Efficient Deep Learning Accelerator Architectures by Model Compression and Data Orchestration

by

Jie-Fang Zhang

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Electrical and Computer Engineering)
in The University of Michigan
2022

Doctoral Committee:

Professor Zhengya Zhang, Chair
Associate Professor Ronald G. Dreslinski
Assistant Professor Hun-Seok Kim
Professor Dennis Sylvester

Jie-Fang Zhang

jfzhang@umich.edu

ORCID iD 0000-0002-6609-4383

© Jie-Fang Zhang 2022

All Rights Reserved

To my family and Yu-Hsien for always being there for me.

And to all the moments of frustration in life.

ACKNOWLEDGEMENTS

I would like to thank my advisor, Professor Zhengya Zhang, who has given me guidance and advice on both research and life. Thank you for being patient and believing in me whenever I struggle to find a more interesting research topic or try to make a research direction more interesting. I would also like to thank Professor Dennis Sylvester, Professor Hun-Seok Kim, and Professor Ronald Dreslinski for being on my dissertation committee and providing valuable feedback.

I have had the pleasure working with many talented people in the VLSISP group. Thanks to Shuanghong, Shiming, Thomas, Wei, Chester, Sung-Gun, Alex, Teyuh, Yaoyu, Reid, Jacob, Junkang, Cheng-Hsun, Jack, and Justin, for the valuable help and discussions about research and life struggles.

Thanks to all the research collaborators. Especially, thanks to Chester Liu, Thomas Chen, Wei Tang, Tim Wesley, and Cheng Fu for their help on the SNAP chip tapeout. And thanks to Ching-En (Alex) Lee, Sophia Shao, Steve Keckler for their helpful advice on the Stitch-X/SNAP project. Thanks to Yi-Chung Wu, Reid Pinkham, Chester Liu, Shang-En Huang, and Wei Tang for their help and discussion on the Point-X project. Thanks to Shang-En Huang, Miao Yin, and Salar Latifi for the insightful advice about the TetriX project.

Thanks to my family, Yu-Hsien, and my friends for always supporting and encouraging me throughout the journey.

TABLE OF CONTENTS

DEDICATION ii

ACKNOWLEDGEMENTS iii

LIST OF FIGURES vii

LIST OF TABLES x

LIST OF ABBREVIATIONS xi

ABSTRACT xiv

CHAPTER

I. Introduction 1

1.1 DNN Computation 2
1.2 Network Model Optimization 3
1.2.1 Model Compression 4
1.2.2 Novel Operation Types 6
1.2.3 Computation Challenges 7
1.3 Dissertation Outline 8

II. SNAP: Accelerator Architecture for Unstructured Sparse Neural Networks 10

2.1 Background 14
2.1.1 Channel-Last Dataflow for Sparse DNN Processing . 15
2.1.2 Other Related Work 17
2.2 Channel-First Processing Dataflow 17
2.2.1 Compression Format 17
2.2.2 Channel-First Dataflow 19
2.3 Channel Index Matching 21
2.3.1 Associative Index Matching 22
2.3.2 Sequence Decoder 22

2.3.3 Design Tradeoff Exploration 24
2.4 Two-Level Partial Sum Reduction 25
2.4.1 PE-level Channel-Dimension Reduction 25
2.4.2 Core-Level Pixel-Dimension Reduction 27
2.4.3 Support for Pointwise CONV and FC 28
2.5 Implementation and Evaluation Results 29
2.5.1 SNAP Architecture Overview 29
2.5.2 Performance Analysis 31
2.5.3 Comparison Against State-of-the-art Works 34
2.6 Summary 38

III. Point-X: Spatial-Locality-Aware Accelerator Architecture for Graph-Based Point-Cloud Neural Networks 39

3.1 Background 44
3.1.1 Edge Convolution Computation 44
3.1.2 Computation Models and Bottlenecks 45
3.2 Spatial-Locality-Aware Clustering 48
3.2.1 Graph Traversal for Spatial Locality 48
3.2.2 Speculative Breadth-First Search (SBFS) Traversal . 52
3.2.3 SLA Clustering Module Implementation 53
3.3 Locality-Aware NoC 55
3.3.1 Chain NoC Architecture 55
3.3.2 Routing Algorithm 56
3.4 CTile Architecture 58
3.5 Point-X System Architecture 61
3.5.1 Multi-Mode Dataflow 62
3.5.2 Workload Partitioning 64
3.6 Benchmarking and Evaluation 65
3.6.1 Evaluation Methodology 66
3.6.2 Area, Performance, Efficiency Analysis 67
3.6.3 Workload Scalability Analysis 68
3.6.4 Performance Comparison 69
3.7 Related Work 72
3.8 Summary 73

IV. TetriX: Efficient Accelerator Architecture for Flexible Tensorized Neural Network Processing 75

4.1 Background 78
4.1.1 Tensor Decomposition Methods 79
4.1.2 TNN Inference with Tensor Contraction 83
4.1.3 Computation Challenges 84
4.2 Optimal Contraction Sequence Search 85
4.2.1 Tensor Network Representation 86

4.2.2 Breadth-First Contraction Search 87
4.2.3 Contraction Sequence Analysis 88
4.3 Hybrid Contraction Sequence Mapping 90
4.3.1 Limitation of Baseline Mapping 90
4.3.2 Hybrid Inner-Outer Product Mapping 93
4.4 TetriX System Architecture 95
4.4.1 Configurable Stationary Dataflow 97
4.4.2 Index Translation 99
4.4.3 Output Gathering 101
4.5 Benchmarking and Evaluation 103
4.5.1 Evaluation Methodology 103
4.5.2 Performance, and Mapping Efficacy Analysis 103
4.5.3 Performance Comparison 106
4.6 Summary 108

V. Conclusion and Outlook 109

BIBLIOGRAPHY 112

LIST OF FIGURES

Figure

1.1 Top-1 accuracy, size, and complexity of modern DNN models. Adapted from [1]. 1
1.2 Core computations of DNNs: (a) vector-matrix multiplication in MLP and RNN, and (b) 2D convolution in CNN. 2
1.3 Evolution of model size in the fields of (a) CV and (b) NLP. Adapted from [2]. 4
1.4 Common model compression techniques: (a) data quantization, (b) network sparsification, and (c) tensor decomposition. 5
1.5 Examples of novel operations: (a) GraphSAGE in GNNs and (b) EdgeCONV in point-cloud networks. Adapted from [3, 4]. 6
2.1 Average density of IA, W data and average effectual work of common DNNs after network pruning. 11
2.2 Convolution computation between unstructured sparse IA and W in a sparse DNN. The colored cells indicate nonzero entries, and the white cells indicate zero entries. 12
2.3 Processing pipeline of a sparse DNN processor. 12
2.4 Illustration of channel-last dataflow for sparse DNN processing. 15
2.5 Channel-first compression in SNAP. 18
2.6 SNAP's channel-first dataflow with channel index matching and psum reduction along the channel dimension. 20
2.7 Comparison between channel-last dataflow and SNAP (channel-first dataflow) for dense, medium and sparse workloads. 21
2.8 (a) Microarchitecture of associative index matching (AIM), and (b) microarchitecture of sequence decoder and step-by-step example of W-IA data pair dispatch in a PE. 23
2.9 Multiplier utilization of AIM designs at different data density levels. 24
2.10 PE microarchitecture and psum reduction along the channel dimension. 25
2.11 Psum reduction configuration across an array of PEs (a) 3×3 CONV in diagonal mode, (b) pointwise CONV in row mode, and (c) FC in row mode. 27
2.12 SNAP system architecture. 30
2.13 Microphoto of the 16nm SNAP test chip. 31

2.14 (a) Measured power consumption at different operating frequencies and the optimal supply voltages, and (b) measured effectual energy efficiency for synthetic sparse workloads at different data density levels. 32
2.15 Processing speedup by SNAP (a channel-first dataflow) and a channel-last dataflow over a dense accelerator baseline running the residual blocks of a sparse ResNet-50 model. 33
3.1 Illustration of (a) point cloud recognition pipeline, (b) EdgeCONV layer divided into KNN graph construction and GraphCONV on vertex point i, and (c) DGCNN's network architecture for 3D object classification [4]. 40
3.2 EdgeCONV computation models: (a) query-based model, and (b) exchange-based model. 46
3.3 (a) KNN graph of input point cloud and its adjacency matrix representation; (b) the KNN graph is traversed and the points clustered following traversal order; the clustered KNN graph and its adjacency matrix are shown. 49
3.4 Clustering performance on KNN graphs: (a) graph edge ratio, and (b) graph edge length, using different graph traversal methods on 1k to 10k point clouds. 51
3.5 Illustration of the SBFS algorithm with 2 traversal lanes. 52
3.6 Architecture and dataflow of SLA clustering module. 54
3.7 Speedup of the SLA clustering module using SBFS with different traversal lane numbers over a BFS implementation baseline. 54
3.8 Architecture of (a) a chain NoC, and (b) a router for chain NoC. . . 56
3.9 Microarchitecture of (a) a compute tile (CTile), (b) a compute engine, and (c) a sort engine. 58
3.10 Microarchitecture of a group engine. 60
3.11 Point-X system architecture. 61
3.12 Multi-mode dataflow for (a) KNN graph construction, (b) shared MLP and FC, and (c) GraphCONV operations. 63
3.13 Workload partition schemes: (a) sequential partition and (b) diagonal partition. 64
3.14 (a) Normalized latency, (b) latency breakdown, and (c) energy breakdown of Point-X for GraphCONV workloads in DGCNN; (d) normalized latency comparison of Point-X to query-based (Q-Base) and the exchange-based (E-Base) baselines for GraphCONV workloads of each point size. 69
3.15 Comparison of (a) throughput and (b) energy efficiency of Point-X to the CPU and GPU baselines. 71
4.1 Illustration of model weight and tensor decomposition for TNNs. . . 76
4.2 Tensor network graph representation of TNN layer with (a) TT, (b) HT, (c) TR, and (d) BT tensor decomposition methods, where the input tensor $\in \mathbb{R}^{n_1 \times n_2 \times n_3 \times n_4}$ and the output tensor $\in \mathbb{R}^{m_1 \times m_2 \times m_3 \times m_4}$. 86
4.3 Illustration of breadth-first approach for optimal contraction sequence search. 88

4.4 Illustration of contraction sequences for an HT-TNN layer inference example: optimal sequence (Optimal) and fixed contraction patterns used in previous works (Pattern-1, Pattern-2). 89
4.5 (a) Contraction sequence space in terms of total MAC operations and required memory size; (b) comparison of contraction sequences in total MAC operations, required memory size, and number of memory accesses. 89
4.6 Illustration of (a) inner product and (b) outer product operation and memory layout. 91
4.7 Example of the mapping options for an optimal contraction sequence: (a) inner-only mapping, (b) outer-only mapping, and (c) hybrid inner-outer mapping, where the dimensions for contraction are indicated in red. 92
4.8 TetriX system architecture. 96
4.9 Illustration of WS and OS dataflows in TetriX architecture. 97
4.10 Illustration of integrated PE microarchitecture for WS/OS dataflows. 98
4.11 Illustration of (a) arbitrary permute and reshape operations on tensor data in memory, and (b) proposed index translation mechanism. . . 99
4.12 Microarchitecture of index translation for (a) WS dataflow and (b) OS dataflow. 100
4.13 Illustration of the hierarchical output gathering mechanism using (a) two gathering stages followed by (b) a switching stage. 102
4.14 Performance comparison of hybrid mapping (TetriX), inner-only mapping (TetriX-Inner), and outer-only mapping (TetriX-Outer) for (a) TT and (b) TR workloads. 104
4.15 Performance comparison of hybrid mapping (TetriX), inner-only mapping (TetriX-Inner), and outer-only mapping (TetriX-Outer) for (a) HT and (b) BT workloads. 105
4.16 Comparison of the TetriX to TIE [5] in (a) normalized latency, and (b) total MAC count, maximum memory size, and number of memory accesses. 106
4.17 End-to-end throughput comparison for TT and HT workloads. . . . 107

LIST OF TABLES

Table

2.1 Selection of Reduction Pattern 26
2.2 Comparison With Prior Works 35
3.1 GraphCONV Computation Comparison with F kernels, N points, and K neighbors per point 47
3.2 Storage and Area Breakdown of Point-X 67
3.3 Layer Evaluation of Point-X on 1k-DGCNN [4] 67
3.4 EdgeCONV Comparison to Existing Works 70
4.1 Comparison of Tensor Decomposition Methods 82

LIST OF ABBREVIATIONS

AI Artificial Intelligence

AIM Associative Index Matching

ASIC Application-Specific Integrated Circuit

BDFS Bounded Depth-First Search

BFS Breadth-First Search

BT Block Term

CAM Content-Addressable Memory

CNN Convolution Neural Network

CONV Convolution

CP Canonical Polyadic

CPU Central Processing Unit

CSC Compressed Sparse Column

CSR Compressed Sparse Row

CTA Compute Tile Array

CTile Compute Tile

CV Computer Vision

DFS Depth-First Search

DNN Deep Neural Network

DP Dot-Product

DRAM Dynamic Random-Access Memory

EdgeCONV Edge Convolution

FC Fully-Connected

F-Map Foreign Map

FPGA Field-Programmable Gate Array

fps Frames per Second

Fpsum Feature Partial Sum

GCN Graph Convolutional Network

GNN Graph Neural Network

GPU Graphics Processing Unit

GraphCONV Graph Convolution

GRU Gated Recurrent Unit

HT Hierarchical Tucker

IA Input Activation

Inf. Inference

KNN K-Nearest Neighbor

L-Map Local Map

LSTM Long Short Term Memory

MAC Multiply-and-Add

ML Machine Learning

MLP Multi-Layer Perceptron

MMM Matrix-Matrix Multiplication

MP Max-Pool

NLP Natural Language Processing

NN Neural Network

NoC Network-on-Chip

OA Output Activation

OS Output Stationary

OP Operation

PE Processing Element

PP Post-Processing

Psum Partial Sum

ReLU Rectifier Linear Unit

RNN Recurrent Neural Network

ROI Region of Interest

SBFS Speculative Breadth-First Search

SE Sorting Element

SIMD Single Instruction Multiple Data

SLA Spatial-Locality-Aware

SNAP Sparse Neural Acceleration Processor

SRAM Static Random-Access Memory

SVD Singular-Value Decomposition

TNN Tensorized Neural Network

TR Tensor Ring

TT Tensor Train

V-ALU Vector Arithmetic Logic Unit

VMM Vector-Matrix Multiplication

W Weight

WS Weight Stationary

ABSTRACT

Deep neural networks (DNNs) have become the primary methods to solve machine learning and artificial intelligence problems in the fields of computer vision, natural language processing, and robotics. The advancements in DNN model development are to a large degree attributed to the increase of model size, complexity, and versatility. The continuous growth of model size, complexity, and versatility causes intense memory storage and compute requirements, and complicates the hardware design, especially for the more resource-constrained mobile and smart sensor platforms.

To resolve the resource bottlenecks, model compression techniques, i.e., data quantization, network sparsification, and tensor decomposition, have been used to reduce the model size while preserving the accuracy of the original model. However, they introduce several computation challenges including 1) irregular computation in an unstructured sparse neural network (NN) from network sparsification, and 2) complex and arbitrary tensor orchestration for tensor contraction in a tensorized NN.

Meanwhile, DNN's capability has been transferred to new domains and applications to handle drastically different modalities and non-Euclidean data, e.g., point clouds and graphs. New computation challenges continue to emerge, for example, irregular memory access for graph-structured data in a graph-based point-cloud NN.

These challenges lead to a low processing efficiency for existing hardware architectures and motivate the exploration of specialized hardware mechanisms and accelerator architectures. This dissertation consists of three works that explore the design of efficient accelerator architectures to overcome the computation challenges by exploiting model compression characteristics and data orchestration techniques.

The first work presents the sparse neural acceleration processor (SNAP) to process sparse NNs resulted from unstructured network pruning. SNAP uses parallel associative search to discover valid weight and input activation pairs for parallel computation. A two-level partial sum (psum) reduction dataflow is used to eliminate access contention at the output buffer and cut the psum writeback traffic. The SNAP chip is implemented and achieves a peak effectual efficiency of 21.55 TOPS/W for sparse workloads and 3.61 TOPS/W for pruned ResNet-50.

The second work presents Point-X, a spatial-locality-aware architecture that exploits the spatial locality in point clouds for efficient graph-based point-cloud NN processing. A clustering method extracts fine-grained and coarse-grained spatial locality from the input point cloud to maximize intra-tile computational parallelism and minimize inter-tile data movement. A chain network-on-chip (NoC) further reduces the data traffic and achieves up to 3.2× speedup over a traditional mesh NoC. The Point-X prototype achieves a throughput of 1307.1 inference/s and an energy efficiency of 604.5 inference/J on the DGCNN workload.

The third work presents TetriX, an architecture-mapping co-design for efficient and flexible tensorized NN inference. An optimal contraction sequence with minimized computation and memory size requirements is identified for inference. A hybrid mapping scheme is used to eliminate complex orchestration operations by alternating between inner and outer product operations. TetriX uses index translation and output gathering to support flexible orchestration operations efficiently. TetriX is the first work to support all existing tensor decomposition methods for tensorized NNs and demonstrates up to 3.9× performance improvement compared to the prior work for tensor-train workloads.

Overall, these three works explore the computation of different network optimization techniques. They exploit the full potentials of model compression and novel operations, and convert them into hardware performance and efficiency. The archi-

tectures can also be used to further enhance and support the development of more effective network models.

CHAPTER I

Introduction

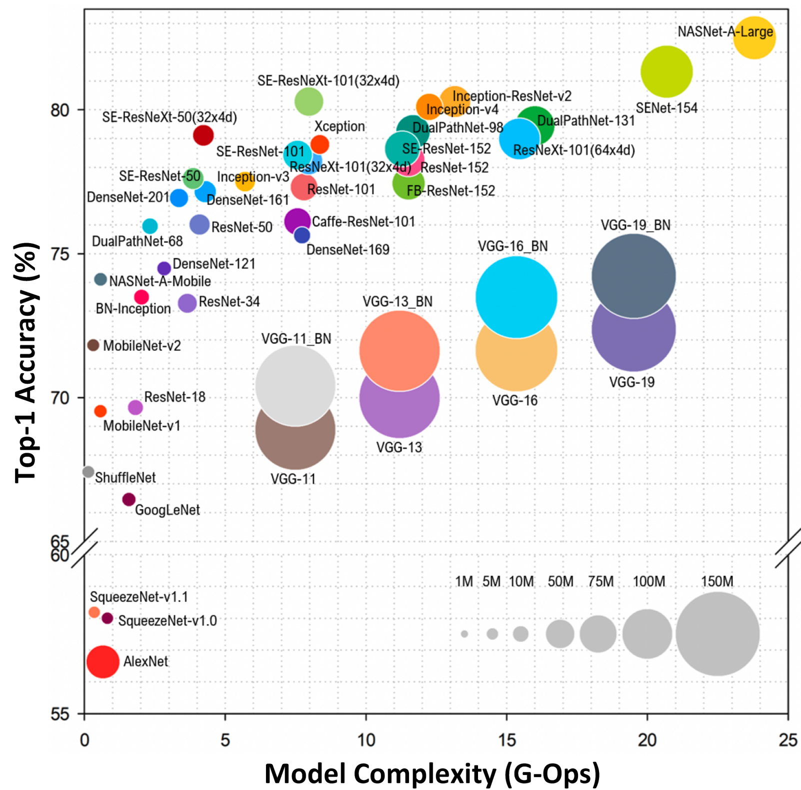


Figure 1.1: Top-1 accuracy, size, and complexity of modern DNN models. Adapted from [1].

Deep learning [6], or more specifically, deep neural networks (DNNs) have become the dominant methods to solve machine learning (ML) and artificial intelligence (AI) problems in the fields of computer vision (CV), natural langurage processing (NLP),

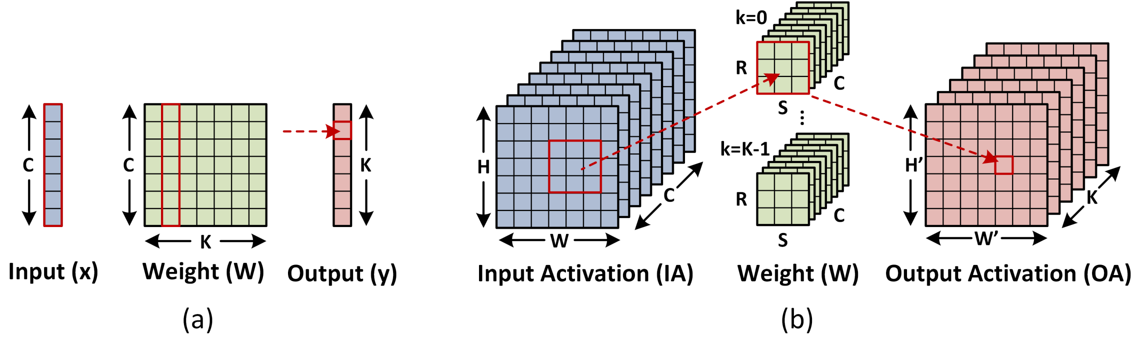


Figure 1.2: Core computations of DNNs: (a) vector-matrix multiplication in MLP and RNN, and (b) 2D convolution in CNN.

autonomous driving, and robotics [1, 2, 7–17]. To achieve an even better accuracy, there is an constant effort to design larger and more powerful models, resulting in increasing number of parameters and computation complexity. Figure 1.1 presents the accuracy of modern network models with their model size and complexity in terms of number of parameters and operation counts.

1.1 DNN Computation

In general, there are three different network structures for DNN models: 1) multi-layer perceptron (MLP), 2) convolutional neural network (CNN), and 3) recurrent neural network (RNN). We present the three network structures along with its core computation.

An MLP consists of multiple feedforward fully-connected (FC) layers cascaded one after another. The computation of an FC layer can be formulated into a vector-matrix multiplication (VMM) between the input vector $\boldsymbol{x} \in \mathbb{R}^C$ and the weight matrix $\boldsymbol{W} \in \mathbb{R}^{C \times K}$ to obtain the output vector $\boldsymbol{y} \in \mathbb{R}^K$, as described in Figure 1.2(a).

A CNN is specialized to 2D image processing and uses convolution (CONV) layers for spatial feature extraction and FC layers for classification. The input and output are often referred as input activation (IA) and output activation (OA). A CONV layer has a weight (W) of size $R \times S \times C \times K$ and receives an IA of size $H \times W \times C$ to

obtain an OA of size $H' \times W' \times K$, as shown in Figure 1.2(b). In terms of computation, a CONV layer performs 2D convolution with a kernel size of $R \times S$ on the IA. The partial sums (Psums) are accumulated across input channel C to obtain an OA. The same operation is repeated for K kernels. The model hyperparameters C and K are the input and output channel sizes, respectively. The output channel size is also known as the (weight) kernel number.

An RNN uses recurrent connections to process the input sequence of the current timestep t and the output sequence from the previous timestep $t-1$. Two popular forms of the recurrent unit are widely adopted: gated recurrent unit (GRU) and long short term memory (LSTM). An LSTM uses input, output, forget gates, and a cell, i.e., i, o, f, c, to keep track of features that are relevant in long term and improves accuracy over traditional recurrent units. The computation of an LSTM can also be formulated into a VMM (Figure 1.2(a)), where the input vectors are the input sequence $\boldsymbol{x}_t$ and the hidden sequence $\boldsymbol{h}_{t-1}$, the matrix is the concatenation of i, f, o, c matrices with respect to the input or hidden sequences, and the output vector is the hidden sequence $\boldsymbol{h}_t$.

1.2 Network Model Optimization

The continuing growth of model size and computation complexity poses a significant challenge to the deployment to real-time applications or on resource-constrained devices. As shown in Figure 1.3, state-of-the-art CV and NLP models requires millions or billions of parameters to obtain the best accuracy [9, 12]. Model compression techniques are proposed to reduce the model size and computation complexity.

On the other hand, there is an increasing effort to apply deep learning into new fields with new modalities or non-Euclidean data beyond the heavily-focused CV and NLP fields [14, 15, 18, 19]. To obtain better accuracy, new operations are proposed from scratch or adapted from traditional operations.

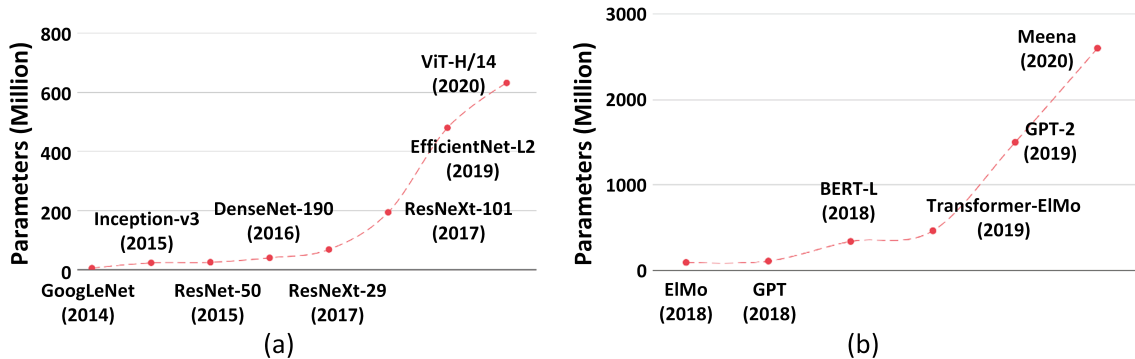


Figure 1.3: Evolution of model size in the fields of (a) CV and (b) NLP. Adapted from [2].

At the algorithm-level, model compression techniques and novel operation types successfully achieve smaller model size and better accuracy, respectively. However, most techniques also introduce sources of inefficiency at the hardware-level, resulting in lower compute utilization and larger computation overhead. Therefore, hardware accelerator and specialized architecture are necessary to bridge network optimization algorithms to hardware performance and computation efficiency.

1.2.1 Model Compression

The rapid growth of the model size and computation complexity leads to a huge cost for model training and inference that only server-scale graphics processing units (GPUs) and central processing units (CPUs) equipped with large compute parallelism, memory storage, and memory bandwidth can support. For example, GPT-3 model contains 175 billion parameters and requires trillions of multiply-and-add (MAC) operations for a single inference [12]. While these heavy models perform extremely well at server-scale, they are impractical and cannot be deployed to resource-constrained platforms, i.e., mobile, drone, and smart sensor, where compute, memory, bandwidth, and battery are largely limited. This has motivated researches on model compression techniques that aim to reduce the model size and the complexity of a DNN model while preserving the accuracy [2, 20–22]. Several model compression techniques, i.e.,

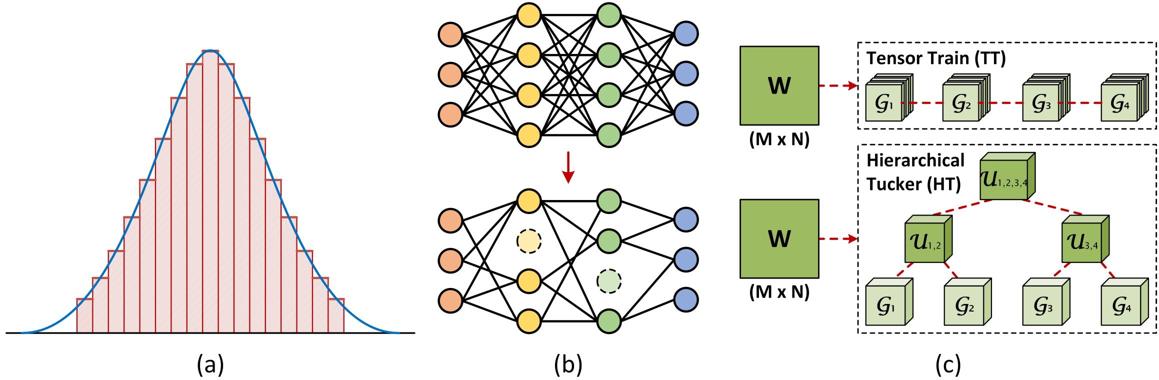


Figure 1.4: Common model compression techniques: (a) data quantization, (b) network sparsification, and (c) tensor decomposition.

data quantization, network sparsification, and tensor decomposition, have been proposed and are widely applied to DNNs.

Data quantization (Figure 1.4(a)) reduces the data bit precision of model parameters by truncating or rounding high-precision values into lower-precision values [23, 24]. A quantized network requires the same amount of computation (at reduced precision), but needs much less bits for memory storage.

Network sparsification (Figure 1.4(b)) is performed via network pruning that removes unimportant connections in the network, generating a sparse model [25–28]. A sparse network can be stored in a sparse format, i.e., compressed sparse row (CSR), to reduce the amount of memory space needed. In addition, a sparse network reduces the amount of computation since most multiplications are unnecessary and can be avoided.

Tensor decomposition (Figure 1.4(c)) aims to find a simpler, low-rank approximation of the original weight matrix or tensor [29–34]. Traditional low-rank approximation methods, i.e., singular-value decomposition (SVD) [29], are used for 2D weight model, whereas, recent tensor decomposition methods, i.e., tensor-train (TT) [33, 34], decomposes the original model into a series of smaller tensors, largely reducing the memory space.

Note that both data quantization and sparsification can be applied to the input

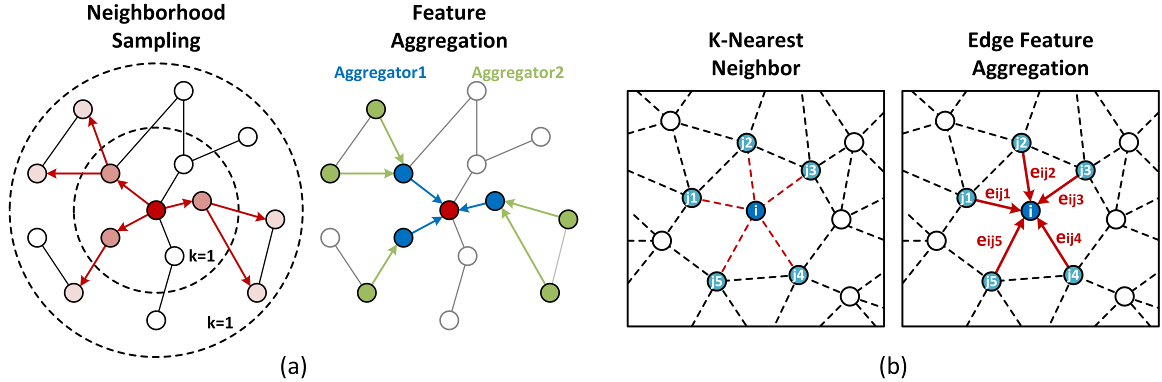


Figure 1.5: Examples of novel operations: (a) GraphSAGE in GNNs and (b) Edge-CONV in point-cloud networks. Adapted from [3, 4].

vector or activation. These compression methods can also be combined when applying on a network, which may generate an even larger compression of the model [21].

1.2.2 Novel Operation Types

After the wide success of CNNs and RNNs in the CV and NLP fields, researchers have worked on converting the insights obtained from traditional network models to new applications and new modalities. For example, a graph is a natural representation of the relationship between the data points in a network and is used for graph mining applications, i.e., community detection, social network analysis [18, 19]; a point cloud is an accurate representation of 3D space and has become a popular modality for 3D vision tasks, i.e., object classification and scene segmentation [14, 16, 17]. These new modalities and non-Euclidean data may have completely different data structures and characteristics compared to the traditional 2D images and 1D text sequences. Therefore, the traditional operations, i.e., CONV, may not be sufficient, leading to the development of novel operations and network models. For example, graph convolution (GraphCONV) [35] with different frameworks i.e., GraphSAGE [3], DeepWalk [36], are used in graph neural networks (GNNs) to obtain better graph embeddings. Similarly, for point-cloud recognition, novel operations, like edge con-

volution (EdgeCONV) [4], were proposed to extract more meaningful features from point clouds.

The novel operations proposed enable the application of DNN into new fields of applications and successfully outperform the accuracy of traditional models [14, 15, 19].

GraphSAGE (Figure 1.5(a)): In GraphSAGE [3], the neighborhoods for each node are first sampled from the graph. Then, each node in the graph aggregates the representations of the nodes in its k-hop neighborhood. The aggregated neighborhood vector is concatenated with the node's current representation, then processed with an FC layer to obtain the new representation of the node. This process is repeated for every hop count $k = 1, \ldots, K$.

EdgeCONV (Figure 1.5(b)): For each point in the point cloud, the EdgeCONV operation forms an edge between the point and its K-nearest neighbors, then computes the edge features based on the point feature difference for every edge. Then, the edge features are aggregated to obtain the output feature of the point [4].

1.2.3 Computation Challenges

At the algorithm-level, model compression techniques succeed in achieving better balance between reasonable accuracy and affordable model size. Similarly, novel operation design can achieve higher accuracy than a traditional model. However, most techniques also introduce sources of inefficiency at the hardware-level. For instance, the sparse formats from network sparsification are highly unstructured and irregular, causing an inefficient parallel computation for a single instruction, multiple data (SIMD) architecture. Similar irregular computation and memory access can be observed for EdgeCONV computation in point-cloud networks. A SIMD architecture must gather and reorganize the data obtained from scattered memory accesses before performing parallel computation, causing a low compute utilization. In addition,

the set of tensors from tensor decomposition require complex and arbitrary data orchestration to map a tensor contraction into matrix multiplication, resulting in additional memory access and control overheads if the mapping is not optimized. These motivate us to design specialized architectures and hardware accelerators to unlock the full potentials of network optimizations and achieve higher processing performance and efficiency.

1.3 Dissertation Outline

To fully benefit from model compression techniques and to support novel operation types, specialized hardware mechanisms and signal processing techniques must be incorporated into the accelerator architecture and design. This dissertation focuses on the design of efficient AI and ML accelerator architecture using optimizations at algorithm, architecture, and microarchitecture levels.

The organization of this dissertation is summarized as follows. Chapter II presents SNAP [37, 38], a sparse neural acceleration processor for unstructured sparse neural network (NN) processing. Specialized associative index matching (AIM) units were designed to pair IA and W operands and provide a sufficient number of them to maintain compute utilization. A two-level reduction dataflow was proposed to eliminate unnecessary memory accesses in the psum reduction process. Chapter III presents Point-X [39, 40], a spatial-locality-aware (SLA) architecture for efficient graph-based point-cloud NN processing. An SLA clustering extracts the spatial locality in the input point cloud to maximize the computational parallelism at each compute tile (CTile) and minimize the inter-CTile data movement. By exploiting the locality, a low-cost chain NoC is designed to reduce the data exchange latency during processing. Chapter IV presents TetriX, an architecture and mapping co-design for efficient and flexible tensorized neural network (TNN) processing. An optimal search is used to identify the contraction sequence with minimized computation for TNN

inference. A hybrid mapping scheme is proposed to eliminate complex tensor orchestration operations by alternating between inner and output product operations. An index translation and an output gathering mechanisms are designed to support arbitrary and scalable tensor permute and reshape operations. Chapter V concludes this dissertation and provides outlooks for future research in ML and AI accelerator design.

CHAPTER II

SNAP: Accelerator Architecture for Unstructured Sparse Neural Networks

Deep learning or more specifically, deep neural network (DNN), has emerged to be a key approach to solving complex cognition and learning problems [6, 41]. State-of-the-art DNNs [7, 8, 42–47] require billions of operations and hundreds of megabytes to store activations and weights. Given the trend towards even larger and deeper networks, the compute and storage requirements will prohibit real-time, low-power deployment on platforms that are resource and energy constrained. The compute and storage challenges motivated efforts in network pruning to zero out a large number of weights (W) in a DNN model with only little inference accuracy degradation [25, 26, 28]. In addition to sparsity in weights, the commonly-used rectifier linear unit (ReLU) clamps all negative activations to zeros, resulting in sparsity in output activations (OAs), which become input activations (IAs) of the next layer.

Figure 2.1 shows that the typical density of nonzero W (after network pruning [26]) and IA (due to ReLU) in well-known network models: AlexNet, VGG-16 and ResNet-50. An average of 50% density is common. Because the nonzero W and IA are nearly randomly distributed, the amount of effectual computation, i.e., computation that does not involve a zero, is only 25%. If a small sacrifice in inference accuracy can be tolerated, the density of operands and the effectual computation can be further

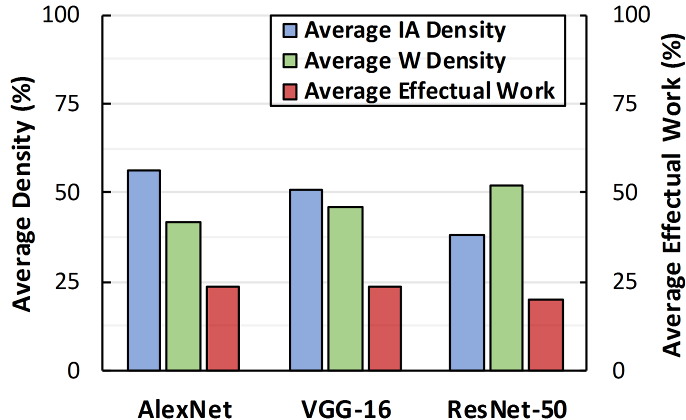


Figure 2.1: Average density of IA, W data and average effectual work of common DNNs after network pruning.

reduced.

Data sparsity can be exploited to save power. Many DNN accelerators, i.e., Eyeriss [48], gates the computation by turning off the clock, whenever a zero in the IA is detected in runtime. Most dense DNN accelerators can incorporate this technique to reduce power, but it does not shorten the latency or improve the throughput.

Cnvlutin [49] and Cambricon-X [50] are well-known early architectures that exploit sparsity in compressed IA for latency reduction and throughput improvement. However, they were designed to work with the sparsity in one of the two operands, W or IA, but not both. A dense processing architecture can be easily adapted to support one-operand sparsity by indirect data access.

To fully exploit sparsity in both operands, W and IA are stored in a compressed form where nonzero elements are represented by value-index pairs. Storage in a compressed form can reduce the memory size and bandwidth. However, unlike the common dense array and matrix storage, a compressed storage is not amenable to regular and efficient vector processing. One approach is to decompress the compressed form before processing, but decompression costs performance, memory, and power. Instead, state-of-the-art sparse DNN accelerators [51–55] process data directly in the compressed form, offering both low memory bandwidth and high degree of accelera-

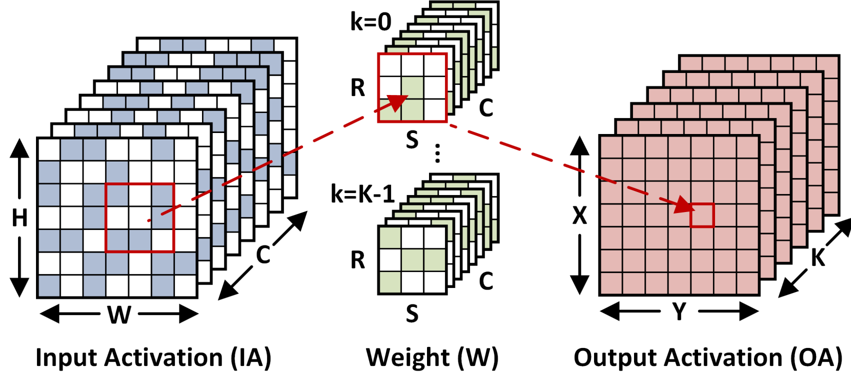


Figure 2.2: Convolution computation between unstructured sparse IA and W in a sparse DNN. The colored cells indicate nonzero entries, and the white cells indicate zero entries.

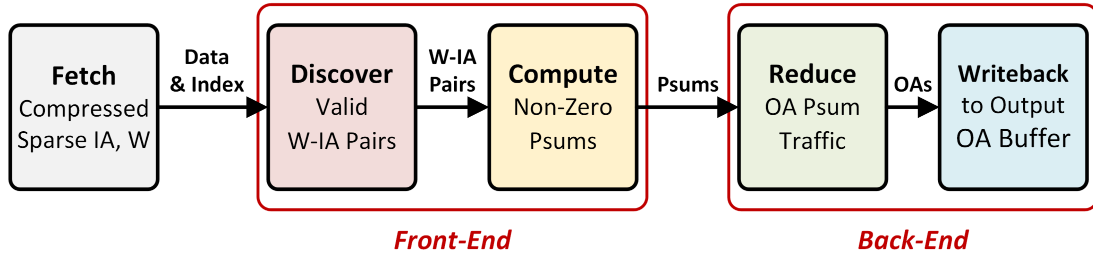


Figure 2.3: Processing pipeline of a sparse DNN processor.

tion.

Figure 2.2 illustrates sparse convolutions, and Figure 2.3 shows the high-level computation pipeline of DNN processing in the compressed format (which will be referred to as sparse DNN processing for simplicity). A sparse DNN processor loads Ws and IAs in a compressed form consisting of a data array (zeros removed) and an index array. Compressed W and IA arrays are paired by matching indices, dispatched to a multiplier array, and the resulting partial sums (Psums) are accumulated to their respective OAs in output buffers.

Data sparsity leads to better performance and efficiency, but major challenges remain:

1. ***Front-end challenge***: Multiplier under-utilization due to an insufficient number of W-IA pairs that can be extracted and dispatched to the multiplier array.

2. ***Back-end challenge***: Data traffic and access contention to support accumulation of psums whose destination addresses are seemingly random.

3. ***Flexibility challenge***: Limited support for different kernel sizes and layer types.

State-of-the-art sparse DNN accelerators including EIE [51], SCNN [52], Sticker [53, 54], and Eyeriss v2 [55] addressed some of the challenges in sparse DNN processing, but did not solve all of them. EIE [51] exploits both W and IA sparsity but is restricted to fully-connected (FC) layers. SCNN [52] is the first attempt at exploiting both W and IA sparsity for convolution (CONV) layers. It maximizes multiplier utilization at the cost of massive psum writeback traffic and access contention, and it supports only CONV layers. Sticker [53, 54] follows SCNN's dataflow and uses 2-way set-associative processing elements (PEs) to alleviate the access contention but requires offline preprocessing to re-arrange IA data. Without the data re-arrangement, the access contention remains as significant as in SCNN. Eyeriss-v2 [55] employs a two-step search front-end to find effectual W-IA pairs by first fetching nonzero IAs, and then using the channel index of the IA to look for nonzero Ws. Eyeriss-v2 adopts an Eyeriss-like row stationary dataflow [56] to avoid memory access contention.

We present Sparse Neural Acceleration Processor (SNAP) that adopts a channel-first dataflow and is optimized for the efficient processing of unstructured sparse DNNs. To solve the front-end challenge, SNAP uses parallel associative index matching (AIM) units and sequence decoders to extract a sufficient number of W-IA pairs to maintain a high multiplier array utilization of 75%. To solve the back-end challenge, SNAP adopts a two-level psum reduction that consists of PE level and core-level reduction to eliminate memory access contention and reduce psum writeback traffic to an average of 2.79 OA reductions/cycle for a core of 63 multipliers. The core-level reduction is configurable to support common layers in vision-based DNN models, including general $R \times S$ CONV ($R, S > 1$), pointwise CONV, and FC.

The rest of the chapter is organized as follows. Section 2.1 introduces the channel-last dataflow, an approach adopted by state-of-the-art sparse accelerators, and analyzes its advantages and inefficiencies in processing sparse DNNs. Section 2.2 presents our channel-first dataflow and quantitatively compares it against the channel-last dataflow to demonstrate its advantages. In Section 2.3, we describe our solution to the front-end challenge using parallel index matching, and in Section 2.4, we describe our solution to the back-end challenge using two-level psum reduction. The configurable core-level reduction makes the SNAP architecture flexible to support different kernel sizes and layer types. Section 2.5 presents the overall SNAP architecture, followed by measurement and evaluation results using both synthetic sparse workloads and commonly-used pruned networks. Finally, Section 2.6 summarizes the contributions of this work.

2.1 Background

A dense CONV operation can be described by Eq. (2.1), where f represents the activation function. For simplicity, the bias is ignored and IA padding is assumed to be zero. FC can be viewed as a special case of CONV. In this work, we will use the following indexing convention: a W index of (r, s, c, k) corresponds to (row, column, channel, kernel), an IA index of (h, w, c) corresponds to (row, column, channel), and an OA index of (x, y, k) corresponds to (row, column, output channel (kernel)).

$$OA_{(x,y,k)} = f\left(\sum_{i=0}^{R-1}\sum_{j=0}^{S-1}\sum_{c=0}^{C-1} IA_{(x+i,y+j,c)} \times W_{(i,j,c,k)}\right). \tag{2.1}$$

As Eq. (2.1) shows, there are two main steps in computing CONV (besides the activation function): 1) IAs and Ws are multiplied to produce psums; and 2) the psums are accumulated (or reduced) along the channel dimension (C) and along the pixel dimension (R, S). The channel-dimension reduction and pixel-dimension

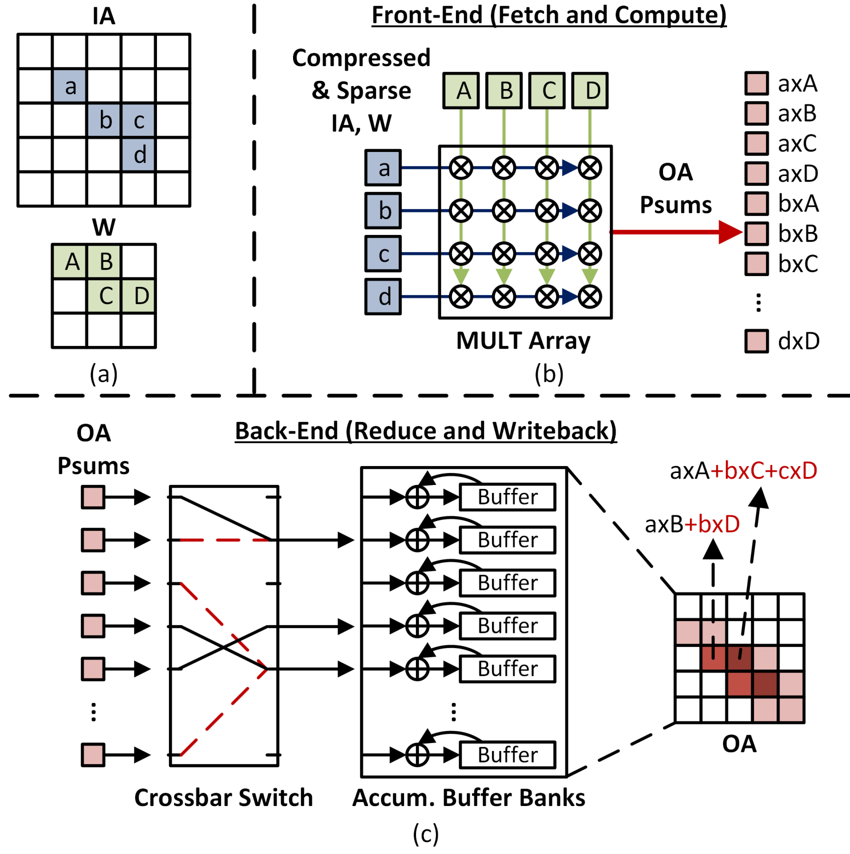


Figure 2.4: Illustration of channel-last dataflow for sparse DNN processing.

reduction are commutative. The channel indices of a W and an IA need to match for the two to be multiplied together.

In ordering inputs for storage and processing, we can choose either the pixel dimension first ((r, s) for W and (h, w) for IA) followed by the channel dimension (c), termed channel-last dataflow, or vice versa, termed channel-first dataflow. Both SCNN [52] and Sticker [54] adopt the channel-last dataflow.

2.1.1 Channel-Last Dataflow for Sparse DNN Processing

The channel-last dataflow is illustrated in Figure 2.4. In the channel-last dataflow, the nonzero W and IA data are ordered in the pixel dimension first for storage and processing. Because data are ordered pixel dimension first, as W and IA data are streamed in, their channel indices are aligned. Since any nonzero W can be multiplied

with any nonzero IA of the same channel in a CONV operation, the W and IA can be freely paired to produce a large number of W-IA pairs for a multiplier array.

The simple W-IA pairing results in a simple front-end for the channel-last dataflow. Shown in Figure 2.4(a) and (b), the compressed W and IA data of size n can be broadcast over a $n \times n$ 2D multiplier array vertically and horizontally, respectively, so that each W is multiplied to every IA. The drawback of the channel-last dataflow is that the address of the OA that a psum needs to be accumulated to (will be referred to as the psum address) does not follow a deterministic ordering. According to Eq. (2.1), if a nonzero W and a nonzero IA have matching channel index c, they can be multiplied to produce a psum with an index of $(x, y, k) = (h - r, w - s, k)$. The 3D index is then translated to a 1D physical address. Hence, the $n \times n$ 2D multiplier array produces n^2 psums whose (x, y) indices are $\{(h - r, w - s)\}$ with random drawings of $h \in [0, H - 1], r \in [0, R - 1], w \in [0, W - 1]$, and $s \in [0, S - 1]$. It is highly likely to have two or more psums that share the same address, and in theory they should be accumulated, or reduced, before writeback. However, it is challenging to organize psums and reduce them before writeback. Without any psum reduction, the writeback traffic becomes congested, and frequent contentions are possible at the OA buffer. It requires complex hardware or wiring, e.g., a crossbar switch, to resolve the contention, and results in pipeline stalls and a low multiplier array utilization.

This back-end challenge is illustrated in Figure 2.4(c). The psums need to be distributed by a switch to the corresponding buffer bank. The red lines indicate the psum writebacks that lead to buffer contentions. To avoid contentions, conflicting psums need to be held. In the example, one output requires the accumulation of 3 psums, resulting in a 3-cycle writeback where the multiplier array stalls for 2 cycles.

2.1.2 Other Related Work

Numerous DNN accelerators have been proposed to exploit the parallelism in DNN inference operations [38, 48–62]. To leverage the sparsity in Ws and IAs, some work implemented power-saving techniques by clock-gating a PE when a zero IA is detected [48, 59] to increase energy efficiency. Some work exploited sparsity at the bit level [60] by skipping the computation for the zero-valued bits in the bit-serial multiplication. Compared to earlier methods, exploiting sparsity in bit-level reduces the overall computation cycles, and increases both efficiency and throughput. However, the dataflows are still similar to traditional dense accelerators, where zero elements are fetched on-chip, incurring unnecessary data transfers.

In this work, we focus on the DNN inference accelerator design that exploits sparsity at data level and operates in the compressed form. Only nonzero elements are fetched on-chip for computation, like in [51, 52, 54, 55]. This type of accelerator skips all unnecessary data transfers and computation to optimize for energy efficiency and computation throughput. We will refer to this type of accelerator as a sparse accelerator for discussion and comparison.

2.2 Channel-First Processing Dataflow

Compared to the channel-last dataflow, the channel-first dataflow orders W and IA data across the channel dimension first for storage and processing. The channel-first dataflow allows the psums computed by the multiplier array to be locally reduced before writeback.

2.2.1 Compression Format

In the channel-first dataflow, nonzero W and IA data are ordered and processed in the channel dimension first, and then in the pixel dimension. An example of a 3×3

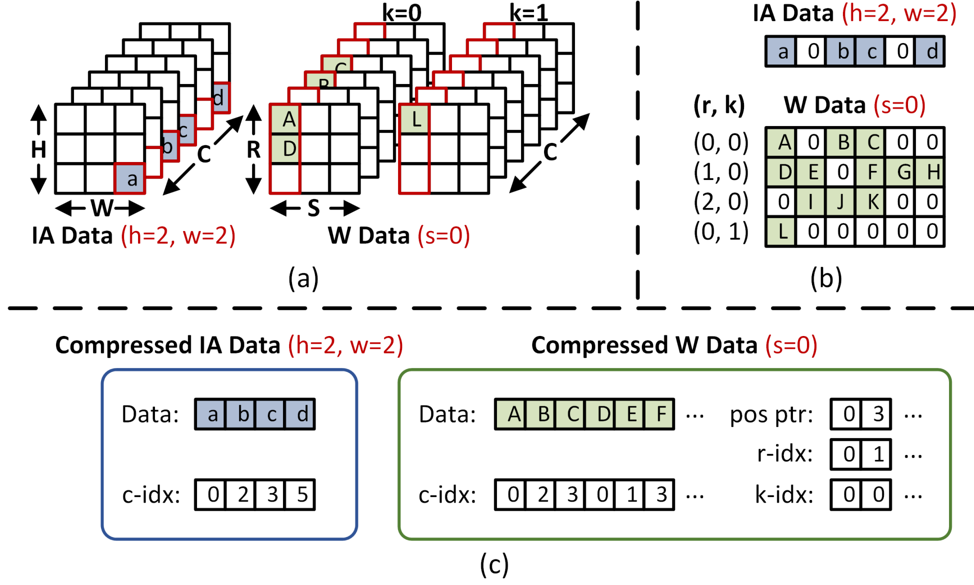


Figure 2.5: Channel-first compression in SNAP.

CONV is shown in Figure 2.5. The channel-first dataflow supports arbitrary IA data size and a 3×3 IA data is chosen for the ease of illustration.

In Figure 2.5(a), the W and IA data are illustrated in the dense form. In our channel-last dataflow, a bundle of nonzero W data and a bundle of nonzero IA data are provided to a PE at a time. An IA bundle contains data in $(h, w, \underline{c})$, where $\underline{c}$ represents all nonzero elements along the channel dimension; and a W bundle is larger and it contains data in $(\underline{r}, s, \underline{c}, \underline{k})$, where $\underline{r}$, $\underline{c}$, $\underline{k}$ represent all nonzero elements along the row, channel and kernel dimension, respectively. In general, a larger data bundle leads to a higher utilization and processing efficiency, but we limit the IA bundle not to span more than one pixel location to simplify channel index matching.

Figure 2.5(b) illustrates the IA bundle of $(h, w, \underline{c}) = (2, 2, \underline{c})$ and the W bundle of $(\underline{r}, s, \underline{c}, \underline{k}) = (\underline{r}, 0, \underline{c}, \underline{k})$. The bundles are stored in the compressed form with all zeros squeezed out as shown in Figure 2.5(c). The IA data are stored channel first. The IA storage consists of a data array that stores nonzero IA data and a channel index (c-idx) array that stores the channel index of the corresponding IA data. The W data are also stored channel first, followed by row (r-idx) and kernel (k-idx). The W

storage consists of a data array, a c-idx array, a r-idx array and a k-idx array, as well as a position pointer (pos-ptr) array to track the starting points in the data array of the next r-idx and/or k-idx. For instance, the pos-ptr array stores 0 and 3, indicating that the first three data values A, B, and C have $(r, k) = (0, 0)$, and the data values D, E, and F have $(r, k) = (1, 0)$. An IA bundle and a W bundle are sent to one PE for processing.

2.2.2 Channel-First Dataflow

In a channel-first dataflow, the nonzero W and IA data are streamed in channel first, the addresses of the psums computed are aligned and the psums can be immediately reduced along the channel dimension. Despite the appeal of the channel-first dataflow, a W and an IA can only be paired and multiplied if their channel index match. Hence, channel index matching must be performed at the front-end to extract the valid W-IA pairs. Compared to the channel-last dataflow, this additional channel index matching step introduces an overhead, but it provides immediately-reducible psums to cut the writeback traffic, leading to potential improvements in both power and performance.

The channel-first dataflow is illustrated in Figure 2.6. Each PE receives a W bundle $(\underline{r}, s, \underline{c}, \underline{k})$ and an IA bundle $(h, w, \underline{c})$, as illustrated in Figure 2.5(c). The W c-idx is matched with the IA c-idx to generate valid W-IA pairs. Valid W-IA data pairs are fetched and multiplied to produce psums. The psums are accumulated and saved to the OAs at $(x, y, k) = (h - r, w - s, k)$. Due to the channel-first input ordering and bundled processing, the address of the psums computed by one PE will stay the same until the PE completes an IA bundle and switches to a new IA bundle (change of h, w), or until the r-idx or k-idx changes (change of r, k) for a W bundle. Due to the high locality of psum address, the majority of the psums are immediately reduced to one within a PE. To process a complete convolution, the IA and W data

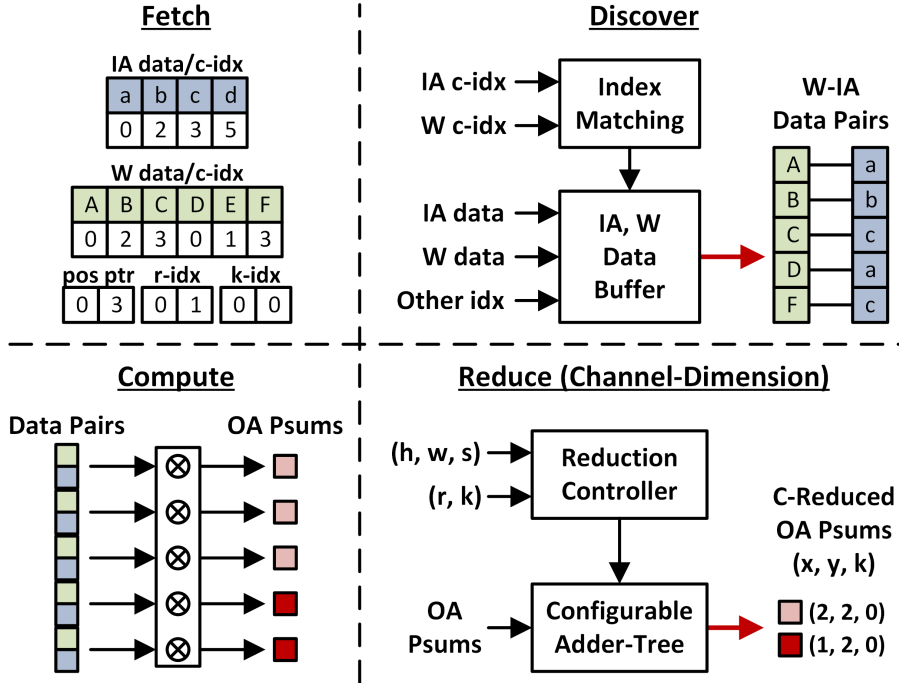


Figure 2.6: SNAP's channel-first dataflow with channel index matching and psum reduction along the channel dimension.

are decomposed and compressed into IA and W bundles. The IA and W bundles are distributed to different PEs for processing as shown in Figure 2.6 until all bundles are fully processed.

In a channel-first dataflow, channel-dimension psum reduction is done first. A second-level pixel-dimension psum reduction can be done on-chip to further reduce the writeback bandwidth. To enable the second-level psum reduction, PEs can be arrayed and coordinated to facilitate the psum reduction across the PEs. The second-level psum reduction will be described in more detail in Section 2.4.2.

To evaluate the benefits of the channel-first dataflow, we prototyped a channel-last dataflow that follows the processing pipeline described in Figure 2.4 and quantify the key differences between the two dataflows as shown in Figure 2.7. The channel-last dataflow is limited by the large number of OA buffer accesses and the access contention, causing the compute to stall and worsening both utilization and processing latency; and in comparison, the channel-first dataflow employs a front-end

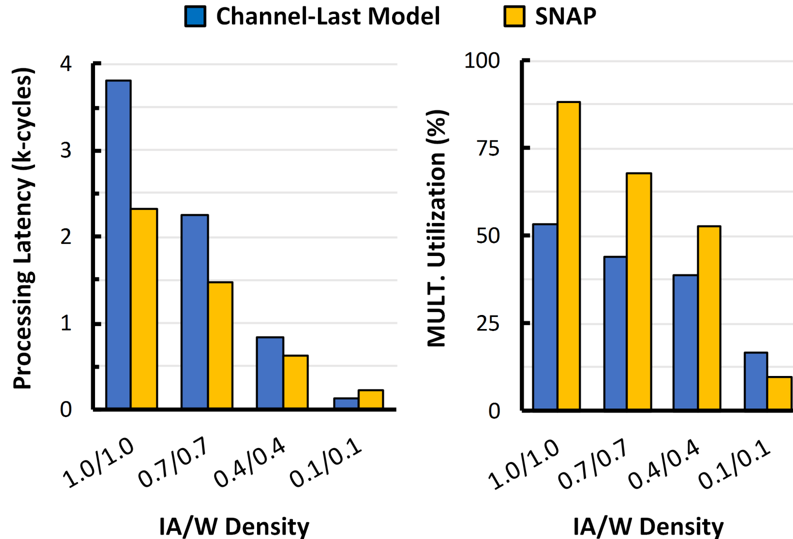


Figure 2.7: Comparison between channel-last dataflow and SNAP (channel-first dataflow) for dense, medium and sparse workloads.

channel index matching to reduce the number of OA buffer accesses and remove the access contention. The channel-first dataflow provides on average 1.51× and 1.45× improvement over the channel-last dataflow in processing latency and multiplier utilization, respectively. Only when the data density drops below a threshold, e.g., 10% W and IA data density, does the channel-first dataflow starts to underperform the channel-last dataflow due to two factors: 1) the lack of reduction opportunities due to highly sparse data, and 2) the imbalance of input bundle sizes causing PE workload imbalance.

The SNAP architecture follows the channel-first dataflow. The two key techniques, channel index matching and two-level psum reduction, will be presented in the next two sections.

2.3 Channel Index Matching

Channel index matching extracts pairs of nonzero W-IA pairs of matching channel index. We propose an associative index matching (AIM) unit to extract a sufficient

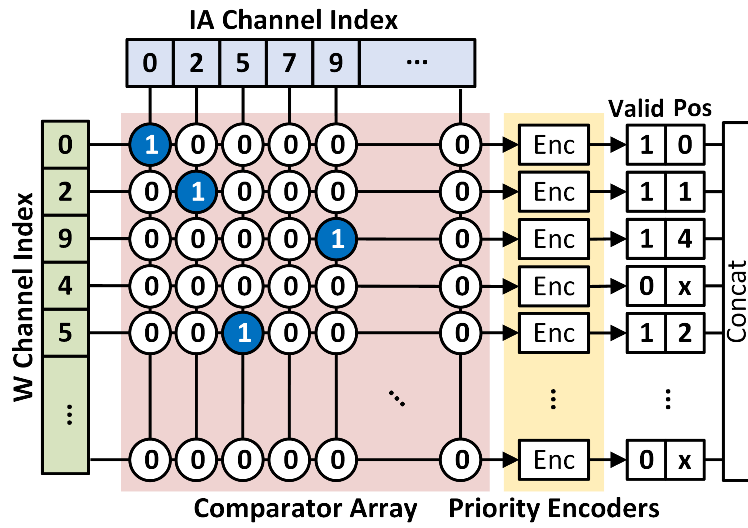
number of W-IA pairs to sustain a high utilization of a parallel multiplier array. The AIM performs index matching, encodes the addresses of valid W-IA pairs, and a sequence decoder decodes the addresses and dispatches the pairs for parallel computation.

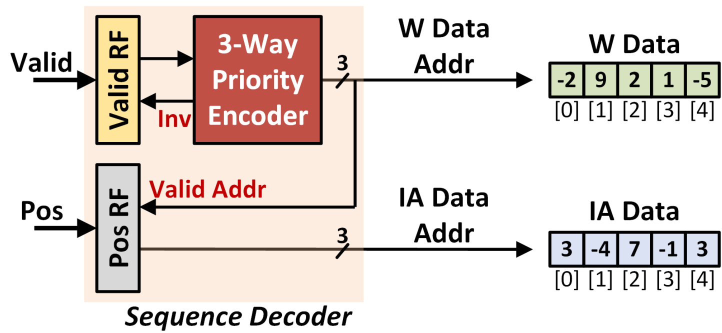
2.3.1 Associative Index Matching

Figure 2.8(a) shows the microarchitecture of the AIM and illustrates its mechanism. The AIM consists of a $N \times N$ comparator array, where each row is connected to a priority encoder. During operation, an AIM receives the W and IA channel index arrays from a PE; and it compares each W channel index to each IA channel index. In Figure 2.8(a), for example, the W channel indices 0, 2, 9, 5 are matched to the IA channel index at position 0, 1, 4, 2, respectively, whereas the W channel index 4 does not have a match. A priority encoder encodes the match result in each row into a valid bit to indicate a match and the matched position in the IA channel index array. Upon completion, an AIM returns a list of valid-position pairs to a PE for processing.

2.3.2 Sequence Decoder

Within a PE, a sequence decoder converts a list of valid-position pairs to W-IA data pairs. Figure 2.8(b) shows the microarchitecture and illustrates the sequence detection mechanism with the valid-position list output from Figure 2.8(a): 1) a 3-way priority encoder converts 3 valid-position pairs (with valid bit = 1) at a time to W-IA data addresses; 2) the positions in the valid-position pairs are used as the addresses to fetch IA data, and the indices of the valid-position pairs are used as the addresses to fetch W data; and 3) the 3 valid-position pairs are invalidated by overwriting the valid bits to 0, and the W and IA data are sent to the multipliers to compute psums. After completing the list of valid-position pairs, the PE requests a new list from the AIM.





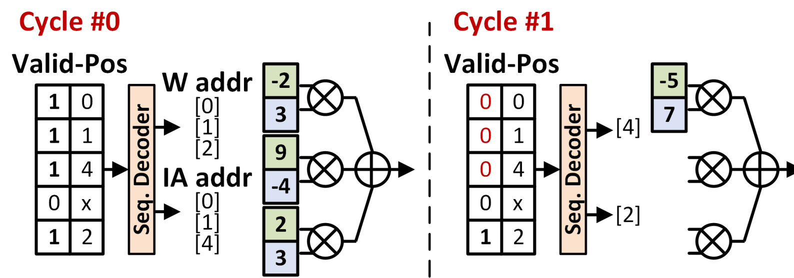


Figure 2.8: (a) Microarchitecture of associative index matching (AIM), and (b) microarchitecture of sequence decoder and step-by-step example of W-IA data pair dispatch in a PE.

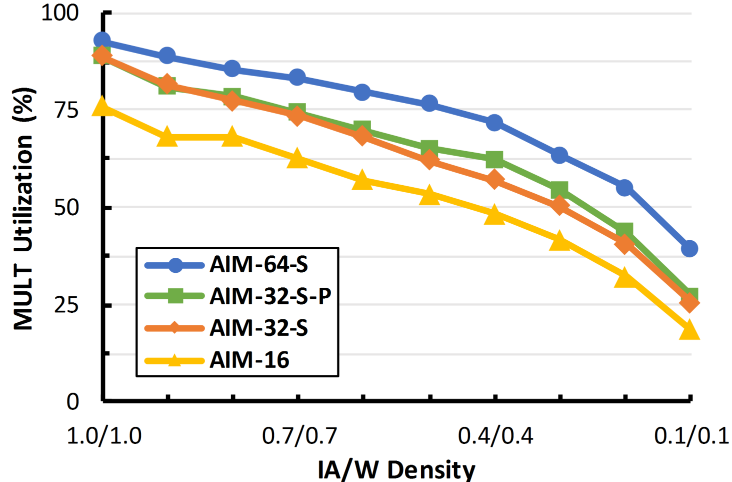


Figure 2.9: Multiplier utilization of AIM designs at different data density levels.

2.3.3 Design Tradeoff Exploration

The size of the comparator array, N, determines AIM's throughput and effectiveness. N needs to be sufficiently large to even out the workload density imbalance and variations across W and IA bundles for extracting enough W-IA pairs to maintain a high multiplier utilization. However, the area and power consumption of AIM increase nearly quadratically with N. To balance these two competing factors, we avoid using a small AIM, e.g., one with a 4×4 or 8×8 comparator array, and instead use a larger AIM and time-multiplex it between multiple PEs to amortize the cost.

Figure 2.9 shows the multiplier utilization across a range of workload densities for $N = 16$, 32, or 64, where a suffix S indicates that a large AIM is time-multiplexed among an appropriate number of PEs, and a suffix P indicates that a simple prefetch mechanism is implemented to further reduce the workload imbalance by pre-requesting valid-position pairs from the AIM. A larger AIM provides a higher multiplier utilization across all workload densities: the utilization improves by 10% from $N = 16$ to 32, and again from 32 to 64. A large AIM with $N = 64$ incurs an area overhead of 50%. A moderate-sized AIM with $N = 32$ cuts the area overhead below 12.5%, and adding prefetching increases the utilization by up to 5%. Therefore, in

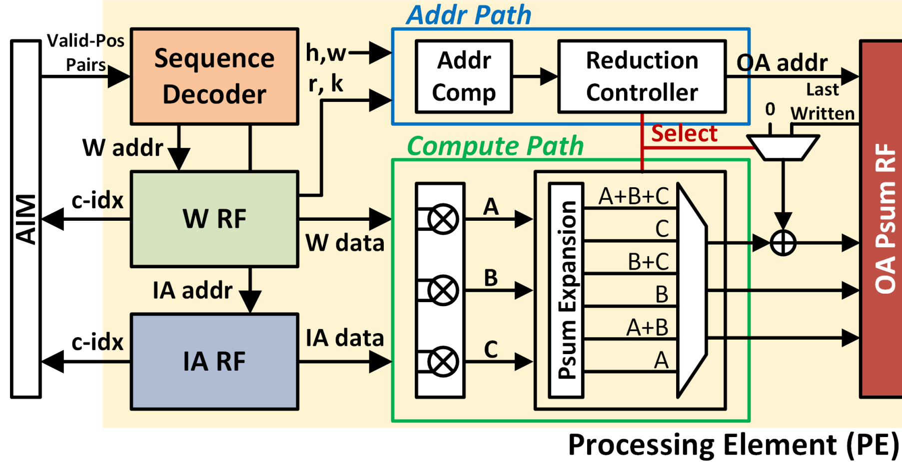


Figure 2.10: PE microarchitecture and psum reduction along the channel dimension.

designing SNAP, we adopt a time-multiplexing, prefetching AIM design with a $N = 32$ comparator array to balance the performance, area and power consumption. This design achieves an average multiplier utilization above 75% for our benchmarks.

2.4 Two-Level Partial Sum Reduction

To reduce the output bandwidth, after the psums are computed by the PEs, they should be maximally accumulated to reduce the number of writebacks to the output buffer. Following the channel-first dataflow, SNAP implements a two-level psum reduction to minimize the read-accumulate-writes to the output buffer. The psums are first reduced along the channel dimension inside the PEs, and then along the pixel dimension across PEs. The across-PE reduction is configurable to support not only CONV, but also pointwise CONV and FC.

2.4.1 PE-level Channel-Dimension Reduction

The PE microarchitecture is shown in Figure 2.10. Each PE contains a compute path consisting of 3 multipliers and psum accumulators, an address path that computes the addresses and then selects the psum reduction pattern, a sequence decoder,

Table 2.1: Selection of Reduction Pattern

OA Addresses	Out[0]	Out[1]	Out[2]
$Addr_A = Addr_B = Addr_C$	A+B+C	-	-
$Addr_A = Addr_B < Addr_C$	A+B	C	-
$Addr_A < Addr_B = Addr_C$	A	B+C	-
$Addr_A < Addr_B < Addr_C$	A	B	C

a W register file (RF) and an IA RF to provide the inputs, and an OA psum RF to store the outputs. In each cycle, the sequence decoder dispatches 3 W-IA data pairs and their indices ((h, w) for IA data and (r, s, k) for W data). The W-IA data pairs are directed to the compute path to produce the psums, A, B, and C; and the W and IA indices are sent to the address path to compute the addresses (recall psum index is $(x, y, k) = (h - r, w - s, k)$, which is translated to a physical 1D address). Given the 3 psum addresses, the reduction controller selects one psum reduction pattern in the compute path.

The channel-first input processing order guarantees that the addresses for A, B, and C are ordered and non-decreasing, i.e., $Addr_A \leq Addr_B \leq Addr_C$. Due to the deep channels seen in modern DNN layers, in most cases, $Addr_A = Addr_B = Addr_C$, and the 3 psums can be accumulated and reduced to one. If not, the reduction controller selects one of the reduction patterns according to Table 2.1. One, two or three psums are produced and sent to the OA psum RF every cycle, avoiding stalls in the computation pipeline. A PE retains a OA psum in the RF until all possible reductions along the channel dimension are complete, cutting the psum writeback traffic by up to the channel depth compared to the channel-last dataflow. Note that the number of multipliers in a PE is set to 3 to keep a reasonable overhead for the reduction pattern selection.

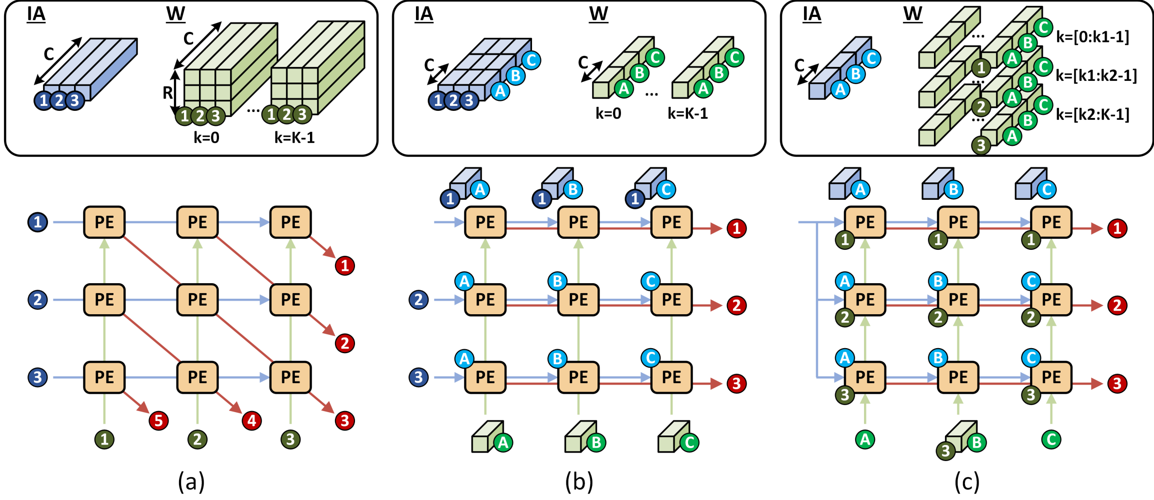


Figure 2.11: Psum reduction configuration across an array of PEs (a) 3×3 CONV in diagonal mode, (b) pointwise CONV in row mode, and (c) FC in row mode.

2.4.2 Core-Level Pixel-Dimension Reduction

To reduce the writeback traffic, after the first-level psum reduction along the channel dimension, a second-level psum reduction across an array of PEs can be done. Figure 2.11(a) illustrates a 3×3 PE array and the input data mapping for a 3×3 CONV kernel. The three W bundles of s = 0, 1, 2 are broadcast to the three PE columns, and the three IA bundles of (h, w) = (2, 2), (2, 3), (2, 4) are broadcast to the three PE rows. This mapping allows W and IA reuse by a column and a row of PEs, respectively. Following this mapping, the PEs on the same diagonal lane produce psums of the same address. For example, assume $(r, k) = (0, 0)$ for the W bundles, PE_{11}, PE_{22}, and PE_{33} compute the psums going to the OA address of $(x, y, k) = (h - r, w - s, k) = (2,\ 2,\ 0)$; and similarly, PE_{12} and PE_{23} compute the psums going to the OA address of $(x, y, k) = (h - r, w - s, k) = (2,\ 1,\ 0)$. Therefore, the psums along the diagonal lanes are accumulated.

We name an array of PEs a core. To support the popular 3×3 CONV seen in modern DNN models, the number of columns can be set to 3 (which also sets the number of PEs along a diagonal to 3) to achieve the full utilization. The number

of rows can be set based on the throughput requirement for an application. Going beyond the 3×3 CONV, the diagonal mode reduction is used for general $R \times S$ CONV ($R, S > 1$). If $R, S > 3$, the CONV kernel is divided into R×3 sub-kernels, and then distributed and processed independently on multiple compute cores. A global accumulator merges the psums from the multiple cores to compute the final OA for writeback.

The core-level psum reduction along the pixel dimension cuts the writeback traffic by 2.3 to 3.0×. The two-level reduction resolves the access contention seen in prior work [52–54]. It reduces the writeback traffic to only 2.79 OAs per cycle on average for a core of 7×3 PEs that contain a total of 63 multipliers. The output bandwidth of a channel-first dataflow using the two-level reduction is 22× lower than the channel-last dataflow with an equal number of multipliers.

2.4.3 Support for Pointwise CONV and FC

In our work, we considered the pointwise CONV and FC as special cases of the CONV computation shown in Figure 2.2 with size constraints $R = S = 1$ and $R = S = H = W = 1$, respectively. The size constraints in pointwise CONV and FC eliminate the possibility of pixel-dimension reduction. The interconnection between the PEs and the core reducer is configurable to support not only the diagonal mode, but also provide a row mode to support DNN layers that do not have any pixel-dimension reduction opportunities.

To reuse the same architecture for a pointwise CONV including the same output bandwidth and to achieve a high utilization, the inputs are divided into groups in the channel dimension, and the core is reconfigured to perform reduction along the channel dimension. For example, in Figure 2.11(b), a W bundle is divided into three groups in the channel dimension, and each group is broadcast to a column of PEs; an IA bundle is divided into three groups in the channel dimension, and multicast to

the three PEs in a row. This mapping allows W reuse by the PEs along a column. Following this mapping, the PEs on the same row produce the psums going to the same OA address. Therefore, the core reducer is configured to accumulate the psums from the PEs along the rows.

In processing an FC, W data cannot be reused in batch-1 processing. Similar to the pointwise CONV, the PE array is utilized in channel-dimension reduction. An IA bundle is divided into groups in the channel dimension; a W bundle is divided into groups in both the channel and kernel dimensions. For example, in Figure 2.11(c), a W bundle is divided into three groups in the channel and kernel dimension, and multicast to the three PEs in a column; and an IA bundle is divided into three groups in the channel dimension and broadcast to the three PEs in a column. This mapping allows IA reuse by the PEs along a column. Similar to the pointwise CONV, the PEs on the same row produce the psums going to the same OA address, and the core reducer is configured to accumulate the psums from the PEs along the rows.

2.5 Implementation and Evaluation Results

We present the SNAP system architecture based on the techniques introduced above. The SNAP architecture is prototyped in a 16nm test chip. The chip is measured and evaluated using workloads of different sparsity levels and a pruned ResNet-50 model. The results are summarized and compared with state-of the-art dense and sparse DNN accelerators.

2.5.1 SNAP Architecture Overview

The SNAP high-level architecture is shown in Figure 2.12. It consists of multiple cores, a control module, and a memory module. The control module provides the configuration of the compute cores and coordinates the communication with the external interface. The memory module is composed of multi-banked IA and OA buffers

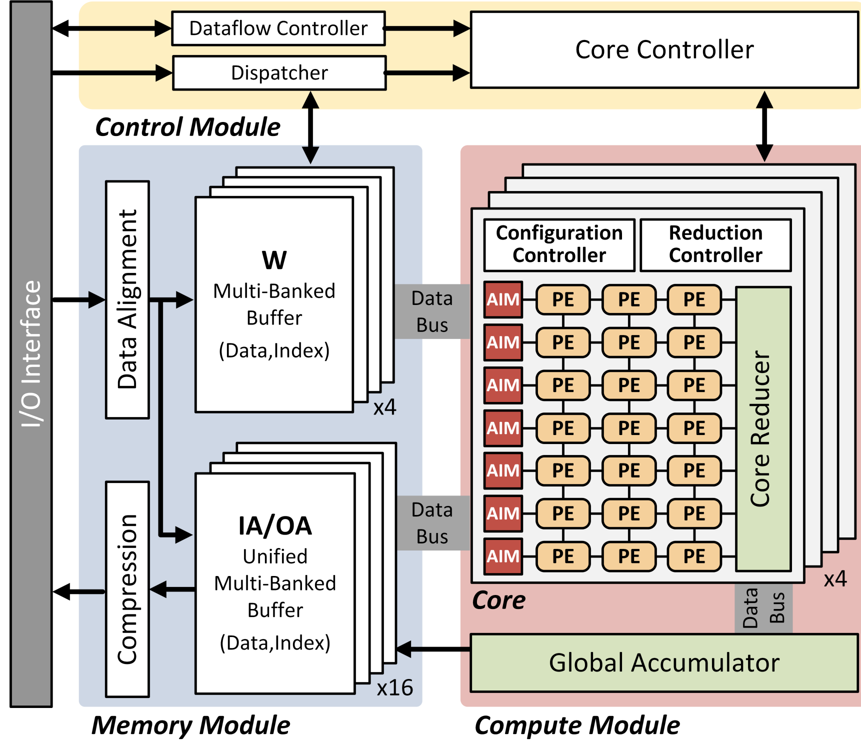


Figure 2.12: SNAP system architecture.

shared between the compute cores, and non-shared W buffers of each compute core. The compressed W and IA data are fetched from off-chip and aligned by bundles. W bundles are stored in each core's W buffers following the system configuration, and IA bundles are stored in the IA buffers. The output OAs are compressed before writeback to the external memory.

Following the high-level architecture, we designed a SNAP test chip that is made of 4 cores, and each core is implemented in a 7×3 PE array. Within a core, 7 AIM units are shared in a time-multiplexed fashion between the 3 PEs in a row. Each PE is implemented with 3 multipliers and a sequence decoder. The PEs output psums, which are accumulated by the core reducer, and a global reducer is used to further accumulate psums before the final writeback.

The SNAP test chip provides a total of 252 multipliers of 16-bit fixed-point precision and a total of 280.6KB SRAM. Note that a 8-bit design would work equally

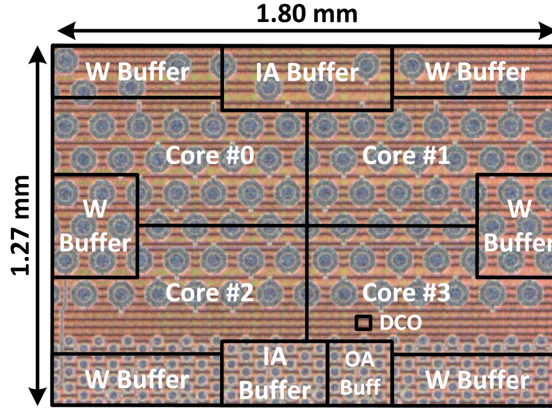


Figure 2.13: Microphoto of the 16nm SNAP test chip.

as well to demonstrate the architectural advantages and show an even better performance and energy efficiency. The only difference is that the overhead of index matching, measured as a fraction of the compute core, increases from 12.5% (in a 16-bit design) to 17% (in a 8-bit design). The test chip is implemented using a 16nm CMOS process technology with a core area of 2.3mm^2. Figure 2.13 shows the chip microphoto.

2.5.2 Performance Analysis

In our evaluations, a 16-bit fixed-point multiply and accumulate (MAC) is counted as 2 operations (OPs). Each PE computes at most 3 MACs or 6 OPs every clock cycle. For the evaluations, we used both synthetic sparse workloads and real workloads of pruned DNN models, including AlexNet, VGG-16, and ResNet-50, pruned with less than 0.5% accuracy loss using the technique from [26], to measure performance, power consumption, and effectual energy efficiency (taking into account the input IA/W density).

The measured chip power consumption is shown in Figure 2.14(a). At each operating frequency, the power is reported at the lowest supply voltage. At 0.55V and 260MHz, the test chip achieves the peak effectual energy efficiency of 3.61TOPS/W

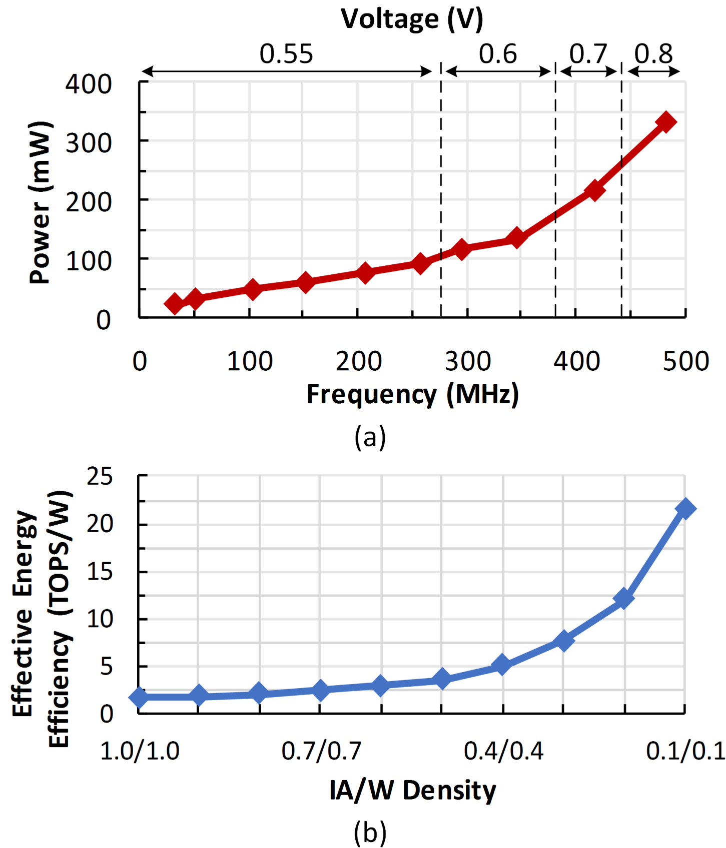


Figure 2.14: (a) Measured power consumption at different operating frequencies and the optimal supply voltages, and (b) measured effectual energy efficiency for synthetic sparse workloads at different data density levels.

for a sparse ResNet-50 model with an average IA/W density of 0.38/0.52. The chip achieves the highest inference throughput of 90.98fps for the same sparse ResNet-50 model at 0.8V, 480MHz, consuming 348mW.

The measured effectual energy efficiency of the SNAP test chip running synthetic sparse workloads of different data density levels is shown in Figure 2.14(b). In previous work, SCNN [52] reports the results at IA/W density of 0.2/0.2 and 0.1/0.1, and Sticker [54] reports the results at IA/W density of 0.15/0.15 and 0.05/0.05. We used IA/W density of 0.1/0.1 to approximately match the previous work so we can fairly compare the designs for energy efficiency. For dense (1.0/1.0), medium (0.4/0.4), and extremely sparse (0.1/0.1) IA/W data density levels, a peak effectual energy efficiency

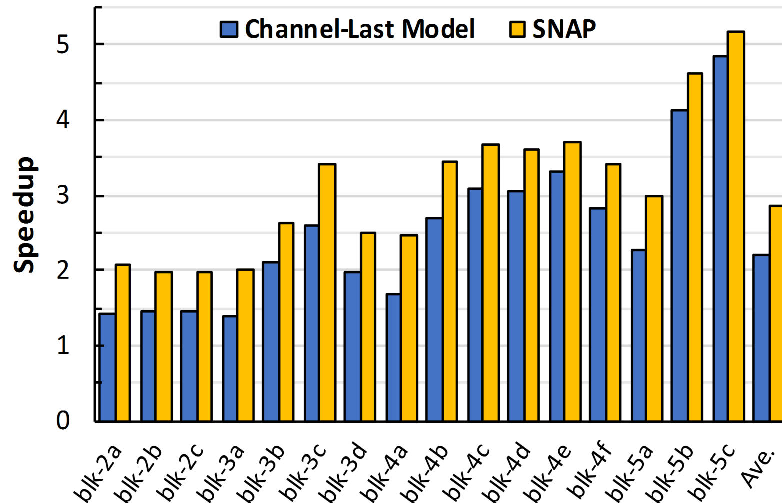


Figure 2.15: Processing speedup by SNAP (a channel-first dataflow) and a channel-last dataflow over a dense accelerator baseline running the residual blocks of a sparse ResNet-50 model.

of 1.67, 5.06 and 21.55TOPS/W is achieved, respectively.

We benchmark the measured throughput of the SNAP architecture and a simulated channel-last dataflow architecture over a dense accelerator baseline running a sparse ResNet-50 model. In Figure 2.15, we show the speedup in processing each residual block. For a fair comparison, the dense accelerator baseline is constructed to be the same as SNAP, but with uncompressed, dense inputs. Overall, the channel-last dataflow demonstrates an average speedup of 2.20× over the dense baseline; and the SNAP design obtains an average speedup of 2.87× over the dense baseline, which is 30.3% better over the channel-last dataflow.

Although SNAP, a channel-first dataflow, generally provides a higher performance than a channel-last dataflow, the performance gap is more visible at the shallow and middle layers, but less visible for deep layers, as shown in Figure 2.15. For instance, SNAP has a 46.2% better performance than the channel-last dataflow for the first residual block blk-2a, but only 7% better performance for the last residual block blk-5c. The difference is attributed to two factors: 1) the deep layers are generally sparser after pruning, resulting in less access contention and better performance for

the channel-last dataflow; and 2) extremely sparse workloads often come with imbalanced zero distribution, causing challenges to SNAP's index matching and psum reduction.

The imbalanced zero distribution, or workload imbalance, is a design challenge for both the SNAP front-end and back-end. At the front-end, the workload imbalance causes PEs to receive varied number of W-IA pairs for computation, and the overall performance is limited by the PE with the heaviest workload. This problem may be resolved by implementing a more aggressive prefetching scheme that provides a larger work chunk for PEs to prevent them from stalling. At the back-end, the workload imbalance causes each PE to have less channel-dimension reduction opportunity before writeback to the output buffer, resulting in a higher writeback bandwidth. Additionally, the PEs on the same core-level reduction lane may have varied reduction progress, requiring the faster PEs to be stalled to wait for the slowest one. The back-end problem may be mitigated by implementing a larger OA psum RF to store more psums in a PE, and implementing the OA output buffers using two-port SRAMs or multi-banked SRAMs to provide a higher accumulation bandwidth.

2.5.3 Comparison Against State-of-the-art Works

Compared with state-of-the-art inference accelerators that support sparsity at data-level or for power-saving shown in Table 2.2, SNAP exploits sparsity in both compressed W and IA data for both CONV and FC layers. SNAP employs a 16-bit fixed-point precision for data storage and computation. SNAP has a comparable number of multipliers as many previous silicon designs. Energy efficiency is reported for the effective efficiency evaluated on both synthetic workloads and real sparse network workloads.

To provide a fair comparison, we also present the synthesis results of SNAP-65nm-8b, the same SNAP design with 8b storage and computation in a 65nm GP process.

Table 2.2: Comparison With Prior Works

	This Work	Sticker [54]	Eyeriss-v2 [55]	SCNN [52]	Envision [59]	Eyeriss-v1 [48]
Sparsity Support	Data-Level	Data-Level	Data-Level	Data-Level	Power-Saving	Power-Saving
Layer Support	CONV+FC	CONV+FC	CONV+FC	CONV	CONV+FC	CONV
Technology	16nm	65nm	65nm	16nm	28nm	65nm
Silicon	Yes	Yes	No	No	Yes	Yes
Core Area	$2.3mm^2$¶	$7.8mm^2$	2695k gates (NAND-2)	$7.9mm^2$	$1.87mm^2$	$12.25mm^2$
Num. of MULTs	252	256	384	1024	1024/512/256	168
Data Width	16b	8b	8b	16b	4/8/16b	16b
On-Chip SRAM	280.6KB	170KB	246KB	1200KB	144KB	108KB
Supply Voltage	0.55–0.80V	0.67–1.0V	N/A	N/A	0.55–1.1V	0.82–1.17V
Frequency	33–480MHz	20–200MHz	200MHz	1000MHz	50–200MHz	100–250MHz
Power	16.3–364mW	20.5–248.4mW	N/A	N/A	7.5–300mW	235–332mW
Peak Efficiency† (TOPS/W)	D: 1.67 (16b) M: 5.06 (16b) S: 21.55 (16b)	D: 0.5 (8b) M: 2.5 (8b) S: 30 (8b)	D: 0.25 (8b) S: N/A	N/A	0.53 (16b)	0.31 (16b)
Sparse AlexNet‡ (TOPS/W)	3.86 (16b)	2.82 (8b)	0.96 (8b)	N/A	0.8-3.8§	0.17 (16b)
Sparse VGG-16‡ (TOPS/W)	3.79 (16b)	N/A	N/A	N/A	2.0§	0.09 (16b)
Sparse ResNet-50‡ (TOPS/W)	3.61 (16b)	N/A	N/A	N/A	N/A	N/A

*One multiply-and-accumulate (MAC) computation is counted as 2 operations (OPs). Data width of the OPs is labeled in bracket.
†Peak effectual energy efficiency in TOPS/W evaluated on synthetic sparse workloads at 1.0/1.0 (D), 0.4/0.4 (M), and 0.1/0.1 (S) IA/W density levels.
‡Effectual energy efficiency in TOPS/W for sparse networks pruned within 0.5% accuracy loss.
§Data width not specified.
¶Post-shrinkage area. (pre-shrinkage area: $2.4mm^2$ [38])

SNAP-65nm-8b was synthesized at 250MHz with the SRAM modules re-generated to support 8b processing. It is estimated to have an area of 9.32mm^2 and consume 500mW. For a sparse AlexNet, SNAP-65nm-8b is estimated to achieve an effective energy efficiency of 0.74TOPS/W running at 250MHz at a nominal voltage of 1.0V.

Dense and Sparse Workloads: For high and medium density workloads, SNAP achieves a higher effectual energy efficiency than all the other sparse accelerators. More specifically, the channel-last accelerators, including SCNN and Sticker, suffer from memory contention and compute stalls, resulting in a lower performance. SNAP also benefits from the large channel-dimension reduction opportunity and a low workload imbalance to achieve a high compute utilization and better efficiency. The NoC in Eyeriss-v2 consumes extra power and latency for conventional DNN workloads. Compared to Eyeriss-v2, SNAP benefits from a larger search depth and is specifically optimized for CONV ($R\times S$ and 1×1) and FC layers. For extremely sparse workloads, Sticker [54] reports better effectual energy efficiency than SNAP, but uses 8-bit storage and computation while SNAP uses 16-bit storage and computation.

In sparse workload evaluations, SNAP outperforms all the other works. Compared to sparse accelerators, Sticker (8b) [54] and Eyeriss-v2 (8b) [55], SNAP achieves up to 3.34× and 6.68× better effectual energy efficiency running synthetic workloads, respectively. For sparse AlexNet, SNAP achieves 1.37× and 4.02× better effectual energy efficiency than Sticker (8b) and Eyeriss-v2 (8b), respectively.

Index Matching Search Depth: Among all the references, Eyeriss-v2 is the closest to SNAP in terms of architectural design. The difference between SNAP and Eyeriss-v2 is mainly attributed to two factors. First is the difference in search granularity in index matching. SNAP adopts a more coarse-grained compression format. During the discover stage, the AIM searches in the first 32 nonzero entries of the compressed W data. On the other hand, Eyeriss-v2 uses the compressed sparse column (CSC) format. During the discover stage, Eyeriss-v2 first identifies

IA's channel index, then searches for nonzero W data only across all k indices. Due to a smaller search depth, Eyeriss-v2's index matching mechanism is more likely to encounter fragmentation of nonzero W data, resulting in an insufficient number of W-IA pairs sent to the MAC array and possibly a lower compute utilization. The second difference is that SNAP's dataflow is specifically optimized for CONV and FC, whereas Eyeriss-v2 implements a fully flexible NoC for routing and switching. The NoC may be a burden on performance and power consumption.

Load Balancing Consideration: Load imbalance can be caused by front-end workload distribution or back-end writeback or both. The channel-last accelerators like Sticker and SCNN are not affected by the front-end workload imbalance seen in channel-first accelerators like SNAP and Eyeriss-v2. The varying number of W-IA pairs assigned to different PEs affects the compute utilization of each PE. With a large enough search depth, SNAP, a channel-first accelerator, can minimize the front-end workload imbalance. Eyeriss-v2, another channel-first accelerator, used a similar approach, except that its search depth is smaller than SNAP.

The channel-last accelerators suffer from the back-end workload imbalance when there are memory access contentions between the accumulation buffers and the multipliers, especially for medium to high density workloads. The memory contention affects the number of cycles needed to complete a chunk of W and IA data and causes progress differences between PEs, resulting in the faster PEs stalling for the slowest one. This imbalance situation is worse for higher density workloads where contention is more frequent. A channel-first accelerator like SNAP effectively mitigates the back-end workload imbalance by streamlined, maximal OA psum reduction before writeback.

2.6 Summary

We present the Sparse Neural Acceleration Processor (SNAP) that exploits unstructured sparsity in sparse DNNs for efficient inference acceleration. SNAP adopts a new channel-first dataflow with channel index matching as the front-end and a two-level psum reduction backend for sparse DNN processing.

At the front-end, channel index matching is done by efficient associative index matching (AIM) units and sequence decoders to discover valid W-IA pairs for computation. It results in an average compute utilization of 75% while limiting the area overhead to only 12.5% of the compute core. At the back-end, the combination of PE-level psum reduction along the channel dimension and core-level psum reduction along the pixel dimension eliminates writeback access contention at the output buffer and reduces the psum writeback traffic by 22× compared to the previous sparse accelerator designs. The core-level psum reduction is configurable to support general $R \times S$ CONV, pointwise CONV, and FC layers.

A SNAP test chip is designed using 252 16-bit multipliers organized in 4 cores of 7×3 PEs. The chip is measured to achieve an effective energy efficiency of up to 21.55TOPS/W running synthetic sparse workloads and 3.61TOPS/W running a pruned ResNet-50. Compared to the state-of-the-art dense and sparse accelerators, SNAP offers competitive performance and energy efficiency by maintaining a high compute utilization and a low writeback data traffic.

CHAPTER III

Point-X: Spatial-Locality-Aware Accelerator Architecture for Graph-Based Point-Cloud Neural Networks

3D deep learning has attracted increasing attention in recent years due to its wide applications in the 3D space, including indoor navigation, object classification, scene segmentation, shape synthesis and modeling [15]. Among all 3D representations, point cloud has gained popularity since it shows an accurate representation of the real world and can be acquired directly as the raw output from most 3D data acquisition devices like light detection and ranging (LiDAR) and infrared (IR) sensors [14, 16, 17, 63]. The raw point clouds undergo common preprocessing steps, including background filtering, noise removal, and region of interest (ROI) identification, and the preprocessed ROI frames are then ready for the point-cloud processing.

After the wide success of deep neural network (DNN) and convolutional neural network (CNN) on 2D image applications [7, 8, 42, 43], researchers have worked on converting the insights from CNNs to the point clouds in the 3D space. The first attempts processed point clouds indirectly using an intermediate representation, i.e., multi-view [64–66] or volumetric [67–69]. These methods either project point clouds into several 2D images of different angles or convert them into voxels in a

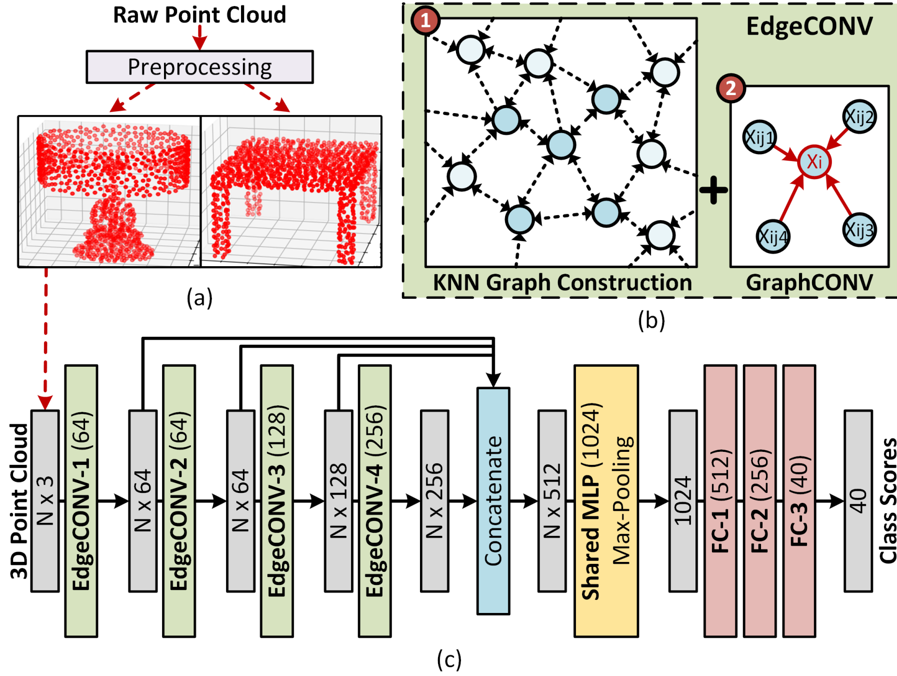


Figure 3.1: Illustration of (a) point cloud recognition pipeline, (b) EdgeCONV layer divided into KNN graph construction and GraphCONV on vertex point i, and (c) DGCNN's network architecture for 3D object classification [4].

3D grid, before applying well-established 2D or 3D CNNs to accomplish indirect point-cloud processing. However, these approaches were unable to capture the fine details and textures due to the data truncation during representation conversion. PointNet [70] proposed a point-based network to process point clouds directly without any projection or voxelization. However, PointNet only considers the global shape structure and not the relations between the points, thus it is unable to capture finer details, limiting its performance [4, 71].

To overcome the limitations, recent works propose to extract local neighborhood information by graph-based methods [4, 72, 73], and integrate them into the global shape structure to improve the performance over the PointNet model. In particular, DGCNN [4] uses a graph-based operator called edge convolution (EdgeCONV) to integrate both local and global features. Figure 3.1 shows the point cloud recognition pipeline using the DGCNN architecture and illustrates the principles of EdgeCONV.

An EdgeCONV first uncovers the spatial relationship of points by constructing a K-nearest neighbor (KNN) graph of the input point cloud. Following the KNN graph edges, a graph convolution (GraphCONV) aggregates the local neighborhood features and the global feature of every vertex point to produce the output feature. The promising DGCNN results make EdgeCONV a widely adopted operator [74–77].

Variants of EdgeCONV have emerged and are employed widely in recent graph-based point-cloud DNN works [78–80]. These variants share EdgeCONV's computation pattern which consists of 1) graph construction to uncover the spatial relationship between the seemingly independent data points, followed by 2) convolution on the constructed graph to extract features for recognition. As such, the exploration and findings on EdgeCONV are applicable to these variants.

EdgeCONV computation cannot be supported efficiently by existing computing solutions. In computing EdgeCONV, a vertex's neighbors may be randomly dispersed in the system memory. Conventional general-purpose SIMD architectures, i.e., CPUs and GPUs, have to fetch scattered graph vertices for vector computation, resulting in a low compute utilization and a low efficiency. Existing DNN/CNN accelerators [48, 57, 59, 81] are incapable of performing such computation because they are purposely designed for sequential memory access and regular-structured data, i.e., 2D image or 1D sequence. Existing graph processing accelerators [82, 83] are optimized for irregular data accesses and cannot fully exploit the data reuse or provide the computational parallelism needed in graph-based networks. Between the two ends of the spectrum, accelerators for graph convolutional neural networks (GNNs/GCNs) [84–90], handle both regular and irregular computation in GCNs. Different from the common GNNs where the graph structures are known in advance, EdgeCONV operates on dynamic graphs that are constructed in runtime. Most previous works [84–88] focused on static graphs, and did not fully support EdgeCONV due to the lack of runtime graph construction. Recently, [89, 90] extended the support to dynamic graphs.

However, they did not consider the community structures of point clouds, and as a result, they failed to capture the best efficiency for graph-based point-cloud DNNs.

To design a practical architecture for graph-based point-cloud processing, three objectives need to be addressed:

1. ***Fetch efficiency***: to deliver a maximal number of neighbor points to compute under a limited transfer bandwidth.

2. ***Computation efficiency***: to provide high compute parallelism and utilization for irregular point cloud workloads.

3. ***Flexibility***: to support diverse computation types in graph-based point-cloud networks.

We present Point-X, a spatial-locality-aware accelerator architecture for efficient graph-based point-cloud processing. Point-X extracts and leverages the spatial locality in the input point cloud to increase computational parallelism and reduce communication overhead, enabling higher fetch and computation efficiency. The main contributions are:

- A speculative breadth-first search (SBFS) graph traversal method is proposed to extract the spatial locality in the input point cloud. It achieves up to 9.2× faster execution over a conventional breadth-first search (BFS) graph traversal. A spatial-locality-aware clustering based on SBFS traversal is used to distribute input points into compute tiles (CTiles) for efficient processing.

- A lightweight chain network-on-chip (NoC) architecture is designed to leverage the spatial locality to effectively reduce the inter-tile traffic and its latency. The chain NoC design demonstrates a 3.2× shorter latency than a mesh NoC while incurring much lower area and energy overheads.

- A flexible CTile and a multi-mode dataflow are designed to support all the common operations in graph-based point-cloud networks, including EdgeCONV, shared multi-layer perceptrons (MLPs), and fully-connected (FC) layers.

A Point-X design is synthesized in a 28nm technology. The design is estimated to occupy an area of 6.8 mm^2 and operate at a 1.0 GHz clock frequency. Point-X demonstrates an average speedup of 7.7× for GraphCONV over a baseline accelerator, and up to 12.1× higher energy efficiency in EdgeCONV over existing accelerators. Point-X achieves an end-to-end throughput of 1307.1 inference/s (Inf./s) and an energy efficiency of 604.5 inference/J (Inf./J) in running DGCNN for 3D object classification [4]. Compared to a general-purpose GPU and CPU, Point-X demonstrates a throughput improvement of 4.5×, 129.7×, respectively, and an energy efficiency improvement of 342.9× and 3160.9×, respectively.

The rest of the chapter is organized as follows. Section 3.1 introduces the EdgeCONV computation and two baseline computation models. Section 3.2 presents the SLA clustering for extracting spatial locality from the input point cloud and our proposed SBFS graph traversal to speedup execution. In Section 3.3, we describe the chain NoC design that exploits data locality to reduce design complexity for data exchange, and in Section 3.4, we describe our versatile CTile design to support common operations used in point-cloud networks. In Section 3.5, we present the Point-X architecture and its multi-mode dataflow to support different operations, then discuss the workload partitioning scheme to support larger point clouds. Section 3.6 presents the evaluation results of Point-X and compares it to the baseline designs, state-of-the-art accelerator works, and general-purpose processors. Finally, Section 3.8 summarizes the contributions of this work.

3.1 Background

An EdgeCONV layer in a point-cloud DNN projects the N points from an input C-dimensional space into an output F-dimensional space. The input point cloud can be described as $\boldsymbol{X} = \{\boldsymbol{x}_0, ..., \boldsymbol{x}_{N-1}\}$, where $\boldsymbol{x}_i \in \mathbb{R}^C$ is the feature of i-th point, and C is the dimensionality of the feature space. For instance, the first EdgeCONV layer receives an input of $C = 3$ which represent the (x, y, z) coordinates in a 3D space. As illustrated in Figure 3.1(b), an EdgeCONV operation is divided into two steps: 1) KNN graph construction and 2) GraphCONV on the KNN graph.

3.1.1 Edge Convolution Computation

KNN Graph Construction

Given an input point cloud $\boldsymbol{X}$, a directed graph is constructed: $G_{(\boldsymbol{X},K)} = (V, E)$ with self-loops, where $V = \{0, \ldots, N-1\}$, $E = \{(i, j_{i1}), \ldots, (i, j_{iK}), \ldots\}$ for $i, j_{ik} \in [0, N-1]$, and j_{ik} represents the k-th nearest neighbor to the vertex point i. The k-th nearest point is found based on the Euclidean distance to vertex point i as described in Eq. (3.1).

$$j_{ik} = \underset{j \in [0,N-1],k}{\arg\min}\ D(i, j) = \underset{j \in [0,N-1],k}{\arg\min}\ \|\boldsymbol{x}_i - \boldsymbol{x}_j\|^2 . \tag{3.1}$$

Graph Convolution

In GraphCONV, the vertex point i is convolved with its K neighbors $j_{i1}, \ldots, j_{iK}$. In DGCNN [4], GraphCONV is defined as the combination of the global shape structure captured by the vertex point's feature $\boldsymbol{x}_i$ and local neighborhood structures captured by the differences between the vertex point and its neighbors $(\boldsymbol{x}_{j_{ik}} - \boldsymbol{x}_i)$. With learnable weights $\boldsymbol{\phi}_0, \ldots, \boldsymbol{\phi}_{F-1}, \boldsymbol{\theta}_0, \ldots, \boldsymbol{\theta}_{F-1} \in \mathbb{R}^C$, the f-th output feature of

GraphCONV for the point i is defined by

$$\boldsymbol{x}'_{if} = \max_{1 \leq k \leq K} \text{ReLU}\left(\boldsymbol{\phi}_f \cdot \boldsymbol{x}_i + \boldsymbol{\theta}_f \cdot (\boldsymbol{x}_{j_{ik}} - \boldsymbol{x}_i)\right), \tag{3.2}$$

where $\boldsymbol{\phi}$ weights are applied to the vertex point to compute the global feature partial sum (Fpsum), and $\boldsymbol{\theta}$ weights are applied to the difference to a neighbor point to compute a local fpsum. A ReLU operation is applied to the sum of the global fpsum and the local fpsum to generate an output fpsum. K output fpsums are aggregated by a max operation to produce an output feature.

3.1.2 Computation Models and Bottlenecks

KNN graph construction is embarrassingly parallel and can be supported easily by SIMD and conventional spatial architectures. However, GraphCONV cannot be parallelized easily for two reasons: 1) there is no knowledge of a vertex point's neighbors prior to the KNN graph construction, limiting scheduling opportunity; and 2) the neighbors are randomly scattered in memory without a fixed pattern, limiting prefetch opportunity. These two factors result in highly inefficient computation when operating with the practical limitations of memory bandwidth, prefetch and scheduling capability. These limitations prohibit the parallel computation on multiple neighbors.

A spatial architecture utilizes multiple compute tiles (CTiles) for parallel computation. To parallelize KNN graph construction and GraphCONV, one possibility is to assign the vertex points to the CTiles to allow the outputs to be computed independently. Under this setup, we present two EdgeCONV computation models, a query-based model and an exchange-based model, which differ in the fetch mechanisms and computation dataflow for GraphCONV computation. Figure 3.2 presents the two computation models for EdgeCONV operation. In both models, a multi-

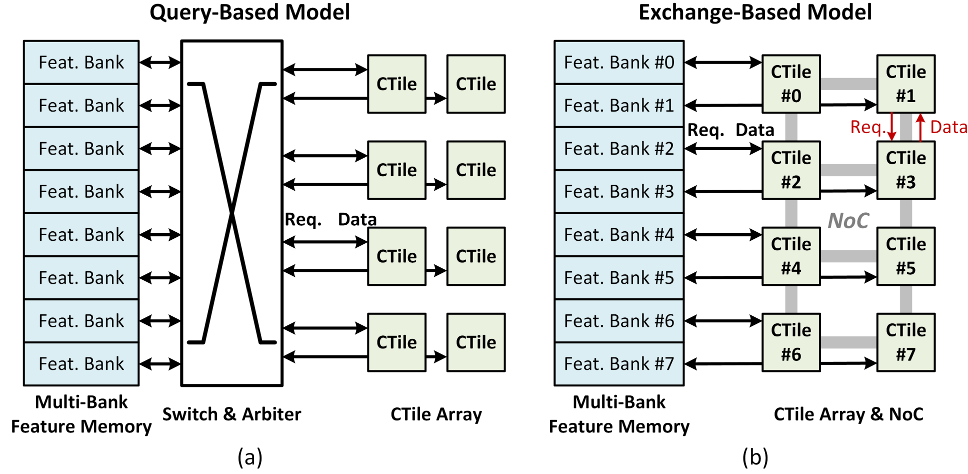


Figure 3.2: EdgeCONV computation models: (a) query-based model, and (b) exchange-based model.

banked system memory is used to hold the point features and to support access by multiple CTiles.

Query-Based Model: The query-based model follows a direct parallelization of Eq. (3.2). In this model, each CTile operates independently and sees the memory as a single shared memory. A CTile is assigned a set of vertex points. To access neighbor points that are not available locally, a CTile sends requests to the centralized memory controller. The controller arbitrates the requests from all CTiles. Although the query-based model only requires a simple dataflow, the all-to-all switch and arbiter design can be complex with poor scalability. Furthermore, frequent memory access conflicts occur when multiple CTiles request access to the same memory bank, resulting in a low fetch efficiency.

Fpsum Reuse: Different CTiles in the query-based model may be requesting and performing dot-product (DP) on overlapping neighbor points, causing duplicated computation during GraphCONV operation. We reformulate Eq. (3.2) to reduce computation redundancy and increase fpsum reuse. Starting from the original form in Eq. (3.2), we separate the DP of the vertex point from the neighbor points; and

Table 3.1: GraphCONV Computation Comparison with F kernels, N points, and K neighbors per point

Computation	Original Form, Eq. (3.2)	Reuse Form, Eq. (3.3)
Dot-Product ($\cdot$)	$F \times N + F \times N \times K$	$2 \times F \times N$
Max-Pool (max)	$F \times N \times K$	$F \times N \times K$
Summation (+)	$F \times N \times K$	$F \times N$
ReLU	$F \times N \times K$	$F \times N$

reorder the max and ReLU operations. The reformulated GraphCONV is written as

$$\boldsymbol{x}'_{if} = \text{ReLU}\left(\max_{1 \leq k \leq K}\left(\boldsymbol{\theta}_f \cdot \boldsymbol{x}_{j_{i_k}}\right) + (\boldsymbol{\phi}_f - \boldsymbol{\theta}_f) \cdot \boldsymbol{x}_i\right). \tag{3.3}$$

Each point can be a vertex point and a neighbor point of other vertex points. Following Eq. (3.3), the **vertex fpsums** (DP of the point with $(\boldsymbol{\phi} - \boldsymbol{\theta})$ weights) and the **neighbor fpsums** (DP of the point with $\boldsymbol{\theta}$ weights) can be computed once, cached and reused to prevent redundant DPs. Table 3.1 shows the comparison of computation of the two forms of GraphCONV defined by Eq. (3.2) and Eq. (3.3). Using the optimized form, the number of DP ($\cdot$), summation (+), and ReLU operations are reduced by a factor of $(K + 1)/2$ or K.

Exchange-Based Model: Following Eq. (3.3), the exchange-based model allows CTiles to exchange neighbor fpsums through an NoC. In this model, each CTile is associated to a memory bank for accessing points locally to compute vertex fpsums and neighbor fpsums. To access a neighbor fpsum not available locally, a CTile requests from the CTile that holds the neighbor fpsum and receives the fpsum through the NoC. The exchange-based model cuts all redundant neighbor fpsum computation. However, a CTile may experience a longer transfer latency when accessing points stored in foreign CTiles far away from it. Furthermore, the feature exchange traffic may overwhelm the NoC bandwidth, resulting in an even lower fetch efficiency.

3.2 Spatial-Locality-Aware Clustering

Prior works [87, 91–97] exploited the community structure of real-world graphs to improve locality for graph applications. Similarly, KNN graphs of real-world point clouds exhibit community structure where groups of points close in space form densely connected subgraphs. Starting from an exchange-based model, a spatial-locality-aware (SLA) architecture extracts and exploits the spatial locality in point clouds to improve fetch and computation efficiencies.

More specifically, we categorize spatial locality into two types: fine-grained and coarse-grained. The fine-grained spatial locality refers to the case that the neighbor points are accessed from the *same* CTile (cluster) in computing GraphCONV of a vertex point. The coarse-grained spatial locality refers to the case that the neighbor points are accessed from *nearby* CTiles (clusters). The fine-grained spatial locality maximizes computational parallelism of a CTile by having most neighbor points in its local memory bank. The coarse-grained spatial locality helps reduce the data movement between CTiles by having the foreign neighbor points needed by a CTile located in nearby CTiles. We present a clustering technique to extract both the fine-grained and coarse-grained spatial locality given a point cloud's KNN graph.

3.2.1 Graph Traversal for Spatial Locality

An example of a point cloud's KNN graph is shown in Figure 3.3(a). The nodes are numbered during point cloud acquisition and preprocessing, and the graph is represented by an adjacency matrix. In the KNN graph, a directed edge from node j to node i indicates that the vertex point i is connected to its neighbor point j, and is represented by the entry (i, j) in the adjacency matrix. Note that, following Eq. (3.1), each point is also neighbor to itself. In Figure 3.3(a), the points of consecutive point indices are grouped into three clusters as illustrated on the adjacency matrix. An edge in the adjacency matrix is a local edge if both the vertex and the neighbor point

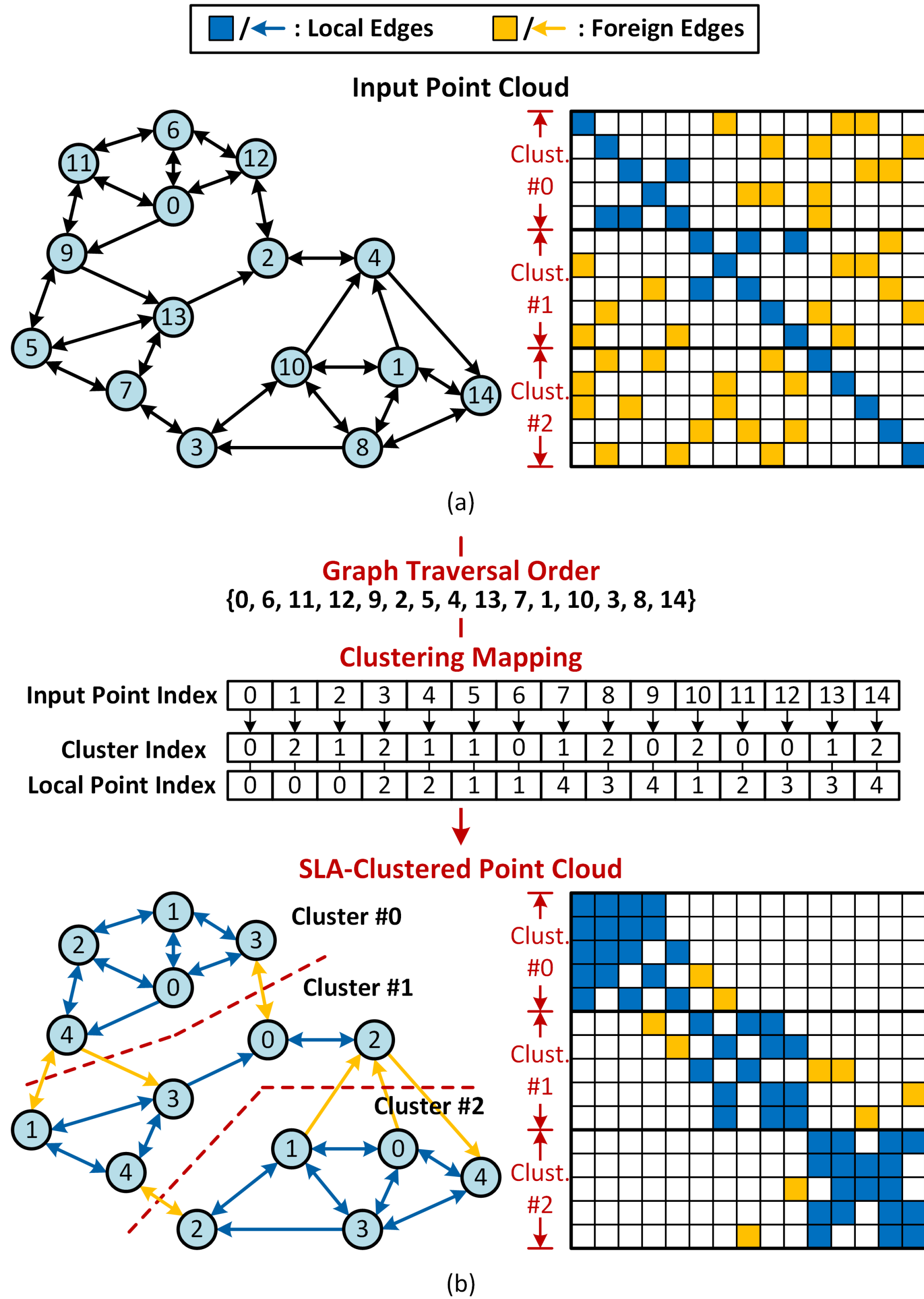


Figure 3.3: (a) KNN graph of input point cloud and its adjacency matrix representation; (b) the KNN graph is traversed and the points clustered following traversal order; the clustered KNN graph and its adjacency matrix are shown.

belong to the same cluster or a foreign edge otherwise.

In an SLA architecture, each cluster of points is assigned to a CTile. A local edge indicates local data access, and a foreign edge indicates access from a foreign cluster. The number of foreign edges suggest the amount of inter-tile data movement and is an indication of the clustering efficiency.

Our SLA clustering aims to maximize both fine-grained and coarse-grained spatial locality by taking advantage of the spatial relationship of the points using graph traversals. The steps and result of SLA clustering are presented in Figure 3.3(b). Following the graph traversal order, the SLA clustering assigns every point into a cluster by mapping its point index to a pair of {cluster index, local point index}. Each cluster comprises a densely connected subgraph with improved fine-grained spatial locality, which can be visualized by the more local edges and fewer foreign edges in each cluster. As the scattered foreign edges are brought closer to the matrix diagonal, the coarse-grained spatial locality is also improved.

Graph traversal methods help uncover the spatial relationship of nodes in a graph [91, 93, 94]. To evaluate the capability of different graph traversal methods, we define two metrics: graph edge ratio to measure the fine-grained spatial locality, and graph edge length to measure the coarse-grained spatial locality. For every vertex point in a cluster, the **graph edge ratio** is defined as the percentage of local edges over all edges. For every foreign edge, the **graph edge length** measures the distance (i.e., number of clusters away) between the vertex point and its foreign neighbor point. For instance, in Figure 3.3(a), the foreign edge from point 10 in cluster 2 to point 4 in cluster 0 has a graph edge length of 2.

The graph edge ratio and graph edge length are plotted in Figure 3.4 for common graph traversal methods, breadth-first search (BFS), depth-first search (DFS), bounded DFS (BDFS) [91] with a depth limit, evaluated using point clouds ranging from 1k to 10k points and compared to the input KNN graph baseline. As shown in

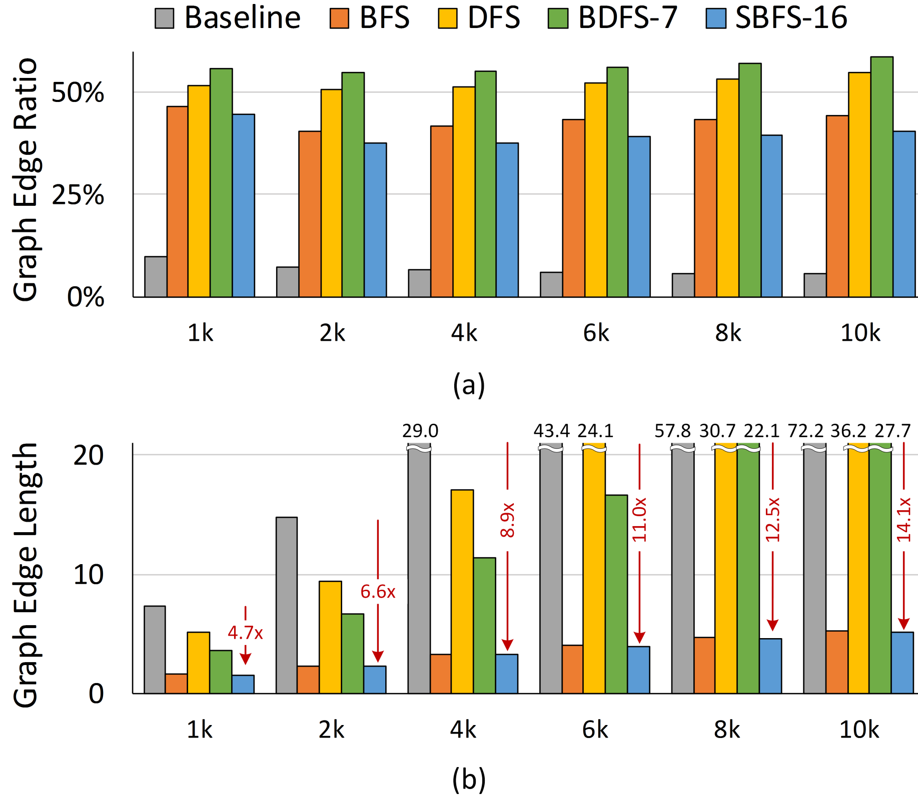


Figure 3.4: Clustering performance on KNN graphs: (a) graph edge ratio, and (b) graph edge length, using different graph traversal methods on 1k to 10k point clouds.

Figure 3.4(a), for 1k points, all traversal methods result in a significant graph edge ratio improvement of 4.8 to 5.8× compared to the baseline. In particular, BDFS provides the highest graph edge ratio of 56%. With increased point sizes, the baseline shows a decrease in graph edge ratio, i.e., only 5.5% at 10k points, while the traversal methods maintain similar graph edge ratio results, leading to an even larger improvement of 7.9 to 10.5× at 10k points. All traversal methods also outperformed the baseline in graph edge length as shown in Figure 3.4(b). Compared to the baseline at 1k points, BFS, DFS, and BDFS reduce the graph edge length by 4.7×, 1.4×, and 2×, respectively. For increased point sizes, the baseline and depth-related methods show significant increases in graph edge lengths, whereas BFS shows limited increase and demonstrates larger graph edge length reduction over the baseline, i.e., 14× at 10k points.

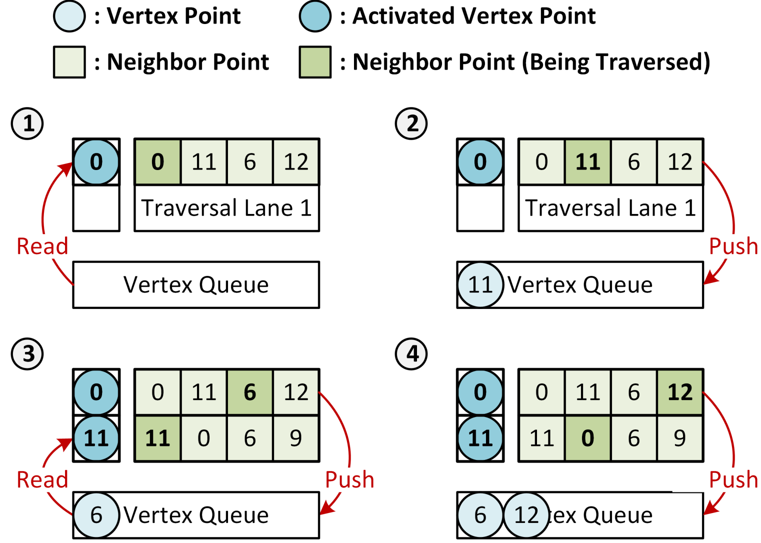


Figure 3.5: Illustration of the SBFS algorithm with 2 traversal lanes.

Combining both fine-grained and coarse-grained spatial locality, BFS stands out as the most promising method, especially notable for its scalable graph edge length.

3.2.2 Speculative Breadth-First Search (SBFS) Traversal

To ensure correctness, BFS only allows traversing neighbors of a vertex point at a time. This requirement hinders the parallelization of BFS traversal execution [98]. We propose the speculative BFS (SBFS) algorithm to approximate BFS traversal and parallelize execution by speculating the traversal order. The speculation is plausible thanks to the community structure in a KNN graph. If two points are connected, they are also likely to share neighbors. For instance, in Figure 3.3(a), point 0 and point 11 are connected and share common neighbors of points 0, 11, and 6. Traversing these two points in parallel does not affect the traversal order.

Figure 3.5 illustrates the first four iterations of traversing the graph in Figure 3.3(a) using an SBFS-2 algorithm, where 2 indicates two active lanes for vertex traversal. Untraversed vertices are read from the vertex queue and its neighbor points are loaded to a traversal lane for traversal. Here is a step-by-step rundown of the process:

1. The root point 0 is read from the vertex queue and its neighbors 0, 11, 6, 12 are loaded into traversal lane 0.

2. From traversal lane 0, neighbor point 11 is traversed. Since point 11 is not visited yet, it is pushed into the vertex queue.

3. From traversal lane 0, neighbor point 6 is traversed and pushed into the vertex queue. In the meantime, vertex point 11 is read from the vertex queue and its neighbors 11, 0, 6, 9 are loaded into traversal lane 1.

4. From traversal lane 0, point 12 is traversed and pushed into the vertex queue, and traversal lane 0 is cleared. From traversal lane 1, neighbor point 0 is traversed. Since point 0 is already visited, it is skipped.

By traversing the neighbors of multiple neighboring vertices at a time, the execution time can be largely reduced with only minor changes to the graph traversal order. The clustering metrics of SBFS-16 are plotted in Figure 3.4. SBFS-16 produces a similar graph edge ratio and graph edge length to BFS across 1k to 10k points. The speculation produces a close approximation to BFS in terms of clustering quality.

3.2.3 SLA Clustering Module Implementation

As Figure 3.6 shows, the SLA clustering module is realized by two submodules: an SBFS traversal module for spatial locality extraction and a cluster graph compiler module for converting a global graph into local subgraphs for each cluster and setting up the inter-cluster connections.

The KNN index buffer stores the constructed KNN graph in the adjacency list format. The SBFS module reads an untraversed vertex point from the vertex queue, and the neighbor indices are requested from the KNN buffer for loading to the traversal lanes. The SBFS module consists of N traversal lanes operating in parallel. Consecutively traversed neighbor points are recorded in the cluster maptable. The cluster

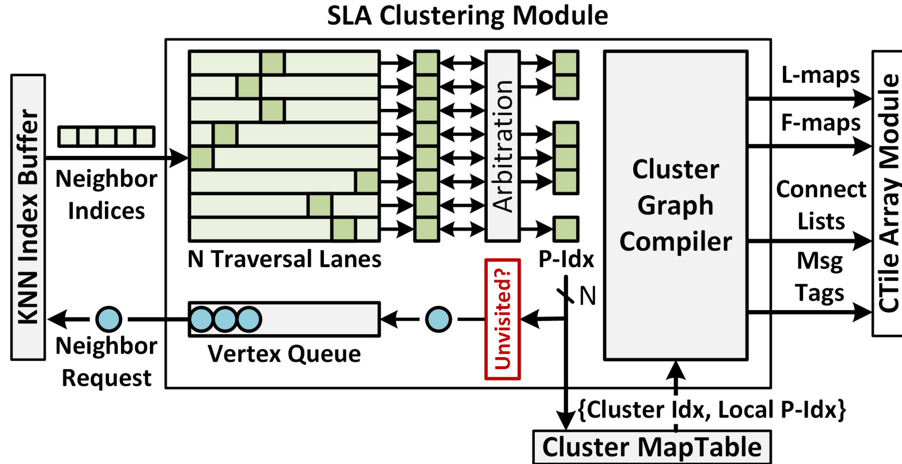


Figure 3.6: Architecture and dataflow of SLA clustering module.

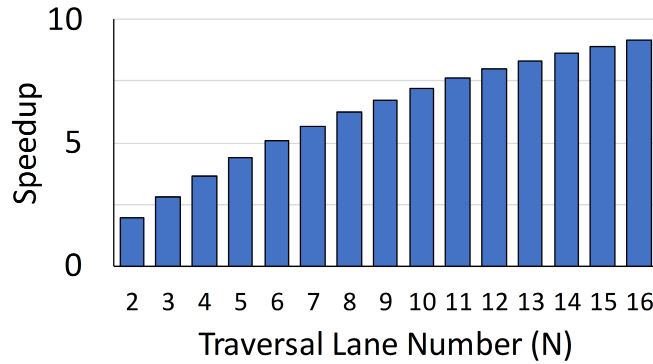


Figure 3.7: Speedup of the SLA clustering module using SBFS with different traversal lane numbers over a BFS implementation baseline.

maptable stores the mapping between the point index and the corresponding pair of {cluster index, local point index}. If a point index is not already recorded in the maptable, a new {cluster index, local point index} pair is assigned in ascending order and written to the maptable. Based on the outputs of the SBFS module and the maptable, the cluster graph compiler module generates subgraphs and inter-cluster connections in the form of local maps (L-Maps) and foreign maps (F-Maps) (see usage in Section 3.4). For inter-CTile data exchange, the cluster graph compiler generates the connectivity list and message tags (see usage in Section 3.3).

The speedup of the SLA clustering module using 2 to 16 traversal lanes for a KNN graph ($K = 20$) are compared to that of BFS, and the results are shown in

Figure 3.7. Nearly linear speedup can be obtained, but the speedup diminishes with more traversal lanes due to arbitration and bandwidth limitation of the vertex queue. For example, SBFS-16 reduces the traversal latency by 9.2×.

3.3 Locality-Aware NoC

The inter-CTile data exchange is supported by an NoC. A general-purpose mesh NoC forwards data in four directions at each router node to allow flexible routing for diverse workloads. In comparison, we propose the locality-aware chain NoC that takes advantage of the extracted spatial locality to achieve a lower complexity and a higher efficiency. The chain NoC is designed with a message reuse strategy to further improve performance and reduce area and power overheads.

3.3.1 Chain NoC Architecture

A chain NoC connects routers using two independent uni-directional networks: an up network transfers messages upward, whereas a down network transfers messages downward as shown in Figure 3.8. A message comprises a data field and a tag: the data is the neighbor fpsum computed by the source CTile and the tag consists of the point index and the directive for routing the message to its destination CTile.

The chain NoC is designed with a message reuse strategy where a message transfer passes through all its destination CTiles in the same direction to save redundant transfers. In SLA clustering, the cluster graph compiler (Figure 3.6) keeps track of the furthest destination CTile that an fpsum needs to travel to in both upward and downward directions. This approach simplifies the routing directive of a message tag into a single distance value between the source and the furthest destination CTile. By adopting the message reuse strategy, the chain NoC cuts the message traffic by 2.1×, which contributes to a higher efficiency for inter-CTile data exchange.

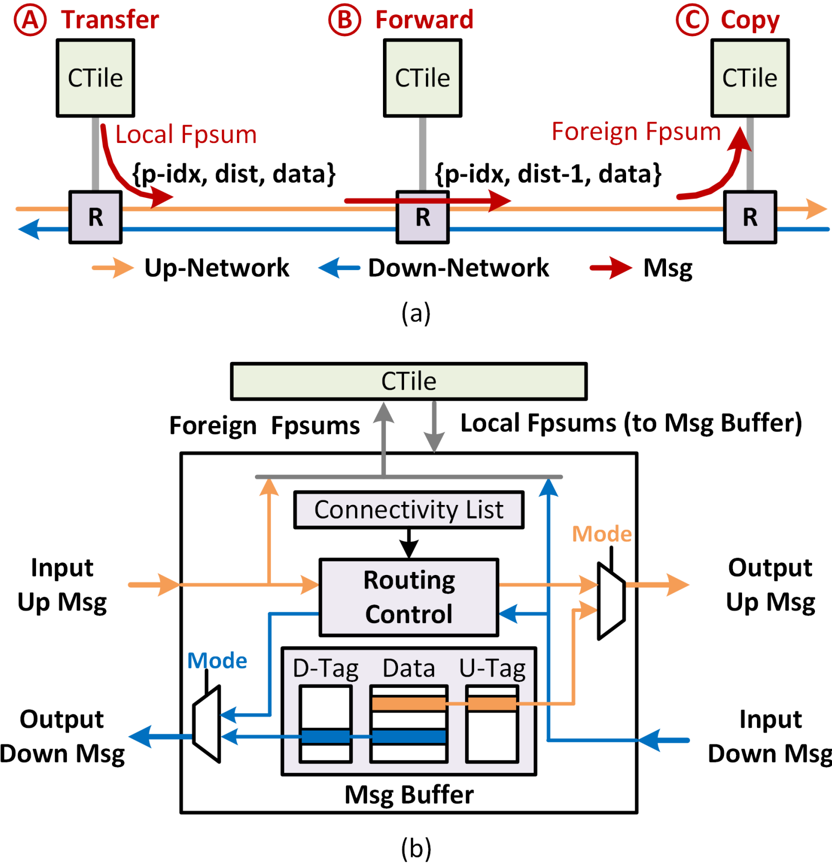


Figure 3.8: Architecture of (a) a chain NoC, and (b) a router for chain NoC.

3.3.2 Routing Algorithm

The routing mechanism of the chain NoC is illustrated in Figure 3.8(a). In preparation for exchanges, the router associated with a CTile receives the connectivity list and the message tags from the SLA clustering module, and the local neighbor fpsums from the CTile. The message tags and local neighbor fpsums are stored in the message buffer. The connectivity list contains the indices of neighbor fpsums that need to be fetched from other CTiles.

An fpsum exchange undergoes three stages as shown in Figure 3.8(a): in the transfer mode, a local router reads a pair of tag and data from the message buffer and sends a message onto the network; in the forward mode, routers forward the message along the way; and when an incoming message tag matches the connectivity list of a

Algorithm 1: Chain NoC Routing

```
Input/Output: msg_in/msg_out
if (msg_in != None) then
    {p-idx, dist, data} = msg_in
    if (dist > 1) then
        Forward: msg_out = {p-idx, dist-1, data}
    else
        Read {p-idx', dist', data'} from Msg Buffer
        Transfer: msg_out = {p-idx', dist', data'}
    if (p-idx match in connectivity list) then
        Copy: send data to CTile
else
    Read {p-idx', dist', data'} from Msg Buffer
    Transfer: msg_out = {p-idx', dist', data'}
```

destination router, the destination router copies the message data to the destination CTile in the copy mode. The routing algorithm is detailed in Algorithm 1. A message is propagated along the up or down direction. The message distance is reduced by 1 when passing through a hop in the forward mode until the message reaches its final destination. By prioritizing the forward mode over the transfer mode, a message is never stalled before reaching its final destination once it is transferred from the source.

Compared to a flexible mesh NoC, the uni-directional design of the chain NoC reduces the complexity and energy spent on complex message switching and arbitration. Furthermore, the routing algorithm ensures that a message is never stalled when traveling in the network, thus eliminating any extra input and output buffers in the router design. As a result, the chain NoC has a low design complexity and requires minimal area and energy overheads. In terms of performance, the chain NoC provides a better fetch efficiency than the mesh NoC even though the chain NoC uses only half of the mesh NoC's physical bandwidth. By reusing the messages, the chain NoC reduces the transfer latency by 2.1× over the mesh NoC. When SLA clustering is applied to both types of NoCs, the chain NoC reduces the transfer latency by

Figure 3.9: Microarchitecture of (a) a compute tile (CTile), (b) a compute engine, and (c) a sort engine.

3.2× over the mesh NoC, demonstrating its suitability for moving data with spatial locality.

3.4 CTile Architecture

A CTile supports the operations required for EdgeCONV, including KNN graph construction and GraphCONV. KNN graph construction is supported by DP for distance computation and K-min sorting; and GraphCONV is supported by DP for

fpsum computation and feature aggregation. A CTile shown in Fig. 3.9(a) comprises of a DP compute engine, a sort engine, and a group engine for feature aggregation. It also consists of buffers for storing fpsums and a controller for interfaces and configurations. Finally, an output buffer stores the results of CTile computation.

Fpsum Computation: The compute engine contains 2 sets of DP units, F_0 phi-DP units and F_0 theta-DP units (F_0 is a design parameter, $F_0 \leq F$), and vector arithmetic units (V-ALUs) as shown in Fig. 3.9(b). A DP unit caches and reuses the respective $\boldsymbol{\phi}$ and $\boldsymbol{\theta}$ weights while the inputs are streamed in to compute DP. A DP unit computes a C_0-way DP per cycle (C_0 is another design parameter, $C_0 \leq C$). Altogether, F_0 neighbor fpsums are computed per cycle by the theta-DP units, and F_0 vertex fpsums are computed per cycle by the subtraction of the outputs of the theta-DP units from the phi-DP units. The vertex and neighbor fpsums are stored in vertex and local neighbor buffers for reuse.

K-Min Sorting for KNN: The sort engine contains F_0 K-min sorter units as illustrated in Fig. 3.9(c). A sorter unit finds the KNN of one given vertex point using insertion sort that is implemented in a 1D systolic array of K sorting elements (SEs) [99]. A distance and point index pair is broadcast to all SEs. Each SE compares the distance value to its current value, inserts the new value or takes the shifted value from the left SE. A K-min sorter unit always maintains the K nearest points of a vertex point sorted by distance values.

Feature Aggregation: Feature aggregation involves max-pool (MP) of the neighbor fpsums, and post-processing (PP) including summing the max neighbor fpsum to the vertex fpsum, and applying ReLU as described by Eq. (3.3). Feature aggregation is done by the group engine shown in Fig. 3.10.

The MP process is of particular importance as it needs to access neighbor fpsums that reside locally or on a foreign CTile. The local controller uses L-maps provided by the SLA clustering module to locate local neighbor fpsums. The L-map of a vertex

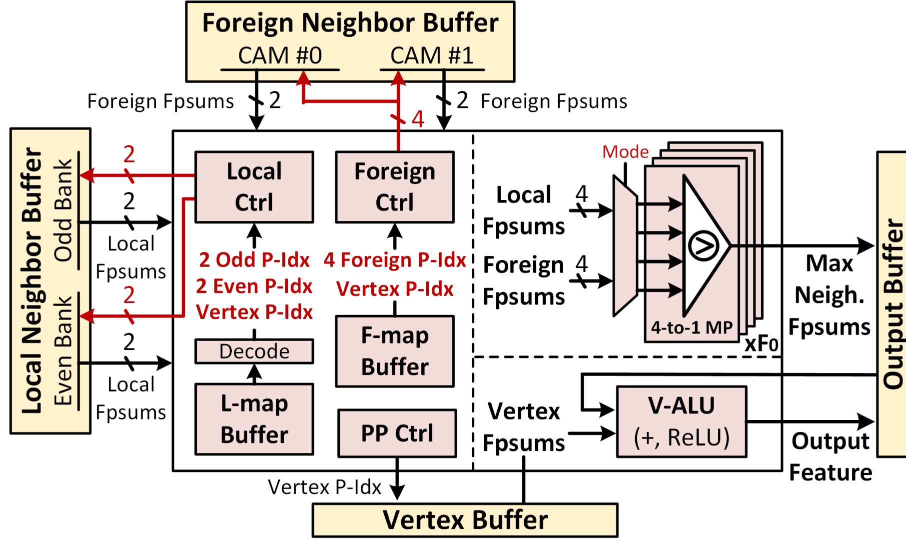


Figure 3.10: Microarchitecture of a group engine.

point stores the local connections to the vertex point in adjacency matrix form. An L-map word is decoded into point indices for accessing the neighbor fpsums stored in the local buffer. To speed up fetching, the local neighbor buffer can be implemented using two 2-port SRAM banks. Two even and two odd point indices are decoded from an L-map every cycle to allow 4 neighbor fpsum words to be fetched per cycle. Each fpsum word contains F_0 neighbor fpsums. The four fpsum words are sent to the MP units.

The foreign controller uses F-maps provided by the SLA clustering module to locate neighbor fpsums from foreign CTiles. An F-map of a vertex point contains its foreign neighbor point indices. An F-map word contains 4 point indices. The foreign controller reads one word from an F-map per cycle, decodes the 4 point indices, and looks up the indices in two content-addressable memories (CAMs). Each CAM returns at most 2 neighbor fpsum words. If a CAM has more than 2 matches, the foreign controller stalls for a cycle before proceeding to the next F-map word. Up to 4 neighbor fpsum words are fetched per cycle. The four fpsum words are sent to the MP units.

Once all local and foreign neighbor fpsums are aggregated to obtain the max

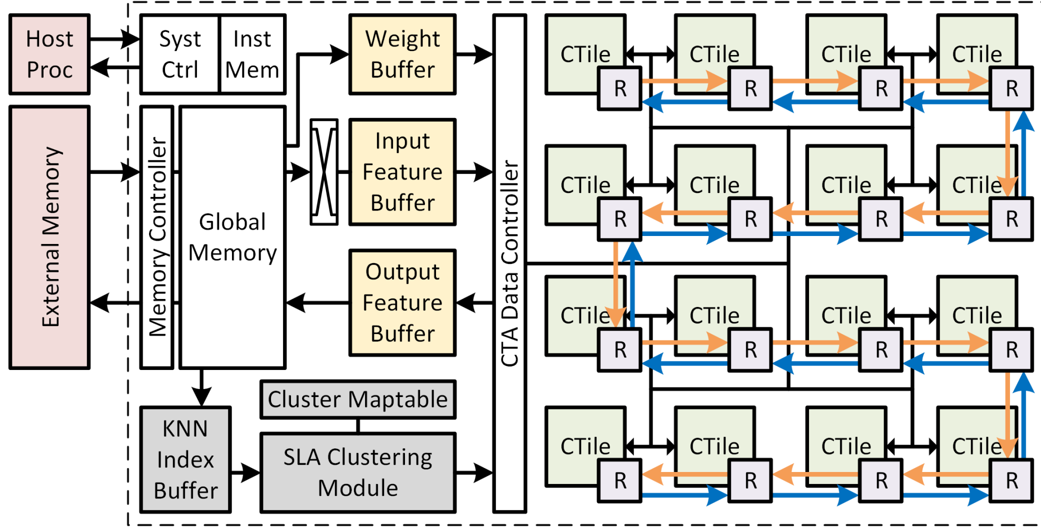


Figure 3.11: Point-X system architecture.

neighbor fpsum, it is summed with the vertex fpsum, followed by ReLU to produce the output feature.

3.5 Point-X System Architecture

The Point-X system architecture is outlined in Figure 3.11. Point-X consists of a clustering module, a CTile array (CTA) module, data buffers, and controllers for system configuration, dataflow, and external communication. The system controller receives the compiled instructions from the host processor, and configures the other modules according to the dataflow mode. The memory controller loads the input features, model weights, and KNN indices, and offloads the output features to the external memory. The global memory holds the data fetched from off-chip and distributes them to respective data buffers following the dataflow mode. The KNN index buffer stores the constructed KNN graph, and the cluster maptable stores the mapping after SLA clustering.

The clustering module performs SLA clustering and compiles the clustering configuration for data exchange and feature aggregation for the CTA module. In the prototype design, the CTA module consists a 4×4 2D array of CTiles connected by a

chain NoC with upward and downward networks. A data controller handles the input distribution to the CTiles using a pipelined H-tree bus. Inside a CTile, a compute engine contains 16 8-way 16b×8b DP units ($F_0 = 8$, $C_0 = 8$), reconfigurable to 8 8-way 16b×16b DP units. A group engine has 8 4-to-1 MP units. A sort engine has 8 K-min sorter units with 20 SEs ($K = 20$) each. Overall, Point-X consists of 16 CTiles, using a total of 2,048 16b×8b MACs or equivalently, 1,024 16b×16b MACs.

3.5.1 Multi-Mode Dataflow

Typical graph-based DNNs are composed of EdgeCONV (KNN graph construction and GraphCONV), shared MLP, and FC layers with different computational requirements. Point-X provides a multi-mode dataflow that coordinates the input loading and data distribution to support these diverse requirements. The dataflow steps for EdgeCONV, shared MLP and FC layers are illustrated in Figure 3.12.

KNN Graph Construction (Figure 3.12(a)): Vertex point features are loaded to DP units of each CTile by unicast and the input point features are streamed in to the CTiles by broadcast. A pair of 16b×8b DP units are combined to compute 16b-16b point feature distances: one DP unit computes the MSB part while the other computes the LSB part, and a V-ALU performs shift and accumulation of the two psums. The distance values and point indices are sent to the sort engine (Figure 3.9(c)). At completion, the K indices in each K-min sorter unit are read out.

GraphCONV (Figure 3.12(c)): The SLA clustering module loads point indices from the KNN index buffer, performs SBFS traversal to produce the clustering mapping, and compiles cluster configurations, including L-maps, F-maps, connectivity lists, and message tags. The L-maps and F-maps are used by the group engine for feature aggregation, and the connectivity list and message tags are used by the routers.

Following the cluster maptable, the input features are stored in the corresponding

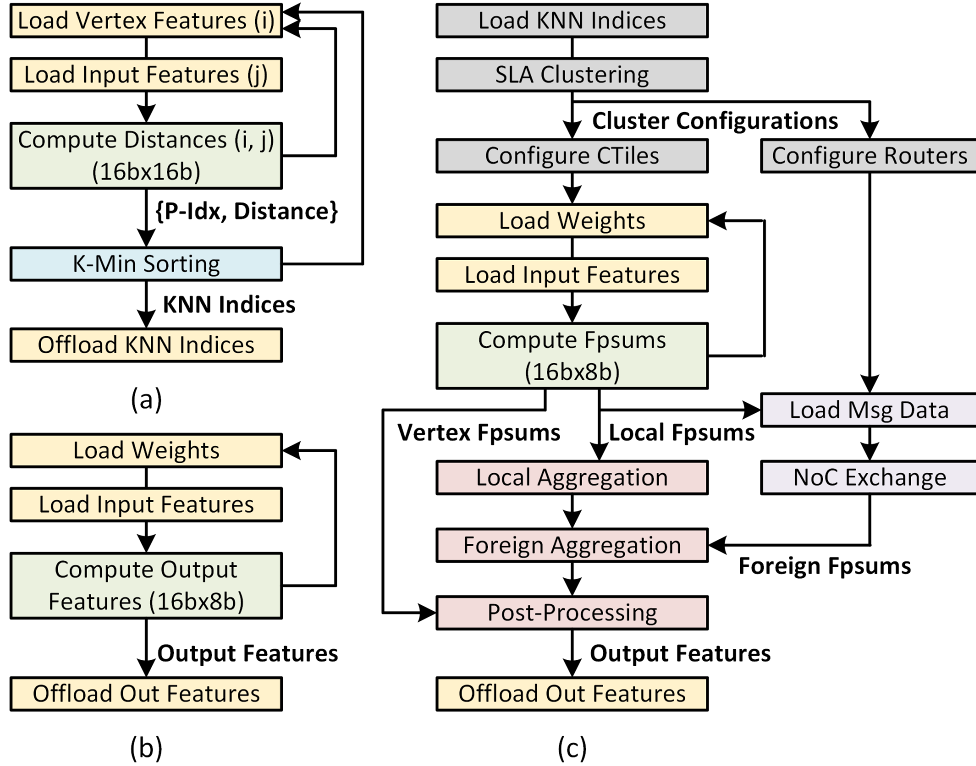


Figure 3.12: Multi-mode dataflow for (a) KNN graph construction, (b) shared MLP and FC, and (c) GraphCONV operations.

cluster bank. The weights are broadcast to all CTiles, and the input features are unicast to the corresponding CTiles. The vertex and neighbor fpsums are computed and stored in the local buffers. NoC data exchange can be executed concurrently with the local fpsum aggregation. The foreign neighbor fpsums are received over the NoC, and the group engine performs the foreign fpsum aggregation. Lastly, post-processing is performed.

Shared MLP and FC Layers (Figure 3.12(b)): The weights are broadcast to the CTiles, and the input features are streamed in and unicast to their corresponding CTiles. In the FC mode, each CTile receives 2 weights via unicast and the input vector is broadcast to the CTiles for computation.

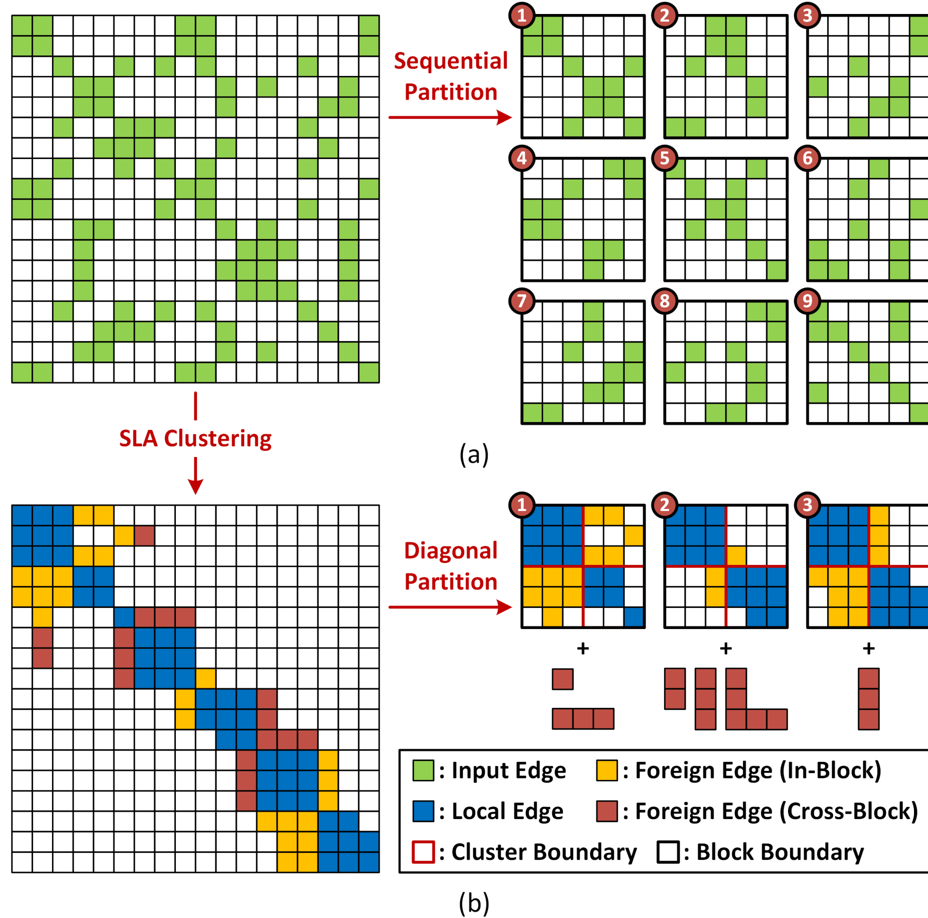


Figure 3.13: Workload partition schemes: (a) sequential partition and (b) diagonal partition.

3.5.2 Workload Partitioning

The Point-X prototype is designed for processing up to 1k points from the input point cloud at a time. Point clouds of size of over 1k points are partitioned into 1k-blocks for Point-X to process.

For KNN graph construction, shared MLP, and FC operations, an input point cloud is sequentially partitioned into 1k-blocks and processed with corresponding inputs or weights. For GraphCONV operation, the sequential partition is not the most efficient. As shown in Figure 3.13(a), the sequential partition divides the adjacency matrix in sequentially-ordered blocks by the source and destination points. Although being simple to execute, as the point size scales up, the scheme results in quadratically

increasing number of blocks to fetch.

In Point-X, we adopt a diagonal partition by exploiting the SLA clustering as illustrated in Figure 3.13(b). The SLA clustering is applied to the entire input point cloud. Only the diagonal blocks in the adjacency matrix are sent to Point-X for processing. As discussed in Section 3, the SLA clustering increases the spatial locality and computation efficiency by increasing the number of local edges and bringing the foreign edges closer to the matrix diagonal. Consequently, the diagonal partition provides two major advantages. First, the number of blocks scales linearly with the point size. Only a small number of additional cross-block foreign edges need to be handled. Second, the diagonal partition provides more spatial locality per block, resulting in a higher computation efficiency.

In handling large point clouds, the SLA clustering module needs to access scattered off-chip memory to fetch KNN indices of vertex points for graph traversal, which can cause a significant delay. To reduce the delay, the global memory is used to prefetch the KNN indices to supplement the KNN index buffer. Following the completion of clustering, Point-X fetches points of diagonal blocks from the off-chip memory. To improve the point fetch efficiency, the clustering results are organized so that the points located in the same DRAM page are accessed together. This approach helps to avoid unnecessary DRAM page changes and provides burst-like access bandwidth.

3.6 Benchmarking and Evaluation

A prototype of Point-X is designed in a 28nm CMOS technology to evaluate its performance, area, and efficiency. The results are compared to a CPU, a GPU, accelerator baselines, and state-of-the-art accelerator works [89, 90] to show Point-X's advantages in throughput and energy efficiency in processing graph-based point-cloud DNNs.

3.6.1 Evaluation Methodology

Following [4, 70, 71, 74–77, 80, 89, 90, 100–104], we used the sampled point cloud dataset from ModelNet40 [105] with a point cloud size of 1,024. Quantization-aware training was applied to reduce the precision of the input features and weights to 16b and 8b, respectively. The higher input feature precision is necessary for a larger point cloud size. For instance, an 8b input precision allows only 256 distinct positions in each dimension for the 1,024 points, making points indistinguishable. The 16b-input, 8b-weight DGCNN achieves 92.8% accuracy on ModelNet40, and shows no accuracy drop compared to the DGCNN trained in FP32 [4]. To evaluate Point-X's performance for scaled-up point cloud data, augmented datasets with 2k, 4k, 6k, 8k, and 10k points are generated by re-sampling the original CAD models in ModelNet40. The augmented datasets represent point clouds of higher resolutions.

A cycle-accurate model was developed to simulate the behavior and analyze the performance of Point-X's architecture and dataflow. For benchmarking, cycle-accurate models were also implemented for a query-based baseline and an exchange-based baseline.

A prototype of Point-X was implemented in RTL and synthesized in a 28nm CMOS technology with SRAMs generated from SRAM compilers to provide more accurate silicon area, timing, and power estimates. PrimeTime PX was used to estimate the energy consumption of Point-X based on activity waveform in running complete network layers of our workloads.

For performance comparison, we evaluated the query-based and exchange-based baselines using the same workloads. Both baselines were designed using the same number of CTiles, memory banks, and MACs in each DP unit as Point-X, but differ from Point-X in the data fetch mechanism. The exchange-based baseline is designed with a mesh NoC for data exchange between CTiles. Lastly, we compare Point-X's efficiency in EdgeCONV to existing accelerators [89, 90] and end-to-end throughput

Table 3.2: Storage and Area Breakdown of Point-X

Module	Storage (KB)	Area (mm^2)	Area Ratio (%)
CTile	18.39	0.288	4.25
Router	2.09	0.023	0.34
CTile Array (CTA)	327.75	4.935	72.90
SLA Clustering	26.88	0.566	8.36
System Ctrl & Mem	190.75	1.270	18.76
Point-X Total	545.4	6.8	100

Table 3.3: Layer Evaluation of Point-X on 1k-DGCNN [4]

Layer (Kernel Size, $F \times C$)		Latency (μs) (Overhead)	Energy (μJ) (Overhead)
EdgeCONV-1 (64×3)	KNN	8.3	17.0
	GraphCONV	10.2 (25.9%)	7.5 (6.5%)
EdgeCONV-2 (64×64)	KNN	74.6	158.5
	GraphCONV	16.8 (15.7%)	18.3 (2.6%)
EdgeCONV-3 (128×64)	KNN	74.6	158.5
	GraphCONV	31.0 (8.5%)	35.9 (1.3%)
EdgeCONV-4 (256×128)	KNN	148.0	316.7
	GraphCONV	89.8 (2.9%)	136.1 (0.4%)
Shared MLP (1024×512)		307.2	799.1
FC-1 (512×1024)		3.2	5.2
FC-2 (256×512)		0.9	1.3
FC-3 (40×256)		0.2	0.2
DGCNN Total		765.0 (1.38%)	1654.3 (0.12%)

and efficiency to the Nvidia GTX-1080Ti GPU and the Intel i7-7700K CPU.

3.6.2 Area, Performance, Efficiency Analysis

The memory and silicon area breakdown of Point-X implemented in a 28nm technology are shown in Table 3.2. The Point-X prototype is 6.8 mm^2 in size and contains 545.4 KB of on-chip memory. The SLA clustering module costs an area overhead of only 8.36%. Inside the CTA module, a CTile is 0.29 mm^2, while a chain NoC router is only 0.023 mm^2, which is less than 1/10 the size of a CTile.

The layer-by-layer latency and energy results of the Point-X prototype when running 1k-DGCNN are presented in Table 3.3. The results for each EdgeCONV

layer are shown separately for KNN graph construction and GraphCONV. For every GraphCONV, the overhead of SLA clustering is noted in parentheses. For a small EdgeCONV layer like EdgeCONV-1, the SLA clustering costs an overhead of 25.9% in latency and 6.5% in energy. The SLA clustering overhead decreases in larger EdgeCONV layers, e.g., it takes only 2.9% and 0.4% of the latency and energy in EdgeCONV-4. For the 1k-DGCNN workload, the Point-X prototype demonstrates an end-to-end latency of 0.77 ms at an average power consumption of 2.2 W. It achieves a throughput of 1307.1 inference/s (Inf./s) at 604.5 inference/Joule (Inf./J).

3.6.3 Workload Scalability Analysis

Point-X is also evaluated for scaled-up workloads. Figure 3.14(a) and (b) show the normalized latency and the latency breakdown of Point-X when running GraphCONV in DGCNN for different point cloud sizes. The latency breakdown looks at the five components of GraphCONV: SLA clustering (CL), fpsum compute (DP), local fpsum aggregation and NoC data exchange (LA&NoC), and foreign fpsum aggregation and post-processing (FA&PP), and system control and memory (MEM). The MEM component includes the configuration latency and additional data fetch latency that cannot be fully hidden by prefetching.

For a point size of under 4k, the global memory serves both the clustering module and the CTA module effectively by prefetching the inputs needed for GraphCONV. This can be observed by the almost linear increase in the normalized latency and consistent latency breakdown. For a point size larger than 4k, the input points needed for processing a block in GraphCONV may be located across many different DRAM pages, resulting in an increase in MEM as shown in Figure 3.14(b). The energy breakdown of Point-X for the GraphCONV workloads is shown in Figure 3.14(c).

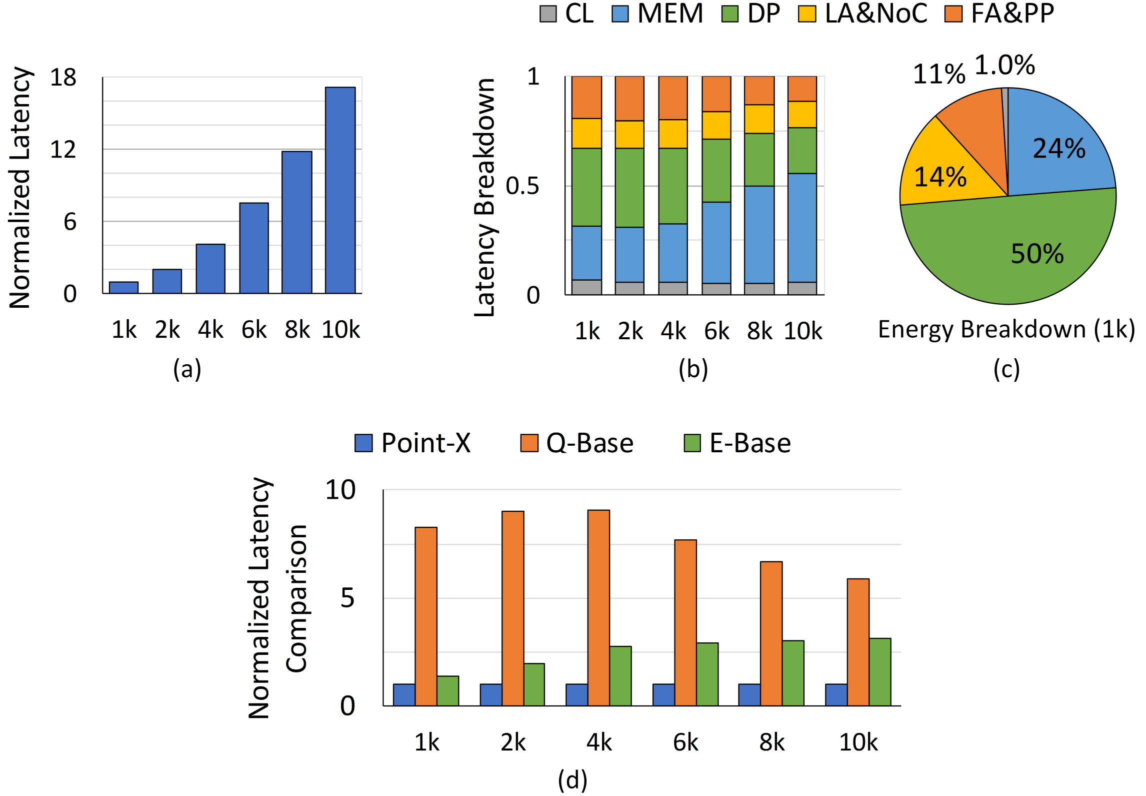


Figure 3.14: (a) Normalized latency, (b) latency breakdown, and (c) energy breakdown of Point-X for GraphCONV workloads in DGCNN; (d) normalized latency comparison of Point-X to query-based (Q-Base) and the exchange-based (E-Base) baselines for GraphCONV workloads of each point size.

3.6.4 Performance Comparison

In Figure 3.14(d), we show Point-X's improvement over a query-based design (Q-Base) and an exchange-based design (E-Base). The differences between Point-X and the two base designs are mainly attributed to the fetch mechanisms and the workload partition schemes for GraphCONV. In Q-Base, CTiles request neighbor points from a centralized shared memory. It incurs a low fetch efficiency due to memory access conflicts. In E-Base, CTiles communicate with each other through a mesh NoC to obtain the needed neighbor points. E-Base suffers from the overloading of message traffic across the network and the long latency of message transfer, which worsen its fetch efficiency. Overall, Point-X provides a speedup of 8.3× and 1.4× over Q-Base

Table 3.4: EdgeCONV Comparison to Existing Works

	Point-X	Cambricon-G	DeepBurning-GL
Type	ASIC	ASIC	FPGA
Technology	28 nm	45 nm	16 nm
Frequency	1.0 GHz	1.0 GHz	200 MHz
Compute Unit	2048 MACs	2048 MACs	5893 DSPs‡
Precision	16b & 16b/8b	16b	16b
On-Chip Mem.	545.4 KB	12.1 MB	3.2 MB‡
Efficiency	858.6 GOPS/W*	360.9 GOPS/W†	71.1 GOPS/W†

* A MAC is counted as 2 OPs; comparisons in KNN are included for energy but not counted in total OPs; Average from all EdgeCONV in [4].
† Numbers derived from [89, 90]; Only EdgeCONV (64×3) was reported.
‡ Derived from FPGA (Alveo U50) resource utilization reported in [90].

and E-Base, respectively, for GraphCONV workloads of 1k points.

The benefit of Point-X's diagonal workload partition is illustrated in Figure 3.14(d) showing normalized comparison for each point cloud size. As the point cloud size increases, the efficiency of Point-X's point fetching from DRAM decreases. As a result, the advantage of Point-X over Q-Base shrinks slightly due to the reduced number of edges per block (for a fixed K) and less frequent memory access conflicts that favor Q-Base. Point-X achieves an average speedup of 7.7× and 2.5× in GraphCONV computation across different point sizes compared to Q-Base and E-Base, respectively.

Table 3.4 compares Point-X to state-of-the-art accelerators for EdgeCONV computation [89, 90]. Cambricon-G [89] has the same peak throughput as Point-X but it uses 22× on-chip memory storage. The large storage holds all intermediate results on-chip for reuse and reduces the latency and energy overheads from external memory communication. DeepBurning-GL [90] is implemented on an FPGA using a customized EdgeCONV template. Both Cambricon-G and DeepBurning-GL process EdgeCONV following Eq. (3.2) and they do not exploit the locality in point cloud, resulting in less-efficient data fetching and redundant computation. Overall, Point-X achieves a 2.4× and 12.1× efficiency improvement in EdgeCONV computation over Cambricon-G and DeepBurning-GL, respectively.

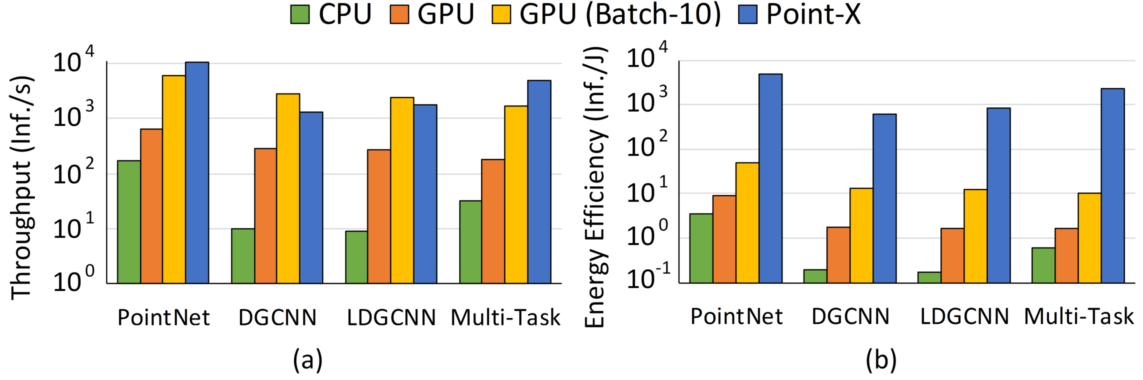


Figure 3.15: Comparison of (a) throughput and (b) energy efficiency of Point-X to the CPU and GPU baselines.

Point-X is also compared to a GPU (Nvidia GTX-1080Ti) and CPU (Intel i7-7700k). Both the GPU and the CPU are in more advanced, faster, and more efficient silicon technologies than the 28nm used for Point-X prototyping. The GPU and the CPU run at higher clock frequencies, have larger silicon footprints, and contain larger on-chip memories for caching. In contrast, Point-X is designed for graph-based DNNs. It has a much smaller silicon footprint and consumes less power. Point-X provides a total of 2,048 16b×8b MACs distributed to 16 CTiles, whereas the GPU has 3,584 cores and the CPU has 8 threads for floating point computation. In Figure 3.15, the throughput and energy efficiency of Point-X are compared to the GPU and CPU baselines for four workloads. When running DGCNN, Point-X achieves a 4.5× and a 129.7× higher throughput over the GPU and the CPU, respectively, at a 342.9× and a 3160.9× better energy efficiency, respectively. We also compared to the GPU with batch-10 inference. On average, the GPU throughput is improved by almost 10× with only 35% power increase. However, batching is often infeasible for devices with limited memory or real-time computing use cases.

3.7 Related Work

Graph-based point-cloud DNNs investigated in this work possess both regular and irregular computation structures. Regular computation structures require hardware designs that offer plenty of data reuse and computational parallelism; and irregular computation structures require hardware designs that provide flexible and efficient data fetch mechanisms. DNN/CNN accelerators [48, 57, 81] are designed for regularly structured computation, and they tend to perform poorly on graph-structured data featuring scattered memory access and limited data reuse. On the other hand, accelerators for graph processing [82, 83] are designed for sparse and irregular data access and computation, and they are unable to fully exploit the parallelism and data reuse in DNN-like workloads.

GNN/GCN accelerators [84–87, 89, 90] address both regular and irregular computation structures. The GNN/GCN accelerators work on arbitrary graphs with power-law distribution, which cause severe workload imbalance during processing. HyGCN [84] and GRIP [85] proposed window sliding/shrinking and parallel prefetching methods to improve fetch efficiency, whereas AWB-GCN [86] proposed runtime workload rebalancing methods to improve compute utilization. These works only focus on static graphs and are unable to fully support EdgeCONV due to the lack of sorting units for KNN graph construction. Recently, Cambricon-G and DeepBurning-GL [89, 90] were proposed to support dynamic graphs, i.e., EdgeCONV operations, but they do not exploit the data locality in point cloud data, making them unable to provide the best efficiency. Compared to these works, Point-X leverages data reuse opportunities to eliminate redundant computation and exploits spatial locality to achieve a higher computational efficiency for EdgeCONV operations.

Rubik and GNNAdvisor [87, 97] proposed graph reordering techniques using locality-sensitive hashing and Rabbit Order [106] to increase locality during GNN processing. The graph is first preprocessed offline by a host processor before being sent to the GPU

or accelerator for processing. For an EdgeCONV operation where the KNN graph is constructed in runtime, offline preprocessing is impractical due to data transfer overheads between the host and the accelerator. In contrast, Point-X avoids the data transfer by having an on-chip clustering based on our parallel SBFS algorithm, which provides a significant speedup over software implementation on the host processor in [87, 97].

General-purpose NoC router designs with lower complexity and area [107, 108] were explored in the past. To reduce the design complexity, [107] limits the routing to either x- or y-directions in a mesh network. The traffic in the network is prioritized to reduce the buffers needed in the router [107, 108]. Similarly, Point-X reduces the 2D mesh to 1D chain to avoid the switching overheads and prioritizes the forward mode to eliminate all buffers in a router. Accelerators [55, 109, 110] also adopted NoCs for inter-PE or memory-to-PE communication, allowing more flexibility to support diverse workloads and dataflows. In comparison, Point-X's NoC is specialized for efficient exchange of data with coarse-grained spatial locality and targets both compact area and stringent power budgets.

3.8 Summary

We present Point-X, an SLA accelerator architecture that extracts and leverages the spatial locality for efficient graph-based DNN processing on point clouds. To extract spatial locality in point clouds, an SBFS graph traversal algorithm is presented to map points into clusters to increase intra-CTile computation parallelism and reduce the inter-CTile communication overhead. Compared to conventional graph traversal methods, SBFS can be parallelized to achieve a 9.2× speedup. To leverage the spatial locality, a lightweight chain NoC is presented to reduce the data exchange latency by 3.2× compared to a mesh NoC. Combining these innovations, Point-X is designed with a multi-mode dataflow to support all operations in graph-based DNNs, i.e.,

EdgeCONV, shared MLP, and FC layers. A 1.0 GHz Point-X prototype is designed in a 28nm technology, occupying 6.8 mm^2 of silicon area. Point-X achieves up to 12.1× higher power efficiency in EdgeCONV compared to state-of-the-art accelerator works. When evaluated on the DGCNN workload, Point-X achieves a throughput of 1307.1 Inf./s and an energy efficiency of 604.5 Inf./J. Compared to the Nvidia GTX-1080Ti GPU, Point-X demonstrates a 4.5× and a 342.9× improvement in throughput and energy efficiency, respectively, on DGCNN workload.

CHAPTER IV

TetriX: Efficient Accelerator Architecture for Flexible Tensorized Neural Network Processing

The recent decade of advancement in AI and ML is to a large degree attributed to the development of DNN, and its CNN and RNN variants. To keep improving the accuracy, DNN models grow larger every year, at a rate of 1.5 to 2× increase in model size and model complexity every year [1, 2, 20, 21, 111]. The increasing model size and complexity create large storage and compute requirements for the underlying compute hardware that is becoming out of reach. State-of-the-art models can still be deployed on server-scale or desktop platforms using high-end GPUs and CPUs, but they are practically infeasible for resource-constrained platforms i.e., mobile, edge, and smart sensors, due to area, power, and cost budgets.

To meet these demanding requirements, researches on model compression have shown promising results in reducing the model size and complexity without degrading the accuracy performance. The popular compression methods through network pruning [26–28], i.e., unstructured pruning, offers good compression of a network model but the sparse data formats, i.e., compressed sparse row (CSR), often result in irregular computation and memory access, leading to lower hardware utilization. Alternatively, low-rank approximation methods, i.e., singular-value decomposition (SVD) [29] and matrix factorization [30], approximate the model's weight matrix

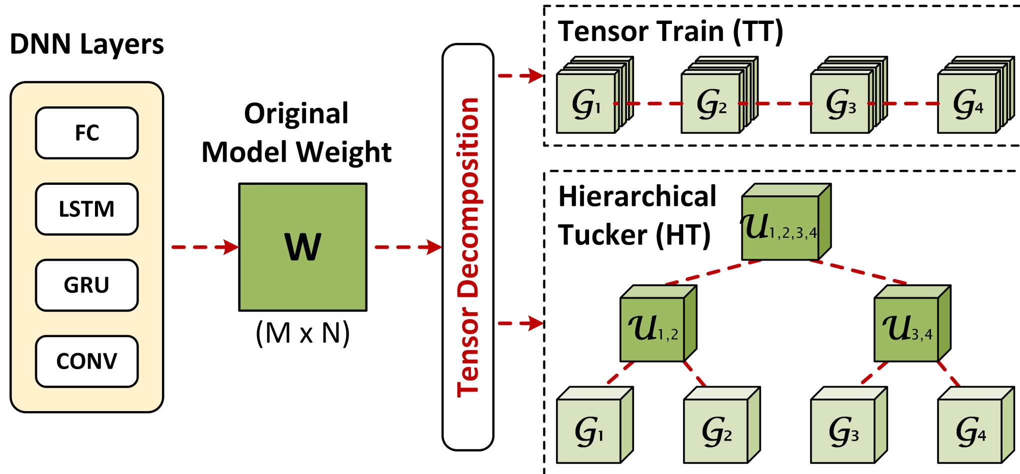


Figure 4.1: Illustration of model weight and tensor decomposition for TNNs.

using low-rank representations. These methods produce regular data structures to facilitate computation, but they often struggle to reach a good balance between compression and accuracy. Recently, tensor decomposition [34, 112, 112–129], a high-order generalization of the low-rank methods, has gained a lot of progress in network model compression by demonstrating a larger compression than the 2D methods [29, 30] while maintaining negligible accuracy drop.

However, a network compressed by tensor decomposition, which we name a tensorized neural network (TNN), requires high-order tensor contraction operations for inference. Figure 4.1 shows an example of a TNN. Compared to a traditional vector or matrix multiplication, a tensor contraction requires additional tensor orchestration operations that consist of arbitrary tensor reshape, permute, and transpose operations to map a tensor into a 2D representation in memory for matrix multiplication. These orchestration operations require additional memory operations, i.e., read-permute-write, read-transpose-write, for general-purpose processors, reducing the computational efficiency. The orchestration operations can be optimized by designing the memory access pattern with the datapath, and coalescing memory access and computation during processing in a custom accelerator. However, existing DNN/CNN accelerators [48, 59, 81] are only optimized for fixed tensor orders (2 for DNN, 4 for

CNN), and lack the support for flexible TNN workloads with arbitrary tensor orders. On the other hand, the existing TNN accelerators [5, 130, 131] are designed for a specific tensor decomposition method using a fixed tensor orchestration pattern, i.e., tensor train for TIE [5], limiting the support of a wider range of TNN workloads.

In this work, we propose TetriX, a co-design between accelerator architecture and workload mapping to enable a flexible and efficient TNN inference. TetriX performs TNN inference on the optimal contraction sequence which requires the minimum computation and memory size. A hybrid inner-outer product mapping scheme is proposed to eliminate complex orchestration operations in a TNN inference by alternating between inner and outer products. The TetriX architecture is designed with a configurable stationary dataflow to support both inner and outer product operations, and uses index translation and output gathering mechanisms to support arbitrary permute and reshape operations efficiently. TetriX is the first design to support all decomposition methods. Through our evaluations, the hybrid mapping scheme is shown to outperform the inner-only and outer-only mapping schemes for both simple and complex decomposition methods. TetriX also shows a latency improvement of up to 3.9× and 1.3× on average compared to TIE for tensor-train TNNs.

In this chapter, we first provide an overview of the basics of TNNs and present the computation challenges in TNN inference in Section 4.1. Then, we present a method to identify the optimal contraction sequence to processing a TNN, and explain the benefits of our proposed hybrid inner-outer product mapping scheme in Sections 4.2 and 4.3. In Section 4.4, we present in detail the architecture and microarchitecture of TetriX. Lastly, we show the evaluation results of TetriX and compare it with state-of-the-art accelerators and general-purpose processors in Section 4.5, before summarizing the work in Section 4.6.

4.1 Background

In general, the DNN computation i.e., FC, LSTM, GRU, and embedding layers, can be formulated as a vector-matrix multiplication (VMM) between an input vector $\boldsymbol{x} \in \mathbb{R}^N$ and a weight matrix $\boldsymbol{W} \in \mathbb{R}^{M \times N}$ to obtain an output vector $\boldsymbol{y} \in \mathbb{R}^M$ using $\boldsymbol{y}_{[j]} = \sum_{k=1}^{N} \boldsymbol{W}_{[j,k]} \cdot \boldsymbol{x}_{[k]}$.

A TNN decomposes a large weight matrix into a series of small and high-order tensors using tensor decomposition methods. To perform tensor decomposition, the 2D weight matrix $\boldsymbol{W}$ is first tensorized into either $\boldsymbol{\mathcal{W}} \in \mathbb{R}^{m_1 \times n_1 \times m_2 \times n_2 \times \cdots \times m_d \times n_d}$ or $\boldsymbol{\mathcal{W}} \in \mathbb{R}^{m_1 \times \cdots \times m_d \times n_1 \times \cdots \times n_d}$, depending on the tensor decomposition method. Similarly, input vector $\boldsymbol{x}$ and output vector $\boldsymbol{y}$ must be tensorized into input tensor $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{n_1 \times \cdots \times n_d}$ and output tensor $\boldsymbol{\mathcal{Y}} \in \mathbb{R}^{m_1 \times \cdots \times m_d}$, respectively, where $M = \prod_{k=1}^{d} m_k$, $N = \prod_{k=1}^{d} n_k$, and d is the order of input and output tensors. The computation of a TNN layer can be represented by a tensor-tensor multiplication as in Eq. (4.1), where the weight tensor $\boldsymbol{\mathcal{W}}$ is represented in tensor-decomposed format, i.e., tensor train (TT), hierarchical Tucker (HT), tensor ring (TR), or block term (BT) [33, 132–135].

$$\boldsymbol{\mathcal{Y}}_{[j_1,j_2,\ldots,j_d]} = \sum_{i_1=1}^{n_1} \sum_{i_2=1}^{n_2} \cdots \sum_{i_d=1}^{n_d} \boldsymbol{\mathcal{W}}_{[j_1,i_1,j_2,i_2,\ldots,j_d,i_d]} \cdot \boldsymbol{\mathcal{X}}_{[i_1,i_2,\ldots,i_d]} \tag{4.1}$$

In this section, we first present common tensor decomposition methods used in TNNs, then describe how to perform TNN inference using tensor contraction. Lastly, we present the computation challenges for TNN processing. Here, we follow the notation convention to represent vectors, matrices, and tensors (order ≥ 3) using boldface lowercase letters ($\boldsymbol{v}$), boldface capital letters ($\boldsymbol{M}$), and boldface script letters ($\boldsymbol{\mathcal{T}}$), respectively [136].

4.1.1 Tensor Decomposition Methods

Recently, research works have demonstrated the ability of several tensor decomposition methods to compress different network models, i.e., MLP, CNN, RNN, for different applications, i.e., video classification, natural language understanding, image classification, and large-scale recommendation model [34, 112, 112–129]. These tensor decomposition methods differ in type, number, order, and topology of the decomposed tensors, which lead to different compression ratio and accuracy performance for TNNs. It is still unclear which decomposition method performs best for a given network model or an application. Futhermore, new and improved decomposition methods are still being proposed in recent works. Here, we present several tensor decomposition methods that are commonly seen in TNNs. For simplicity and consistency, we modify the naming and the letter notations from existing literatures, and call a core tensor when it contains at least one input or output dimension, or a transfer tensor when it only contains rank values.

Tensor Train (TT): The TT decomposition method decomposes a tensor into a series of core tensors connected in a chain form [33]. In a TT-TNN, the weight matrix $\boldsymbol{W}$ is first tensorized into a d-th-order tensor $\boldsymbol{\mathcal{W}} \in \mathbb{R}^{(m_1 \times n_1) \times (m_2 \times n_2) \times \cdots \times (m_d \times n_d)}$. Then, $\boldsymbol{\mathcal{W}}$ is decomposed into d 4th-order core tensors $\boldsymbol{\mathcal{G}}^{(k)} \in \mathbb{R}^{r_{k-1} \times m_k \times n_k \times r_k}$, $k \in [1, d]$. The TT-format of $\boldsymbol{\mathcal{W}}$ can be represented as

$$\boldsymbol{\mathcal{W}}_{[j_1,i_1,j_2,i_2,\ldots,j_d,i_d]} = \sum_{k_1=1}^{r_1} \sum_{k_2=1}^{r_2} \cdots \sum_{k_{d-1}=1}^{r_{d-1}} \boldsymbol{\mathcal{G}}^{(1)}_{[j_1,i_1,k_1]} \cdot \boldsymbol{\mathcal{G}}^{(2)}_{[k_1,j_2,i_2,k_2]} \cdots \boldsymbol{\mathcal{G}}^{(d)}_{[k_{d-1},j_d,i_d]}. \tag{4.2}$$

$\boldsymbol{\mathcal{G}}^{(k)}$ and r_k are the k-th core tensor and rank, respectively, and $r_0 = r_d = 1$ by definition. In TT-format, $\boldsymbol{\mathcal{W}}$ requires $\sum_{k=1}^{d} r_{k-1} n_k m_k r_k = O(dmnr^2)$ parameters, where $r = \max(r_k), m = \max(m_k)$, and $n = \max(n_k)$, giving a compression ratio of $(MN)/\sum_{k=1}^{d}(r_{k-1} n_k m_k r_k)$. In practice, the rank values are selected to be much smaller than the input and output dimensions, that is, $r \ll m, n$, so a large compres-

sion ratio can be achieved.

Hierarchical Tucker (HT): The HT decomposition method decomposes a tensor hierarchically into core tensors and transfer tensors connected in a binary tree form [132, 133]. In the general HT decomposition, each HT node $\boldsymbol{\mathcal{G}}^{(s)} \in \mathbb{R}^{r_s \times n_{\mu_s} \times \cdots \times n_{\upsilon_s}}$ is associated with a dimension set $s \subsetneq D$, $D = \{1, 2, \ldots, d\}$, $\mu_s = \min(s)$, and $\upsilon_s = \max(s)$. An HT node can be recursively decomposed into a transfer tensor $\boldsymbol{\mathcal{U}}^{(s)} \in \mathbb{R}^{r_s \times r_{s_1} \times r_{s_2}}$, a left node $\boldsymbol{\mathcal{G}}^{(s_1)} \in \mathbb{R}^{r_{s_1} \times n_{\mu_{s_1}} \times \cdots \times n_{\upsilon_{s_1}}}$, and a right node $\mathcal{G}^{(s_2)} \in \mathbb{R}^{r_{s_2} \times n_{\mu_{s_2}} \times \cdots \times n_{\upsilon_{s_2}}}$ following Eq. (4.3) until its dimension set contains only one dimension.

$$\boldsymbol{\mathcal{G}}^{(s)} = \boldsymbol{\mathcal{U}}^{(s)} \times \boldsymbol{\mathcal{G}}^{(s_1)} \times \boldsymbol{\mathcal{G}}^{(s_2)} \tag{4.3}$$

In an HT-TNN, the weight matrix $\boldsymbol{W}$ is first tensorized into a d-th-order tensor $\boldsymbol{\mathcal{W}} \in \mathbb{R}^{(m_1 \times n_1) \times (m_2 \times n_2) \times \cdots \times (m_d \times n_d)}$. Then, $\boldsymbol{\mathcal{W}}$ is decomposed recursively following Eq. (4.3) and can be represented recursively as

$$\begin{aligned}
\boldsymbol{\mathcal{W}}_{[j_1, i_1, j_2, i_2, \ldots, j_d, j_d]} &= \sum_{k=1}^{r_D} \sum_{p=1}^{r_{D_1}} \sum_{q=1}^{r_{D_2}} \boldsymbol{\mathcal{U}}^{(D)}_{[k,p,q]} \cdot \boldsymbol{\mathcal{G}}^{(D_1)}_{[p, \varphi_{D_1}(i,j)]} \cdot \boldsymbol{\mathcal{G}}^{(D_2)}_{[q, \varphi_{D_2}(i,j)]} \qquad (4.4) \\
\boldsymbol{\mathcal{G}}^{(s)}_{[k, \varphi_s(i,j)]} &= \sum_{p=1}^{r_{s_1}} \sum_{q=1}^{r_{s_2}} \boldsymbol{\mathcal{U}}^{(s)}_{[k,p,q]} \cdot \boldsymbol{\mathcal{G}}^{(s_1)}_{[p, \varphi_{s_1}(i,j)]} \cdot \boldsymbol{\mathcal{G}}^{(s_2)}_{[q, \varphi_{s_2}(i,j)]},
\end{aligned}$$

where $D = \{1, 2, \ldots, d\}$, $D_1 = \{1, \ldots, \lfloor d/2 \rfloor\}$, and $D_2 = \{\lceil d/2 \rceil, \ldots, d\}$ are associated to the root node and its left and right child nodes, respectively. The mapping function $\varphi_s(i, j)$ produces the correct indices $i \subseteq \{i_1, i_2, \ldots, i_d\}$ and $j \subseteq \{j_1, j_2, \ldots, j_d\}$ for a given node $\boldsymbol{\mathcal{G}}^{(s)}$ with the given s and d. For example, with $d = 6$ and $s = \{3, 4\}$, the output of $\varphi_s(i, j)$ is (i_3, i_4, j_3, j_4).

The HT method decomposes $\boldsymbol{\mathcal{W}}$ into 3rd-order core tensors $\boldsymbol{\mathcal{G}}^{(k)} \in \mathbb{R}^{r_k \times m_k \times n_k}$, $k = 1, 2, \ldots, d$, and 3rd-order transfer tensors $\boldsymbol{\mathcal{U}}^{(s)} \in \mathbb{R}^{r_s \times r_{s_1} \times r_{s_2}}$. In HT-format, $\boldsymbol{\mathcal{W}}$ requires $\sum_{k=1}^{d} r_k m_k n_k + \sum_{\Phi(s, s_1, s_2)} r_s r_{s_1} r_{s_2} = O(dmnr + dr^3)$ parameters, where

$\Phi(s, s_1, s_2)$ lists all valid tuples of (s, s_1, s_2) that appear in the HT decomposition.

Tensor Ring (TR): The TR decomposition method decomposes a tensor into core tensors similarly to TT decomposition method, but forms a ring structure by connecting the first and last core tensors using $r_0 = r_d = R$, and $R \geq 1$ to achieve higher expressiveness than the TT-format [134]. In a TR-TNN, a weight matrix $\boldsymbol{W} \in \mathbb{R}^{M \times N}$ is first tensorized into a d-th-order tensor $\boldsymbol{\mathcal{W}} \in \mathbb{R}^{n_1 \times \cdots \times n_{d_n} \times m_1 \times \cdots \times m_{d_m}}$, where $N = \prod_{k=1}^{d_n} n_k$, $M = \prod_{k=1}^{d_m} m_k$, and $d = d_n + d_m$. Then, $\boldsymbol{\mathcal{W}}$ is decomposed into d_n 3rd-order core tensors $\boldsymbol{\mathcal{G}}^{(k_n)} \in \mathbb{R}^{r_{k_n - 1} \times n_k \times r_{k_n}}$, $k_n \in [1, d_n]$, and d_m 3rd-order core tensors $\boldsymbol{\mathcal{G}}^{(k_m)} \in \mathbb{R}^{r_{k_m - 1} \times m_k \times r_{k_m}}$, $k_m \in [d_n + 1, d]$. The TR-format of $\boldsymbol{\mathcal{W}}$ can be represented as

$$\begin{aligned} &\boldsymbol{\mathcal{W}}_{[i_1, \ldots, i_{d_n}, j_1, \ldots, j_{d_m}]} \\ &\quad = \sum_{k_0}^{r_0} \cdots \sum_{k_{d-1}}^{r_{d-1}} \boldsymbol{\mathcal{G}}^{(1)}_{[k_0, i_1, k_1]} \cdots \boldsymbol{\mathcal{G}}^{(d_n)}_{[k_{d_n - 1}, i_{d_n}, k_{d_n}]} \cdot \boldsymbol{\mathcal{G}}^{(d_n + 1)}_{[k_{d_n}, j_1, k_{d_n + 1}]} \cdots \boldsymbol{\mathcal{G}}^{(d)}_{[k_{d-1}, j_{d_m}, k_0]}. \end{aligned} \tag{4.5}$$

In practice, TNNs in TR-format have tensor cores that separate input and output dimensions, as opposed to TNNs in TT-format. This allows a more flexible tensorization of input and output vectors where d_n and d_m may be different. In TR-format, $\boldsymbol{\mathcal{W}}$ requires $\sum_{k_n=1}^{d_n} r_{k_n - 1} n_k r_{k_n} + \sum_{k_m = d_n + 1}^{d} r_{k_m - 1} m_k r_{k_m} = O(dnr^2 + dmr^2)$ parameters.

Block Term (BT): The canonical polyadic (CP) method decomposes a tensor into a sum of several component rank-1 tensors whereas the Tucker method decomposes a tensor into a high-order transfer tensor multiplied by a tensor along each order [31, 32, 136]. BT decomposition method combines the key features of CP decomposition and Tucker decomposition, and decomposes a tensor into a sum of Tucker formats [135]. In a BT-TNN, the weight matrix $\boldsymbol{W}$ is first tensorized into a d-th-order tensor $\boldsymbol{\mathcal{W}} \in \mathbb{R}^{(m_1 \times n_1) \times (m_2 \times n_2) \times \cdots \times (m_d \times n_d)}$. With a CP-rank S, $\boldsymbol{\mathcal{W}}$ is decomposed into S transfer tensors $\boldsymbol{\mathcal{U}}^{(s)} \in \mathbb{R}^{r_1 \times \cdots \times r_d}$, where $s \in [1, S]$, and $(S \times d)$ 3rd-order core

Table 4.1: Comparison of Tensor Decomposition Methods

Method	Tensor Topology	Tensor Type	Tensor Order	Tensor Number	Parameter Number	Applications
Baseline	-	Matrix	2	1	MN	-
TT	Chain	Core	4	d	$O(dmnr^2)$	[34, 113–119]
HT	Binary Tree	Transfer Core	3 3	$(d-1)$ d	$O(dmnr + dr^3)$	[122–124]
TR	Ring	Core	3	$d_n + d_m$	$O(dnr^2 + dmr^2)$	[120, 121]
BT	Tree	Transfer Core	d 3	S $S \times d$	$O(Sdmnr + Sr^d)$	[125–128]

tensors $\boldsymbol{\mathcal{G}}^{(s,k)} \in \mathbb{R}^{r_k \times m_k \times n_k}$, $k \in [1, d]$. The BT-format of $\boldsymbol{\mathcal{W}}$ can be represented as

$$\boldsymbol{\mathcal{W}}_{[j_1,i_1,j_2,i_2,\ldots,j_d,i_d]} = \sum_{s=1}^{S} \sum_{k_1=1}^{r_1} \cdots \sum_{k_d=1}^{r_d} \boldsymbol{\mathcal{U}}^{(s)}_{[k_1,\ldots,k_d]} \cdot \boldsymbol{\mathcal{G}}^{(s,1)}_{[k_1,j_1,i_1]} \cdots \boldsymbol{\mathcal{G}}^{(s,d)}_{[k_d,j_d,i_d]}. \tag{4.6}$$

$\boldsymbol{\mathcal{G}}^{(s,k)}$ and $\boldsymbol{\mathcal{U}}^{(s)}$ are the k-th Tucker core tensor and the transfer tensor of the s-th Tucker model, respectively. If $S = 1$, then the BT-format is reduced to the Tucker-format. In BT-format, $\boldsymbol{\mathcal{W}}$ requires $S\left(\sum_{k=1}^{d} r_k m_k n_k + \prod_{k=1}^{d} r_k\right) = O(Sdmnr + Sr^d)$ parameters.

In general, the TT is considered as the the simplest decomposition because of its simple and regular structure, and can be supported by most existing TNN accelerator works [5, 130, 131]. HT, TR, and BT are more complex decomposition methods, that rely on more complex structures to achieve higher compression ratio or better accuracy. Besides the tensor decomposition methods mentioned above, novel or improved decomposition methods, i.e., KCP [129], are proposed. As a general observation, the decomposed weight tensors have relatively small parameter sizes compared to the original weight matrix. A core tensor is either a 3rd-order or a 4th-order tensor and has at most one input and one output dimension. A transfer tensor is either a 3rd-order or a d-th-order tensor and only contains rank values. Table 4.1 compares the general property of TT, HT, TR, and BT decomposition methods.

4.1.2 TNN Inference with Tensor Contraction

The inference of a TNN layer can be computed element by element following Eq. (4.1), where the weight tensor $\boldsymbol{\mathcal{W}}$ can be represented using Eq. (4.2), (4.4), (4.5), or (4.6). However, it requires a massive amount of MAC operations where each component of a tensor is accessed repeatedly to compute for every element in the output tensor, causing a lot of redundant memory accesses and MAC operations [5]. Alternatively, the inference of a TNN with K decomposed tensors can be computed through a sequence of K tensor contraction operations. A tensor contraction can be performed between two tensors if some of their dimensions are matched [136]. For instance, given two tensors $\boldsymbol{\mathcal{A}} \in \mathbb{R}^{n_1 \times n_2 \times n_3}$ and $\boldsymbol{\mathcal{B}} \in \mathbb{R}^{m_1 \times m_2 \times m_3}$, where $n_2 = m_3$, the tensor contraction $\boldsymbol{\mathcal{C}} = (\boldsymbol{\mathcal{A}} \times_3^2 \boldsymbol{\mathcal{B}}) \in \mathbb{R}^{n_1 \times n_3 \times m_1 \times m_2}$ can be mathematically expressed as

$$(\boldsymbol{\mathcal{A}} \times_3^2 \boldsymbol{\mathcal{B}})_{[i_1,i_3,j_1,j_2]} = \sum_{k=1}^{n_2} \boldsymbol{\mathcal{A}}_{[i_1,k,i_3]} \cdot \boldsymbol{\mathcal{B}}_{[j_1,j_2,k]}. \tag{4.7}$$

In practice, a tensor contraction is often converted into a matrix-matrix multiplication (MMM) for processing. For example, in Eq. (4.7), tensor $\boldsymbol{\mathcal{A}}$ undergoes a mode-2 unfold operation to obtain matrix $\boldsymbol{A} \in \mathbb{R}^{(n_1 \times n_3) \times n_2}$ and tensor $\boldsymbol{\mathcal{B}}$ undergoes a mode-3 unfold and transpose operations to obtain matrix $\boldsymbol{B} \in \mathbb{R}^{m_3 \times (m_1 \times m_2)}$. The MMM between $\boldsymbol{A}$ and $\boldsymbol{B}$ produces matrix $\boldsymbol{C} \in \mathbb{R}^{(n_1 \times n_3) \times (m_1 \times m_2)}$, which is then folded to obtain the output tensor $\boldsymbol{\mathcal{C}}$. In this work, we refer to all fold and unfold operations as tensor orchestration operations, which can be viewed as a combination of fundamental tensor permute, reshape, and transpose operations.

TNN processing requires a large variety of tensor orchestration operations. Beyond the example in Eq. (4.7), both the number of dimensions and the dimension for contraction in $\boldsymbol{\mathcal{A}}$ and $\boldsymbol{\mathcal{B}}$ can be arbitrary. In addition, there could be multiple dimensions for contraction between $\boldsymbol{\mathcal{A}}$ and $\boldsymbol{\mathcal{B}}$, and permutation on the dimensions

of $\mathcal{C}$ may be required. All the possible combinations make the tensor orchestration operation in TNN processing impractical to generalize.

4.1.3 Computation Challenges

The TNN inference is computed via a sequence of tensor contraction operations which are broken down into tensor orchestration operations and MMM operations. The general-purpose SIMD processors can perform MMMs efficiently. However, tensor orchestrations require additional memory operations, i.e., read-permute-write, read-transpose-write, to organize data in memory before MMMs. The memory operations render a SIMD architecture less efficient than a custom accelerator that coalesces tensor orchestration and computation by optimizing the memory access along with the datapath design. The decomposed TNN layers are all regular-structured tensors, but the inference cannot be executed efficiently by existing DNN accelerators due to their lack of support for flexible tensor orchestrations. Recent TNN accelerators [5, 130, 131], were proposed for a specific tensor decomposition method and designed for a specific tensor orchestration pattern, e.g., TT and backward processing in TIE [5]. As a result, they are either unable to support other popular tensor decomposition methods or suffer from unnecessary computation and memory access when processing TNN inference. We identify the following major challenges for TNN computation.

Flexibility of TNN Workloads: Beyond the well-established decomposition methods presented, a range of new decomposition methods are also receiving attention [129]. In addition, researches showed that the rank hyperparameter selection of a TNN can be optimized by neural architecture search, giving potentially better compression and performance [137–139]. Therefore, a TNN processing architecture must be flexible in supporting all decomposition methods, including TT, HT, TR, BT, and others, that may have varying orders, dimensions, and rank hyperparameters to remain relevant as this field continues to evolve.

Performance of TNN Inference: For a TNN layer of K decomposed weight tensors, the output tensor can be computed in $O(K!)$ different ways, each corresponding to a unique contraction sequence. Each contraction sequence requires different amount of MAC operations, intermediate memory storage sizes, and tensor orchestrations, depending on the input, weight, and output tensor dimensions and rank values. Existing TNN accelerators, i.e., TIE [5], are designed with a fixed contraction sequence that may be far from the optimal in many cases, causing redundant MAC computation and memory access for a TNN inference.

Efficiency of Tensor Orchestration: Versatile orchestrations are required to support decomposition methods with more complex orchestration requirements, i.e., HT, TR, BT, and different contraction sequences optimized for different workloads. For the hardware, different combinations of the permute, transpose, and reshape operations require different memory read/write patterns, as well as the coordination and buffering of data accessed. The diverse operations and patterns cause a large design complexity and control overhead. Prior works [5, 130, 131] avoid the large design complexity and overhead by only supporting limited orchestration patterns, causing a limitation in the TNN type they can support and the contraction pattern they can use for TNN inference. For example, TIE [5] can only support TT-TNN inference by using a fixed contraction sequence (back to front) which only requires one tensor orchestration operation that can be supported by a customized memory read/write mechanism and buffering scheme.

4.2 Optimal Contraction Sequence Search

We abstract every TNN layer inference into a tensor network graph. We conduct contraction sequence search to identify the optimal contraction sequence with the minimum computation and memory size requirement for processing.

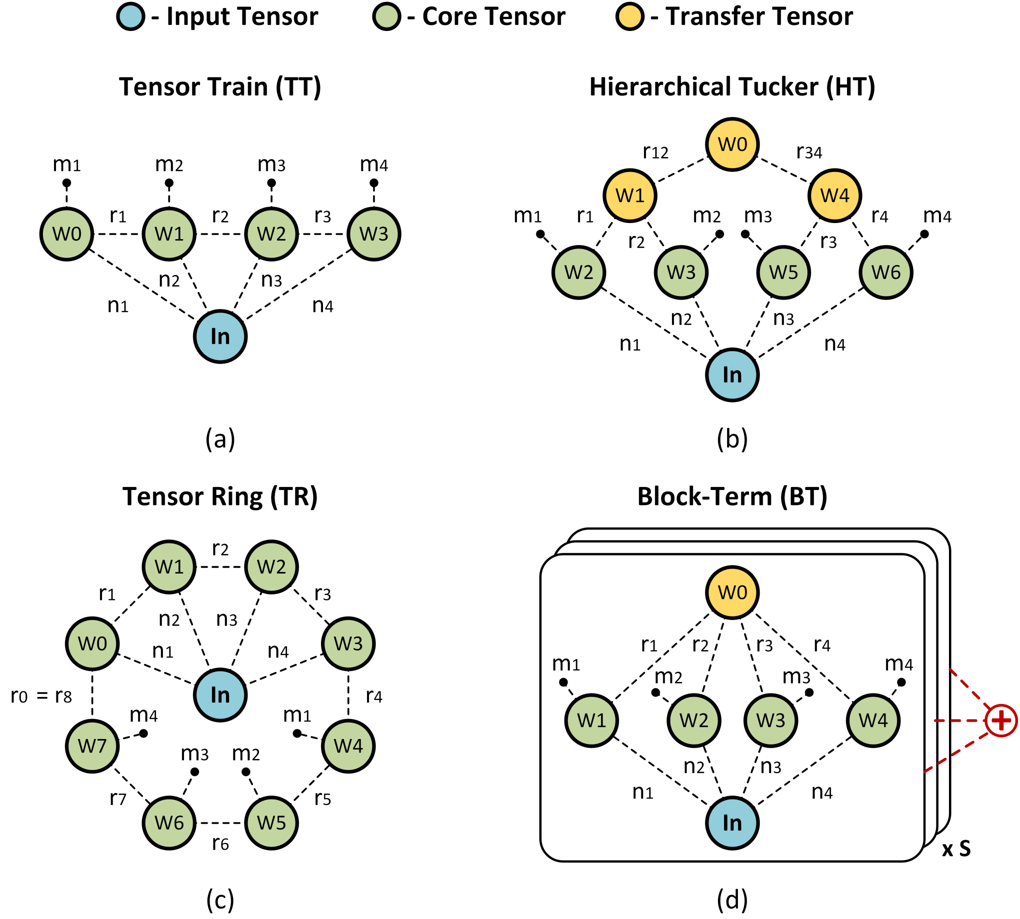


Figure 4.2: Tensor network graph representation of TNN layer with (a) TT, (b) HT, (c) TR, and (d) BT tensor decomposition methods, where the input tensor $\in \mathbb{R}^{n_1 \times n_2 \times n_3 \times n_4}$ and the output tensor $\in \mathbb{R}^{m_1 \times m_2 \times m_3 \times m_4}$.

4.2.1 Tensor Network Representation

We use the tensor network graph to represent a TNN layer inference. A tensor network graph provides an unified abstraction for the diversified TNN layers with different decomposition formats and input tensor specifications. Furthermore, the abstraction is agnostic to the permutation of a tensor which facilitates the analysis and exploration of different tensor orchestration possibilities.

Figure 4.2 illustrates common tensor decomposition methods in the tensor network graph representation. In a tensor network graph, each node represents a decomposed weight tensor or an input tensor. For a tensor represented by a node, every tensor

dimension is associated with an edge on the node, unless the dimension is 1. Two nodes are connected by an edge if they have one or more shared dimensions that can be contracted via a tensor contraction. A node has a loose edge if it has a free dimension which cannot be contracted with any other node. When contracting two tensors, the two nodes are merged into a single node that inherits all edges from both sides except the edge connecting them. After all contractions are done, the final node represents the output tensor and the loose edges represent the output dimensions.

4.2.2 Breadth-First Contraction Search

Different contraction sequences yield distinct computation and memory storage costs, and searching the optimal sequence is known as a NP-hard problem. Several approaches were proposed to search for the optimal contraction sequence. [140–142]. In this work, we focus on the breadth-first (BF) approach described in [140, 142].

For a tensor network graph with K initial nodes, a general breath-first approach constructs all sets of candidate nodes (Set_k), for $k \in [1, K]$. Set_1 contains the K initial nodes, and Set_k contains C_K^k candidate nodes resulted from contractions of k initial nodes. For each candidate node in a set, we can find a split to divide the node into two source nodes that can be contracted to obtain the candidate node. For instance, a node (012) in Set_3 may have 3 splits: $(0|12), (1|02)$, and $(2|01)$. To explore all possible contraction paths, the BF approach constructs the set of candidate nodes sequentially from $k = 1$ to K. For every candidate node in each set level k, the computation and memory storage costs from every split are calculated, and the split with the least cost is recorded. Once the final set Set_K is constructed, a backward tracing on the split with lowest computation or memory storage cost at each set level is performed to identify the contraction sequence with minimal cost.

In the context of TNN inference in this work, we limit the tensor contraction to be performed only between an initial or intermediate input tensor and a weight tensor.

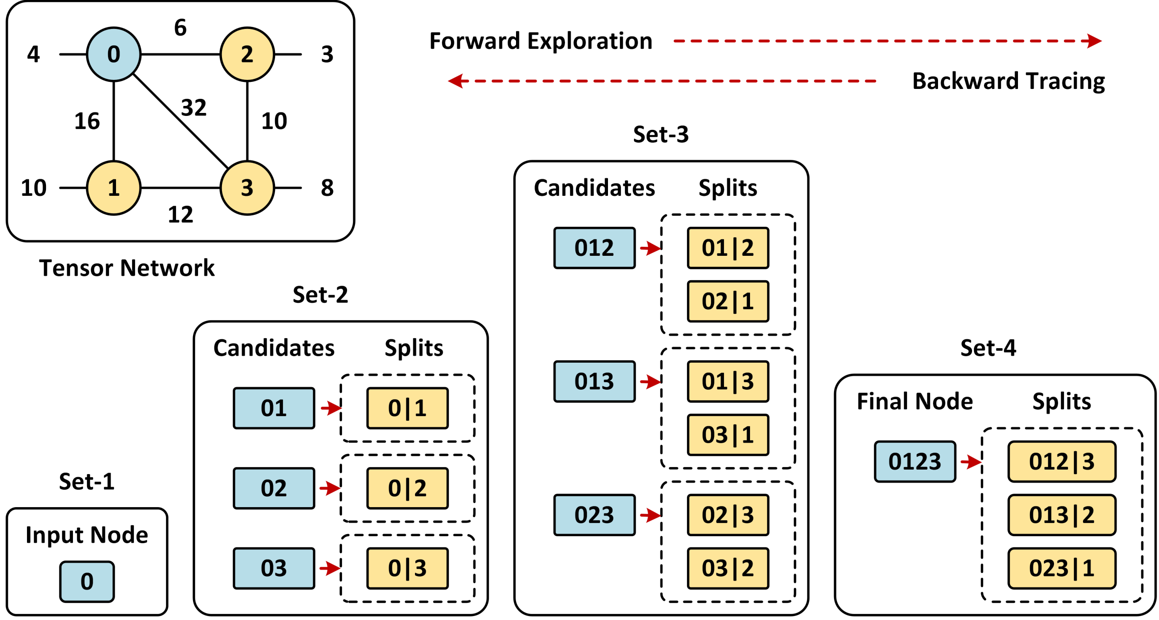


Figure 4.3: Illustration of breadth-first approach for optimal contraction sequence search.

This additional constraint allows the obtained contraction sequence to match the execution pattern of a tensor-by-tensor inference. In addition, it allows the removal of any candidate node that is not a result of a contraction on the initial and intermediate input tensor, thus reducing the search space significantly. Figure 4.3 shows an example of the optimal contraction sequence search on a given tensor network using the BF-approach with the input tensor constraint for TNN inference.

4.2.3 Contraction Sequence Analysis

In this work, we aim to find the optimal contraction sequence considering the hardware cost, including the total number of MAC operations and the required memory size to hold all intermediate tensors during inference. Figure 4.4 and Figure 4.5 demonstrate the benefits of identifying the optimal contraction sequence using an HT-TNN layer example. Figure 4.4 illustrates the optimal contraction sequence (Optimal) compared to two fixed contraction patterns (Pattern-1 and Pattern-2) used in [122, 123]. Figure 4.5(a) shows the contraction sequence space in terms of total

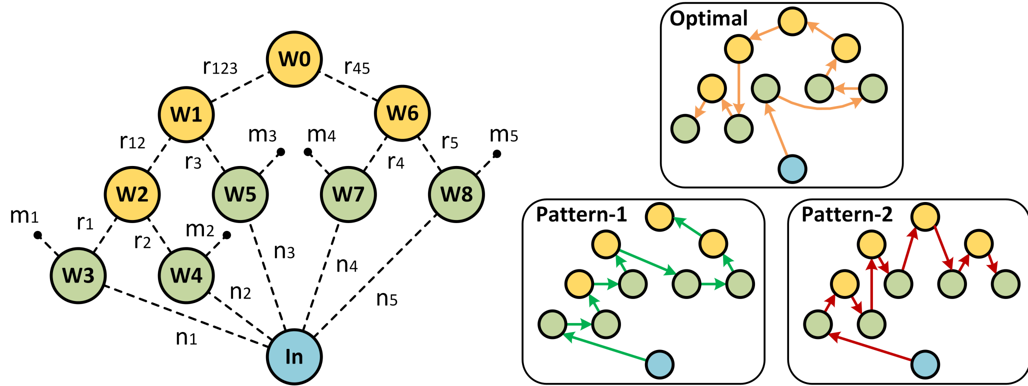
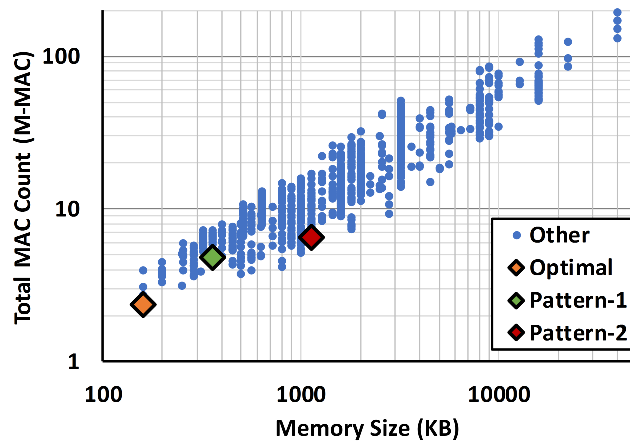


Figure 4.4: Illustration of contraction sequences for an HT-TNN layer inference example: optimal sequence (Optimal) and fixed contraction patterns used in previous works (Pattern-1, Pattern-2).



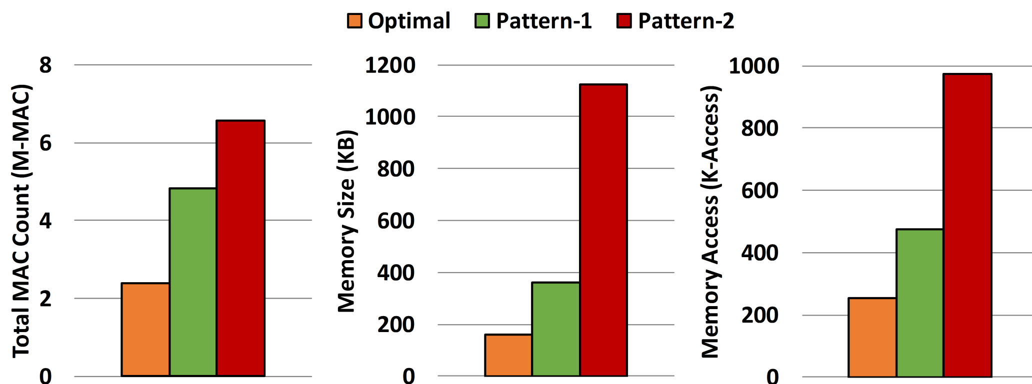


Figure 4.5: (a) Contraction sequence space in terms of total MAC operations and required memory size; (b) comparison of contraction sequences in total MAC operations, required memory size, and number of memory accesses.

MAC operations and required memory size for all intermediate tensors. By positioning the three contraction sequences on the search space, we can observe that even though Pattern-1 and Pattern-2 are well positioned in the entire space compared to most of the other possible sequences, there is still a significant gap from Optimal. Figure 4.5(b) compares the metrics for the three contraction sequences. The optimal sequence requires 2 to 3× fewer MAC operations, 2 to 7× less memory, and 2 to 4× less memory accesses, compared to fixed contraction sequences Pattern-1 and Pattern-2 used in [122, 123]. This demonstrates the importance of searching the optimal contraction sequence for improving TNN inference performance.

4.3 Hybrid Contraction Sequence Mapping

In the above, the contraction sequence obtained from the optimal search considers the number of MACs and the memory size and access, but not the tensor orchestration ability of the hardware. As a result, a tensor contraction from the optimal sequence may still be too complex and costly to be supported on hardware. A standard TNN hardware may rely on either inner product or outer product to perform tensor contraction, which limits the mapping options and costs expensive control and buffering for complex tensor orchestrations. In this section, we propose a hybrid mapping scheme that alternates between inner-product and outer-product mapping to eliminate complex tensor transpose operations completely, and greatly reduces the complexity of tensor orchestration to simpler tensor reshape and permute operations.

4.3.1 Limitation of Baseline Mapping

Existing DNN accelerator works use either inner product or outer product for VMM or MMM computation using a specific dataflow. Figure 4.6 shows the general execution of inner and outer product in an accelerator. The inner and outer product operations mainly differ in the data layout in the memory and the accumulation

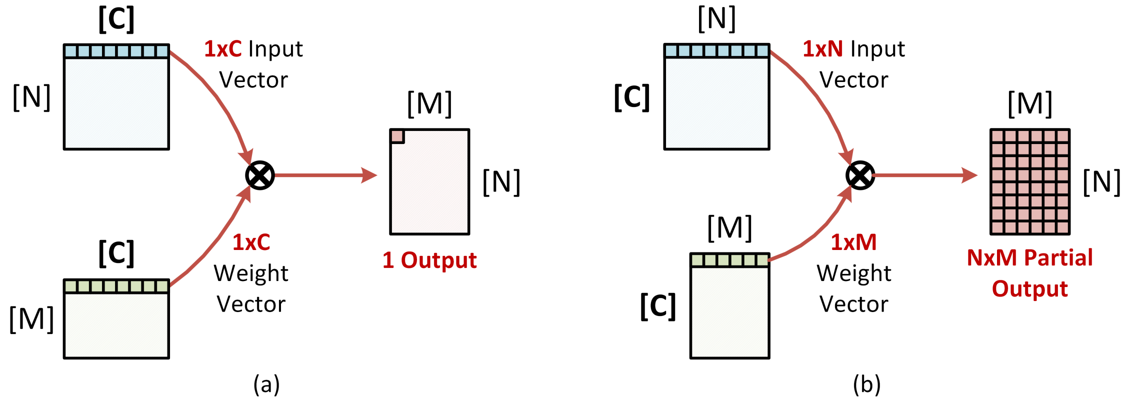


Figure 4.6: Illustration of (a) inner product and (b) outer product operation and memory layout.

pattern. In an inner product, vectors of size C are accessed from the memory for computation, where C represents the dimension for contraction, and the partial sums are accumulated spatially. The weight stationary (WS) dataflow is often used to support the inner-product operation [57, 143]. On the other hand, in an outer product, vectors of size N and M are accessed from the memories, where N and M are non-contractable dimensions, and the partial sums are accumulated temporally. The output stationary (OS) dataflow is often used to support the outer-product operation [5, 144, 145].

With a fixed dataflow to support either the inner or outer product, existing DNN accelerators are limited in options when mapping an arbitrary contraction sequence, leading to possible inefficiencies and overheads. For instance, Figure 4.7(a) illustrates a possible contraction sequence obtained after the optimal contraction sequence search and shows how existing DNN accelerators would map the contraction sequence for inner product or outer product execution as shown in Figure 4.7(b) and (c), respectively. For an inner-product design, if the tensor data is organized in the correct inner-product format, then it can begin the computation directly. However, if the data is organized in an incorrect format, then the inner-product design must first perform a complex orchestration on the tensor data in the memory that involves

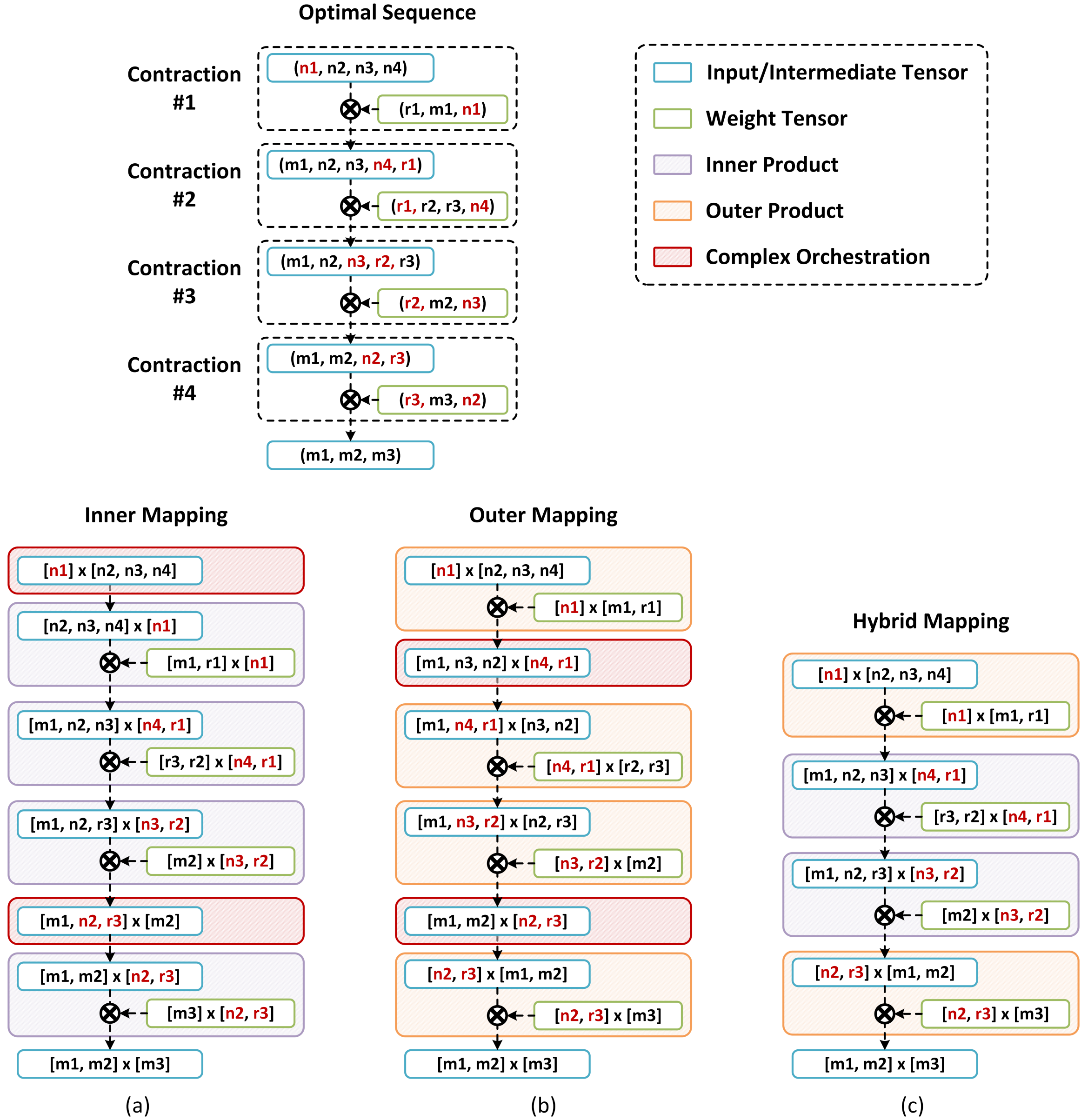


Figure 4.7: Example of the mapping options for an optimal contraction sequence: (a) inner-only mapping, (b) outer-only mapping, and (c) hybrid inner-outer mapping, where the dimensions for contraction are indicated in red.

a transpose operation. The same applies to an outer-product design. Overall, if only inner product or outer product is supported, the accelerator design must handle complex orchestration operations that require additional latency or large design overheads.

4.3.2 Hybrid Inner-Outer Product Mapping

To avoid the cost of complex tensor orchestration operations, we propose a hybrid mapping scheme that alternates between inner and outer product. We co-designed the architecture with the proposed mapping scheme to identify two mapping operations that can be used to map arbitrary contraction sequences.

More specifically, we first define two formats: 1) I-format for inner product, and 2) O-format for outer product as follows.

$$\begin{aligned} \text{I-Format} &: [i_F, i_1, i_2, \ldots, i_n] \times [C] \\ \text{O-Format} &: [i_F, i_1, \ldots i_{k-1}, C] \times [i_k, \ldots, i_n] \end{aligned}$$

where $i_F, i_1, \ldots, i_n$ and C are the tensor dimensions, i_F is the fixed dimension that does not participate in orchestration, $i_1, \ldots, i_n$ are the free dimensions for orchestration, and C is the contractable dimension. The $[\cdot] \times [\cdot]$ notation indicates the 2D matrix format when the tensor data is stored in memory.

Then, the two mapping operations of the proposed hybrid mapping can be formulated as follows.

$$\begin{aligned} \text{Input (I-Format)} &: [i_F, i_1, i_2, \ldots, i_n] \times [C] \\ \text{Weight (I-Format)} &: [w_1, w_2, \ldots, w_m] \times [C] \\ \text{Output} &: \{i_F, \Phi(i_1, i_2, \ldots, i_n, w_1, w_2, \ldots, w_{m-1}), w_m\} \end{aligned} \tag{4.8}$$

$$
\begin{aligned}
\text{Input (O-Format)} &: [i_F, i_1, \dots i_{k-1}, C] \times [i_k, \dots, i_n] \\
\text{Weight (O-Format)} &: [w_1, \dots w_{\ell-1}, C] \times [w_\ell, \dots, w_m] \\
\text{Output} &: \{i_F, \Phi(i_1, \dots, i_n, w_1, \dots, w_{m-1}), w_m\} \\
&\quad \text{or } \{i_F, \Phi(i_1, \dots, i_{n-1}, w_1, \dots, w_m), i_n\} \qquad (4.9)
\end{aligned}
$$

where $i_F, i_1, \dots, i_n, C$ are the input tensor dimensions, $w_1, \dots, w_m, C$ are the weight tensor dimensions, and the permute function $\Phi(\cdot)$ represents any possible permutation of the input and weight dimensions it encloses. Here, the output is presented in the reshape-free notation. For instance, arbitrary reshape operations can be applied to $\{a, b, c, d\}$ to convert it into $[a, b] \times [c, d]$ or $[a] \times [b, c, d]$ and so on, where the only requirement is the dimension ordering must remain the same after applying the reshape operation.

Eq. (4.8) and Eq. (4.9) are referred to as the inner mapping operation and the outer mapping operation, respectively. The inner mapping operation takes an input and weight both in I-format, and generates an output with the last weight dimension w_m as the last output dimension. The outer mapping operation takes an input and weight both in O-format, and generates an output that uses either of the last input dimension i_n or the last weight dimension w_m as the last output dimension. In both mapping operations, the free dimensions of input and weight that are not used as the last output dimension can be freely permuted for the output dimension. From the outputs obtained from the mapping operations, a new fixed dimension i'_F and a new contractable dimension C' are identified to form the input in either I-Format or O-Format for the next contraction.

With the mapping operations, the hybrid mapping scheme can avoid complex tensor orchestration operations in an arbitrary contraction sequence by alternating between inner product and outer product operations, as shown in Figure 4.7(d). Compared to existing inner-product or outer-product accelerator designs, the hybrid

mapping scheme can effectively reduce any possible latency overheads due to unnecessary orchestration operations and avoid costly designs with large control and area overheads.

The assumptions and heuristic for permute function $\Phi(\cdot)$ for hybrid mapping in TNN inference are listed as follows.

- Following the prior work [5], we assume that the weight tensors are pre-orchestrated by a host processor before loading on-chip for TNN inference. Therefore, the weights can be organized into either I-format or O-format according to the input format for a tensor contraction.

- For the first contraction in the sequence, the input tensor dimension must be a direct tensorization of the shape of the input vector for TNN inference; For the last contraction in the sequence, the output tensor dimension must be able to vectorize into the shape of the output vector of TNN inference.

- For the permute function $\Phi(\cdot)$ in our design, we permute the free dimensions contractable through inner product to the lower parts of the output dimension, whereas the free dimensions contractable through outer product are permuted to the upper parts of the output dimension. This heuristic limits the unnecessary moving of the free dimensions that may result in excessive memory traffic as the backend module (more details in Section 4.4).

4.4 TetriX System Architecture

The TetriX system architecture is presented in Figure 4.8. TetriX consists of a compute module, an index module, and a backend module. A system controller receives the instructions and configurations from the host processor, and coordinates the modules for processing. The compute module uses a configurable dataflow that allows it to perform inner and outer product operations using WS and OS dataflows.

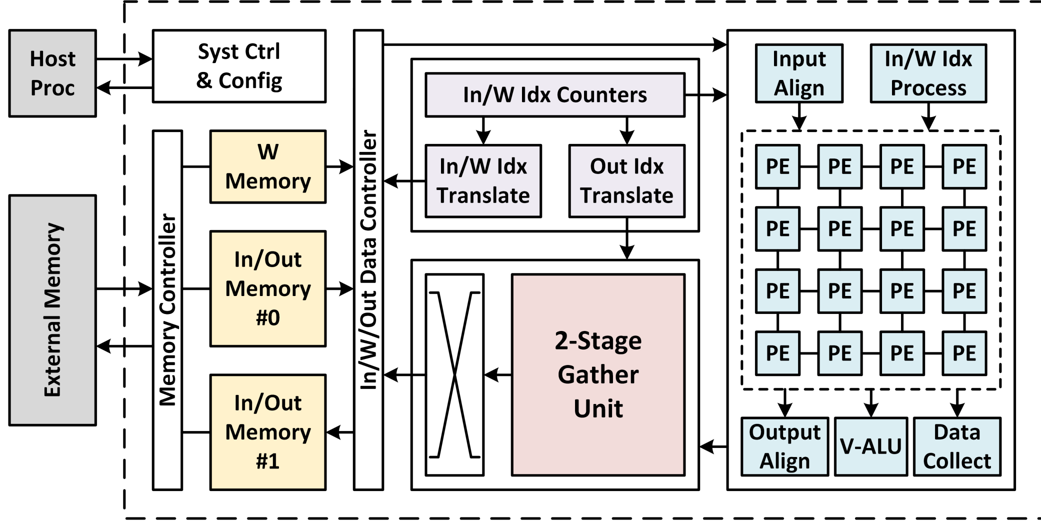


Figure 4.8: TetriX system architecture.

A 16×16 PE array is designed for the MMM operation. Pre-processing and post-processing units, i.e., input/output align, index processing, data collect, are used to assist the input streaming and output collection in both the WS and OS dataflow. A 32KB weight memory is used to store all the decomposed weight tensors for a TNN layer, and two 256KB tensor memories are used alternately to store the input and output tensor data for a tensor contraction operation. The memories are designed to be multi-banked to hold flexible input and output tensor shapes. The index module performs index translation to convert N-dimensional input and weight tensor indices into 2D memory addresses for input, weight, and output memories. The backend module performs hierarchical output gathering to form output data words and coordinates the writeback of data words into corresponding output memory bank. Overall, TetriX has a total of 256 PEs for 16b×16b MAC computation for a tensor contraction operation, and has a total on-chip memory storage of 544KB to store weight tensors and intermediate tensors during an end-to-end TNN layer inference.

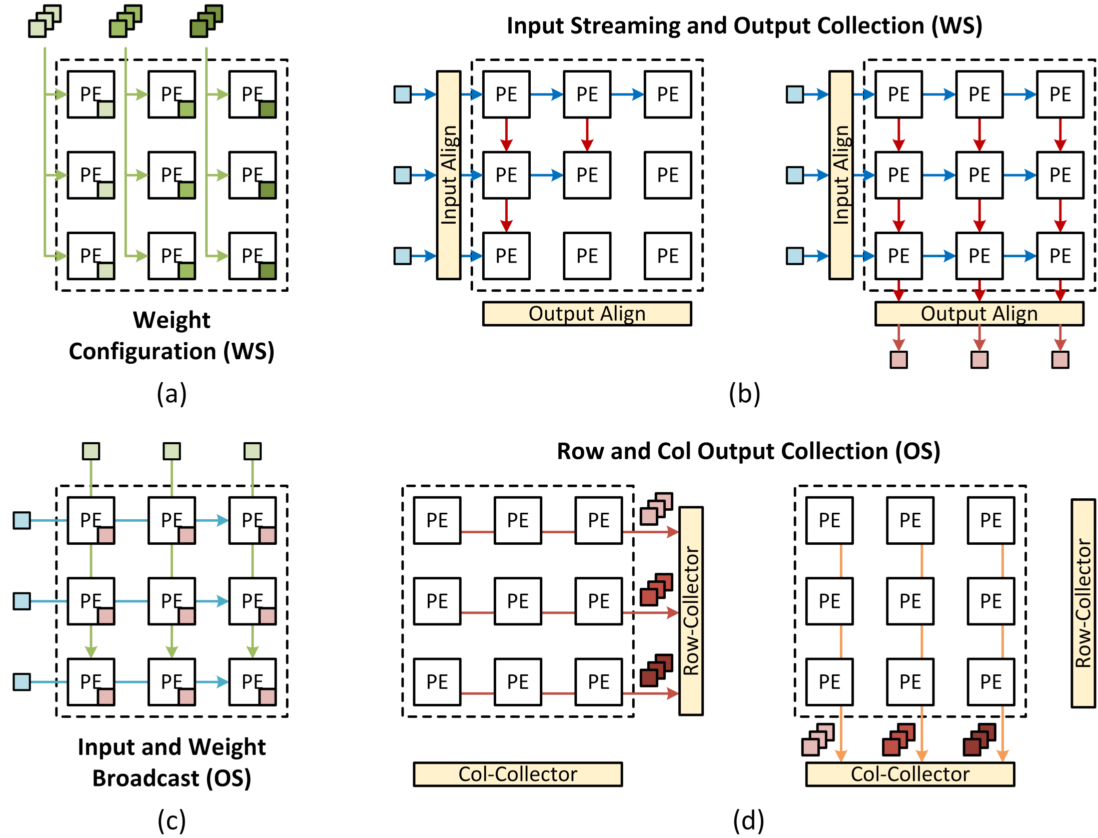


Figure 4.9: Illustration of WS and OS dataflows in TetriX architecture.

4.4.1 Configurable Stationary Dataflow

The compute module is designed to support both WS and OS dataflows for inner and outer product operations, respectively. Figure 4.9 illustrates the different steps of WS and OS dataflows in TetriX. In the WS dataflow, the compute module behaves as a systolic array to support inner-product operation. For processing, the weight vectors are first sent to the PE array columns for reuse (Figure 4.9(a)). The input vectors are streamed in to the PE array where the inputs are organized in a pipelined fashion for systolic processing [146]. Similarly, the pipelined outputs collected from the PE array are aligned to form the output vectors (Figure 4.9(b)). The WS dataflow presents an efficient processing scheme where data only propagates between neighboring PEs and accumulation occurs spatially during data propagation, requiring the least bandwidth and control overhead.

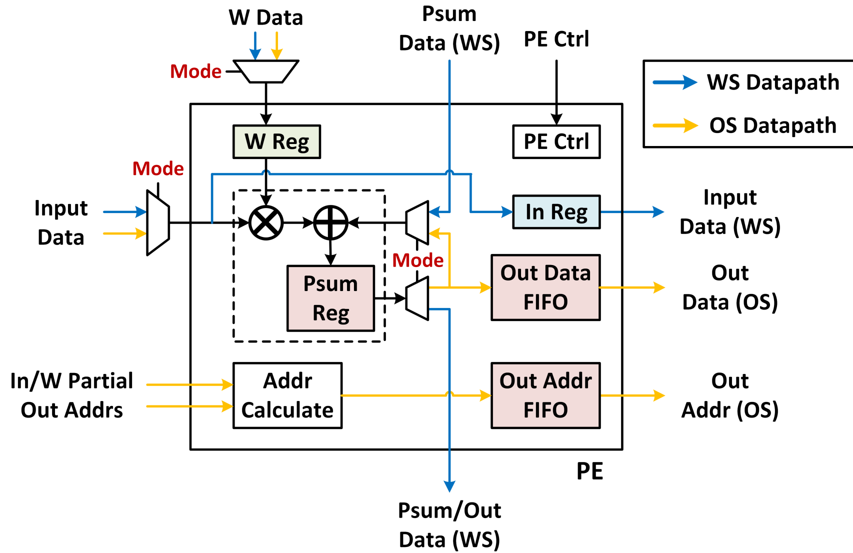


Figure 4.10: Illustration of integrated PE microarchitecture for WS/OS dataflows.

A PE holds a weight data that is reused for multiplication across multiple input data. During processing, a PE multiplies the input data from the left PE with the weight data it holds, then accumulates with the input psum data from the top PE to obtain the output psum data. Then, the input data is sent to the right PE and the output psum data is sent to the bottom PE. The PE datapath for WS dataflow is presented in blue in Figure 4.10.

In the OS dataflow, the compute module behaves as a spatial array to support the outer-product operation. For processing, the input vectors and weight vectors are broadcast across the PE array horizontally and vertically, respectively, and the output data are accumulated temporally at each PE (Figure 4.9(c)). The output data stored at each PE cannot be read at once due to bandwidth limitation. We propose two output collection modes: 1) row collection, and 2) column (col) collection. In the row collection mode, a row collector arbitrates between rows that are ready for collection, and reads the output data and memory addresses from the granted row (Figure 4.9(d)). Similarly for the col collection mode. The output collection modes allow the OS dataflow to have more flexibility and offer additional workload mapping

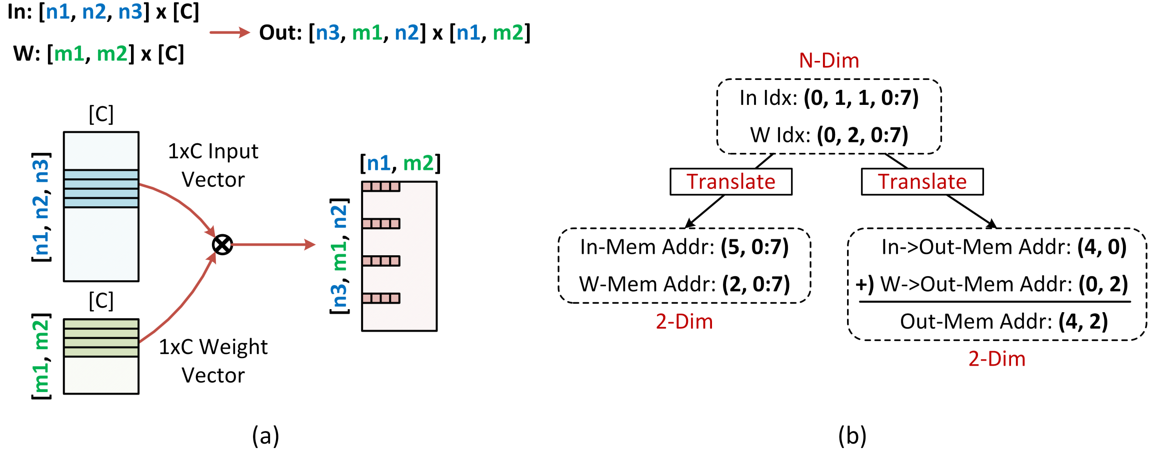


Figure 4.11: Illustration of (a) arbitrary permute and reshape operations on tensor data in memory, and (b) proposed index translation mechanism.

options for a better mapping performance. However, compared to the WS dataflow, the OS dataflow requires a more fine-grained control on the PEs and has larger design overheads due to the output memory address calculation and collector arbitration.

During processing, a PE computes the psum data using the received input and weight data from broadcast, then accumulates to the psum register. Once accumulated temporally, the psum is sent to the output data FIFO for output collection. In addition to the data, lower parts of input and weight tensor indices are sent to the compute module for index translation. They are first used to compute partial output memory addresses, which are broadcast along with the data to each PE to form a complete output memory address stored in the output address FIFO for output collection. The PE datapath for OS dataflow is presented in yellow in Figure 4.10.

4.4.2 Index Translation

With the hybrid mapping scheme, TetriX only has to support orchestration operations that consists of reshape and permute operations, as shown in Figure 4.11(a). However, these operations are still challenging to support since they have to operate on a N-dimensional tensor data that is stored in a 2D matrix form in the memory.

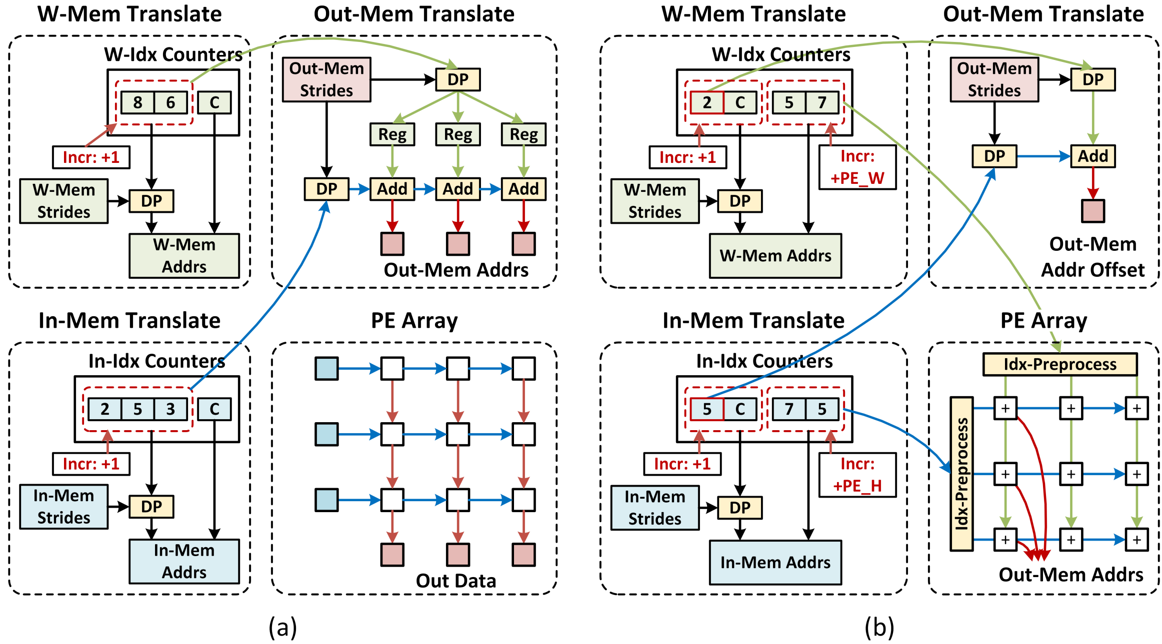


Figure 4.12: Microarchitecture of index translation for (a) WS dataflow and (b) OS dataflow.

We propose an index translation mechanism that uses grouped counters to keep track of the N-dimensional tensor indices of input and weight during processing, and translates those tensor indices into 2D memory addresses for input, weight, and output memory access by using dot-product operations with pre-configured memory strides. The index translation mechanism is illustrated in Figure 4.11(b).

Figure 4.12(a) and (b) illustrate the microarchitecture of index translation mechanism for WS and OS dataflow, respectively. In both WS and OS dataflow, the tensor indices from the input and weight index counters are used to calculate the input and weight memory address to access the N-dimensional tensor data stored in the input and weight memory, respectively. However, the output memory address is calculated differently in the WS and OS dataflow.

In the WS dataflow, the output translate unit receives the upper parts of weight and input index counters to calculate the output memory address, as illustrated in Figure 4.12(a). When configuring weight data to the PE array columns (Figure 4.9(a)),

the output translate units computes a partial output memory address from the weight tensor indices and stores it to the corresponding register for reuse. During input streaming (Figure 4.9(b)), the output translate unit computes the partial output memory address from the input tensor indices and propagates it in a systolic fashion across the registers to calculate the complete output memory addresses.

In the OS dataflow, the upper parts of weight and input index counters are sent to the output translate unit to calculate the output memory address offset, whereas the lower parts are sent to the PE array to calculate the output memory address, as illustrated in Figure 4.12(b). During processing, the output translate unit operates similarly as in the WS dataflow. The PE array first processes the weight and input tensor indices into partial output memory addresses, and then broadcasts them across the array along with the input and weight data (Figure 4.9(c)). Each PE calculates the output memory address with partial addresses from input and weight. During output collection (Figure 4.9(d)), the output memory addresses from PEs must be accumulated with the address offset from the output translate unit to form the final output memory addresses.

4.4.3 Output Gathering

Once the output data and their output memory address are computed by the compute and index modules, they need to be grouped into data words before being written to an output memory bank. We refer to the process of grouping contiguous data elements into data words as output gathering. Due to the flexible tensor dimensions, the output gathering needs to flexibly adapt to all possible scenarios. For example, with different output tensor dimensions, 16 output data obtained from the PE array can be gathered into 4 data words of length 4, or 5 data words of length 3 and 1 data word of length 1.

To handle the flexibility of output gathering, we propose a hierarchical output

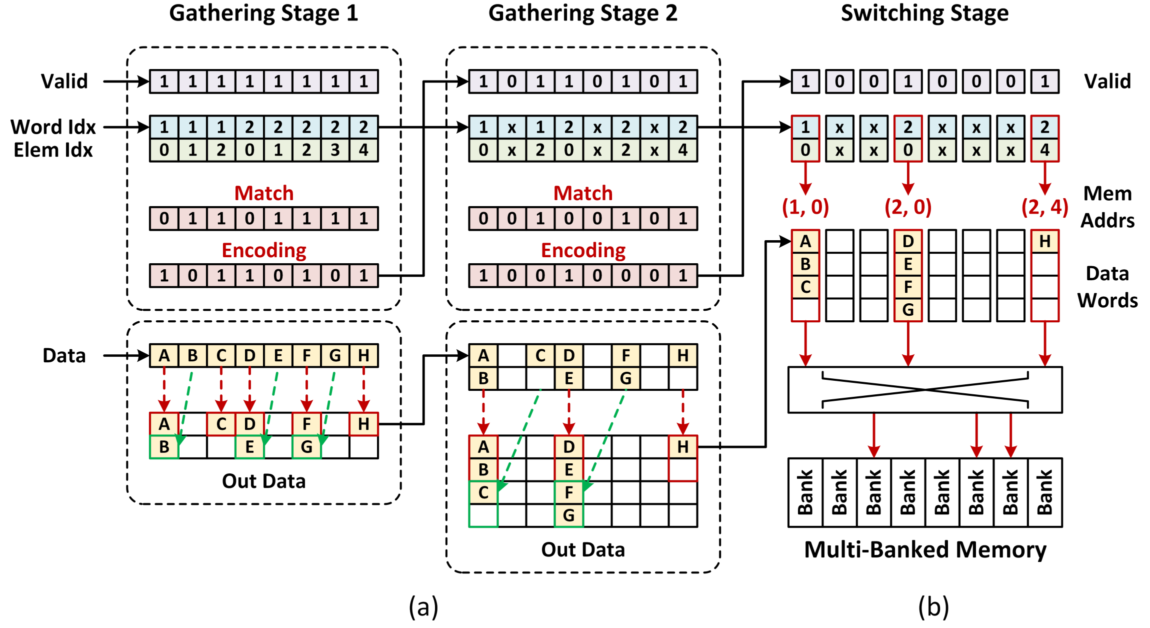


Figure 4.13: Illustration of the hierarchical output gathering mechanism using (a) two gathering stages followed by (b) a switching stage.

gathering mechanism as illustrated in Figure 4.13. The hierarchical output gathering performs gathering across multiple stages where each stage receives the output from the previous stage for gathering. In this approach, the d-th stage compares the memory address (word and element indices) of every data to its neighbor data at 2^{d-1}-distance on the left, to generate a match vector. With the match vector, an encoding vector is calculated and used to form data words of length 2^d. The boolean expressions used to calculate the match and encoding vectors are presented as follows.

$$\begin{aligned} \text{match}[i] &= \text{valid}[i] \ \& \ (\text{word index}[i] == \text{word index}[i - 2^{(d-1)}]) \\ \text{encoding}[i] &= \text{valid}[i] \ \& \ (\sim (\text{match}[i] \ \& \ \text{encoding}[i - 2^{(d-1)}]) \end{aligned}$$

Figure 4.13(a) and (b) illustrate two gathering stages and a switching stage, respectively. In Figure 4.13(a), stage 1 receives the initial data, and forms data words of length 2, then, stage 2 receives the output from stage 1 and forms data words of length 4. After the data words are formed, they are sent to the switching stage

where a switch uses the memory addresses to coordinate and write the data words to corresponding output memory banks, as shown in Figure 4.13(b).

4.5 Benchmarking and Evaluation

4.5.1 Evaluation Methodology

To evaluate TetriX, we collected the specifications of the TNN workloads from existing TNN literature [34, 112–128]. This collection is composed of more than 100 distinct TNN workloads of TT, HT, TR, BT, and Tucker formats where each TNN workload requires 3 to 13 tensor contractions, depending on the specification. Following the prior work [5], we used quantized inputs and weights of 16b fixed-point precision for the TNN workload inference in our design.

Cycle-accurate models were developed to simulate the behavior and analyze the performance of TetriX's architecture and dataflow. To compare the effectiveness of our hybrid mapping scheme, baseline designs using inner-only mapping and outer-only mapping are also evaluated using the same dataset and optimal contraction sequences as for Tetrix. We refer to the baseline designs as TetriX-Inner and TetriX-Outer. For benchmarking and comparing with prior accelerator works, a cycle-accurate model was also implemented for TIE accelerator. We also compare TetriX's performance to a commercial general-purpose GPU (Nvidia GTX-1080Ti).

4.5.2 Performance, and Mapping Efficacy Analysis

TetriX is designed with a total of 256 MACs and 544KB on-chip memory and runs at a 1.0 GHz frequency. It achieves a compute throughput of 508.3 and 350.2 GOPS for TT and HT workloads, respectively, where 1 MAC represents 2 OPs.

Figure 4.14 and Figure 4.15 show the performance comparison of TetriX to the mapping baselines TetriX-Inner and TetriX-Outer on TT, TR, HT, and BT workloads.

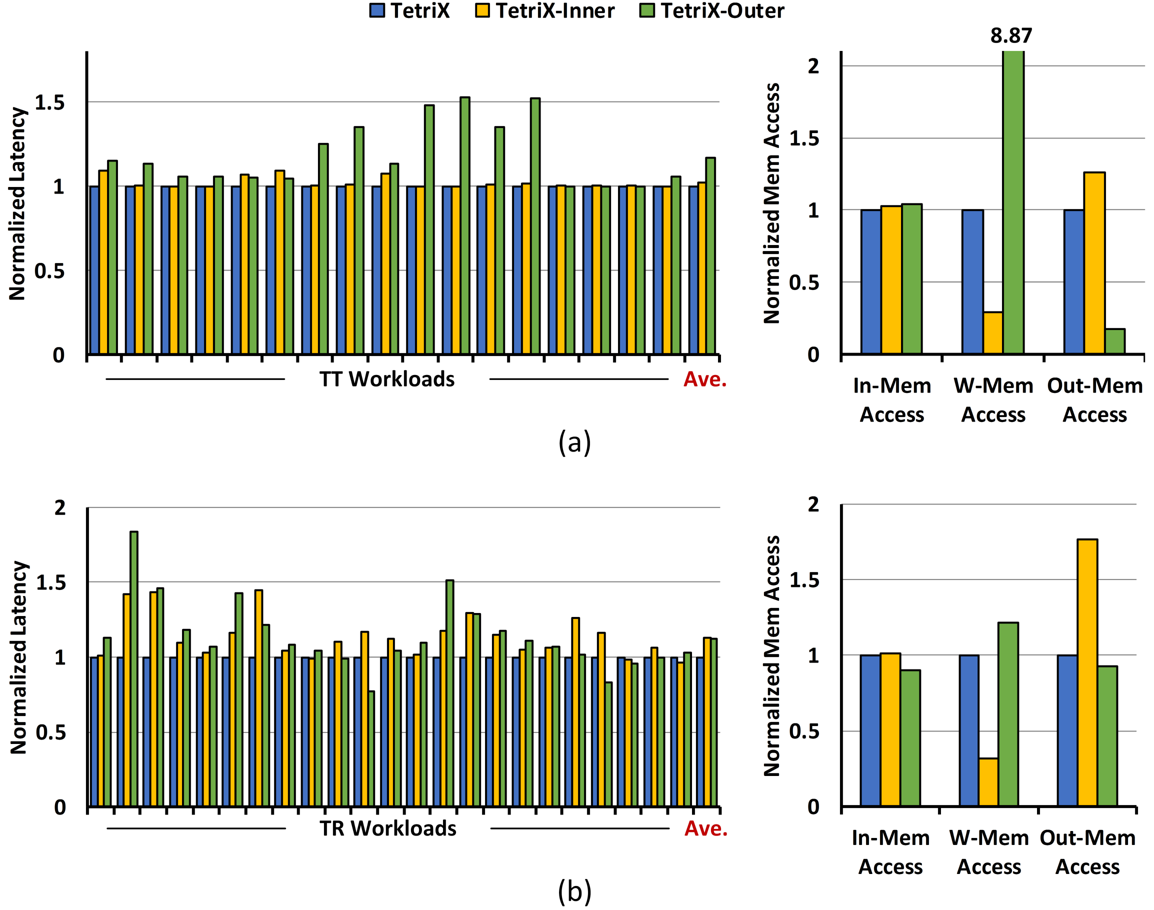


Figure 4.14: Performance comparison of hybrid mapping (TetriX), inner-only mapping (TetriX-Inner), and outer-only mapping (TetriX-Outer) for (a) TT and (b) TR workloads.

For each decomposition method, we show the normalized latency and memory accesses for input, weight, and output on-chip memories. Note that, all the mapping schemes run on the same optimal contraction sequence, thus the number of MAC operations and required memory sizes are identical.

Compared to the hybrid mapping, the inner-only and outer-only mapping schemes have to handle complex tensor orchestrations involving tensor transpose when the data layout is not organized correctly in memory for a contraction. However, without the explicit support for complex tensor orchestration operations in the TetriX architecture, additional read-orchestrate-write operations must be executed and cannot be

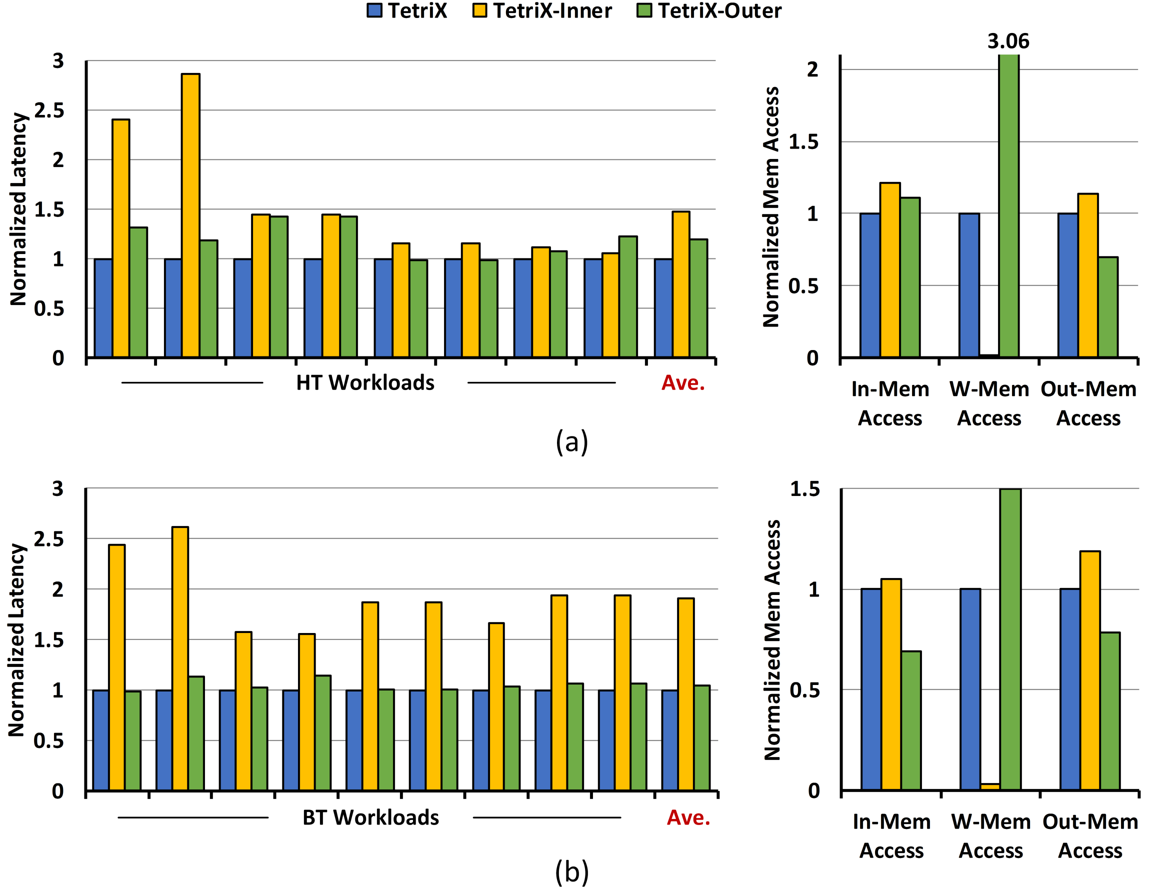


Figure 4.15: Performance comparison of hybrid mapping (TetriX), inner-only mapping (TetriX-Inner), and outer-only mapping (TetriX-Outer) for (a) HT and (b) BT workloads.

interleaved with the MMM processing, thus causing additional orchestration latency.

In Figure 4.14(a) and (b), TetriX is compared to the two mapping baselines for TT and TR workloads. Compared to TetriX-Inner, TetriX shows a latency improvement of 1.02× and 1.1× for TT and TR workloads, respectively. Compared to TetriX-Outer, TetriX shows a latency improvement of 1.2× and 1.1× for TT and TR workloads. The little latency improvement in TetriX is mainly due to the simplicity of TT and TR decomposition structures where the contraction sequence is often regular with little variation and does not require complex tensor orchestration throughout the workload inference.

On the other hand, Figure 4.15(a) and (b) show the comparison of TetriX to

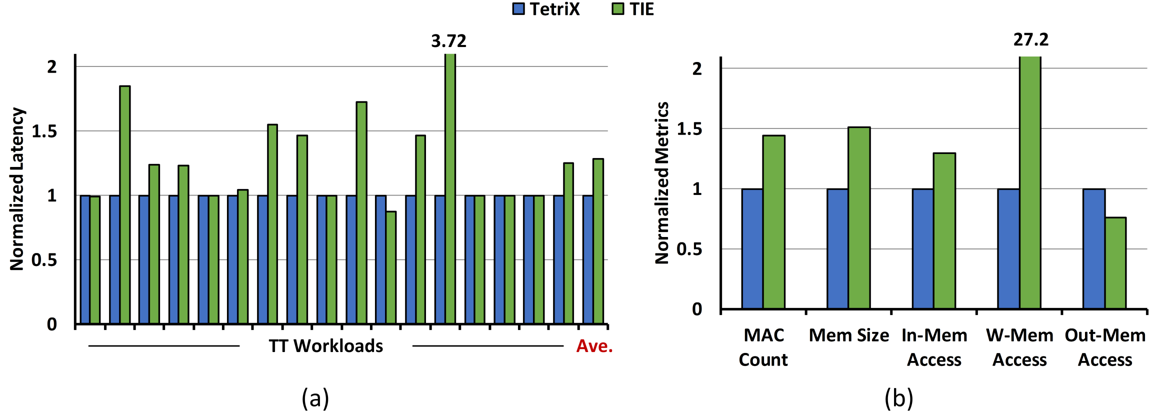


Figure 4.16: Comparison of the TetriX to TIE [5] in (a) normalized latency, and (b) total MAC count, maximum memory size, and number of memory accesses.

the two mapping baselines for HT and BT workloads. Different from TT and TR workloads, HT and BT workloads can take better advantage of the hybrid mapping scheme, resulting in more significant latency improvements of TetriX over the two mapping baselines. Compared to TetriX-Inner, TetriX shows a latency improvement of 1.5× and 1.9× for HT and BT workloads, respectively. Compared to TetriX-Outer, TetriX shows a latency improvement of 1.2× and 1.1× for HT and BT workloads, respectively. A larger improvement can be achieved because of the complex structures in HT and BT decomposition methods where the contraction sequence is highly dynamic and arbitrary, requiring frequent complex tensor orchestrations and causing additional orchestration latency for the mapping baselines.

4.5.3 Performance Comparison

In Figure 4.16, we compared the performance of TetriX to a prior accelerator work, TIE [5], which can only support the TT decomposition method. In our optimal sequence search for the TT decomposition, we observe a narrower contraction sequence space due to the simple and regular decomposition structure. The optimal search mostly results in two contraction patterns: 1) forward processing from the first core tensor to the last core tensor, and 2) backward processing from the last core tensor

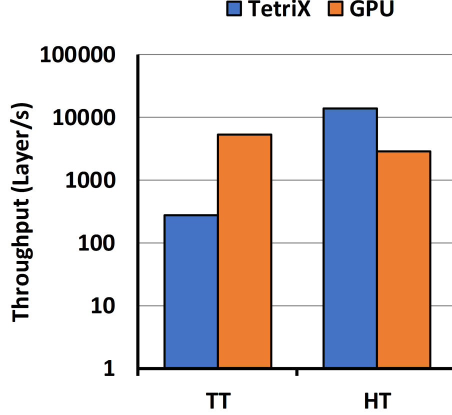


Figure 4.17: End-to-end throughput comparison for TT and HT workloads.

to the first core tensor. TIE adopted the backward processing contraction pattern for all TT workloads [5]. For TetriX, if the optimal contraction sequence overlaps with TIE's contraction pattern, a similar performance is achieved. However, if the optimal contraction sequence differs from the TIE's contraction pattern, TetriX outperforms TIE with significant latency improvement of up to 3.9×, as shown in Figure 4.16(a).

Overall, TetriX requires on average 1.4× less MAC operations and 1.5× less memory size, and achieves on average 1.3× latency improvement across the TT workloads of our dataset, as shown in Figure 4.16(b). With the reduction of memory size thanks to the optimal contraction sequence, more end-to-end TNN workloads can be fully mapped on TetriX for on-chip processing without incurring external memory access.

TetriX is also compared to a commercial general-purpose GPU (Nvidia GTX-1080Ti). The GPU has 3,584 cores for massively parallel computation, whereas TetriX is only designed with 256 MACs. Figure 4.17 compares the throughput of TetriX and GPU when evaluated on end-to-end TT and HT workloads. When running on TT workloads with simple and regular structures, the GPU achieves a 19× higher throughput compared to TetriX. In contrast, TetriX achieves a 4.8× higher throughput compared to the GPU for HT workloads with more complex structures. This demonstrates the effectiveness of the hybrid mapping scheme in alleviating the orchestration overheads for complex-structured contraction sequences.

4.6 Summary

We present TetriX, an architecture and mapping co-design for efficient and flexible TNN processing. TetriX can flexibly adapt to arbitrary tensor contraction operations required in optimal contraction sequences to achieve higher performance and efficiency for TNN inference. The optimal contraction sequence shows 3× and 7× less computation and memory over contraction patterns used in previous works. A hybrid mapping scheme is proposed to eliminate complex tensor orchestrations by alternating between inner and outer product operations. The hybrid mapping shows up to 1.9× performance improvement compared to the baseline mapping schemes for complex tensor decomposition methods. Configurable dataflow, index translation, and output gathering mechanisms are designed to support flexible tensor permute, reshape and matrix-matrix multiplication operations for arbitrary tensor contractions. Compared to previous accelerator works, TetriX supports all tensor decomposition methods used in TNNs with varying orders, dimensions, ranks. Compared to the state-of-the-art accelerator, TetriX achieves up to 3.9× and on average 1.3× latency improvement for TT workloads across our collected dataset. Lastly, compared to a general-purpose GPU, TetriX shows a 4.8× throughput improvement for complex decomposition methods, i.e., HT workloads, across our collected dataset.

CHAPTER V

Conclusion and Outlook

In this dissertation, three accelerator architectures and designs are presented to address the computation challenges by exploiting model compression characteristics and data orchestration techniques.

SNAP addresses the computation challenges in unstructured sparse DNN processing. With the design of AIM unit, the sequence decoder, and the two-level psum reduction, SNAP can achieve higher parallelism and efficiency for the irregular computation in unstructured sparse NNs. A 16nm SNAP test chip demonstrates an effective energy efficiency of up to 21.55 TOPS/W for sparse workloads and 3.61 TOPS/W for a pruned ResNet-50.

Point-X addresses the inefficiencies in graph-based point-cloud NN processing. The SLA clustering extracts fine-grained and coarse-grained spatial locality to achieve higher data reuse and parallelism while having less data movement during processing. The SBFS graph traversal enables a speedup for SLA clustering execution. The chain NoC reduces design complexity and improves efficiency for data exchange between CTiles. A Point-X design in 28nm can achieve a throughput of 1307.1 Inf./s and an energy efficiency of 604.5 Inf./J when running on DGCNN.

TetriX explores the co-design opportunities in architecture and workload mapping to address the flexibility and performance challenges in TNN processing. To

improve performance, the optimal contraction sequence with minimized computation requirement is identified, then mapped to the processing architecture by using our proposed hybrid inner-outer mapping scheme. TetriX is designed with the configurable dataflow, the index translation, and the output gathering mechanisms to support flexible tensor contraction operations. TetriX can support all existing decomposition methods used in TNNs and shows up to 3.9× performance improvement compared to the prior work.

Throughout these accelerator architecture works, optimization techniques across algorithm, architecture, and microarchitecture levels were investigated in order to obtain the best performance and efficiency for compressed NNs and point-cloud networks. These accelerator designs and optimization techniques can guide the design of future algorithms and more effective network models.

Research of accelerator architecture and design is always at the intersection of multiple research fields ranging from algorithm, software, compiler, architecture, circuit design, device, and many more. The research presented in this dissertation attempts to cover the whole processing system but the emphasis is placed on the novel aspects of the design. The discussions are not all-inclusive, and many possible research opportunities have not been fully considered. Here, we briefly outline some directions related to this work.

On the algorithm level: 1) joint model compression techniques on NNs for extreme compression; and 2) co-design of network operations and hardware features of an accelerator.

On the architecture level: 1) architecture design for domain-specific operations that broadens the hardware's flexibility to support more applications; and 2) full system-on-chip (SoC) integration that includes a general-purpose processor, bus interface, external memory, system intellectual properties (IPs), and the accelerator.

On the microarchitecture and circuit level: 1) 2.5D and 3D chiplet integration as

a scalable solution; 2) mix-design with novel technologies, like processing-in-memory using resistive random-access memory (RRAM) or SRAM; and 3) design with non-volatile memories, like magnetoresistive random-access memory (MRAM), that offer different design tradeoffs.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] S. Bianco, R. Cadene, L. Celona, and P. Napoletano, “Benchmark analysis of representative deep neural network architectures,” *IEEE Access*, vol. 6, pp. 64 270–64 277, 2018.

[2] G. Menghani, “Efficient deep learning: A survey on making deep learning models smaller, faster, and better,” *arXiv preprint arXiv:2106.08962*, 2021.

[3] W. L. Hamilton, R. Ying, and J. Leskovec, “Inductive representation learning on large graphs,” in *Proceedings of the Advances in Neural Information Processing Systems (NIPS)*, vol. 30, 2017, pp. 1025–1035.

[4] Y. Wang, Y. Sun, Z. Liu, S. E. Sarma, M. M. Bronstein, and J. M. Solomon, “Dynamic graph CNN for learning on point clouds,” *ACM Transactions on Graphics*, vol. 38, no. 5, 2019.

[5] C. Deng, F. Sun, X. Qian, J. Lin, Z. Wang, and B. Yuan, “TIE: Energy-efficient tensor train-based inference engine for deep neural network,” in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2019, pp. 264–277.

[6] Y. LeCun, Y. Bengio, and G. Hinton, “Deep learning,” *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[7] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” in *Proceedings of the Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 770–778.

[8] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, “Rethinking the Inception architecture for computer vision,” in *Proceedings of the Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 2818–2826.

[9] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly *et al.*, “An image is worth 16x16 words: Transformers for image recognition at scale,” *arXiv preprint arXiv:2010.11929*, 2020.

[10] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, “Attention is all you need,” in *Proceedings of the Advances in Neural Information Processing Systems (NIPS)*, 2017, pp. 6000–6010.

[11] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[12] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," in *Proceedings of the Advances in Neural Information Processing Systems (NIPS)*, 2020, pp. 1877–1901.

[13] A. Canziani, A. Paszke, and E. Culurciello, "An analysis of deep neural network models for practical applications," *arXiv preprint arXiv:1605.07678*, 2017.

[14] Y. Guo, H. Wang, Q. Hu, H. Liu, L. Liu, and M. Bennamoun, "Deep learning for 3D point clouds: A survey," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 43, no. 12, pp. 4338–4364, 2021.

[15] E. Ahmed, A. Saint, A. E. R. Shabayek, K. Cherenkova, R. Das, G. Gusev, D. Aouada, and B. E. Ottersten, "Deep learning advances on different 3D data representations: A survey," *arXiv preprint arXiv:1808.01462*, 2018.

[16] J. Zhang, X. Zhao, Z. Chen, and Z. Lu, "A review of deep learning-based semantic segmentation for point cloud," *IEEE Access*, vol. 7, pp. 179 118–179 133, 2019.

[17] Y. Xie, T. Jiaojiao, and X. X. Zhu, "Linking points with labels in 3D: A review of point cloud semantic segmentation," *IEEE Geoscience and Remote Sensing Magazine*, vol. 8, no. 4, pp. 38–59, 2020.

[18] M. M. Bronstein, J. Bruna, Y. LeCun, A. Szlam, and P. Vandergheynst, "Geometric deep learning: Going beyond Euclidean data," *IEEE Signal Processing Magazine*, vol. 34, no. 4, pp. 18–42, 2017.

[19] S. Zhang, H. Tong, J. Xu, and R. Maciejewski, "Graph convolutional networks: A comprehensive review," *Springer Computational Social Networks*, vol. 6, no. 1, 2019.

[20] J. Cheng, P.-S. Wang, G. Li, Q.-H. Hu, and H.-q. Lu, "Recent advances in efficient computation of deep convolutional neural networks," *Springer Frontiers of Information Technology & Electronic Engineering*, vol. 19, pp. 64–77, 2018.

[21] L. Deng, G. Li, S. Han, L. Shi, and Y. Xie, "Model compression and hardware acceleration for neural networks: A comprehensive survey," *Proceedings of the IEEE*, vol. 108, no. 4, pp. 485–532, 2020.

[22] Y. Cheng, D. Wang, P. Zhou, and T. Zhang, "Model compression and acceleration for deep neural networks: The principles, progress, and challenges," *IEEE Signal Processing Magazine*, vol. 35, no. 1, pp. 126–136, 2018.

[23] S. Dai, R. Venkatesan, M. Ren, B. Zimmer, W. Dally, and B. Khailany, "VS-Quant: Per-vector scaled quantization for accurate low-precision neural network inference," in *Proceedings of the Conference on Machine Learning and Systems (MLSys)*, 2021, pp. 873–884.

[24] D. D. Kalamkar, D. Mudigere, N. Mellempudi, D. Das, K. Banerjee, S. Avancha, D. T. Vooturi, N. Jammalamadaka, J. Huang, H. Yuen, J. Yang, J. Park, A. Heinecke, E. Georganas, S. Srinivasan, A. Kundu, M. Smelyanskiy, B. Kaul, and P. Dubey, "A study of BFLOAT16 for deep learning training," *arXiv preprint arXiv:1905.12322*, 2019.

[25] 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.

[26] S. Han, J. Pool, J. Tran, and W. Dally, "Learning both weights and connections for efficient neural network," in *Proceedings of the Advances in Neural Information Processing Systems (NIPS)*, 2015, pp. 1135–1143.

[27] S. Anwar, K. Hwang, and W. Sung, "Structured pruning of deep convolutional neural networks," *ACM Journal on Emerging Technologies in Computing Systems*, vol. 13, no. 3, 2017.

[28] T. Zhang, S. Ye, K. Zhang, J. Tang, W. Wen, M. Fardad, and Y. Wang, "A systematic DNN weight pruning framework using alternating direction method of multipliers," in *Proceedings of the European Conference on Computer Vision (ECCV)*, 2018, pp. 184–199.

[29] E. Denton, W. Zaremba, J. Bruna, Y. LeCun, and R. Fergus, "Exploiting linear structure within convolutional networks for efficient evaluation," in *Proceedings of the Advances in Neural Information Processing Systems (NIPS)*, 2014, pp. 1269–1277.

[30] T. N. Sainath, B. Kingsbury, V. Sindhwani, E. Arisoy, and B. Ramabhadran, "Low-rank matrix factorization for deep neural network training with high-dimensional output targets," in *Proceedings of the International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2013, pp. 6655–6659.

[31] J. D. Carroll and J.-J. Chang, "Analysis of individual differences in multidimensional scaling via an N-way generalization of "Eckart-Young" decomposition," *Springer Psychometrika*, vol. 35, no. 3, pp. 283–319, 1970.

[32] L. R. Tucker, "Some mathematical notes on three-mode factor analysis," *Springer Psychometrika*, vol. 31, no. 3, pp. 279–311, 1966.

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

[34] A. Novikov, D. Podoprikhin, A. Osokin, and D. Vetrov, “Tensorizing neural networks,” in *Proceedings of the Advances in Neural Information Processing Systems (NIPS)*, 2015, pp. 442–450.

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

[36] B. Perozzi, R. Al-Rfou, and S. Skiena, “DeepWalk: Online learning of social representations,” in *Proceedings of the International Conference on Knowledge Discovery and Data Mining (KDD)*, 2014, pp. 701–710.

[37] J.-F. Zhang, C.-E. Lee, C. Liu, Y. S. Shao, S. W. Keckler, and Z. Zhang, “SNAP: An efficient sparse neural acceleration processor for unstructured sparse deep neural network inference,” *IEEE Journal of Solid-State Circuits*, vol. 56, no. 2, pp. 636–647, 2021.

[38] ——, “SNAP: A 1.67-21.55 TOPS/W sparse neural acceleration processor for unstructured sparse deep neural network inference in 16nm CMOS,” in *Proceedings of the Symposium on VLSI Circuits (VLSI)*, 2019, pp. 306–307.

[39] J.-F. Zhang and Z. Zhang, “Point-X: A spatial-locality-aware architecture for energy-efficient graph-based point-cloud deep learning,” in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2021, pp. 1078–1090.

[40] ——, “Exploration of energy-efficient architecture for graph-based point-cloud deep learning,” in *Proceedings of the Workshop on Signal Processing Systems (SiPS)*, 2021, pp. 260–264.

[41] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, “ImageNet: A large-scale hierarchical image database,” in *Proceedings of the Conference on Computer Vision and Pattern Recognition (CVPR)*, 2009, pp. 248–255.

[42] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “ImageNet classification with deep convolutional neural networks,” in *Proceedings of the Advances in Neural Information Processing Systems (NIPS)*, 2012, pp. 1097–1105.

[43] K. Simonyan and A. Zisserman, “Very deep convolutional networks for large-scale image recognition,” *International Conference on Learning Representations (ICLR)*, 2015.

[44] 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 Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015, pp. 1–9.

[45] R. Girshick, J. Donahue, T. Darrell, and J. Malik, “Rich feature hierarchies for accurate object detection and semantic segmentation,” in *Proceedings of the Conference on Computer Vision and Pattern Recognition (CVPR)*, 2014, pp. 580–587.

[46] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, “You only look once: Unified, real-time object detection,” in *Proceedings of the Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 779–788.

[47] J. Long, E. Shelhamer, and T. Darrell, “Fully convolutional networks for semantic segmentation,” in *Proceedings of the Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015, pp. 3431–3440.

[48] 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, 2017.

[49] J. Albericio, P. Judd, T. H. Hetherington, T. M. Aamodt, N. D. E. Jerger, and A. Moshovos, “Cnvlutin: Ineffectual-neuron-free deep neural network computing,” in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2016, pp. 1–13.

[50] 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 *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2016, pp. 1–12.

[51] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, “EIE: Efficient inference engine on compressed deep neural network,” in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2016, pp. 243–254.

[52] 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 International Symposium on Computer Architecture (ISCA)*, 2017, pp. 27–40.

[53] Z. Yuan, J. Yue, H. Yang, Z. Wang, J. Li, Y. Yang, Q. Guo, X. Li, M.-F. Chang, H. Yang, and Y. Liu, “STICKER: A 0.41-62.1 TOPS/W 8bit neural network processor with multi-sparsity compatible convolution arrays and online tuning acceleration for fully connected layers,” in *Proceedings of the Symposium on VLSI Circuits (VLSI)*, 2018, pp. 33–34.

[54] Z. Yuan, Y. Liu, J. Yue, Y. Yang, J. Wang, X. Feng, J. Zhao, X. Li, and H. Yang, “STICKER: An energy-efficient multi-sparsity compatible accelerator for convolutional neural networks in 65-nm CMOS,” *IEEE Journal of Solid-State Circuits*, vol. 55, no. 2, pp. 465–477, 2020.

[55] 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.

[56] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2016, pp. 367–379.

[57] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2017, pp. 1–12.

[58] Y.-H. Chen, T. Krishna, J. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," in *Proceedings of the International Solid-State Circuits Conference (ISSCC)*, 2016, pp. 262–263.

[59] B. Moons, R. Uytterhoeven, W. Dehaene, and M. Verhelst, "Envision: A 0.26-to-10TOPS/W subword-parallel dynamic-voltage-accuracy-frequency-scalable convolutional neural network processor in 28nm FDSOI," in *Proceedings of the International Solid-State Circuits Conference (ISSCC)*, 2017, pp. 246–247.

[60] J. Albericio, A. Delmás, P. Judd, S. Sharify, G. O'Leary, R. Genov, and A. Moshovos, "Bit-pragmatic deep neural network computing," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2017, pp. 382–394.

[61] J. Lee, C. Kim, S. Kang, D. Shin, S. Kim, and H.-J. Yoo, "UNPU: An energy-efficient deep neural network accelerator with fully variable weight bit precision," *IEEE Journal of Solid-State Circuits*, vol. 54, no. 1, pp. 173–185, 2018.

[62] C.-E. Lee, Y. S. Shao, J.-F. Zhang, A. Parashar, J. Emer, S. W. Keckler, and Z. Zhang, "Stitch-X: An accelerator architecture for exploiting unstructured sparsity in deep neural networks," *Conference on Machine Learning and Systems (MLSys)*, 2018.

[63] Y. Li and J. Ibanez-Guzman, "Lidar for autonomous driving: The principles, challenges, and trends for automotive lidar and perception systems," *IEEE Signal Processing Magazine*, vol. 37, no. 4, pp. 50–61, 2020.

[64] H. Su, S. Maji, E. Kalogerakis, and E. Learned-Miller, "Multi-view convolutional neural networks for 3D shape recognition," in *Proceedings of the International Conference on Computer Vision (ICCV)*, 2015, pp. 945–953.

[65] T. Yu, J. Meng, and J. Yuan, "Multi-view harmonized bilinear network for 3d object recognition," in *Proceedings of the Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018, pp. 186–194.

[66] C. R. Qi, H. Su, M. Nießner, A. Dai, M. Yan, and L. J. Guibas, "Volumetric and multi-view CNNs for object classification on 3D data," in *Proceedings of the Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 5648–5656.

[67] D. Maturana and S. Scherer, "VoxNet: A 3D convolutional neural network for real-time object recognition," in *Proceedings of the International Conference on Intelligent Robots and Systems (IROS)*, 2015, pp. 922–928.

[68] G. Riegler, A. O. Ulusoy, and A. Geiger, "OctNet: Learning deep 3D representations at high resolutions," in *Proceedings of the Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017, pp. 6620–6629.

[69] P.-S. Wang, Y. Liu, Y.-X. Guo, C.-Y. Sun, and X. Tong, "O-CNN: Octree-based convolutional neural networks for 3D shape analysis," *ACM Transactions on Graphics*, vol. 36, no. 4, 2017.

[70] C. R. Qi, H. Su, K. Mo, and L. J. Guibas, "PointNet: Deep learning on point sets for 3D classification and segmentation," in *Proceedings of the Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017, pp. 77–85.

[71] C. R. Qi, L. Yi, H. Su, and L. J. Guibas, "PointNet++: Deep hierarchical feature learning on point sets in a metric space," in *Proceedings of the Advances in Neural Information Processing Systems (NIPS)*, 2017, pp. 5105–5114.

[72] M. Simonovsky and N. Komodakis, "Dynamic edge-conditioned filters in convolutional neural networks on graphs," in *Proceedings of the Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017, pp. 29–38.

[73] Y. Shen, C. Feng, Y. Yang, and D. Tian, "Mining point cloud local structures by kernel correlation and graph pooling," in *Proceedings of the Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018, pp. 4548–4557.

[74] K. Zhang, M. Hao, J. Wang, C. W. de Silva, and C. Fu, "Linked dynamic graph CNN: Learning on point cloud via linking hierarchical features," *arXiv preprint arXiv:1904.10014*, 2019.

[75] C. Chen, G. Li, R. Xu, T. Chen, M. Wang, and L. Lin, "ClusterNet: Deep hierarchical cluster network with rigorously rotation-invariant representation for point cloud analysis," in *Proceedings of the Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019, pp. 4989–4997.

[76] H. You, Y. Feng, R. Ji, and Y. Gao, "PVNet: A joint convolutional network of point cloud and multi-view for 3D shape recognition," in *Proceedings of the International Conference on Multimedia (MM)*, 2018, pp. 1310–1318.

[77] Y. Wang and J. M. Solomon, "Deep closest point: Learning representations for point cloud registration," in *Proceedings of the International Conference on Computer Vision (ICCV)*, 2019, pp. 3522–3531.

[78] K. Hassani and M. Haley, "Unsupervised multi-task feature learning on point clouds," in *Proceedings of the International Conference on Computer Vision (ICCV)*, 2019, pp. 8159–8170.

[79] S. Lan, R. Yu, G. Yu, and L. S. Davis, "Modeling local geometric structure of 3D point clouds using Geo-CNN," in *Proceedings of the Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019, pp. 998–1008.

[80] C. Chen, L. Z. Fragonara, and A. Tsourdos, "GAPNet: Graph attention based point neural network for exploiting local feature of point cloud," *arXiv preprint arXiv:1905.08705*, 2019.

[81] D. Shin, J. Lee, J. Lee, J. Lee, and H.-J. Yoo, "DNPU: An energy-efficient deep-learning processor with heterogeneous multi-core architecture," *IEEE Micro*, vol. 38, no. 5, pp. 85–93, 2018.

[82] T. J. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi, "Graphicionado: A high-performance and energy-efficient accelerator for graph analytics," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2016, pp. 1–13.

[83] M. M. Ozdal, S. Yesil, T. Kim, A. Ayupov, J. Greth, S. Burns, and O. Ozturk, "Energy efficient architecture for graph analytics accelerators," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2016, pp. 166–177.

[84] M. Yan, L. Deng, X. Hu, L. Liang, Y. Feng, X. Ye, Z. Zhang, D. Fan, and Y. Xie, "HyGCN: A GCN accelerator with hybrid architecture," in *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, 2020, pp. 15–29.

[85] K. Kiningham, C. Re, and P. Levis, "GRIP: A graph neural network accelerator architecture," *arXiv preprint arXiv:2007.13828*, 2020.

[86] T. Geng, A. Li, R. Shi, C. Wu, T. Wang, Y. Li, P. Haghi, A. Tumeo, S. Che, S. Reinhardt, and M. C. Herbordt, "AWB-GCN: A graph convolutional network accelerator with runtime workload rebalancing," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2020, pp. 922–936.

[87] X. Chen, Y. Wang, X. Xie, X. Hu, A. Basak, L. Liang, M. Yan, L. Deng, Y. Ding, Z. Du, Y. Chen, and Y. Xie, "Rubik: A hierarchical architecture for efficient graph learning," *arXiv preprint arXiv:2009.12495*, 2020.

[88] R. Garg, E. Qin, F. M. Martínez, R. Guirado, A. Jain, S. Abadal, J. L. Abellán, M. E. Acacio, E. Alarcón, S. Rajamanickam, and T. Krishna, "Understanding the design space of sparse/dense multiphase dataflows for mapping graph neural networks on spatial accelerators," *arXiv preprint arXiv:2103.07977*, 2021.

[89] X. Song, T. Zhi, Z. Fan, Z. Zhang, X. Zeng, W. Li, X. Hu, Z. Du, Q. Guo, and Y. Chen, "Cambricon-G: A polyvalent energy-efficient accelerator for dynamic graph neural networks," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, no. 1, pp. 116–128, 2022.

[90] S. Liang, C. Liu, Y. Wang, H. Li, and X. Li, "DeepBurning-GL: An automated framework for generating graph neural network accelerators," in *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*, 2020, pp. 1–9.

[91] A. Mukkara, N. Beckmann, M. Abeydeera, X. Ma, and D. Sanchez, "Exploiting locality in graph analytics through hardware-accelerated traversal scheduling," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2018, pp. 1–14.

[92] A. Mukkara, N. Beckmann, and D. Sanchez, "Cache-guided scheduling: Exploiting caches to maximize locality in graph processing," *International Workshop on Architecture for Graph Processing*, 2017.

[93] J. Banerjee, W. Kim, S.-J. Kim, and J. F. Garza, "Clustering a DAG for CAD databases," *IEEE Transactions on Software Engineering*, vol. 14, no. 11, pp. 1684–1699, 1988.

[94] P. Yuan, C. Xie, L. Liu, and H. Jin, "PathGraph: A path centric graph processing system," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 10, pp. 2998–3012, 2016.

[95] H. Wei, J. X. Yu, C. Lu, and X. Lin, "Speedup graph processing by graph ordering," in *Proceedings of the International Conference on Management of Data (SIGMOD)*, 2016, pp. 1813–1828.

[96] P. Yao, L. Zheng, Z. Zeng, Y. Huang, C. Gui, X. Liao, H. Jin, and J. Xue, "A locality-aware energy-efficient accelerator for graph mining applications," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2020, pp. 895–907.

[97] Y. Wang, B. Feng, G. Li, S. Li, L. Deng, Y. Xie, and Y. Ding, "GNNAdvisor: An adaptive and efficient runtime system for GNN acceleration on GPUs," in *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*, 2021, pp. 515–531.

[98] C. E. Leiserson and T. B. Schardl, "A work-efficient parallel breadth-first search algorithm (or how to cope with the nondeterminism of reducers)," in *Proceedings of the Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2010, pp. 303–314.

[99] Y.-C. Wu, C.-H. Chang, J.-H. Hung, and C.-H. Yang, "A 135-mW fully integrated data processor for next-generation sequencing," *IEEE Transactions on Biomedical Circuits and Systems*, vol. 11, no. 6, pp. 1216–1225, 2017.

[100] Y. Liu, B. Fan, G. Meng, J. Lu, S. Xiang, and C. Pan, "DensePoint: Learning densely contextual representation for efficient point cloud processing," in *Proceedings of the International Conference on Computer Vision (ICCV)*, 2019, pp. 5238–5247.

[101] Q. Xu, X. Sun, C.-Y. Wu, P. Wang, and U. Neumann, "Grid-GCN for fast and scalable point cloud learning," in *Proceedings of the Conference on Computer Vision and Pattern Recognition (CVPR)*, 2020, pp. 5660–5669.

[102] Y. Liu, B. Fan, S. Xiang, and C. Pan, "Relation-shape convolutional neural network for point cloud analysis," in *Proceedings of the Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019, pp. 8887–8896.

[103] M. Atzmon, H. Maron, and Y. Lipman, "Point convolutional neural networks by extension operators," *ACM Transactions on Graphics*, vol. 37, no. 4, 2018.

[104] H. Thomas, C. R. Qi, J.-E. Deschaud, B. Marcotegui, F. Goulette, and L. J. Guibas, "KPConv: Flexible and deformable convolution for point clouds," in *Proceedings of the International Conference on Computer Vision (ICCV)*, 2019, pp. 6410–6419.

[105] Z. Wu, S. Song, A. Khosla, F. Yu, L. Zhang, X. Tang, and J. Xiao, "3D ShapeNets: A deep representation for volumetric shapes," in *Proceedings of the Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015, pp. 1912–1920.

[106] J. Arai, H. Shiokawa, T. Yamamuro, M. Onizuka, and S. Iwamura, "Rabbit order: Just-in-time parallel reordering for fast graph analysis," in *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, 2016, pp. 22–31.

[107] J. Kim, "Low-cost router microarchitecture for on-chip networks," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2009, pp. 255–266.

[108] S. Park, T. Krishna, C.-H. Chen, B. Daya, A. Chandrakasan, and L.-S. Peh, “Approaching the theoretical limits of a mesh NoC with a 16-node chip prototype in 45nm SOI,” in *Proceedings of the Design Automation Conference (DAC)*, 2012, pp. 398–405.

[109] H. Kwon, A. Samajdar, and T. Krishna, “MAERI: Enabling flexible dataflow mapping over DNN accelerators via reconfigurable interconnects,” in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operation Systems (ASPLOS)*, 2018, pp. 461–475.

[110] ——, “Rethinking NoCs for spatial neural network accelerators,” in *Proceedings of the International Symposium on Networks-on-Chip (NOCS)*, 2017, pp. 1–8.

[111] N. P. Jouppi, D. Hyun Yoon, M. Ashcraft, M. Gottscho, T. B. Jablin, G. Kurian, J. Laudon, S. Li, P. Ma, X. Ma, T. Norrie, N. Patil, S. Prasad, C. Young, Z. Zhou, and D. Patterson, “Ten lessons from three generations shaped Google’s TPUv4i: Industrial product,” in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2021, pp. 1–14.

[112] Y.-D. Kim, E. Park, S. Yoo, T. Choi, L. Yang, and D. Shin, “Compression of deep convolutional neural networks for fast and low power mobile applications,” *arXiv preprint arXiv:1511.06530*, 2016.

[113] A. Tjandra, S. Sakti, and S. Nakamura, “Compressing recurrent neural network with tensor train,” in *Proceedings of the International Joint Conference on Neural Networks (IJCNN)*, 2017, pp. 4451–4458.

[114] Y. Yang, D. Krompass, and V. Tresp, “Tensor-train recurrent neural networks for video classification,” in *Proceedings of the International Conference on Machine Learning (ICML)*, 2017, pp. 3891–3900.

[115] A. Tjandra, S. Sakti, and S. Nakamura, “Tensor decomposition for compressing recurrent neural network,” in *Proceedings of the International Joint Conference on Neural Networks (IJCNN)*, 2018, pp. 1–8.

[116] O. Hrinchuk, V. Khrulkov, L. Mirvakhabova, E. Orlova, and I. Oseledets, “Tensorized embedding layers for efficient model compression,” *arXiv preprint arXiv:1901.10787*, 2019.

[117] H. Huang and H. Yu, “LTNN: A layerwise tensorized compression of multilayer neural network,” *IEEE Transactions on Neural Networks and Learning Systems*, vol. 30, no. 5, pp. 1497–1511, 2019.

[118] C. C. Onu, J. E. Miller, and D. Precup, “A fully tensorized recurrent neural network,” *arXiv preprint arXiv:2010.04196*, 2020.

[119] C. Yin, B. Acun, C.-J. Wu, and X. Liu, “TT-Rec: Tensor train compression for deep learning recommendation models,” in *Proceedings of the Conference on Machine Learning and Systems (MLSys)*, 2021, pp. 448–462.

[120] V. Aggarwal, W. Wang, B. Eriksson, Y. Sun, and W. Wang, "Wide compression: Tensor ring nets," in *Proceedings of the Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018, pp. 9329–9338.

[121] Y. Pan, J. Xu, M. Wang, J. Ye, F. Wang, K. Bai, and Z. Xu, "Compressing recurrent neural networks with tensor ring for action recognition," in *Proceedings of the Conference on Artificial Intelligence (AAAI)*, 2019, pp. 4683–4690.

[122] M. Yin, S. Liao, X.-Y. Liu, X. Wang, and B. Yuan, "Compressing recurrent neural networks using hierarchical Tucker tensor decomposition," *arXiv preprint arXiv:2005.04366*, 2020.

[123] B. Wu, D. Wang, G. Zhao, L. Deng, and G. Li, "Hybrid tensor decomposition in neural network compression," *Elsevier Neural Networks*, vol. 132, pp. 309–320, 2020.

[124] M. Yin, S. Liao, X.-Y. Liu, X. Wang, and B. Yuan, "Towards extremely compact RNNs for video recognition with fully decomposed hierarchical Tucker structure," in *Proceedings of the Conference on Computer Vision and Pattern Recognition (CVPR)*, 2021, pp. 12 085–12 094.

[125] G. Li, J. Ye, H. Yang, D. Chen, S. Yan, and Z. Xu, "BT-Nets: Simplifying deep neural networks via block term decomposition," *arXiv preprint arXiv:1712.05689*, 2017.

[126] J. Ye, L. Wang, G. Li, D. Chen, S. Zhe, X. Chu, and Z. Xu, "Learning compact recurrent neural networks with block-term tensor decomposition," in *Proceedings of the Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018, pp. 9378–9387.

[127] X. Ma, P. Zhang, S. Zhang, N. Duan, Y. Hou, M. Zhou, and D. Song, "A tensorized transformer for language modeling," in *Proceedings of the Advances in Neural Information Processing Systems (NIPS)*, 2019, pp. 2232–2242.

[128] J. Ye, G. Li, D. Chen, H. Yang, S. Zhe, and Z. Xu, "Block-term tensor neural networks," *Elsevier Neural Networks*, vol. 130, pp. 11–21, 2020.

[129] D. Wang, B. Wu, G. Zhao, M. Yao, H. Chen, L. Deng, T. Yan, and G. Li, "Kronecker CP decomposition with fast multiplication for compressing RNNs," *IEEE Transactions on Neural Networks and Learning Systems*, pp. 1–15, 2021.

[130] R. Guo, Z. Yue, X. Si, T. Hu, H. Li, L. Tang, Y. Wang, L. Liu, M.-F. Chang, Q. Li, S. Wei, and S. Yin, "A 5.99-to-691.1TOPS/W tensor-train in-memory-computing processor using bit-level-sparsity-based optimization and variable-precision quantization," in *Proceedings of the International Solid-State Circuits Conference (ISSCC)*, 2021, pp. 242–244.

[131] H. Huang, L. Ni, K. Wang, Y. Wang, and H. Yu, "A highly parallel and energy efficient three-dimensional multilayer CMOS-RRAM accelerator for tensorized neural network," *IEEE Transactions on Nanotechnology*, vol. 17, no. 4, pp. 645–656, 2018.

[132] W. Hackbusch and S. Kühn, "A new scheme for the tensor representation," *Springer Journal of Fourier Analysis and Applications*, vol. 15, no. 5, pp. 706–722, 2009.

[133] L. Grasedyck, "Hierarchical singular value decomposition of tensors," *SIAM Journal on Matrix Analysis and Applications*, vol. 31, no. 4, pp. 2029–2054, 2010.

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

[135] L. De Lathauwer, "Decompositions of a higher-order tensor in block terms–part II: Definitions and uniqueness," *SIAM Journal on Matrix Analysis and Applications*, vol. 30, no. 3, pp. 1033–1066, 2008.

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

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

[138] C. Hawkins, X. Liu, and Z. Zhang, "Towards compact neural networks via end-to-end training: A Bayesian tensor approach with automatic rank determination," *SIAM Journal on Mathematics of Data Science*, vol. 4, no. 1, pp. 46–71, 2022.

[139] F. Sedighin, A. Cichocki, and A.-H. Phan, "Adaptive rank selection for tensor ring decomposition," *IEEE Journal of Selected Topics in Signal Processing*, vol. 15, no. 3, pp. 454–463, 2021.

[140] R. N. C. Pfeifer, J. Haegeman, and F. Verstraete, "Faster identification of optimal contraction sequences for tensor networks," *APS Physics Review E*, vol. 90, p. 033315, 2014.

[141] C.-C. Lam, P. Sadayappan, and R. Wenger, "On optimizing a class of multi-dimensional loops with reduction for parallel execution," *World Scientific Parallel Processing Letters*, vol. 7, no. 2, pp. 157–168, 1997.

[142] L. Liang, J. Xu, L. Deng, M. Yan, X. Hu, Z. Zhang, G. Li, and Y. Xie, "Fast search of the optimal contraction sequence in tensor networks," *IEEE Journal of Selected Topics in Signal Processing*, vol. 15, no. 3, pp. 574–586, 2021.

[143] L. Cavigelli and L. Benini, "Origami: A 803-GOp/s/W convolutional network accelerator," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 27, no. 11, pp. 2461–2475, 2017.

[144] S. Pal, J. Beaumont, D.-H. Park, A. Amarnath, S. Feng, C. Chakrabarti, H.-S. Kim, D. Blaauw, T. Mudge, and R. Dreslinski, "OuterSPACE: An outer product based sparse matrix multiplication accelerator," in *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, 2018, pp. 724–736.

[145] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam, "ShiDianNao: Shifting vision processing closer to the sensor," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2015, pp. 92–104.

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