. Das, B. Kaul, and T. Krishna, “SIGMA: A Sparse and Irregular GEMM Accelerator with Flexible Interconnects for DNN Training,” in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2020, pp. 58–70. DOI: 10.1109/HPCA47549.2020.00015.
- [73] F. Muñoz-Martínez, R. Garg, M. Pellauer, J. L. Abellán, M. E. Acacio, and T. Krishna, “Flexagon: A Multi-Dataflow Sparse-Sparse Matrix Multiplication Accelerator for Efficient DNN Processing,” in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, 2023, pp. 252–265.

- [74] Y. N. Wu, P.-A. Tsai, S. Muralidharan, A. Parashar, V. Sze, and J. Emer, “HighLight: Efficient and Flexible DNN Acceleration with Hierarchical Structured Sparsity,” in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, 2023, pp. 1106–1120.
- [75] Y.-H. Chen, T.-J. Yang, J. Emer, and V. Sze, “Eyeriss v2: A Flexible Accelerator for Emerging Deep Neural Networks on Mobile Devices,” *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 9, no. 2, pp. 292–308, 2019, ISSN: 2156-3357. DOI: 10.1109/jetcas.2019.2910232.
- [76] Y. Hilewitz and R. B. Lee, “Fast Bit Gather, Bit Scatter and Bit Permutation Instructions for Commodity Microprocessors,” *Journal of Signal Processing Systems*, vol. 53, no. 1, pp. 145–169, 2008.
- [77] R. Balasubramonian, A. B. Kahng, N. Muralimanohar, A. Shafiee, and V. Srinivas, “CACTI 7: New tools for interconnect exploration in innovative off-chip memories,” *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 14, no. 2, pp. 1–25, 2017.
- [78] Y. Kim, W. Yang, and O. Mutlu, “Ramulator: A Fast and Extensible DRAM Simulator,” *IEEE Computer Architecture Letters*, vol. 15, no. 1, pp. 45–49, 2015.
- [79] K. Chandrasekar, C. Weis, Y. Li, S. Goossens, M. Jung, O. Naji, B. Akesson, N. Wehn, and K. Goossens, *DRAMPower: Open-source DRAM Power & Energy Estimation Tool*, <http://www.drampower.info>.
- [80] A. Parashar, P. Raina, Y. S. Shao, Y.-H. Chen, V. A. Ying, A. Mukkara, R. Venkatesan, B. Khailany, S. W. Keckler, and J. Emer, “Timeloop: A Systematic Approach to DNN Accelerator Evaluation,” in *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2019, pp. 304–315. DOI: 10.1109/ISPASS.2019.00042.
- [81] isakedo. “Github - isakedo/DNNsim.” (2020), [Online]. Available: <https://github.com/isakedo/DNNsim> (visited on 11/17/2020).
- [82] X. Yang, M. Gao, J. Pu, A. Nayak, Q. Liu, S. E. Bell, J. O. Setter, K. Cao, H. Ha, C. Kozyrakis, *et al.*, “DNN Dataflow Choice Is Overrated,” *arXiv preprint arXiv:1809.04070*, 2018.
- [83] M. Naumov, D. Mudigere, H.-J. M. Shi, J. Huang, N. Sundaraman, J. Park, X. Wang, U. Gupta, C.-J. Wu, A. G. Azzolini, *et al.*, “Deep Learning Recommendation Model for Personalization and Recommendation Systems,” *arXiv preprint arXiv:1906.00091*, 2019.
- [84] M. Wilkening, U. Gupta, S. Hsia, C. Trippel, C.-J. Wu, D. Brooks, and G.-Y. Wei, “RecSSD: Near Data Processing for Solid State Drive Based Recommendation Inference,” in

Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, 2021, pp. 717–729.

- [85] J. H. Lee, H. Zhang, V. Lagrange, P. Krishnamoorthy, X. Zhao, and Y. S. Ki, “SmartSSD: FPGA Accelerated Near-Storage Data Analytics on SSD,” *IEEE Computer architecture letters*, vol. 19, no. 2, pp. 110–113, 2020.
- [86] “PCIe Peer-to-Peer (P2P) - Xilinx XRT.” (2021), [Online]. Available: <https://xilinx.github.io/XRT/master/html/p2p.html> (visited on 11/11/2021).
- [87] S. Van Doren, “HOTI 2019: Compute Express Link,” in *2019 IEEE Symposium on High-Performance Interconnects (HOTI)*, IEEE, 2019, pp. 18–18.
- [88] *Compute Express Link (CXL) specification*, Revision 3.0, Version 1.0, Computer Express Link Consortium Inc., Aug. 2022.
- [89] M. Jung, “Hello bytes, bye blocks: PCIe storage meets compute express link for memory expansion (CXL-SSD),” in *Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems*, 2022, pp. 45–51.
- [90] S.-P. Yang, M. Kim, S. Nam, J. Park, J.-y. Choi, E. H. Nam, E. Lee, S. Lee, and B. S. Kim, “Overcoming the Memory Wall with CXL-Enabled SSDs,” in *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, 2023, pp. 601–617.
- [91] “CMM-H (CXL Memory Module, H: Hybrid).” (2023), [Online]. Available: <https://samsungsl.com/cmmh/>.
- [92] Y. Kwon, Y. Lee, and M. Rhu, “TensorDIMM: A Practical Near-Memory Processing Architecture for Embeddings and Tensor Operations in Deep Learning,” in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 740–753.
- [93] R. Hwang, T. Kim, Y. Kwon, and M. Rhu, “Centaur: A Chiplet-based, Hybrid Sparse-Dense Accelerator for Personalized Recommendations,” in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, IEEE, 2020, pp. 968–981.
- [94] L. Ke, U. Gupta, B. Y. Cho, D. Brooks, V. Chandra, U. Diril, A. Firoozshahian, K. Hazelwood, B. Jia, H.-H. S. Lee, *et al.*, “RecNMP: Accelerating Personalized Recommendation with Near-Memory Processing,” in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, IEEE, 2020, pp. 790–803.
- [95] L. Ke, X. Zhang, J. So, J.-G. Lee, S.-H. Kang, S. Lee, S. Han, Y. Cho, J. H. Kim, Y. Kwon, *et al.*, “Near-Memory Processing in Action: Accelerating Personalized Recommendation with AxDIMM,” *IEEE Micro*, 2021.

- [96] X. Sun, H. Wan, Q. Li, C.-L. Yang, T.-W. Kuo, and C. J. Xue, “RM-SSD: In-Storage Computing for Large-Scale Recommendation Inference,” in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, IEEE, 2022, pp. 1056–1070.
- [97] K. Balasubramanian, A. Alshabanah, J. D. Choe, and M. Annavaram, “cDLRM: Look Ahead Caching for Scalable Training of Recommendation Models,” in *Fifteenth ACM Conference on Recommender Systems*, 2021, pp. 263–272.
- [98] Y. Kwon and M. Rhu, “Training Personalized Recommendation Systems from (GPU) Scratch: Look Forward not Backwards,” in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, 2022, pp. 860–873.
- [99] S. Agarwal, C. Yan, Z. Zhang, and S. Venkataraman, “Bagpipe: Accelerating Deep Recommendation Model Training,” *SOSP ’23*, pp. 348–363, 2023. DOI: 10.1145/3600006.3613142.
- [100] W. Zhao, J. Zhang, D. Xie, Y. Qian, R. Jia, and P. Li, “AIBox: CTR prediction model training on a single node,” in *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*, 2019, pp. 319–328.
- [101] G. Sethi, B. Acun, N. Agarwal, C. Kozyrakis, C. Trippel, and C.-J. Wu, “RecShard: Statistical Feature-Based Memory Optimization for Industry-Scale Neural Recommendation,” in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2022, pp. 344–358.
- [102] D. Mudigere, Y. Hao, J. Huang, Z. Jia, A. Tulloch, S. Sridharan, X. Liu, M. Ozdal, J. Nie, J. Park, L. Luo, J. (Yang, L. Gao, D. Ivchenko, A. Basant, Y. Hu, J. Yang, E. K. Ardestani, X. Wang, R. Komuravelli, C.-H. Chu, S. Yilmaz, H. Li, J. Qian, Z. Feng, Y. Ma, J. Yang, E. Wen, H. Li, L. Yang, C. Sun, W. Zhao, D. Melts, K. Dhulipala, K. Kishore, T. Graf, A. Eisenman, K. K. Matam, A. Gangidi, G. J. Chen, M. Krishnan, A. Nayak, K. Nair, B. Muthiah, M. khorashadi, P. Bhattacharya, P. Lapukhov, M. Naumov, A. Mathews, L. Qiao, M. Smelyanskiy, B. Jia, and V. Rao, “Software-Hardware Co-Design for Fast and Scalable Training of Deep Learning Recommendation Models,” in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ser. ISCA ’22, New York, New York, 2022, pp. 993–1011, ISBN: 9781450386104. DOI: 10.1145/3470496.3533727.
- [103] T. Allen and R. Ge, “Characterizing Power and Performance of GPU Memory Access,” in *2016 4th International Workshop on Energy Efficient Supercomputing (E2SC)*, IEEE, 2016, pp. 46–53.
- [104] “Pytorch/torchrec: Pytorch domain library for recommendation systems.” (2021), [Online]. Available: <https://github.com/pytorch/torchrec>.

- [105] E. Oldridge, J. Perez, B. Frederickson, N. Koumchatzky, M. Lee, Z. Wang, L. Wu, F. Yu, R. Zamora, O. Yilmaz, *et al.*, “Merlin: A GPU Accelerated Recommendation Framework,” in *Proceedings of IRS*, 2020.
- [106] B. Acun, M. Murphy, X. Wang, J. Nie, C.-J. Wu, and K. Hazelwood, “Understanding Training Efficiency of Deep Learning Recommendation Models at Scale,” in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, IEEE, 2021, pp. 802–814.
- [107] “UltraScale Architecture Memory Resources.” (2021), [Online]. Available: https://www.xilinx.com/support/documentation/user_guides/ug573-ultrascale-memory-resources.pdf.
- [108] “SmartSSD Computational Storage Drive – Installation and User Guide.” (2021), [Online]. Available: <https://docs.xilinx.com/v/u/en-US/ug1382-smartssd-csd> (visited on 11/13/2021).
- [109] J. Lowe-Power, A. M. Ahmad, A. Akram, M. Alian, R. Amslinger, M. Andreozzi, A. Armejach, N. Asmussen, B. Beckmann, S. Bharadwaj, *et al.*, “The gem5 Simulator: Version 20.0+,” *arXiv preprint arXiv:2007.03152*, 2020.
- [110] N. Agarwal, T. Krishna, L.-S. Peh, and N. K. Jha, “Garnet: A Detailed on-Chip Network Model inside a Full-System Simulator,” in *2009 IEEE international symposium on performance analysis of systems and software*, IEEE, 2009, pp. 33–42.
- [111] H. Li, D. S. Berger, L. Hsu, D. Ernst, P. Zardoshti, S. Novakovic, M. Shah, S. Rajadnya, S. Lee, I. Agarwal, M. D. Hill, M. Fontoura, and R. Bianchini, “Pond: CXL-Based Memory Pooling Systems for Cloud Platforms,” *ASPLOS 2023*, pp. 574–587, 2023. DOI: 10.1145/3575693.3578835.
- [112] S. Li, Z. Yang, D. Reddy, A. Srivastava, and B. Jacob, “DRAMsim3: a Cycle-accurate, Thermal-Capable DRAM Simulator,” *IEEE Computer Architecture Letters*, vol. 19, no. 2, pp. 106–109, 2020.
- [113] “Display Advertising Challenge.” (2014), [Online]. Available: <https://www.kaggle.com/c/criteo-display-ad-challenge> (visited on 10/03/2021).
- [114] “Download Criteo 1TB Click Logs dataset.” (2014), [Online]. Available: <https://ailab.criteo.com/download-criteo-1tb-click-logs-dataset/> (visited on 10/03/2021).
- [115] “Github - lkp411/cDLRM: Enabling pure data parallel training of DLRM via caching and prefetching.” (2021), [Online]. Available: <https://github.com/lkp411/cDLRM> (visited on 10/10/2022).

- [116] “PG150 - UltraScale Architecture FPGAs Memory IP Product Guide.” (2021), [Online]. Available: https://www.xilinx.com/content/dam/xilinx/support/documentation/ip_documentation/ultrascale_memory_ip/v1_4/pg150-ultrascale-memory-ip.pdf.
- [117] Samsung. “Samsung SZ1735a Z-SSD.” (2021), [Online]. Available: https://samsungsemiconductor-us.com/app/uploads/2021/03/Updated_Samsung-SZ1735a-U.2-Product-Brief_final.pdf (visited on 11/12/2021).
- [118] R. Bittner and E. Ruf, “Direct GPU/FPGA Communication via PCI Express,” in *2012 41st International Conference on Parallel Processing Workshops*, 2012, pp. 135–139. DOI: 10.1109/ICPPW.2012.20.
- [119] E. Hanson, S. Li, G. Zhou, F. Cheng, Y. Wang, R. Bose, H. Li, and Y. Chen, “Si-Kintsugi: Towards Recovering Golden-Like Performance of Defective Many-Core Spatial Architectures for AI,” in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, 2023, pp. 972–985.
- [120] B. Kim, S. Li, and H. Li, “INCA: Input-stationary Dataflow at Outside-the-box Thinking about Deep Learning Accelerators,” in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, IEEE, 2023, pp. 29–41.
- [121] Q. Zheng, S. Li, Y. Wang, Z. Li, Y. Chen, and H. H. Li, “Accelerating Sparse Attention with a Reconfigurable Non-volatile Processing-In-Memory Architecture,” in *2023 60th ACM/IEEE Design Automation Conference (DAC)*, IEEE, 2023, pp. 1–6.
- [122] Z. Xie, S. Li, M. Ma, C.-C. Chang, J. Pan, Y. Chen, and J. Hu, “DEEP: Developing extremely efficient runtime on-chip power meters,” in *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design*, 2022, pp. 1–9.
- [123] Q. Zhang, S. Li, G. Zhou, J. Pan, C.-C. Chang, Y. Chen, and Z. Xie, “PANDA: Architecture-level power evaluation by unifying analytical and machine learning solutions,” in *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, IEEE, 2023, pp. 01–09.
- [124] E. Hanson, S. Li, X. Qian, H. H. Li, and Y. Chen, “DyNNamic: Dynamically Reshaping, High Data-Reuse Accelerator for Compact DNNs,” *IEEE Transactions on Computers*, vol. 72, no. 3, pp. 880–892, 2022.
- [125] Z. Du, S. Li, Y. Wu, X. Jiang, J. Sun, Q. Zheng, Y. Wu, A. Li, H. Li, Y. Chen, *et al.*, “SiDA: Sparsity-Inspired Data-Aware Serving for Efficient and Scalable Large Mixture-of-Experts Models,” in *Proceedings of Machine Learning and Systems*, vol. 6, 2024.

Biography

Shiyu Li is a Ph.D. candidate in Electrical and Computer Engineering at Duke University under the supervision of Prof. Yiran Chen. He received the B.Eng. degree in Automation from Tsinghua University, Beijing, China in 2019. His research interests include computer architecture, algorithm-hardware co-design of deep learning systems, and near-data processing.

Shiyu's research work, including three first-author papers, appeared in top-tier conferences and journals, including ICML [1], MICRO [2], [119], ISCA [6], HPCA [120], DAC [121], ICCAD [122], [123], IEEE TC [3], [124], and MLSys [125]. He has received the Duke Graduate School Conference Travel Award, the Duke Electrical and Computer Engineering Conference Travel Fellowship, and travel awards for SRC @ ICCAD and ACM Ph.D. Forum @ DAC. Shiyu has served as a reviewer for multiple prestigious journals, including ACM TODAES, ACM TECS, IEEE TCAD, IEEE TVLSI, IEEE TCAS-I, IEEE D&T, IEEE TNNLS, IEEE TSP, IEEE TAI, IEEE TCE. He has served as a reviewer for top-tier conferences, including ECCV, ICCV, CVPR, NeurIPS, ICML, and ASP-DAC.