

2023

Achieving Real-Time Dnn Execution On Mobile Devices With Compiler Optimizations

Wei Niu

William & Mary - Arts & Sciences, niuwei95@gmail.com

Follow this and additional works at: <https://scholarworks.wm.edu/etd>



Part of the [Computer Sciences Commons](#)

Recommended Citation

Niu, Wei, "Achieving Real-Time Dnn Execution On Mobile Devices With Compiler Optimizations" (2023). *Dissertations, Theses, and Masters Projects*. William & Mary. Paper 1697552565.
<https://dx.doi.org/10.21220/s2-mfpe-jh96>

This Dissertation is brought to you for free and open access by the Theses, Dissertations, & Master Projects at W&M ScholarWorks. It has been accepted for inclusion in Dissertations, Theses, and Masters Projects by an authorized administrator of W&M ScholarWorks. For more information, please contact scholarworks@wm.edu.

Achieving Real-time DNN Execution on Mobile Devices with Compiler
Optimizations

Wei Niu

Yiyang, Hunan, China

Bachelor of Software Engineering, Beihang University, China, 2016

A Dissertation presented to the Graduate Faculty of
The College of William & Mary in Candidacy for the Degree of
Doctor of Philosophy

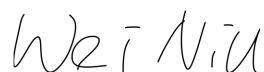
Department of Computer Science

College of William & Mary
May, 2023

APPROVAL PAGE

This Dissertation is submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy



Wei Niu

Approved by the Committee, May 2023



Committee Chair

Bin Ren, Associate Professor, Computer Science
William & Mary




Gang Zhou, Professor, Computer Science
William & Mary



Evgenia Smirni, Professor, Computer Science
William & Mary



Pieter Peers, Associate Professor, Computer Science
William & Mary



Gagan Agrawal, Professor, Computer & Cyber Sciences
Augusta University

ABSTRACT

Deep learning, particularly deep neural networks (DNNs), has led to significant advancements in various fields, such as autonomous driving, natural language processing, extended reality (XR), and view synthesis. Mobile and edge devices, with their efficient and specialized processors and suitability for real-time scenarios, have become the primary carriers for these emerging applications. The advancements in AutoML tools (e.g., Network Architecture Search) and training techniques have resulted in increasingly complex and deep DNN architectures with larger computational requirements. However, achieving real-time DNN execution (inference) on mobile devices is a challenging task due to the limited computing and storage resources available on embedded chips. Moreover, there is a considerable performance gap between the theoretical peak and the actual performance of DNN workloads on mobile devices due to the lack of understanding between the hardware and parallel algorithms. This dissertation aims to enable the real-time execution of DNNs on mobile devices by proposing a range of compiler-based optimizations.

Traditional convolutional neural networks (CNNs) consist of computation-intensive convolution layers, which are responsible for a significant portion of the entire computational workload. To this end, we present PatDNN, an innovative compression-compilation co-design framework that facilitates the compression of large-scale, computation-intensive DNNs to fit within the constrained storage and computation resources available on mobile devices. The PatDNN framework incorporates a hardware-friendly, pattern-based pruning method to compress DNN model parameters, and a range of sophisticated compiler optimizations tailored for pattern-based pruning to further enhance the efficiency of the system.

As higher accuracy requirements for different machine learning tasks, researchers have designed deeper and deeper models that involve moving substantial amounts of data among the layers. We present DNNFusion, an advanced operator fusion framework that can fuse multiple successive operators within a DNN into a single operator, significantly decreasing the number of memory accesses. Moreover, we propose a novel mathematical-property-based graph rewriting framework to simplify the computation even further.

To take advantage of the emerging dedicated accelerators in mobile SOCs, we propose GCD² specifically for mobile Digital Signal Processors (DSPs). In contrast to mainstream processors, DSPs feature a wider SIMD width and a wider variety of vector instructions. GCD² incorporates our novel compiler optimizations to capitalize on the unique capabilities of mobile DSPs and enhance hardware utilization.

Lastly, we propose SOD² for optimizing dynamic DNNs, where the tensor shapes and even the set of operators used are dependent upon the input and/or execution.

We derive a classification of common operators that form DNNs, and propose a Rank and Dimension Propagation (RDP) algorithm based on the classification. SOD² statically determines the shapes of operators, and then enables a series of other compiler optimizations.

TABLE OF CONTENTS

Acknowledgments	vi
Dedication	viii
List of Tables	ix
List of Figures	xiii
1 Introduction	1
1.1 Contributions	2
1.1.1 Real-time DNN Execution with Compression and Compilation Co-Design	2
1.1.2 Real-time DNN Execution with Advanced DNN Operator Fusion	3
1.1.3 Real-time DNN Execution on Mobile Digital Signal Processor	4
1.1.4 Dynamic DNN Execution with Static Compiler Optimization .	5
1.2 Dissertation Organization	6
2 PatDNN: Achieving Real-Time DNN Execution on Mobile Devices with Pattern-based Weight Pruning	8
2.1 Introduction	8
2.2 Background and Motivation	11
2.2.1 Layerwise Computation of DNNs	11
2.2.2 Mobile Acceleration of DNNs	12
2.2.3 DNN Model Compression and Challenges	14

2.2.4	ADMM-based DNN Model Compression Framework	15
2.2.5	Motivation	16
2.3	Overview of PatDNN	17
2.3.1	Pattern-based Pruning	17
2.3.2	Overview of PatDNN Acceleration Framework	19
2.4	PatDNN Training with Pattern-based Pruning	21
2.4.1	Designing the Pattern Set	21
2.4.2	Kernel Pattern and Connectivity Pruning Algorithm	22
2.4.3	Accuracy Validation and Analysis	24
2.5	PatDNN Inference Code Optimization	26
2.5.1	Compiler-based PatDNN Inference Framework	26
2.5.2	Filter Kernel Reorder (FKR)	28
2.5.3	Compressed DNN Weight Storage (FKW Format)	30
2.5.4	Load Redundancy Elimination (LRE)	31
2.5.5	Parameter Auto-tuning	33
2.6	Evaluation	34
2.6.1	Methodology	34
2.6.2	Overall Performance	35
2.6.3	Optimization Evaluation	37
2.6.4	PatDNN Performance Analysis in GFLOPS	39
2.6.5	Portability Study	41
2.6.6	Impact of Pattern Counts	41
2.7	Discussion	41
2.8	Summary	42
3	DNNFusion: Accelerating Deep Neural Networks Execution with Advanced Operator Fusion	43

3.1	Introduction	43
3.2	Blessing and Curse of Deep Layers	46
3.3	Classification of DNN Operators and Fusion Opportunity Analysis	48
3.3.1	DNN Operators Classification	48
3.3.2	Fusion Opportunity Analysis	50
3.4	DNNFusion’s Design	53
3.4.1	Overview of DNNFusion	53
3.4.2	Mathematical-Property-Based Graph Rewriting	53
3.4.3	Light-Weight Profile-Driven Fusion Plan Exploration	57
3.4.3.1	Overall Idea	57
3.4.3.2	Fusion Plan Generation Algorithm	58
3.4.4	Fusion Code Generation and Optimizations	61
3.4.4.1	Fusion Code Generation	61
3.4.4.2	Other Fusion-related Optimizations	63
3.5	Evaluation	64
3.5.1	Evaluation Setup	65
3.5.2	Overall Mobile Inference Evaluation	66
3.5.3	Understanding Fusion Optimizations	69
3.5.4	Portability	73
3.6	Related Work	73
3.7	Summary	75
4	GCD ² : A Globally Optimizing Compiler for Mapping DNNs to Mobile DSPs	77
4.1	Introduction	77
4.2	Executing DNNs on Mobile DSPs	80
4.3	Instructions and Layouts	82
4.4	System Design of GCD ²	86

4.4.1	SIMD Global Opt. Problem Formulation	86
4.4.2	Layout & Instruction Select Solution	88
4.4.3	VLIW Optimization	90
4.4.4	Putting Everything Together	96
4.5	Evaluation	97
4.5.1	Evaluation Setup	99
4.5.2	Comparison with Other Frameworks	100
4.5.3	Impact of Opt. and Algorithmic Features	103
4.5.4	Power Consumption and Energy Efficiency	106
4.5.5	Comparison with Other DNN Accelerators	107
4.6	Related Work	108
4.7	Summary	111
5	SOD ² : Statically Optimizing Dynamic Deep Neural Network Execution	112
5.1	Introduction	112
5.2	Existing Frameworks and Limitations	115
5.3	Operator Classification based on Dynamism	116
	Background and Notation.	116
5.4	Design of SOD ²	120
5.4.1	Pre-Deployment Data-Flow Analysis	120
	Formal Definition of Operator Rank and Dimension	
	Propagation (RDP).	121
	RDP Solution.	123
5.4.2	Operator Fusion for Dynamic DNN based on RDP	125
5.4.3	Static Execution Planning based on RDP	126
5.4.4	Other Optimizations	127
5.4.4.1	Memory Allocation Plan	127

5.4.4.2	RDP-based Multi-Version Code Generation	128
5.5	Evaluation	129
5.5.1	Evaluation Setup	130
5.5.2	Overall Comparison	132
5.5.3	Optimization Breakdown Analysis	133
5.5.4	Further Performance Analysis	136
	Latency Comparison with the Same Execution Path.	136
	Latency Comparison with Different Input Sizes.	137
	Latency with Fixed Memory Budget.	137
	Portability Evaluation	138
5.6	Related Work	138
5.7	Summary	140
6	Conclusion and Future Plan	141
6.1	Conclusion	141
6.2	Future Plan	141
	Bibliography	143
	Vita	167

ACKNOWLEDGMENTS

I would like to express my heartfelt thanks to everyone who has helped me along the way in my academic career. I wish I could mention them all; however, I am able to give some of them a special mention here.

First and foremost, I want to express my deepest appreciation to my advisor, Professor Bin Ren, for all of the guidance, mentorship, and patience he showed me throughout the entirety of my Ph.D. studies. His invaluable insights and expertise have greatly enriched my research experience. He is always patient and creative, encouraging and supportive, respects his students, and listens seriously to our ideas. His enthusiasm for research and teaching, commitment to excellence, and hard work have inspired me to strive for excellence in my academic and professional pursuits. He is the best mentor I've ever had, as well as a role model for my future career. I am extremely grateful for the opportunity to work with him. Now, I am ready to start my new journey with these invaluable experiences and knowledge he has given me.

I would like to extend my appreciation to my research collaborators for their help and dedication. They are Professor Gagan Agrawal, Professor Caiwen Ding, Professor Hui Guan, Professor Xue Lin, Dr. Shaoshan Liu, Professor Sijia Liu, Professor Kaisheng Ma, Professor Xiaolong Ma, Professor Xuehai Qian, Dr. Minghai Qin, Professor Xipeng Shen, Professor Xulong Tang, Professor Dingwen Tao, Professor Yanzhi Wang, and Professor Gang Zhou. Their expertise and creativity have helped propel our research goals forward and achieve great results. It is impossible to have my research papers and dissertation without their guidance and collaboration.

I am also grateful to my committee members, Professor Gang Zhou, Professor Evgenia Smirni, Professor Pieter Peers, and Professor Gagan Agrawal, for their time and effort in providing valuable feedback and suggestions. Their constructive and insightful feedback has greatly improved the quality of my dissertation and contributed greatly to its completion.

I am grateful to Dr. Rui Xia for being an excellent mentor during my internship at Bytedance, CA, USA. I am particularly appreciative of Wenwei Chen, my former mentor at Bytedance, Beijing, China, for his guidance and encouragement. Without his support, I might not have had the courage to pursue my Ph.D. degree. I also extend my thanks to my friends and colleagues in the lab for their continuous support and encouragement. We have had many interesting discussions and brainstorming sessions, which have helped me broaden my horizons and think outside the box. Their companionship and positive attitude have also made my time in graduate school more enriching and enjoyable.

I would like to express my sincere thanks to all the faculty members and staff who have contributed to my education and growth in the department of Computer Science at William & Mary, as well as the staff at Sadler Center and Commons Dining Hall for their great dining service.

At last, a special thanks to my parents and Jiewei Wang for their unconditional support and love. Without these, I would not be where I am today. Thank you for always being there for me.

For my grandfather, I wish he could see my graduation.

To my family.

LIST OF TABLES

2.1	DNN acceleration frameworks on mobile devices.	13
2.2	Qualitative comparison of different pruning schemes on accuracy and speedup under the same pruning rate.	19
2.3	Top-5 accuracy comparison on kernel pattern pruning.	24
2.4	Top-5 accuracy and CONV weight reduction on joint kernel pattern pruning (8 patterns in the set) and connectivity pruning.	25
2.5	DNNs characteristics (under kernel pattern and connectivity pruning): Accu: ImageNet top-5, CIFAR top-1; the negative values in Accuracy Loss actually mean accuracy improvement.	34
2.6	VGG unique CONV layers' filter shapes and given names.	35
2.7	Pattern counts impact (with $3.6\times$ connectivity pruning): accuracy loss and exe time for VGG.	40
3.1	An empirical study to motivate this work: The relation of overall computation, layer count, and execution efficiency of multiple DNNs. Results are collected on Qualcomm Adreno 650 GPU with an optimized baseline framework with fixed-pattern operator fusion that outperforms all state-of-the-art DNN execution frameworks (called OurB+ and will be introduced later).	47
3.2	Classification of DNN operators in mapping types. These operators are defined in ONNX [160].	49

3.3	Mapping type analysis. The first column and the first row (both without color) show the mapping types of first and second operators, respectively, before fusion, and the colored cells show the mapping type of the operator after fusion. Green implies that these fusion combinations can be fused directly (i.e., they are profitable). Red implies that these fusions are unprofitable. Yellow implies that further profiling is required to determine profitability.	50
3.4	Graph rewriting with mathematical properties. Only representative graph rewriting rules are listed due to space limitation. In summary, DNNFusion derives 45, 38, and 66 graph rewriting rules in the category of Associative, Distributive, and Communicative, respectively. We omit unrelated operators for better readability. $\odot, +, -, Abs, Recip, Square, \sqrt{\quad}$ mean element-wise multiplication, addition, subtraction, absolute, reciprocal, square, and square root, respectively. <code>BitShift</code> calculates the bit shifted value of elements of a given tensor element-wisely. <code>ReduceSum</code> and <code>ReduceProd</code> calculate the reduced summation and production of elements of an input tensor along an axis. <code>Exp</code> calculates the exponent of elements in a given input tensor element-wisely. <code>#FLOPS</code> denotes the number of floating point operations	56
3.5	Fusion rate evaluation: computation layer count and intermediate result size for all evaluated DNNs. CIL (Compute-Intensive Layer): each input is used more than once, e.g. <code>MatMul</code> , <code>CONV</code> . MIL (Memory-Intensive Layer): each input is used only once, e.g. <code>Activation</code> . IRS: intermediate results. '?' means this framework does not support this model.	67

3.6	Inference latency comparison: DNNFusion, MNN, TVM, TFLite, and PyTorch on mobile CPU and GPU. #FLOPS denotes the number of floating point operations. OurB is our baseline implementation by turning off all fusion optimizations and OurB+ is OurB with a fixed-pattern fusion as TVM. DNNF is short for DNNFusion, i.e., our optimized version. '-' denotes this framework does not support this execution.	68
4.1	Latency and Power Comparisons among Mobile CPU, GPU, and DSP. Experiments are conducted on a Samsung Galaxy S20 with TFLite [1]. CPU, GPU, and DSP uses int8, float16, and int8, respectively. Power is collected by the Android system interface. Results are for each inference.	78
4.2	Execution Latency w/ Different SIMD Instructions (and Layouts) for Matrix Multiplication $C = A \times B$. M , K , and N denote the dimension size of Matrix A ($M \times K$), B ($K \times N$), and C ($M \times N$), respectively. Execution latency and total data size with padding are normalized by vmpy for readability. Smaller numbers mean better latency or less padding. Bold ones denote the best case.	82
4.3	SIMD Instructions Selected and Performance by RAKE [5] and GCD². Representative Conv2d kernels (w/ varied shapes, 7×7 , 1×1 , and 3×3) are from ResNet-50.	86

4.4	Overall Performance Comparison among TFLite, SNPE, and GCD² on Mobile DSP. “-” means this model is not supported by the framework yet. OverT and OverS are the speedup of GCD ² over TFLite, and SNPE, respectively. GCD ² ’s overall <i>compilation time</i> for these models ranges from 5 minutes (WDSR-b) to 25 minutes (EfficientDet-d0).	98
4.5	Inference Speed and Energy Efficiency Comparison with ResNet-50 on EdgeTPU [63] and NVIDIA Jetson Xavier [217]. FPS is short for frames per second, and FPW represents for inference frames per Watt.	108
5.1	Inference overhead for shape dynamism w/ execution re-initialization. SL: shape propagation and layout selection. ST: schedule and tuning. Alloc: memory allocation. Infer: inference time. Experiments are conducted on a Samsung Galaxy S21 w/ MNN [96].	115
5.2	Classification of DNN operators based on dynamism degrees. Operators are from ONNX (Open Neural Network Exchange) [160]. .	118
5.3	Definition of Rank and Dimensions Propagation (RDP). . . .	123
5.4	Memory consumption (allocated for intermediate results) for ONNX Runtime, MNN, TVM with Nimble extension (TVM-N), and SOD² on a mobile CPU. “-” means this model is not supported by the framework yet. “S” stands for shape dynamism, and “C” represents for control-flow dynamism.	131
5.5	End-to-end execution latency comparison among ONNX Runtime, MNN, TVM-N, and SOD² on mobile CPU and mobile GPU. “-” means this model is not supported by the framework yet.	132

LIST OF FIGURES

2.1	DNN CONV layer computation.	12
2.2	(a) Non-structured weight pruning and (b) two types of structured weight pruning.	14
2.3	Illustration of (a) kernel pattern pruning on CONV kernels, and (b) connectivity pruning by removing kernels.	17
2.4	Illustration of connectivity pruning.	18
2.5	Overview of PatDNN acceleration framework.	20
2.6	The algorithm-level overview of PatDNN training.	21
2.7	PatDNN’s compiler-based optimization and code generation flow: compiler takes both model codes with graph-based optimizations and a layerwise representation (as an example in Figure 2.8) to generate low-level C/C++ and OpenCL codes (as No-opt). This low-level code is further optimized with filter kernel reorder and our FKW compact model storage (+Reorder), the register-level load redundancy elimination (+LRE), and other optimizations like auto-tuning. Finally, the code is deployed on mobile devices.	27
2.8	An LR example for a CONV layer.	28
2.9	An example of filter kernel reorder.	28
2.10	An example of FKW compressed weight storage.	30
2.11	Load redundancy elimination (left: kernel-level; right: filter-level). . .	31
2.12	Overall performance: x-axis: different trained DNN models; y-axis: average DNN inference execution time on a single input.	36

2.13	Speedup of opt/no-opt on each unique CONV layer.	37
2.14	Profiling result: reorder and redundancy elimination.	38
2.15	Effect of different loop permutations and loop tiling.	38
2.16	Extra data structure overhead: FKW over CSR on unique VGG CONV layers with different pruning rates.	39
2.17	GFLOPS performance study: PatDNN vs dense.	39
2.18	Portability study: performance on two other platforms.	40
3.1	DNNFusion overview.	54
3.2	Examples of graph rewriting with mathematical properties. Associative property explores the optimal execution order of operators and replaces the expensive combination of operators with a cheaper one. Distributive property explores the common combination of operators and simplifies the computation structure. Commutative property switches the execution order of operators to reduce the overall computation. Note: the letter below each operator (e.g., B below Conv in (a)) or the letter in rectangle (e.g., C in (b)) denotes that this input is from model weights rather than an intermediate result. The letter in diamond (e.g., A) means that this is the input of this operator block, which could be the input of the model or intermediate result from a prior block. The intermediate results within this block are omitted for readability.	55
3.3	An example of fusion plan exploration. Assume Add, Conv, ReLU, Mul, and Sub have identical output shape and IRS size.	58
3.4	Code generation.	61
3.5	Data movement operators optimization.	63

3.6 **Speedup over TASO optimized execution on mobile CPU.**
The models (computational graphs) are optimized by TASO and then
executed on TFLite. 69

3.7 **Optimization breakdown on y-axis: speedup over OurB, i.e.
a version w/o fusion opt.** GR, Fuse, and Other denote graph
rewriting, fusion, and other fusion-related optimizations, respectively. 70

3.8 **Memory (left) and cache miss (right) analysis.** MA and MC
denote memory access and memory consumption, respectively. Cache
miss count is compared on L1/L2/L3 data cache and L1/L2 TLB
cache on mobile CPU, and on L1/L2 data cache only on mobile GPU.
All values are normalized w.r.t DNNF (the optimal version). 71

3.9 **(a) Mobile CPU and GPU utilization.** CPU utilization is av-
eraged on 8-cores. **(b) Compilation time.** Comparison between
TVM and DNNF for YOLO-V4 on mobile CPU. DNNF (w/o db) is
without the presence of an existing profiling database; DNNF (w/ db)
assumes such a database is pre-computed. **Fusion** is invisible as it
spends very little time on both TVM and DNNF. 72

3.10 **Portability evaluation.** It is on Samsung Galaxy S10 and Honor
Magic 2. Left two figures are YOLO-V4 and right two are GPT-2.
Only TFLite supports GPT-2 on mobile CPU (no mobile GPU support). 73

4.1 **SIMD/Vector Multiply Instruction Examples in Mobile DSP
Chip 82**

4.2	Data Layouts to Support Usage of Varied SIMD Instructions for Matrix Multiplication. Each number denotes the linear storage offset of an element. A blue, yellow, and orange cell takes 1, 2, and 4 bytes, respectively. Left shows data storage, and right shows computation.	84
4.3	Examples of Computational Graphs. Left and right show partial CGs in ResNet [77] and TinyBERT [97].	87
4.4	Two Examples of Packing Instructions with Soft Dependencies. Different colors show different VLIW execution pipeline stages (read in green, execute in orange, and write in blue). In (a), the second stage (Assign R1+R2 to R3) of the second instruction requires to wait for the completion of the first instruction, incurring packing penalty. A similar situation happens to (b) between Assign and Store.	91
4.5	An Instruction Packing Example. The left part shows part of the pseudo assembly code for the innermost nested loop performing 2D Element-wise Addition: $R = A + B + C$, where A, B, and C are two-dimensional uint8 arrays and R is a two-dimensional int16 array. v2:1 denotes a 16-bit register combining 2 8-bit registers v2 and v1. The middle part shows an IDG, in which, solid edges denote hard dependency, dot edges denote soft, and critical path is colored in red. Right shows the packing results from our solution and an sub-optimal solution that treats all soft dependencies as hard (<i>soft to hard</i>). N denotes an empty instruction slot.	94
4.6	System Workflow of GCD².	97

4.7	Performance Comparison of GCD², Halide, TVM, and RAKE with Individual Kernels. Left shows the speedup and right shows the packet counts, both normalizing Halide as 1. Conv2D operators (from ResNet-50) are used. GCD ^b is a sub-optimal version of GCD ² that contains tensor optimizations only without VLIW packing. . . .	101
4.8	DSP Utilization and Memory Bandwidth Comparison. These results are as reported by Snapdragon Profiler [176], and normalized with GCD ²	102
4.9	Performance Breakdown Analysis. Speedup over the baseline (normalized with the no-opt version). DSP utilization and memory bandwidth analysis (both normalized with the GCD ² optimal version as 100%). The results are collected from Snapdragon Profiler [176]. . .	103
4.10	Layout Optimization Analysis. X-axis denotes the number of operators in the computational graph. The left figure shows the speedup over local optimal with different numbers of operators. The right figure shows the search time, and its y-axis is logarithmically scaled.	104
4.11	VLIW Scheduling Analysis. The version treating all soft dependencies as hard ones is used as the baseline.	105
4.12	Unrolling Factor Analysis on a Single MatMul Kernel and on Multiple MatMul Kernels. The x-axis in the left figure denotes the unrolling factors. The right figure shows the performance comparison among the best settings of three unrolling strategies (Out, Mid, and GCD ²) on 8 operators (from O1 to O8). For comparison, it also shows versions w/o unrolling and w/ exhaustive search.	106

4.13	Comparison of Total Power Consumption (left) and Energy Efficiency in Inference Frames/Watt (right). Three DSP frameworks and TFLite with GPU back-end on 4 representative DNNs. . . .	107
5.1	Different degrees of dynamism. Each node is a DNN operator. Yellow, blue, red, and purple mean <i>Input Shape Determined Output</i> , <i>Input Shape Determined Output Shape</i> , <i>Input Shape & Value Determined Output Shape</i> , and <i>Execution Determined Output</i> , respectively. In (d), Switch 's execution path is decided dynamically during runtime and red dot edges represent both the computation dependency and control flow.	120
5.2	Domain of RDP dataflow analysis. It includes known, symbolic, and operation-inferred constants that form a lattice.	121
5.3	Examples of forward and backward transfer. Each node is an operator. Yellow, blue, and red mean <i>Input Shape Determined Output</i> , <i>Input Shape Determined Output Shape</i> , and <i>Input Shape & Value Determined Output Shape</i> , respectively. Ids (e.g., ①) indicate the location where transfer functions apply and their applying orders for a forward transfer (a backward transfer reverses this order). S and V equations map values in the RDP domain to the shape and value of each tensor, in which, F denotes the transfer function. <i>fs</i> and <i>bs</i> of F denote forward and backward, and F's subscript is a short form of its type (e.g., <i>ISDOS</i> means <i>Input Shape Determined Output Shape</i>).	122

5.4	Operator fusion with dynamic shapes. The top code snippet shows that fusion is not feasible because of broadcasting [44]. Specifically, <i>Add</i> requires <i>A</i> 's indices <i>I'</i> , <i>J'</i> , and <i>K'</i> to be either 1 or <i>I</i> , <i>J</i> , and <i>K</i> , resulting in 8 fusion scenarios. With RDP, such fusion is feasible (shown in the below code snippet). This fusion significantly reduces intermediate result materialization requirements.	126
5.5	Memory reduction of different optimizations on CPU. Over the baseline w/o any RDP-enabled optimization (No opt.)	133
5.6	Execution speedup of different opt. on CPU and GPU. Over the baseline w/o any RDP-enabled optimization (No opt.)	134
5.7	Further break down effect of existing static fusion (SFusion) and RDP-based fusion (RDP Fusion). For both layer count and intermediate result size, normalized by no fusion opt.	134
5.8	The percentage of different types of sub-graph.	135
5.9	Latency and memory consumption comparison between SOD² and MNN with the same execution path.	136
5.10	Performance variation with different input sizes (shapes). The data is collected from YOLO-V6. A larger input size means more computations.	137
5.11	Speedup with the same memory consumption.	137
5.12	Portability evaluation. The results are collected on Snapdragon 835. An empty bar means the model is not supported by the framework. Results are normalized by MNN for readability.	138

Chapter 1

Introduction

In recent years, there has been an explosion of deep learning applications such as computer vision (CV) [76, 142, 195, 199, 206, 213, 218, 239], natural language processing (NLP) [97, 115, 188, 201, 212], and autonomous driving [13, 24, 55, 235]. Mobile devices have emerged as the ideal platform for these applications due to their capability and privacy protection. The computational capability of mobile devices has increased considerably as a result of breakthroughs in transistor density. The Snapdragon 8 Gen 1 [187] offers 20 times more peak performance than the Snapdragon 850 [179]. In addition, the memory capacity of mobile devices has increased, allowing for the deployment of larger deep neural networks (DNNs). However, the ever-increasing size of DNNs [201] has posed a significant challenge to achieving real-time inference performance (typically 30 processing per second, or 33 milliseconds per processing) for DNNs. The limited computational resources of mobile devices, such as limited memory bandwidth and power budgets, have exacerbated this issue. There are several challenges behind the scenes. First, the computation size of DNNs is growing exponentially, leading to a significant increase in computing resource requirements. Second, as the number of layers in DNNs increases, data flow between layers becomes more intense, which is one of the most significant performance bottlenecks. In addition, traditional frameworks are not suited to the emergence of specialized mobile device accelerators, resulting in inefficient DNN execution. Real-time DNN inference on

mobile devices has therefore become a challenging issue.

The dissertation aims to address the challenges of achieving real-time DNN inference on mobile devices. We approach this issue from three distinct perspectives. Firstly, we propose a compression-compilation co-design framework for large-scale deep neural networks that introduces a new optimization dimension: fine-grained pruning patterns within coarse-grained structures. Secondly, we develop a framework for advanced operator fusion specifically designed for extremely deep DNNs, which reduces intermediate results among layers by up to $4.7\times$. Thirdly, we propose a compilation system that incorporates global optimization for emerging dedicated accelerators that fully exploit the new features offered by mobile DSPs. Lastly, we design a comprehensive dynamic DNN execution system, which includes both static compiler optimizations and dynamic runtime optimization to achieve real-time performance for DNNs on mobile devices.

1.1 Contributions

1.1.1 Real-time DNN Execution with Compression and Compilation Co-Design

Considering the limited computational resources of mobile devices and the intensive computation in DNNs, there has been a widely recognized challenge to achieving real-time performance for large-scale DNN inference. Model (DNN) pruning [34, 69, 73, 74, 79, 150, 233] serves as a promising technique that can dramatically reduce the computation size (FLOPs) for computation-intensive operators (e.g, Convolution and Fully-connected). However, existing model pruning represents two extremes in the design space, non-structured pruning (e.g., random pruning) [69, 74] and structured pruning (e.g., filter/kernel pruning) [79, 150, 233]. Non-structured pruning applies fine-grained pruning and achieves high pruning rates, but it results in poor hardware performance due to its irregular computation pattern. In contrast, structured pruning employs coarse-grained pruning, making it hardware efficient, but with high accuracy loss.

We present PatDNN [158], a solution to address the existing limitations by introducing fine-grained pruning patterns within coarse-grained structures. This innovative approach advances the state-of-the-art and exposes an unexplored design dimension. By leveraging the benefits of fine-grained pruning patterns, we introduce a set of architecture-aware compiler and code generation-based optimizations that mitigate the irregular computation pattern. The proposed optimizations include filter kernel reordering, compressed weight storage, register load redundancy elimination, and parameter autotuning. These optimizations eliminate uncertainty and enhance parallelism at both the thread-level and instruction-level in the sparse computation. Our evaluation results demonstrate a significant improvement in performance, up to $44\times$ faster than the current state-of-the-art frameworks (TFLite [1], TVM [25], and MNN [96]). Our sparse computation can even yield a speedup ratio comparable to the compression rate for certain convolutional layers in VGG-16 [206]. Most importantly, PatDNN enables for the first time, the real-time execution of complex DNNs on mobile devices.

1.1.2 Real-time DNN Execution with Advanced DNN Operator Fusion

The need for higher model accuracy has led to the design of neural networks with more and more layers. Due to the high volume of data movement between layers and the limited memory throughput on mobile devices, however, the depth of DNNs is the most significant obstacle to efficient execution. Operator fusion (or kernel/layer fusion) is a common method for reducing data movement and eliminating unnecessary intermediate result materialization. Popular end-to-end frameworks such as TFLite [1], TVM [25], MNN [96], and Pytorch-Mobile [165] employ operator fusion optimizations based on recognition of specific fusion patterns. These operator fusion transformations are generally based on a computational graph representation that views an application as a set of tensor operations and represents dependencies as a Directed Acyclic Graph (DAG). However, the fusion patterns considered in these works are insufficient to cover all fusion possibilities. It is unlikely to have happened that the fusion patterns can be extended to cover all possible operator combinations.

DNNFusion [157] is a flexible and comprehensive operator fusion framework that introduces an alternative approach to traditional end-to-end frameworks, such as TFLite and TVM, which rely heavily on pattern matching to recognize limited patterns. Our proposed solution categorizes various operator types into a limited number of groups, develops rules for the possible combinations of these groups, and generates the fusion plans and associated code. We also implement a novel mathematical property-based graph rewriting framework to optimize DNN computation and facilitate subsequent fusion plan generation, an integrated fusion plan generation that leverages the high-level operator abstraction and accurate light-weight profiling, and additional advanced optimizations enabled by the newly designed fusion analysis and fused code generation. DNNFusion has been integrated into PatDNN as an optimized version of dense model inference and has been exhaustively evaluated on 15 key DNNs, including BERT and its variants, GPT-2 [188], YOLO-V4 [13]. The evaluation results demonstrate that DNNFusion outperforms the state-of-the-art frameworks (TFLite, PyTorch, TVM, and MNN) with up to $9.3\times$ speedup on the popular models, enabling for the first time many of the most recent DNN models that are not supported by existing frameworks to run on mobile devices efficiently, even in real-time.

1.1.3 Real-time DNN Execution on Mobile Digital Signal Processor

On the recent hardware side, the trend of designing specialized accelerators for real-time execution of Deep Neural Networks (DNN) on mobile devices is emerging along with advances in hardware technology. For example, modern mobile Digital Signal Processors (DSPs) feature: 1) wider SIMD (Single Instruction Multiple Data) widths, 2) more capable vector instructions, and 3) more flexible instruction pipelines that can tolerate certain data dependencies. In comparison to mobile CPU and GPU, the mobile DSP is superior in terms of both power and performance for executing DNNs. Existing DNN compilers and libraries, however, are unable to fully leverage the DSP’s computational power because: 1) they failed to achieve a global optimization of selecting the best vector instructions for each

layer; and 2) an effective instruction packing strategy with regard to data dependencies is missing at the VLIW (Very Long Instruction Word) packing level. Compiler optimizations for the DSP chip turns out to involve dealing with many advanced features, especially with respect to low-level parallelism exposed through its instruction set, requiring techniques well beyond the ones implemented in current systems or otherwise developed.

We present GCD² [156], which addresses the problem of efficiently mapping real-world complex DNN workloads on modern mobile DSP architectures. To this end, GCD² incorporates several optimizations which are designed to better harness the power of advanced SIMD instructions while also making it possible to combine different instructions with varying data dependencies into a single VLIW (Very Long Instruction Word) packet. Specifically, we first design different tensor data layouts to improve the utilization of SIMD instructions, followed by the adoption of a global instruction selection algorithm that facilitates selection of the most suitable SIMD instructions and their accompanying layouts for each layer. Lastly, we develop a novel VLIW packing algorithm that results in an efficient instruction scheduling process for the DNN execution. Evaluation results demonstrate that GCD² is able to deliver a speedup of up to 6.0× when compared to other cutting-edge frameworks for mobile DSP by using optimized data layouts in combination with efficient VLIW packing strategies. It also turns out that, with our optimizations, mobile DSP even achieves 6.1× and 1.48× better energy efficiency with the same data type (int8) over EdgeTPU and Jetson Xavier, respectively.

1.1.4 Dynamic DNN Execution with Static Compiler Optimization

Dynamic DNN are increasingly common today due to its higher representative power and flexibility. On the systematic optimization side, however, the majority of work for DNNs considers *static models* characterized by two properties: input and output shapes and sizes known in advance for each layer, and a fixed execution path independent of the input. In contrast, *dynamic models* deviate from one or both of these properties. For instance, some dynamic models, such as those used in cutting-edge computer vision models [107, 191, 198]

or transformers for natural language processing like BERT [46], can handle inputs with varying shapes and/or apply variable portions of filter kernels during execution. Those dynamisms pose many challenges for the optimizations that have been key to obtaining high efficiency. For example, the lack of static information about the input shapes and sizes makes it difficult to perform operator fusion, which is a key optimization for reducing data movement and eliminating unnecessary intermediate result materialization.

We present SOD², a static compiler optimization framework for DNN inference in the presence of dynamic features towards minimizing inference latency and memory requirements. We start by conducting a comprehensive investigation into the operators commonly used in modern DNNs. These operators are classified based on the relationship between output shapes, input shapes, and their computation patterns. We introduce the Rank and Dimension Propagation (RDP) framework, which performs data-flow analysis to deduce intermediate tensor shapes and dimensions. Leveraging the insights gained from RDP analysis, we enable various optimizations, including operator fusion, fused code generation, static execution planning, runtime memory allocation, and multi-version code generation. To evaluate the effectiveness of SOD², we extensively test it on 10 state-of-the-art DNN models featuring shape dynamism and/or control-flow dynamism. These models include those related to emerging artificial general intelligence (AGI), such as StableDiffusion [198] and SegmentAnything [107]. Our evaluation demonstrates that SOD² achieves significant reductions in memory consumption up to 88%, while accelerating execution by up to $3.9 \times$ compared to four state-of-the-art DNN execution frameworks.

1.2 Dissertation Organization

The remainder of this dissertation is structured as follows. Chapter 2 introduces an end-to-end framework to efficiently execute DNN on mobile devices with the help of a novel model compression technique – pattern-based pruning and a set of thorough architecture-aware compiler/code generation-based optimizations. Chapter 3 presents a rigorous and extensive

loop fusion framework that can exploit the operator view of computations in DNNs, and yet can enable a set of advanced transformations. Chapter 4 reports on a compilation system GCD², developed to support complex Deep Neural Network (DNN) workloads on modern mobile DSP chips. Chapter 5 shows a static compiler optimization framework named SOD², to support dynamic neural networks on mobile devices. Finally, 6 concludes this dissertation and discusses future research directions.

Chapter 2

PatDNN: Achieving Real-Time DNN Execution on Mobile Devices with Pattern-based Weight Pruning

2.1 Introduction

Deep learning or deep neural networks (DNNs) have become the fundamental element and core enabler of ubiquitous artificial intelligence. After obtaining DNN models trained with a huge amount of data, they can be deployed for inference, perception and control tasks in various autonomous systems and internet-of-things (IoT) applications. Recently, along with the rapid emergence of high-end mobile devices¹, executing DNNs on mobile platforms gains popularity and is quickly becoming the mainstream [42, 116, 118, 161, 254] for broad applications such as sensor nodes, wireless access points, smartphones, wearable devices, video streaming, augmented reality, robotics, unmanned vehicles, smart health devices, etc. [12, 17, 117, 167, 197].

Considering the nature of these applications, achieving *real-time DNN inference* is an

¹Modern mobile platforms become increasingly sophisticated, usually equipped with both CPUs and GPUs, e.g., Qualcomm Snapdragon 855 [180] has an octa-core Kryo 485 CPU and an Adreno 640 GPU.

ideal but yet a very challenging goal for mobile devices due to the limited computing resources of embedded processors. For example, consider VGG-16 [206], one of the key DNN models in transfer learning with broad application scenarios. For an embedded GPU (Adreno 640, with 16-bit floating-point for weights/intermediate results), it takes 242ms to perform inference using TVM [25], and is not even supported in TensorFlow-Lite (TFLite) [1] — these are two representative mobile-oriented, end-to-end DNN inference acceleration frameworks. It is clearly far from real-time execution.

To achieve the goal, it is necessary to consider algorithm-level innovations. To this end, *DNN model compression* techniques, including *weight pruning* [34, 69, 73, 74, 79, 150, 233] and *weight/activation quantization* [30, 31, 70, 85, 86, 127, 134, 164, 192, 236, 259], have been proposed and studied intensively for model storage reduction and computation acceleration. Early efforts on DNN model compression [34, 69, 73, 74, 79, 150, 233] mainly rely on iterative and heuristic methods, with limited and non-uniform model compression rates. Recently, a systematic DNN model compression framework (ADMM-NN) has been developed using the powerful mathematical optimization tool ADMM (Alternating Direction Methods of Multipliers) [18, 82, 141], currently achieving the best performance (in terms of model compression rate under the same accuracy) on weight pruning [193, 256] and one of the best on weight quantization [127].

Despite the high compression ratio, there is a significant gap between algorithm-level innovations and hardware-level performance optimizations for DNN inference acceleration. Specifically, the general but *non-structured* weight pruning (i.e., arbitrary weight can be pruned) [69, 74] can seriously affect processing throughput because the indices for the compressed weight representation prevent achieving high parallelism [79, 150, 233]. While ADMM-NN achieves higher and more reliable compression ratios, hardware implementation obstacle due to the non-structured nature still stays the same. Alternatively, the *structured* pruning [79, 150, 233], e.g., filter and channel pruning, can generate more hardware-friendly models but result in relatively higher accuracy drop. To achieve the real-time inference for representative DNNs in mobile devices, it is imperative to develop an end-to-end DNN

acceleration framework that achieves *both high accuracy and high hardware efficiency*.

We make a key observation that the general non-structured pruning and current structured pruning represent two extremes in the design space. In non-structured pruning, *any* weight can be pruned, while in structured pruning, the pruning is done for the *whole filter or channel*. Thus, non-structured pruning is completely *fine-grained*, which achieves high compression ratio but is not hardware or software optimization friendly, while structured pruning is *coarse-grained*, which generates hardware-efficient regular models with higher accuracy loss.

In this paper, we advance the state-of-the-art by naturally introducing a new dimension, *fine-grained pruning patterns inside the coarse-grained structures*, revealing a previously *unknown* point in design space. This new dimension allows more flexible exploration of the trade-off between accuracy and hardware efficiency. In this paradigm, the key question is *how to “recover” the hardware efficiency lost due to the fine-grained patterns*. The unique insight of our solution is to use *compiler* to seamlessly close the gap between hardware efficiency of fully structured pruning and the pattern-based “semi-structured” pruning.

Specifically, we propose *PatDNN*, a novel end-to-end mobile DNN acceleration framework that can generate highly accurate DNN models using pattern-based pruning methods and guarantee execution efficiency with compiler optimizations. PatDNN consists of two stages: (1) *pattern-based training stage*, which performs kernel pattern and connectivity pruning (termed *pattern-based pruning* in general) with a pattern set generation and an extended ADMM solution framework. (2) *execution code generation stage*, which converts DNN models into computational graphs and applies multiple optimizations including: a high-level and fine-grained DNN layerwise representation, filter kernel reorder, load redundancy eliminations, and automatic parameter tuning. All design optimizations are general, and applicable to both mobile CPUs and GPUs.

In sum, this paper makes several major contributions:

- First, it proposes a novel *pattern-based* DNN pruning approach that achieves the ben-

efits of both non-structured and structured pruning while avoiding their weaknesses.

- Second, it enhances the recent ADMM-NN framework [193,249] with pattern selection capability to map a pattern to each kernel, and train non-zero weights.
- Third, it identifies the compatibility of the proposed pattern-based pruning scheme with compiler code generation, and develop multiple novel compiler optimizations for compressed DNN execution. These optimization opportunities are enabled only by our pattern-based design, and do not exist in any prior DNN execution frameworks.
- Fourth, it implements an end-to-end DNN acceleration framework *PatDNN* on mobile platforms, compatible with modern embedded CPU and GPU architectures, achieving real-time performance on representative DNNs without accuracy loss for the first time.

We compare PatDNN with three state-of-the-art end-to-end DNN frameworks on both mobile CPU and GPU, TensorFlow Lite [1], TVM [25], and Alibaba Mobile Neural Networks [96] using three widely used DNNs, VGG-16, ResNet-50, and MobileNet-V2 and two benchmark datasets, ImageNet and CIFAR-10. Our evaluation results show that PatDNN achieves up to $44.5\times$ speedup without any accuracy compromise. Using Adreno 640 embedded GPU, PatDNN achieves 18.9ms inference time of VGG-16 on ImageNet dataset. To the best of our knowledge, it is the first time to achieve real-time execution of such representative large-scale DNNs on mobile devices.

2.2 Background and Motivation

2.2.1 Layerwise Computation of DNNs

DNN models can be viewed as cascaded connections of multiple functional layers, such as convolutional (CONV), fully-connected (FC), and pooling (POOL) layers, to extract features for classification or detection [101,125,250]. Take the most computation-intensive

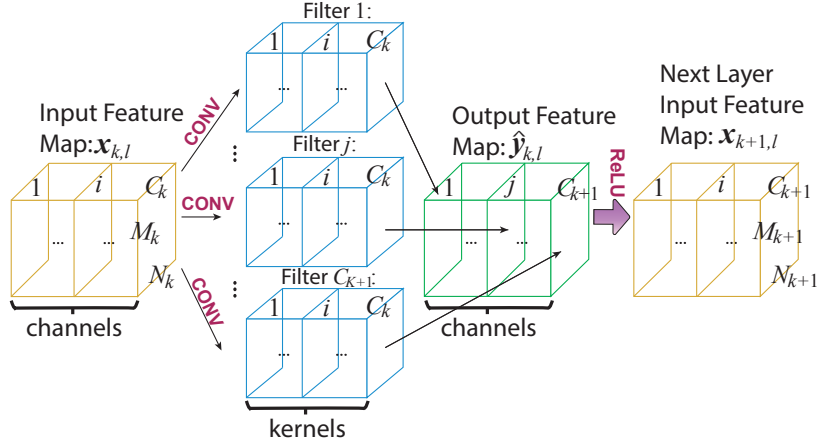


Figure 2.1: DNN CONV layer computation.

CONV layer as an example, as shown in Figure 2.1, the input feature map of the k -th layer has a size of $M_k \times N_k \times C_k$, where C_k is the number of *channels* of the input feature map. This layer uses C_{k+1} CONV filters, each with a size of $P_k \times Q_k \times C_k$. Note that the number of *kernels* C_k in a CONV filter should match the number of channels C_k in the input feature map to perform convolution. Each j -th CONV filter performs convolution with the input feature map, using a stride of S_k , resulting in the j -th channel in the output feature map. Therefore, the number of channels in the output feature map equals to the number of filters C_{k+1} , while the size of the output feature map i.e., M_{k+1} and N_{k+1} is determined by M_k , N_k , P_k , Q_k , and S_k . The CONV layer is followed by an activation layer, which performs an activation operation, typically ReLU, on the output feature map. Besides the functional layers in DNNs, batch normalization becomes an essential operation to increase the stability of DNN training by overcoming the gradient vanishing issue [89].

2.2.2 Mobile Acceleration of DNNs

In recent years, there have been intensive efforts on DNN inference acceleration frameworks targeting mobile devices, include DeepX [116], TFLite [1], DeepEar [119], TVM [25], Alibaba Mobile Neural Network (MNN) [96], DeepCache [241], DeepMon [88], DeepSense [248], and MCDNN [72]. Most of these prior works do not fully utilize model compression techniques.

Other efforts that explore model sparsity and model compression to accelerate the DNN execution include Liu et al. [139], DeftNN [81], SCNN [163], AdaDeep [140]. However, they either do not target mobile platforms, or require new hardware, or trade off compression rate and accuracy, introducing various drawbacks compared to our work.

Table 2.1 compares the major optimization techniques offered by three state-of-the-art, end-to-end DNN inference frameworks (TFLite [1], TVM [25], and MNN [96]). We do not include other efforts, e.g., DeepCache [241] and DeepMon [88], since they mainly focus on specific DNN applications rather than general DNNs. In this work, our goal is to find the most appropriate weight pruning scheme for mobile DNN acceleration and the corresponding full-stack acceleration framework. We utilize 16-bit floating point representation on GPU for both weights and intermediate results which is supported in mobile devices and shown to incur no accuracy loss [1, 25, 96] for DNNs.

Table 2.1: DNN acceleration frameworks on mobile devices.

DNNs	Optimization Knobs	TFLite	TVM	MNN	Ours
Dense	Parameters auto-tuning	N	Y	N	Y
	CPU/GPU support	Y	Y	Y	Y
	Half-floating support	Y	Y	Y	Y
	Computation graph optimization	Y [!]	Y [*]	Y [!]	Y^{**}
	Tensor optimization	Y [!]	Y [†]	Y [!]	Y^{††}
Sparse	Sparse DNN model support	N	N	N	Y
	Pattern-based pruning	N	N	N	Y
	Connectivity pruning	N	N	N	Y
	Filter kernel reordering	N	N	N	Y
	Opt. sparse kernel code generation	N	N	N	Y
	Auto-tuning for sparse models	N	N	N	Y

* Operator fusion, constant folding, static memory plan, and data layout transform

** Besides above in *, operation replacement

† Scheduling, nested parallelism, tensorization, explicit memory latency hiding

†† Besides above in †, dense kernel reordering, SIMD operation optimization

! Similar optimizations as TVM, but less advanced

2.2.3 DNN Model Compression and Challenges

DNN model compression has been proposed for simultaneously reducing the storage/computation and accelerating inference with minor classification accuracy (or prediction quality) loss. Model compression is performed during DNN training. Two important categories of DNN model compression techniques are weight pruning [34, 69, 74, 79, 150, 233] and weight quantization [30, 85, 127, 134, 164, 192, 236, 259].

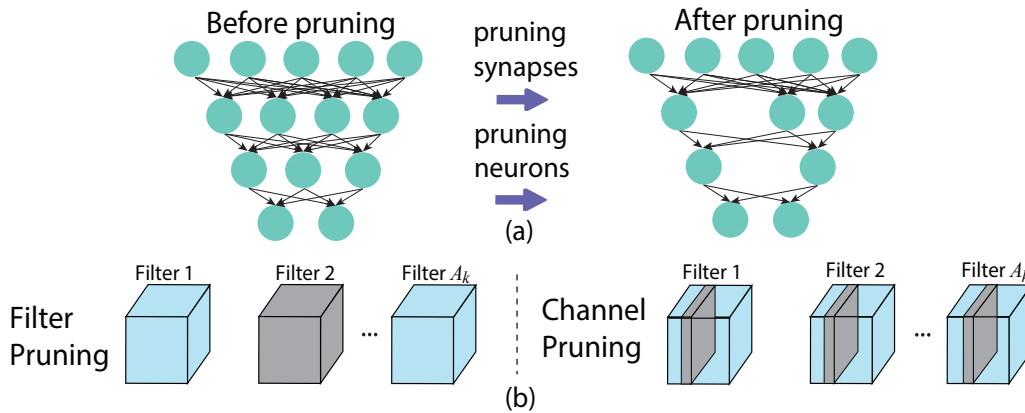


Figure 2.2: (a) Non-structured weight pruning and (b) two types of structured weight pruning.

Weight pruning reduces the redundancy in the number of weights. As shown in Figure 2.2, two main approaches of weight pruning are (1) the general and non-structured pruning; and (2) structured pruning, which produces irregular and regular compressed DNN models.

Non-Structured Pruning: In this method, arbitrary weight can be pruned. It can result in a high pruning rate, i.e., reduction in the number of weights, which can reduce the actual computation. For compiler and code optimization, non-structured pruning incurs several challenges due to the irregularity in computation and memory access. First, the irregular and sparse kernel weights require *heavy control-flow instructions*, which degrade instruction-level parallelism. Second, it introduces *thread divergence and load imbalance* due to the fact that kernels in different filters have divergent workloads and they are usually processed by multiple threads — a key concern for efficient thread-level parallelism. Third, it usually incurs *low memory performance* due to poor data locality and cache

performance. More importantly, it prohibits advanced memory optimizations such as eliminating redundant loads that widely exist in convolution operations. Similarly, for hardware acceleration, since the pruned models are stored in some sparse matrix format with indices, they often lead to performance degradation in GPU and CPU implementations [34, 69, 74].

Structured Pruning: This method can produce regular, but smaller weight matrices. Figure 2.2 (b) illustrates the representative structured pruning schemes: *filter pruning* and *channel pruning* [233]. Filter and channel pruning can be considered as equivalent in that pruning a filter in the k -th layer is equivalent to pruning the corresponding channel in the $(k + 1)$ -th layer. Filter/channel pruning is compatible with Winograd algorithm [122, 234] that has been used to accelerate computation of the original DNNs. Due to the regular structure, the GPU/CPU implementations typically lead to more significant acceleration [79, 150, 233]. However, the structured pruning suffers from notable accuracy loss [79, 233].

2.2.4 ADMM-based DNN Model Compression Framework

Recent work ADMM-NN [193, 249] leverages Alternating Direction Methods of Multipliers (ADMM) method for joint DNN weight pruning and quantization. ADMM is a powerful tool for optimization, by decomposing an original problem into two subproblems that can be solved separately and efficiently. For example, considering optimization problem $\min_{\mathbf{x}} f(\mathbf{x}) + g(\mathbf{x})$. In ADMM, this problem is decomposed into two subproblems on \mathbf{x} and \mathbf{z} (auxiliary variable), which will be solved iteratively until convergence. The first subproblem derives \mathbf{x} given \mathbf{z} : $\min_{\mathbf{x}} f(\mathbf{x}) + q_1(\mathbf{x}|\mathbf{z})$. The second subproblem derives \mathbf{z} given \mathbf{x} : $\min_{\mathbf{z}} g(\mathbf{z}) + q_2(\mathbf{z}|\mathbf{x})$. Both q_1 and q_2 are quadratic functions.

As a unique property, ADMM can effectively deal with a subset of combinatorial constraints and yield optimal (or at least high quality) solutions [82, 141]. Luckily, the necessary constraints in the DNN weight pruning and quantization belong to this subset of combinatorial constraints, making ADMM applicable to DNN model compression.

Due to the unprecedented results on accuracy and pruning rate, ADMM-NN [193]

is considered as the state-of-art results for non-structured weight pruning and one of state-of-art methods for weight quantization. For non-structured pruning, ADMM-NN achieves $167\times$, $24\times$, and $7\times$ weight reductions on LeNet-5, AlexNet, and ResNet-50 models, respectively, without accuracy loss. However, the framework only focuses on non-structured weight pruning, in which the pruning rate does not directly translate to performance improvements.

ADMM-NN can be extended to perform structured pruning, i.e., filter/channel pruning, and our results show that it leads to 1.0% Top-5 accuracy degradation with $3.8\times$ weight reduction on VGG-16 CONV layers using ImageNet dataset. Although better than prior work (1.7% in [79] and 1.4% in AMC [78]), this accuracy loss is not negligible for many applications.

2.2.5 Motivation

Based on the discussion of prior work on weight pruning, we rethink the design space and observe that non-structured and structured represent two extremes in the design space. In non-structured pruning, any weight can be pruned, we consider it as a fine-grained method; in structured pruning, the weights of whole filter or channel are pruned together, we consider it as a coarse-grained method. Correspondingly, the two methods have different implications on hardware acceleration and software optimization: non-structured pruning is not hardware or software optimization friendly, so the higher pruning ratio cannot fully translate to performance gain, while structured pruning incurs higher accuracy loss.

The motivation of our study is to seek an approach that can offer the best of both methods. To achieve that, we naturally introduce a new dimension, *fine-grained pruning patterns inside the coarse-grained structures*, revealing a previously *unknown* point in design space. With the higher accuracy enabled by fine-grained pruning pattern, the key question is how to re-gain similar hardware efficiency as coarse-gained structured pruning. We take a unique approach and leverage compiler optimizations to close the performance gap between full structured pruning and pattern-based “semi-structured” pruning.

2.3 Overview of PatDNN

2.3.1 Pattern-based Pruning

In pattern-based pruning, the key consideration is how to design and select the patterns. To achieve high accuracy and execution efficiency, we need to design the patterns considering the implication for *theory and algorithm*, *compiler optimization*, and *hardware execution*. Good patterns should have two key properties: flexibility and regularity.

The *Flexibility* is not only desirable at theory and algorithm level but also enables efficient compiler code generation. Specifically, it allows compilers to maximize or maintain both instruction-level and thread-level parallelism. The *regularity* not only results in highly efficient hardware execution but also enables efficient compiler optimizations such as *redundant load elimination* to further improve performance. Compared to irregular structures, recent works also show from theory and algorithm level that high accuracy or function approximation capability can be achieved at the same time with certain regularity. Given these two key properties, we propose two pattern-based pruning techniques: kernel pattern pruning and connectivity pruning.

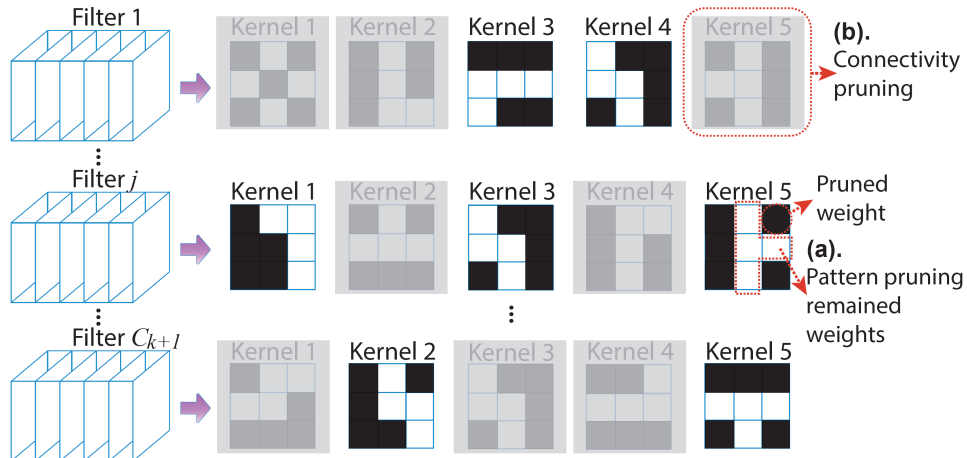


Figure 2.3: Illustration of (a) kernel pattern pruning on CONV kernels, and (b) connectivity pruning by removing kernels.

Kernel Pattern Pruning is illustrated in Figure 2.3. For each kernel (in a CONV

filter), a fixed number of weights are pruned, and the remaining weights (white cells) form specific “kernel patterns”. We define the example in Figure 2.3 as 4-entry pattern pruning, since every kernel reserves 4 non-zero weights out of the original 3×3 kernel (the most commonly used kernel). The same approach is also applicable to other kernel sizes and the FC layer. For each kernel, it possesses *flexibility* in choosing among a number of pre-defined patterns.

At *theory and algorithm* level, it is shown in [123, 132] that the desirable kernel shape has certain patterns to match the connection structure in human visual systems, instead of a square shape. The selection of appropriate pattern for each kernel can be naturally done by extending ADMM-based framework. In Section 2.4.3, we achieve accuracy enhancement in all representative DNNs in our testing. At *compiler* level, the pre-defined pattern allows compiler to *re-order and generate codes* at filter and kernel level so that kernels with the same pattern can be grouped for consecutive executions to maximize instruction-level parallelism. At *hardware* level, the 4-entry patterns are extremely friendly to the SIMD architecture in embedded processors based on either GPUs or CPUs. Note that our approach is general and can be applied to any pre-defined patterns, not just the 4-entry considered in the paper.

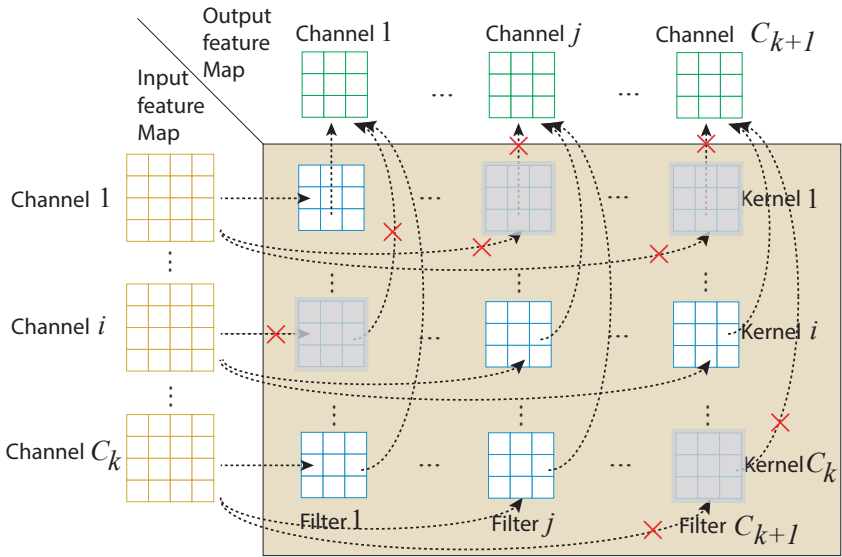


Figure 2.4: Illustration of connectivity pruning.

Connectivity Pruning is illustrated in Figure 2.4. The key insight is to *cut the connections* between certain input and output channels, which is equivalent to removal of corresponding kernels. In CONV layers, the correlation between input channel i and output channel j is represented by the i -th kernel of filter j . This method is proposed for overcoming the limited weight pruning rate by kernel pattern pruning.

At *theory and algorithm* levels, the connectivity pruning matches the desirability of locality in layerwise computations inspired by human visual systems [243, 244]. It is more flexible than the prior filter/channel pruning schemes that remove whole filters/channels, thereby achieving higher accuracy. At *compiler and hardware* level, removed kernels and associated computations can be grouped by compiler using the *re-ordering* capability without affecting the other computations, thereby maintaining parallelism degree.

Table 2.2: Qualitative comparison of different pruning schemes on accuracy and speedup under the same pruning rate.

Pruning Scheme	Accuracy				Hardware Speedup			
	Highest	Minor Loss	Moderate Loss	Highest Loss	Highest	High	Moderate	Minor
Non-structured	X							X
Filter/Channel				X	X			
Pattern	X				X			
Connectivity		X				X		

2.3.2 Overview of PatDNN Acceleration Framework

Based on the above discussions, we propose *PatDNN*, a novel end-to-end mobile DNN acceleration framework that can generate highly accurate DNN models using pattern-based pruning methods and guarantee execution efficiency with compiler optimizations. Compared to recent prior works [78, 79, 193, 233], PatDNN uniquely enables *cross-layer vertical integration*, making it desirable across theory/algorithm, compiler and hardware. Allowing compilers to treat pruned kernels as special patterns, our approach not only achieves high pruning rate with high accuracy, but also effectively converts into performance improvements due to hardware friendly properties.

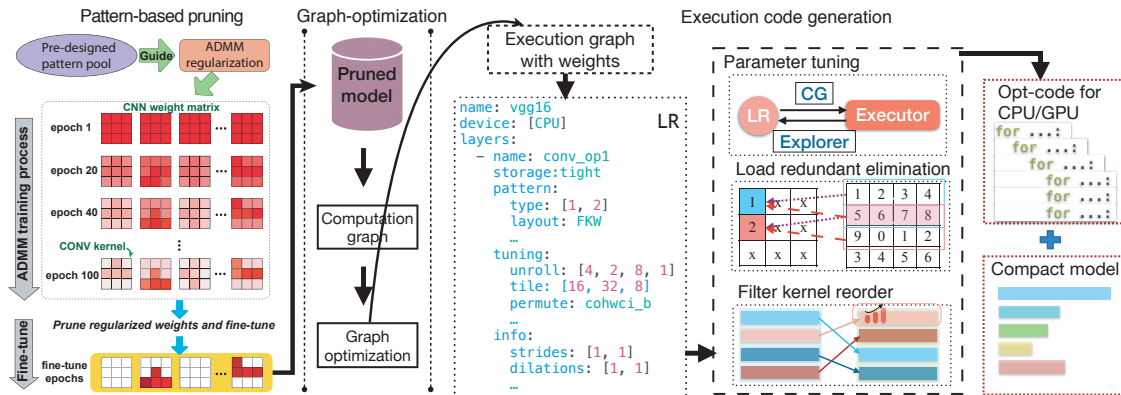


Figure 2.5: Overview of PatDNN acceleration framework.

As shown in Table 2.2, PatDNN can achieve the benefits of both non-structured and structured pruning. The key enabler to achieving this goal is to leverage compiler to maintain the efficiency of structured pruning based on kernel pattern and connectivity pruning. Our approach is an excellent example of hardware and software co-design, which can be compared to an intuitive analogy: the multi-level cache memory hierarchy provides sufficient hardware supports to hide memory access latency and explore locality, but compiler and software optimizations are still needed to fully realize effective cache management policy.

Figure 2.5 shows the overview of PatDNN which consists of two stages: (1) *pattern-based training stage* (Section 2.4), which performs kernel pattern and connectivity pruning with an extended ADMM solution framework. (2) *execution code generation stage* (Section 2.5), which performs multiple effective optimizations based on the patterns. Similar to TVM [25], PatDNN converts DNN models into computational graphs and applies multiple graph-based optimizations. Based on these optimizations, we focus on layerwise design and optimization including a high-level and fine-grained DNN layerwise representation (LR), filter kernel reorder, load redundancy eliminations, and automatic parameter tuning. All of these designs and optimizations are general, and applicable to both mobile CPUs and GPUs. The second stage generates optimized execution codes as well as DNN models with weights stored in a novel compact format.

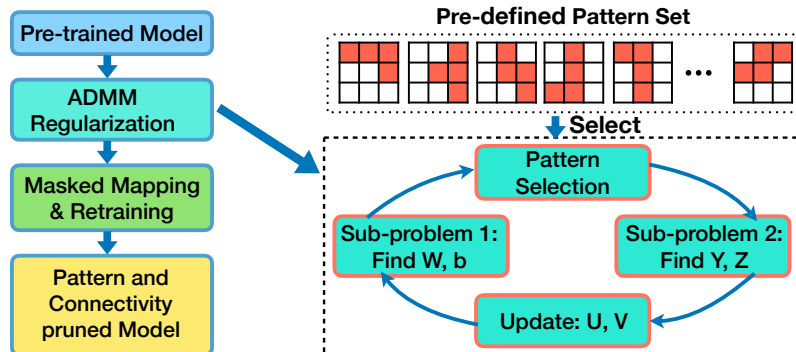


Figure 2.6: The algorithm-level overview of PatDNN training.

2.4 PatDNN Training with Pattern-based Pruning

This section describes the methods to generate compressed DNN models for PatDNN. The procedure is composed of two steps: (1) we design a set of desired patterns to be selected for each kernel; (2) assign a pattern for each kernel (kernel pattern pruning) or prune the whole kernel (connectivity pruning), and train the pattern-based weights for maintaining accuracy. The overall flow is shown in Figure 2.6. Essentially, it reflects the algorithm aspects of PatDNN. Our method can be applied to either a pre-trained DNN or train a model from scratch.

2.4.1 Designing the Pattern Set

We need to determine the number of patterns, and design each specific candidate pattern in the pattern set. The number of patterns is an important hyperparameter that should be carefully considered. If it is too large, it is more challenging to generate efficient codes, thereby affecting performance; if it is too small, the lack of flexibility may lead to accuracy degradation. Through empirical study, we validate that 6-8 patterns in the set achieves as a desirable tradeoff for the most common 3×3 kernel—ensuring low compiler overhead while maintaining high accuracy.

When the number of patterns is determined and 4-entry patterns are utilized, the compiler optimization and hardware efficiency are oblivious to the specific pattern shapes.

However, the specific patterns to use need to be carefully optimized to maintain high accuracy after kernel pattern pruning. The key insights of pattern design are: (1) both theory and empirical studies [243, 244] show that the central weight in a 3×3 kernel is critical and shall not be pruned; and (2) it is desirable that the distortion is small for each kernel before and after kernel pattern pruning. Hence, we propose the following heuristic. First, for the pre-trained DNN, we scan all the kernels, and for each kernel, we find the four weights with largest magnitudes (including the central weight). These four weights form a 4-entry pattern, called the *natural pattern* of the kernel. According to the definition of natural patterns, there are a total of $\binom{8}{3} = 56$ number of possible patterns. Suppose we aim at k different patterns in the candidate set. We count and select the Top- k most commonly appeared natural patterns across all kernels in the DNN, thereby forming the pattern candidate set (to select from in the subsequent step).

Our study on pattern number and pattern style selection is consistent with the pattern pruning theory work that is proposed in [147]. Different from pattern theory derivation in [147], our approach focuses on system-level design and compiler optimization of the pattern-based acceleration framework.

2.4.2 Kernel Pattern and Connectivity Pruning Algorithm

Problem Formulation: Consider an N -layer DNN, and we focus on the most computation-intensive CONV layers. The weights and biases of layer k are respectively denoted by \mathbf{W}_k and \mathbf{b}_k , and the loss function of DNN is denoted by $f(\{\mathbf{W}_k\}_{k=1}^N, \{\mathbf{b}_k\}_{k=1}^N)$, refer to [256] for more details. In our discussion, $\{\mathbf{W}_k\}_{k=1}^N$ and $\{\mathbf{b}_k\}_{k=1}^N$ respectively characterize the collection of weights and biases from layer 1 to layer N . Then the pattern and connectivity pruning is formulated as an optimization problem:

$$\begin{aligned}
 & \underset{\{\mathbf{W}_k\}, \{\mathbf{b}_k\}}{\text{minimize}} && f(\{\mathbf{W}_k\}_{k=1}^N, \{\mathbf{b}_k\}_{k=1}^N), \\
 & \text{subject to} && \mathbf{W}_k \in \mathcal{S}_k, \mathbf{W}_k \in \mathcal{S}'_k, \quad k = 1, \dots, N.
 \end{aligned} \tag{2.1}$$

The collection of weights in the k -th CONV layer forms a four-dimensional tensor, i.e., $\mathbf{W}_k \in \mathbb{R}^{P_k \times Q_k \times C_k \times C_{k+1}}$, where P_k, Q_k, C_k , and C_{k+1} are respectively the height of kernel, the width of kernel, the number of kernels, and the number of filters, in layer k . Suppose \mathbf{X} denotes the weight tensor in a specific layer, then $(\mathbf{X})_{::,a,b}$ denotes a specific kernel.

In *kernel pattern pruning*, the constraint in the k -th CONV layer is $\mathbf{W}_k \in \mathcal{S}_k := \{\mathbf{X} \mid \text{each kernel in } \mathbf{X} \text{ needs to satisfy one specific pattern shape in the pattern set (and non-zero weight values can be arbitrary)}\}$. In *connectivity pruning*, the constraint in the k -th CONV layer is $\mathbf{W}_k \in \mathcal{S}'_k := \{\mathbf{X} \mid \text{the number of nonzero kernels in } \mathbf{X} \text{ is less than or equal to } \alpha_k\}$ (α_k is a predetermined hyperparameter with more discussions later). Both constraints need to be simultaneously satisfied.

Extended ADMM-based Solution Framework: The constraint $\mathbf{W}_k \in \mathcal{S}_k$ in problem (2.1) is different from the clustering-like constraints in ADMM-NN [193], in that it is flexible to select a pattern for each kernel from the pattern set. As long as a pattern is assigned for each kernel, constraints in problem (2.1) become clustering-like and ADMM compatible. Similar to ADMM-NN [193], the ADMM-based solution is an iterative process, starting from a pre-trained DNN model. We assign an appropriate pattern for each kernel based on the L_2 -norm metric in each iteration, to achieve higher flexibility.

By incorporating auxiliary variables \mathbf{Z}_k 's and \mathbf{Y}_k 's, and dual variables \mathbf{U}_k 's and \mathbf{V}_k 's, we decompose (2.1) into three subproblems, and iteratively solve until convergence. In iteration l , after assigning patterns we solve the first subproblem

$$\begin{aligned} & \underset{\{\mathbf{W}_k\}, \{\mathbf{b}_k\}}{\text{minimize}} f(\{\mathbf{W}_k\}_{k=1}^N, \{\mathbf{b}_k\}_{k=1}^N) + \sum_{k=1}^N \frac{\rho_k}{2} \|\mathbf{W}_k - \mathbf{Z}_k^l + \mathbf{U}_k^l\|_F^2 \\ & + \sum_{k=1}^N \frac{\rho_k}{2} \|\mathbf{W}_k - \mathbf{Y}_k^l + \mathbf{V}_k^l\|_F^2. \end{aligned} \quad (2.2)$$

The first term is the loss function of the DNN, while the other quadratic terms are convex. As a result, this subproblem can be solved by stochastic gradient descent (e.g., the ADAM algorithm [106]) similar to training the original DNN.

The solution $\{\mathbf{W}_k\}$ of subproblem 1 is denoted by $\{\mathbf{W}_k^{l+1}\}$. Then we aim to derive

$\{\mathbf{Z}_k^{l+1}\}$ and $\{\mathbf{Y}_k^{l+1}\}$ in subproblems 2 and 3. These subproblems have the same form as those in ADMM-NN [193]. Thanks to the characteristics in combinatorial constraints, the optimal, analytical solution of the two subproblems are Euclidean projections, and are polynomial time solvable. For example, for connectivity pruning, the projection is: keeping α_k kernels with largest L_2 norms and setting the rest of kernels to zero. For kernel pattern pruning it is similar. Finally, we update dual variables \mathbf{U}_k and \mathbf{V}_k according to the ADMM rule [18] and thereby complete the l -th iteration in the ADMM-based solution.

The hyperparameter determination process is relatively straightforward for joint pattern and connectivity pruning. There is no additional hyperparameters for kernel pattern pruning when the pattern set has been developed. For connectivity pruning we need to determine the pruning rate α_k for each layer. In this paper, we adopt a heuristic method of uniform pruning rate for all layers except for the first layer (which is smaller, yet more sensitive to pruning).

2.4.3 Accuracy Validation and Analysis

We validate the accuracy of ADMM-based joint kernel pattern and connectivity pruning, based on ImageNet ILSVRC-2012 and CIFAR-10 datasets, using VGG-16 [206], ResNet-50 [77], and MobileNet-V2 [200] DNN models. Our implementations are based on PyTorch, and the baseline accuracy results are in many cases higher than prior work, which reflects the recent progress in DNN training. With a pre-trained DNN model, we limit the number of epochs in kernel pattern and connectivity pruning to 120, similar to the original DNN training in PyTorch and much lower than iterative pruning [74].

Table 2.3: Top-5 accuracy comparison on kernel pattern pruning.

Network	Original DNN	6-pattern	8-pattern	12-pattern
VGG16	91.7%	92.1%	92.3%	92.4%
ResNet50	92.7%	92.7%	92.8%	93.0%

Table 2.3 illustrates the Top-5 accuracy comparison on kernel pattern pruning only, applied on the CONV layers of VGG-16 and ResNet-50 using ImageNet dataset. The

baseline is the original DNN without patterns, and we demonstrate the accuracy results with 6, 8, and 12 patterns (all 4-entry patterns) in the pattern set. Our first observation is that *the accuracy will improve when the number of candidate patterns is sufficient* — typically 4 - 8 patterns are sufficient. This is attributed to the compatibility of kernel pattern pruning with human visual system and the ability to eliminate overfitting (compared with square kernel shape). This observation has been also validated for other types of DNNs and data sets (e.g., CIFAR-10).

Table 2.4: Top-5 accuracy and CONV weight reduction on joint kernel pattern pruning (8 patterns in the set) and connectivity pruning.

	Method	Top-5 Accuracy	CONV compression rate
VGG16	Deep compression [14]	89.1%	3.5×
	NeST [8]	89.4%	6.5×
	ADMM-NN [49] (non-structured)	88.9%	10.2×
	Our’s (8-pattern + connectivity)	91.6%	8.0×
ResNet50	Fine-grained Pruning [42]	92.3%	2.6×
	ADMM-NN [49] (non-structured)	92.3%	7.0×
	Our’s (8-pattern + connectivity)	92.5%	4.4×

Table 2.4 illustrates the Top-5 accuracy comparison on joint kernel pattern pruning (8 patterns in the set) and connectivity pruning, on VGG-16 and ResNet-50 using ImageNet dataset. For VGG-16, all kernels are 3×3 . After applying 4-entry patterns on all kernels and $3.6\times$ uniform connectivity pruning, we achieve around $8\times$ weight reduction on CONV layers of VGG-16. For ResNet-50, a portion of kernels are 1×1 besides the majority of 3×3 kernels. We apply kernel pattern pruning on all 3×3 ones, and apply uniform $3.6\times$ connectivity pruning on all kernels. We achieve $4.4\times$ weight reduction on CONV layers. One can observe from the table that (1) *no Top-5 accuracy drop with this setup*; (2) *under the same accuracy, the weight reduction rate is close to ADMM-based (and outperforms prior heuristic based) non-structured pruning on CONV layers*.

For the CIFAR-10 dataset, we observe consistent accuracy improvements with 8 patterns on 3×3 kernels and $3.6\times$ connectivity pruning, with results shown in Section 2.6.

2.5 PatDNN Inference Code Optimization

For DNN models with kernel pattern and connectivity pruning, PatDNN ensures hardware execution efficiency of DNN inference with optimized compiler and code generation. As aforementioned, compiler optimizations play the key role in “recovering” the performance loss due to the fine-grained pattern-based pruning compared to fully structured pruning. This stage includes two-levels of optimizations: (1) optimizations on computational graphs that explore the potential opportunities among multiple DNN layers; and (2) optimizations within each layer. PatDNN adopts an enhanced TVM [25]-like approach together with other innovations from the latest efforts in this direction (e.g., Tensor Comprehensions [224]) to implement the former (with major optimizations summarized in Table 2.1). Due to space limit, we do not elaborate each as they are not the main research contribution and not specific to DNN execution optimization leveraging pattern-based pruning.

This section focuses on PatDNN’s layerwise optimizations based on kernel pattern and connectivity pruning that are specifically designed to address the challenges in DNN acceleration with non-structured weight pruning, i.e., *heavy control-flow instructions, thread divergence and load imbalance, and poor memory performance*. These optimizations are general, and applicable to both mobile CPUs and GPUs. Our framework can generate both optimized CPU (vectorized C++) code and GPU (OpenCL) code. Figure 2.7 illustrates PatDNN’s compiler-based optimization and code generation flow with a CONV layer example.

2.5.1 Compiler-based PatDNN Inference Framework

Layerwise Representation: The key feature of PatDNN is its *sparsity- and pruning-aware* design. To support it, PatDNN proposes a high-level fine-grained Layerwise Representation (LR) to capture the sparsity information. This LR includes intensive DNN layer specific information to enable aggressive layerwise optimizations. In particular, it includes detailed *kernel pattern and connectivity-related information* (e.g., the pattern types presented in this

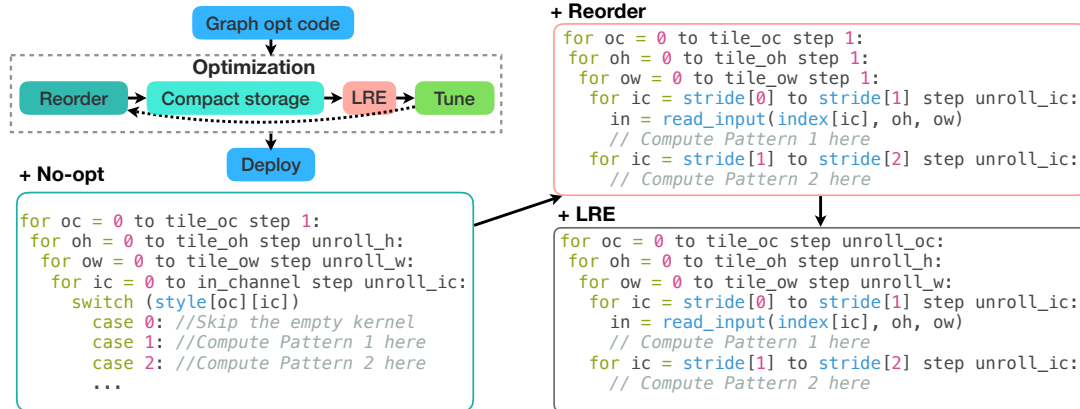


Figure 2.7: PatDNN’s compiler-based optimization and code generation flow: compiler takes both model codes with graph-based optimizations and a layerwise representation (as an example in Figure 2.8) to generate low-level C/C++ and OpenCL codes (as No-opt). This low-level code is further optimized with filter kernel reorder and our FKW compact model storage (+Reorder), the register-level load redundancy elimination (+LRE), and other optimizations like auto-tuning. Finally, the code is deployed on mobile devices.

layer, the pattern order in each filter, the connection between kernels and input/output channels, etc.); and *tuning-decided parameters* (e.g., the input and output tile sizes, unrolling factors, the loop permutation of this layer, etc.).

PatDNN extracts the pattern/connectivity information from DNN models with computational graph optimizations, and determines the tuning-related parameters by the auto-tuning. This LR is used for PatDNN’s following optimizations: (1) filter kernel reordering, which operates on kernel pattern and connectivity-related information, i.e., specifically the compressed weight storage structure; and (2) load redundancy elimination, which requires each kernel’s pattern, the connectivity between kernels and input/output channels, and the exact input/output tile size and unroll factor. After these optimizations, high-level LR can generate compressed model and associated optimized model execution code by using the pattern-related information and other basic layer information extracted from DNN models, (e.g., the kernel size, computation stride, computation dilation, etc). Figure 2.7 shows the optimization flow and two sample code skeletons (+Reorder and +LRE) for these two optimizations, respectively.

```

device: [CPU]
layers:
- name: "conv_op1"
  storage: "tight"
  pattern: {"type": [1, 2], "layout": FKW, ...}
  tuning: {"unroll": [4, 2, 8, 1], "tile": [16, 32, 8],
           "permute": cohwc_i_b, ...}
  info: {"strides": [1, 1], "dilations": [1, 1], ...}

```

Figure 2.8: An LR example for a CONV layer.

Figure 2.8 shows a simplified LR example for a CONV layer (with 2-D kernels). This LR will generate execution code for CPU (`device`). Two types of kernel patterns (`[1, 2]`) present in this layer (`patterns`) and the filter kernels' pattern layout is specified by our FKW compressed weight storage format (clarified in Section 2.5.3 in detail)². Its computation loop permutation is `cohwc_i_b`, i.e., in the order of output channel, output height, output width, and input channel, with blocking and unrolling. Their blocking sizes are specified in `tile`. Their unrolling factors are specified in `unroll`. Figure 2.7 (+`Reorder`) also shows the execution code generated from this LR, in which the outer loops iterating on all tiles are omitted. The inner-most iteration processes kernels in each filter in the order of their pattern types, i.e., all kernels with pattern 1 in each filter will be processed at first, then kernels with pattern 2. This code optimization does not require any loop control-flows. This is guaranteed by our filter kernel reorder that is introduced in Section 2.5.2 in details.

2.5.2 Filter Kernel Reorder (FKR)

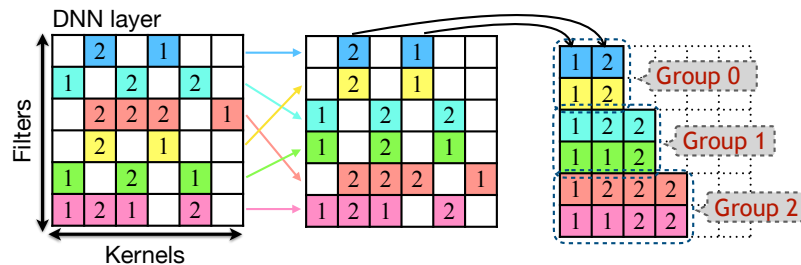


Figure 2.9: An example of filter kernel reorder.

²This LR is used after our filter kernel reorder, so the pattern information is stored in the optimized FKW format. Before reorder, a relatively loose data format is used, which is omitted due to the space limit.

Kernel pattern and connectivity pruning offer better opportunities to address the performance challenges in non-structured pruning thanks to its better regularity. Specifically, Filter kernel reorder (FKR) is designed to address two key challenges, i.e., heavy control-flow instructions, and thread divergence and load imbalance. Our basic insight is: for a specific DNN layer, the patterns of all kernels are already known after model training, so the inference computation pattern is also known before model deployment. FKR leverages this knowledge to organize the filters with similar kernels together to improve *inter-thread* parallelization and order the same kernels in a filter together to improve *intra-thread* parallelization.

Figure 2.9 explains FKR with a simplified example. Here, a matrix represents a CONV layer of DNN and each cell is a kernel with pattern type denoted by the number on it. Empty kernels are the ones pruned by **connectivity pruning**. The kernels in the same row belong to the same filter, and are marked with the same color.

Before the reorder, kernels with different patterns are distributed in this DNN layer. When performing the convolution operation directly, the execution code will contain many branches (as the `+No-opt` code in Figure 2.7) that incur significant instruction pipeline stalls and thread divergences, hurting both instruction- and thread-level parallelism. According to our experimental results in Section 2.6, this version results in sub-optimal performance.

FKR is composed of two steps: *filter reorder* and *kernel reorder*. The filter reorder organizes similar filters next to each other and the kernel reorder groups kernels with identical patterns in each filter together. Particularly, the *filter similarity* used in filter reorder is decided by two factors: first, the number of non-empty kernels in each filter (i.e., the length of each filter); and second, for filters with the same length, the number of kernels at identical positions with identical pattern IDs when the kernels in each filter are ordered according to these IDs.

After the reorder, the filters with the same length are grouped together, and in each group, the filters with the highest degree of similarity are ordered next to each other. The code `+Reorder` in figure 2.7 is for the execution of this reordered layer. This code shows

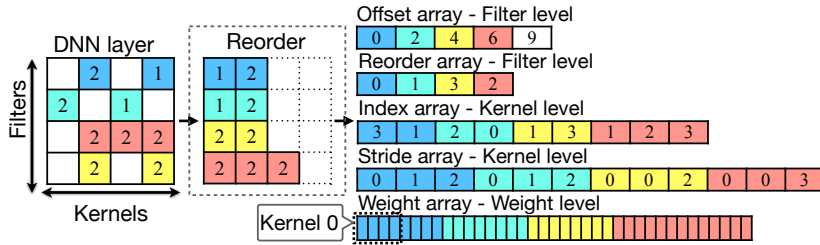


Figure 2.10: An example of FKW compressed weight storage.

much better instruction-level parallelism because it eliminates all branches. In addition, it also allows the better exploration of thread-level parallelism, because it results in large thread execution similarity and good load balance, particularly, considering the example of mapping the filters in the same group to the same GPU thread block.

2.5.3 Compressed DNN Weight Storage (FKW Format)

After FKR, our LR stores the DNN’s weights in a novel compact format (called FKW, standing for Filter-Kernel-Weight format). Compared with existing compact data formats (like CSR), FKW is higher-level and results in much less extra structure overhead (i.e., the total size of all index arrays that are used for weights data access). In addition, FKW leverages the pattern information, and stores the kernels with the FKR information that will support later branch-less DNN execution. Other compact data format cannot support this.

Figure 2.10 shows an example. This DNN layer consists of four filters, each with 2, 2, 2, and 3 (after FKR) non-empty kernels, respectively. The two kernels in the first filter (marked as blue) have pattern 1 and 2, corresponding to the input channel 3 and 1, respectively. FKW uses five arrays to represent this DNN layer: offset array, reorder array, index array, stride array, and weight array. The offset array and reorder array store filter-level information, index array and stride array store kernel-level, and the weight array stores actual weights.

More specifically, the offset array stores the offset of each filter (in terms of the number of non-empty kernels). In Figure 2.10, the offset of filter 0 is 0, and the offset of filter

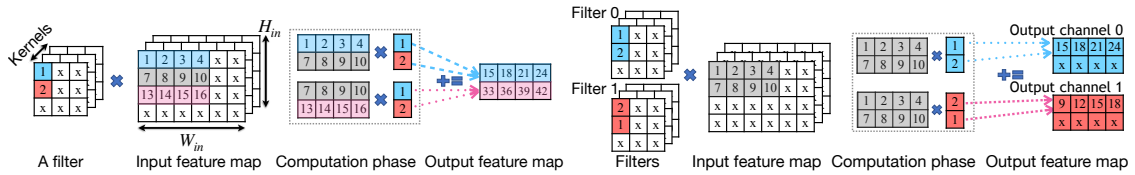


Figure 2.11: Load redundancy elimination (left: kernel-level; right: filter-level).

1 is 2 because there are two kernels in filter 0, and so on. The reorder array shows the reorder information that is used for accumulating the computation output to the correct output channel. In Figure 2.10, the reorder array tells us that filter 2 and filter 3 have been switched and their computation results should also be switched to the corresponding output channel. The index array represents the corresponding input channel for each non-empty kernel. In Figure 2.10, kernel 1 in filter 0 corresponds to the input channel 3, and kernel 2 corresponds to the input channel 1. So, the first two elements in the index array are 3 and 1, respectively. The stride array denotes the number of kernels in each pattern within the same filter. In Figure 2.10, the filter 0 has the stride array values 0, 1, and 2, denoting that the filter 0 has 1 kernel with pattern 1 ($1 = 1 - 0$), and 1 kernel with pattern 2 ($1 = 2 - 1$). In this example, each kernel has four (non-zero) weights, so each filter has 8, 8, 8, and 12 weights (after FKR), respectively.

2.5.4 Load Redundancy Elimination (LRE)

As discussed before, irregular memory access (in the form of array indirection) is also a major cause of inefficient execution of weight pruned DNNs. PatDNN uses two techniques to address this issue: (1) a conventional input tiling to improve the cache performance; and (2) the optimized code generation with the help of the pre-defined pattern information. The first one, specifically the determination of the optimal tiling size will be introduced in Section 2.5.5. This section focuses on the second, specifically, introducing our novel redundant *register* load elimination optimization applied in code generation procedure.

Our key insight is: in DNN execution, such as a convolution operation, the data access pattern of the input and output is decided by the (non-zero elements) patterns of kernels

that are already known after training. Therefore, it is possible to generate the optimized data access code with this information for each pattern of kernels and call them dynamically during the DNN execution. The generated codes consist of all statically determined data access instructions for the kernel-level computation with a careful instruction reorganization to 1) eliminate all indirect memory accesses; and 2) eliminate all redundant *register* load operations. The elimination of all indirect memory accesses is relatively straightforward, because in all data access instructions, the index of input data can be directly calculated from kernel pattern. We next explain two novel register-level load redundancy elimination methods in details.

Figure 2.11 illustrates both register-level load redundancy eliminations: the left one is within each kernel, and the right one is among multiple kernels. Within each kernel, the load redundancy is caused by the convolution operation. In the example (shown on the left part of Figure 2.11), the kernel value 1 requires the elements in the first two rows of the input matrix while value 2 requires the second and third rows. The elements in the second row [7,8,9,10] are loaded twice (from cache to register). PatDNN eliminates this load redundancy by explicitly reusing the (SIMD) registers that already hold the required data (like the second row in the above example).

Multiple kernels on the same position of different filters may share the same pattern and input channel. The input data required by these kernels are exactly identical. The right-hand side of Figure 2.11 shows a concrete example. If the computation of these filters on identical data is packed together, the possible redundant load of this input can be eliminated. PatDNN explores this optimization when it generates the optimized memory access code. The FKR organizes the kernels (in different filters) with identical patterns together. Together with a filter-level (or output channel) loop unrolling when processing these kernels, the redundant register load is eliminated. Figure 2.7 (+LRE) shows an example of this unrolling code.

It is worth noting that the above two redundancy elimination opportunities are more straightforward to exploit for dense models where the memory accesses of kernel weights

are continuous and the data reuse pattern is periodically repeated. However, it is very challenging (or even not possible) to exploit for pruned sparse models with irregular memory accesses, because it is hard to detect the data reuse pattern (or the data reuse pattern does not even exist). Our pattern-based pruning can preserve the data reuse patterns and help the compiler to detect them, thus re-enabling these two kinds of register-level load redundancy elimination.

2.5.5 Parameter Auto-tuning

Many configuration parameters require careful tuning to guarantee the performance of the generated execution code. However, manual tuning is tedious, and hard to yield the optimal code. Therefore, PatDNN also includes an auto-tuning component for selecting the best execution configuration.

It consists of two parts: first, an *explorer model* based on Genetic Algorithm to generate the configuration exploration space; and second, a *performance estimation model* created from our historical data to predict the possible best configuration and performance for a given hardware. Compared with the simulated annealing in TVM, our explorer model supports better parallelism because it allows the initialization of an arbitrary number of chromosomes to start the search. For a typical (large-scale) DNN like VGG-16, our exploration can complete in 3-5ms. During the exploration, history data is also collected for training the performance estimator (based on Multilayer Perceptron and least square regression loss). The advantage of this approach is that when deploying PatDNN on a new platform, it can give a quick prediction of the optimal configuration parameters as well as the possible execution time. In addition, these tuning parameters are crucial to the performance of our PatDNN execution, thus need to be carefully tuned by our auto-tuning module including: data placement configurations on GPU, tiling sizes, loop permutations, and loop unrolling factors.

Table 2.5: DNNs characteristics (under kernel pattern and connectivity pruning): Accu: ImageNet top-5, CIFAR top-1; the negative values in Accuracy Loss actually mean accuracy improvement.

Name	Network	Dataset	Layers	Conv	Patterns	Accu(%)	Accu Loss (%)
VGG	VGG-16	ImageNet	16	13	8	91.6	0.1
		CIFAR-10	16	13	8	93.9	-0.4
RNT	ResNet-50	ImageNet	50	49	8	92.5	0.2
		CIFAR-10	50	49	8	95.6	-1.0
MBNT	MobileNet-V2	ImageNet	53	52	8	90.3	0.0
		CIFAR-10	54	53	8	94.6	-0.1

2.6 Evaluation

This section evaluates the execution performance of PatDNN by comparing it with three state-of-the-art DNN inference acceleration frameworks, TFLite [1], TVM [25], and MNN [96]. All major optimizations of these frameworks (and our PatDNN) are summarized in Table 2.1.

2.6.1 Methodology

Evaluation Objective: Our overall evaluation demonstrates that achieving real-time inference of large-scale DNNs on modern mobile devices is possible with PatDNN. Specifically, the evaluation has five objectives: (1) demonstrating that PatDNN outperforms existing state-of-the-art DNN frameworks without any accuracy compromise; (2) studying the performance effect of our key compiler optimizations and explaining the reasons for performance improvement; (3) further confirming the performance of PatDNN by comparing its pure GFLOPS with our optimized dense baseline; (4) showing that PatDNN performs similarly on different mobile platforms, i.e., PatDNN has a good portability; and (5) unveiling the impact of pattern count selections on both the accuracy and performance.

DNNs and Datasets: PatDNN is evaluated on three mainstream DNNs, VGG-16 (VGG), ResNet-50 (RNT), and Mobile-Net-V2 (MBNT). They are trained on two datasets, ImageNet and CIFAR-10. Table 2.5 characterizes these trained DNNs. Some information is omitted

Table 2.6: VGG unique CONV layers’ filter shapes and given names.

Name	Filter shape	Name	Filter shape	Name	Filter shape
L1	[64,3,3,3]	L4	[128,128,3,3]	L7	[512,256,3,3]
L2	[64,64,3,3]	L5	[256,128,3,3]	L8	[512,512,3,3]
L3	[128,64,3,3]	L6	[256,256,3,3]	L9	[512,512,3,3]

due to the space constraint, e.g., a uniform CONV pruning rate for VGG and RNT is $8\times$, and $4.4\times$, respectively (with uniform $3.6\times$ connectivity pruning rate). VGG has 13 CONV layers, and 5 of them have identical structures to others. Table 2.6 lists the filter shape ([#output channel, #input channel, kernel height, and kernel width]) of these 9 unique layers and gives them a short name each.

Evaluation Platforms and Running Configurations: Our experiments are conducted on a Samsung Galaxy S10 cell phone with the latest Qualcomm Snapdragon 855 mobile platform that consists of a Qualcomm Kryo 485 Octa-core CPU and a Qualcomm Adreno 640 GPU. Our portability tests are conducted on a Xiaomi POCOPHONE F1 phone with a Qualcomm Snapdragon 845 that consists of a Kryo 385 Octa-core CPU and an Adreno 630 GPU, and an Honor Magic 2 phone with a Kirin 980 that consists of an ARM Octa-core CPU and a Mali-G76 GPU. All tests run 50 times on different input (images) with 8 threads on CPU, and all pipelines on GPU. Because multiple runs do not vary significantly, this section only reports the average time for readability. Because CONV layers are most time-consuming, accounting for more than 95% (90% for VGG) of the total execution time, our evaluation focuses on the CONV layers. All runs are tuned to their best configurations, e.g., Winograd optimization [122] is used for all dense runs, and 16-bit float point is used for all GPU runs.

2.6.2 Overall Performance

Figure 2.12 shows the overall CPU and GPU performance of PatDNN compared to TFLite, TVM, MNN on all six trained DNNs. PatDNN outperforms all other frameworks for all cases. On CPU, PatDNN achieves $12.3\times$ to $44.5\times$ speedup over TFLite, $2.4\times$ to $5.1\times$ over

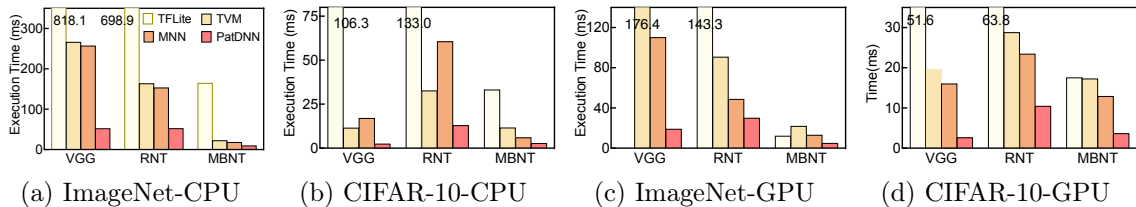


Figure 2.12: Overall performance: x-axis: different trained DNN models; y-axis: average DNN inference execution time on a single input.

TVM, and $1.9\times$ to $7.1\times$ over MNN, respectively. On GPU, PatDNN achieves $2.5\times$ to $20\times$, $2.8\times$ to $11.4\times$, and $1.6\times$ to $6.2\times$ speedup over TFLite, TVM, and MNN, respectively³. For the largest DNN (VGG) and largest data set (ImageNet), PatDNN completes CONV layers on a single input within 18.9 ms on GPU. Even including the other rest layers (like FC), PatDNN can still meet the real-time requirement (usually 30 frames/sec, i.e., 33 ms/frame).

PatDNN outperforms other frameworks because of two major reasons. First, its dense version is already $1.1\times$ to $1.6\times$ faster than TVM and MNN on mobile platforms because of some extra optimizations (as shown in Table 2.1). Figure 2.17(a) shows that PatDNN’s dense version is faster than MNN on VGG, our largest DNN. Second, the pattern-based pruning reduces the overall computation by $3\times$ to $8\times$. Such computation reduction unfortunately cannot transfer to performance gains directly. We confirmed this by implementing an optimized sparse matrix version of PatDNN based on CSR [66], which shows almost the same speed to PatDNN’s dense version. However, the subsequent compiler-level optimizations (filter kernel reorder, load redundancy elimination, auto-tuning, and compressed weight storage) successfully convert this computation reduction into real performance gains. We conduct a more detailed study on these optimizations in the next Section, and Figure 2.13 shows a break-down of these optimizations’ contributions. Figures 2.14 to 2.16 provide a detailed analysis of the underlying reasons.

³TFLite does not support executing VGG on ImageNet data set on GPU due to its too large memory footprint.

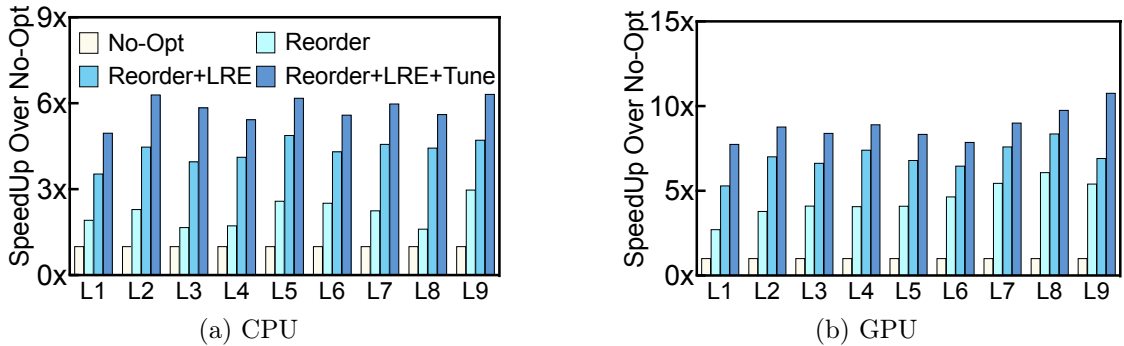


Figure 2.13: Speedup of opt/no-opt on each unique CONV layer.

2.6.3 Optimization Evaluation

This section studies the effect of our key compiler optimizations and shows that our PatDNN’s good performance mainly comes from these pattern-enabled optimizations. This part also compares the extra structure overhead between FKW and CSR. Constrained by space, we only report the results of VGG, our most complex DNN, on the most widely accepted dataset (ImageNet). Experiments on other DNNs and datasets show the same trend. The rest parts also use VGG on ImageNet as a representative example.

Figure 2.13 reports the speedup of the versions with optimizations over the version without any optimization on each unique CONV layer of VGG on CPU and GPU, respectively. On CPU, reorder brings 1.6 \times to 3.0 \times speedup, load redundancy eliminations bring additional 1.6 \times to 2.8 \times speedup, and parameter tuning brings additional 1.2 \times to 1.9 \times speedup. On GPU, these numbers are 2.7 \times to 6.1 \times , 1.5 \times to 3.3 \times and 1.4 \times to 3.8 \times . It is interesting that FKR brings more benefits on GPU than on CPU, because GPU’s performance is more sensitive to the thread divergence and load balance due to its massive parallel nature. We next study why these optimizations work.

Filter Kernel Reorder: Figure 2.14 (a) reports the filter length distribution of VGG L4 before and after FKR. Before reorder, the filters with varied lengths are distributed randomly, resulting in significant load imbalance if assigning them to different threads. After reorder, the filters are grouped into three groups, and the filters within each group

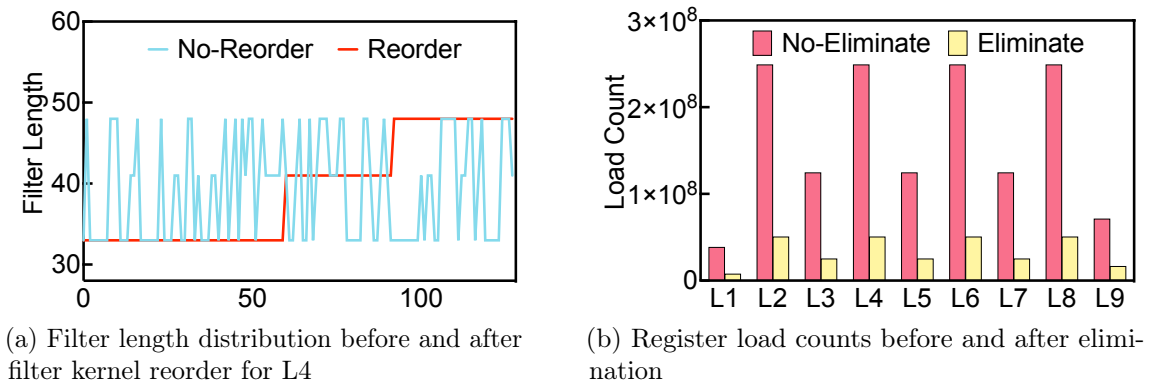


Figure 2.14: Profiling result: reorder and redundancy elimination.

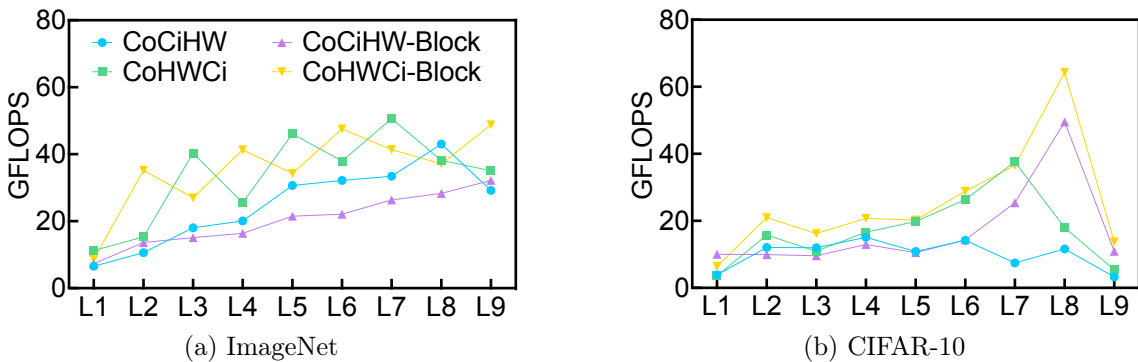


Figure 2.15: Effect of different loop permutations and loop tiling.

have identical lengths. Each group could be executed by CPU threads simultaneously, or mapped to the same GPU thread block.

Load Redundant Elimination: Figure 2.14 (b) reports the register load counts before and after LRE for each unique CONV of VGG. It shows that our register LRE can significantly reduce the number of register loads. Note that even if register load has lower latency than cache or memory load, the memory/cache performance has nevertheless been aggressively optimized by conventional tiling. Thus, the significant performance gains must have been achieved with the reduced number of register loads.

Auto-tuning: Figure 2.15 reports the CPU performance (in GFLOPS) of each unique VGG CONV layer with varied loop permutations, and with or w/o blocking on ImageNet and CIFAR-10, respectively. It shows that different inputs and layers may require different

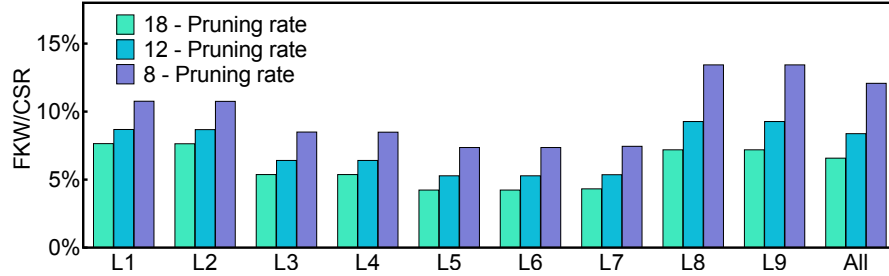


Figure 2.16: Extra data structure overhead: FKW over CSR on unique VGG CONV layers with different pruning rates.

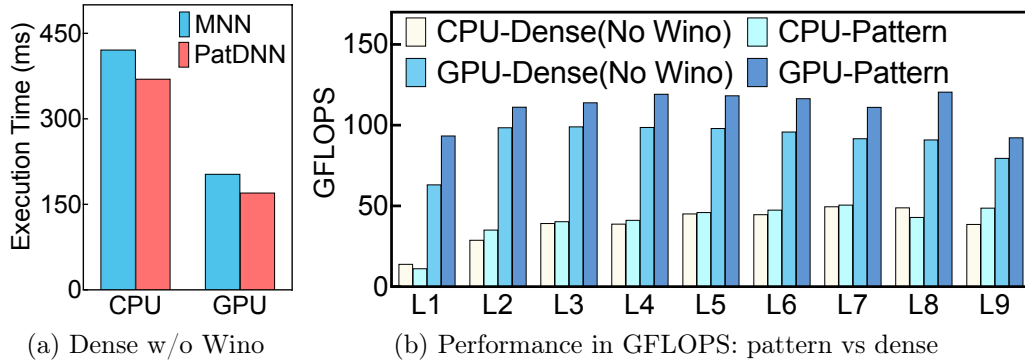


Figure 2.17: GFLOPS performance study: PatDNN vs dense.

configurations. Proper tuning will bring significant benefits. Constrained by space, we omit the GPU results and tuning results about GPU data placement.

Compressed Weight Storage: Figure 2.16 shows the extra data structure overhead (i.e., the size of data structures other than weights) of FKW over CSR on each unique VGG CONV layer with three kinds of pruning rates, $18\times$, $12\times$, and $8\times$ respectively. For each one, FKW saves 93.4%, 91.6%, and 87.9% extra data structure overhead over CSR in total, resulting in 46.7%, 45.8%, and 43.9% overall storage space saving.

2.6.4 PatDNN Performance Analysis in GFLOPS

To further analyze the performance of PatDNN, this part compares its pure GFLOPS with our dense implementation. To conduct an apple-to-apple comparison, we turn off the Winograd optimization that transforms the convolution operation to matrix-multiplication for a trade-off between the computation reduction and operation conversion overhead.

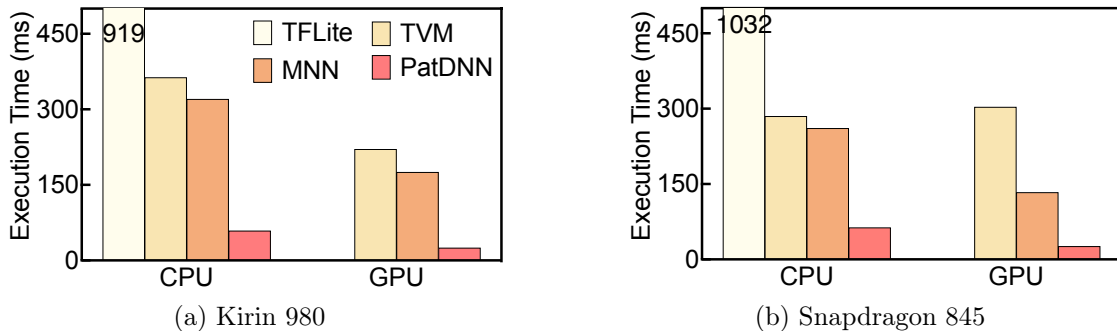


Figure 2.18: Portability study: performance on two other platforms.

Table 2.7: Pattern counts impact (with $3.6\times$ connectivity pruning): accuracy loss and execution time for VGG.

Network	Dataset	#Patterns	Accu (%)	Accu Loss (%)	Device	Time (ms)
VGG-16	ImageNet	6	91.4	0.3	CPU	50.5
					GPU	18.6
		8	91.6	0.1	CPU	51.8
					GPU	18.9
		12	91.7	0.0	CPU	92.5
					GPU	27.6

Figure 2.17 (a) shows that our dense version can serve as an optimized baseline, because it is even faster than MNN.

Figure 2.17 (b) shows that our pattern-based (sparse) PatDNN achieves comparable GFLOPS to our optimized dense baseline on CPU, and outperforms it on GPU. It implies that the memory performance of PatDNN is comparable to the dense baseline on CPU and even better than it on GPU. This benefits from our model compression and memory load (and register load) reductions. Without pattern-based pruning, the input, output, and DNN model compete for the limited memory/cache resource; after pruning, only the input and output compete for it. PatDNN also reduces the overall computation; thus, it significantly outperforms all other mobile frameworks. We cannot achieve this performance without our pattern-based design, and our other sparse implementation with conventional sparse matrix optimizations can only get either comparable or even slower speed than other mobile frameworks.

2.6.5 Portability Study

PatDNN is also evaluated on two other platforms to confirm its portability. Figure 2.18 shows the result. On these platforms, PatDNN also outperforms other frameworks. Particularly, other frameworks run much slower on Magic 2 than on Snapdragon 855; however, PatDNN performs more stably. This is because our pattern-based pruning leads to fewer computations and fewer memory accesses thus reducing the memory bandwidth pressure.

2.6.6 Impact of Pattern Counts

Table 2.7 reports the impact of the pattern count selection on both the accuracy and execution time, under $3.6\times$ uniform connectivity pruning rate. As increasing pattern counts, the accuracy increases slightly, however, the performance drops quickly. Our evaluation selects 8 patterns that result in ideal performance with a negligible accuracy loss.

2.7 Discussion

Generality: The techniques proposed in PatDNN are general enough to be applied to other platforms. Compared to laptops or servers, mobile platforms are more resource-constrained, making it is more challenging to achieve real-time execution. However, the need for real-time DNN execution is crucial due to many important mobile applications. In fact, in addition to the mobile platforms in our paper, we also tested PatDNN on the latest Raspberry Pi 4 platform. It shows a similar speedup over other frameworks like TVM. We believe that it is a promising research direction to improve PatDNN’s portability by incorporating it with TVM that emphasizes the DNN execution on varied computing devices.

Dense vs. Sparse DNNs: General end-to-end DNN inference acceleration frameworks like TFLite, TVM, and MNN do not support sparse DNN execution. If we simply add sparse DNN support with random pruning and general compression storage (like CSR) to these frameworks, it is expected that their speed cannot be improved significantly as shown

in the results of PatDNN’s CSR implementation. Although there is potential to improve the performance with coarse-grained structured pruning (that prunes whole filters/channels), the accuracy will be obviously degraded as we discussed before. From this perspective, PatDNN opens a new door to accelerate DNN execution with a compression/compiler-optimization co-design. With such co-design, sparse (or compressed) DNN execution becomes a more promising solution in resource-constraint environments than dense DNN.

2.8 Summary

This paper presents PatDNN, an end-to-end framework to achieve real-time DNN execution on mobile devices. PatDNN consists of two stages, a pattern-based pruning stage based on extended ADMM solution framework, and an optimized execution code generation stage including a high-level, fine-grained DNN layerwise representation and a set of architecture-aware optimizations. This design allows PatDNN to benefit from both high accuracy and hardware efficiency. Our evaluation results demonstrate that PatDNN outperforms other state-of-the-art end-to-end DNN execution frameworks with up to $44.5\times$ speedup and no accuracy compromise, and achieves real-time execution of large-scale DNNs on mobile devices.

Chapter 3

DNNFusion: Accelerating Deep Neural Networks Execution with Advanced Operator Fusion

3.1 Introduction

The past ten years have witnessed a resurgence of Machine Learning, specifically in the form of Deep Learning. Deep Neural Networks (DNNs) such as Convolution Neural Networks (CNN) and Recurrent Neural Networks (RNN) serve as the state-of-the-art foundation and core enabler of many applications that have only emerged within the last few years and yet have become extremely popular among all users of computing today [12, 197]. Behind the success of deep learning are the increasingly large model sizes and complex model structures that require tremendous computation and memory resources [39]. There is a difficult trade-off between increasing complexity of DNNs (required for increasing accuracy) and deployment of these DNNs on resource-constrained mobile devices (required for wider reach).

In recent years, there has been a significant emphasis on optimizing the execution of

large DNNs. Operator fusion (or *kernel/layer* fusion) has been a common approach towards improving efficiency of DNN execution [1, 25]. The basic idea of such fusion is the same as the traditional loop fusion done by optimizing compilers [100, 103, 151], and they lead to the following benefits: (i) eliminating unnecessary materialization of intermediate results, (ii) reducing unnecessary scans of the input; and (iii) enabling other optimization opportunities. Traditional end-to-end frameworks like TensorFlow Lite [1], TVM [25], MNN [96], and Pytorch-Mobile [165] all have operator fusion optimizations, which are broadly based on recognizing certain fusion patterns. These transformations have generally been based on a representation called *computational graph* [25], which views the application as a set of operations on tensors, and representation of dependencies in the form of consumption of tensor(s) output by an operation by another operation.

In this paper, we observe that the fusion patterns considered in the past work [1, 25] are too restricted to cover the diversity of operators and layer connections that are emerging. For example, ONNX (Open Neural Network Exchange) [160] lists 167 distinct operators, and creating fusion patterns based on their combinations is unlikely to be a feasible approach. At the same time, traditional compiler loop transformations (including fusion [103, 151]) work on a low-level view of the computation, i.e., loop (indices) and dependence between array elements. More recent work on loop fusion has been based on polyhedral analysis [16], with several different resulting algorithms [2, 3, 15]. Polyhedral analysis, while providing an excellent foundation to rigorously reason the legality of, and explore the space of, loop transformations, can be an “overkill” to capture the relatively simple data structures (tensors) and operations (without loop-carried dependencies) in DNNs. Moreover, polyhedral analysis is normally limited to affine-loop analysis and transformations (although latest efforts [202, 226, 228] do extend it to certain non-affine loop optimizations), and cannot capture certain operation (combinations) in DNNs. An example will be a combination of **Gather**, which copies input to output indirectly using an index array followed by **Flatten**, which changes the dimensionality of a tensor. Finally, the operator view in computational graphs can enable us to exploit properties of these computations, which may be lost when

a lower-level view of the computation is considered.

This paper presents DNNFusion, a rigorous and extensive loop fusion framework that can exploit the operator view of computations in DNNs, and yet can enable a set of advanced transformations. The core idea is to classify operators into different types, and develop rules for different combinations of the types, as opposed to looking for patterns with specific combination of operations. Particularly, we first classify the existing operations in a DNN into several groups based on the mapping between their input and output, such as One-to-One, One-to-Many, and others. We also enhance the computational graph representation into the Extended Computational Graph (ECG) representation, where the type (and other properties) of the operation are explicitly noted. Then, we design a mapping type analysis to infer the profitability of fusing operations of different combinations of these types of operators, binning the combination into three groups: likely profitable (and legal), likely not profitable, and ones where profitability may need to be determined through profile information.

Next, on the ECG representation, we apply a series of *graph rewriting rules* that we have developed. These rules exploit the mathematical properties of the operations and have a similar flavor to the classical optimization called *strength reduction* [29]. Unlike traditional compiler work, however, we apply these rules on operations on tensors (and not scalars) and our set of rules go well beyond the traditional ones. The rest of our framework comprises algorithms for determining fusion of specific operations (based on certain heuristics) and generating optimized fused code. Almost each fusion generates a new operator (and its implementation) that is not present in the original library; however, once a new operator is generated, its implementation can be reused when the same pattern is detected in the same or a different model. Overall, we show that an operator view of the DNN can enable rigorous optimizations, beyond what will be possible with a lower-level view of the computation or the existing (simplistic) work on applying a small set of fusion patterns on the operator view.

In summary, this paper makes the following contributions:

- It designs high-level abstractions (including mapping type analysis and ECG) for operator fusion by leveraging high-level DNN operator information. The approach can handle a diversity of operators and yet enable aggressive optimizations.
- It proposes a novel mathematical-property-based graph rewriting to simplify ECG structure, optimize DNN computations, and facilitate subsequent fusion plan generation.
- It presents an integrated fusion plan generation by combining the benefit of efficient machine-independent mapping type analysis while leveraging a profiling result database.
- It implements optimized fusion code generation, integrating the approach into a state-of-the-art end-to-end DNN execution framework. The optimized framework with operator fusion is called DNNFusion.

DNNFusion is extensively evaluated on 15 cutting-edge DNN models with 5 types of tasks, varied model sizes, and different layer counts on mobile devices. Comparing with four popular state-of-the-art end-to-end DNN execution frameworks, MNN [96], TVM [25], TensorFlow-Lite [1], and Pytorch-Mobile [165], DNNFusion achieves up to $8.8\times$ more loop fusions, $9.3\times$ speedup with our proposed advanced operator fusion. Particularly, DNNFusion *for the first time* allows many latest DNN models that are not supported by any existing end-to-end frameworks to run on mobile devices efficiently, even in real-time. Moreover, DNNFusion improves cache performance and device utilization – thus, enabling execution on devices with more restricted resources – and reduces performance tuning time during compilation.

3.2 Blessing and Curse of Deep Layers

This section presents a study that motivates our work, by demonstrating that it is challenging to execute deep(er) neural networks efficiently, particularly on resource-constraint mobile devices, due to the high memory and computation requirements.

Table 3.1: An empirical study to motivate this work: The relation of overall computation, layer count, and execution efficiency of multiple DNNs. Results are collected on Qualcomm Adreno 650 GPU with an optimized baseline framework with fixed-pattern operator fusion that outperforms all state-of-the-art DNN execution frameworks (called **OurB+** and will be introduced later).

Model	#Total layer	IR size	#FLOPS	Speed (FLOPs/S)
VGG-16 [206]	51	161M	31.0B	320G
YOLO-V4 [13]	398	329M	34.6B	135G
DistilBERT [201]	457	540M	35.3B	78G
MobileBERT [212]	2,387	744M	17.6B	44G
GPT-2 [188]	2,533	1,389M	69.1B	62G

As we stated earlier, there has been a trend towards deeper DNNs. With increasing amount of computation, there has also been a trend towards reducing the computation by reducing the weight size. Consider the well-known Natural Language Processing (NLP) model, BERT [46] as an example. TFLite takes 985ms to inference BERT on the latest CPU of Snapdragon 865. In recent efforts [188, 212] (MobileBERT, GPT-2), machine learning researchers have addressed this issue by reducing the weight size on each layer and thus training thinner and deeper models to balance the computation workload and model accuracy.

However, we have observed that the depth of the model is the critical impediment to efficient execution. Our experimental study has correlated execution efficiency with the total amount of computation and the number of layers (Table 3.1). Particularly, we can see that although DistilBERT [201] and VGG-16 [206] have a similar number of computations (while having 457 and 51 layers, respectively), DistilBERT’s execution performance (78 GFLOPs/S) is much worse than VGG’s (320 GFLOPs/S). This is mainly because of two reasons. First, models with more layers usually generate more intermediate results, thus increasing the memory/cache pressure. Second, deep models usually have an insufficient amount of computations in each layer, thus degrading the processor’s utilization, particularly for GPUs. Operator fusion can be an effective technique to reduce memory requirements and improve efficiency, and is the focus of our study.

3.3 Classification of DNN Operators and Fusion Opportunity Analysis

This section establishes the basis for our approach, by classifying DNN operators and their combinations.

3.3.1 DNN Operators Classification

This work carefully studied all operators supported by a popular (and general) DNN ecosystem ONNX (Open Neural Network Exchange) [160], and finds that the mapping relation between (each) input and output of each operator is critical to determine both the profitability and correct implementation of fusion optimization. Moreover, it is possible for us to classify all operators into five high-level abstract types based on the relationship between input elements and output elements. These five types are One-to-One, One-to-Many, Many-to-Many (which includes Many-to-One, but we do not consider it separately here), Reorganize, and Shuffle. This classification serves as the foundation of our proposed fusion framework. Table 3.2 shows more details of this operator classification and gives one or two representative examples for each mapping type. If an operator has only one input or multiple inputs with the same mapping type to the output, the mapping type of this operator is decided by its any input/output pair. If multiple input/output pairs with varied mapping types exist, this operator’s mapping type is decided by the more complex mapping type¹.

Assuming each input element can be denoted as $x[d_1, \dots, d_n]$, where x means the operand of an operator and d_1, \dots, d_n denotes the index for an element of an operand, the mapping types between one input and one output are classified as follows.

¹Order in increasing order of complexity: One-to-One, Reorganize, Shuffle, One-to-Many, and Many-to-Many

Table 3.2: Classification of DNN operators in mapping types. These operators are defined in ONNX [160].

Mapping type	Operators	Representative
One-to-One	Add, Asin, BatchNormalization, Cast, Ceil, Clip, Concat, Cos, Erf, Exp, Greater, Where, LeakyRelu, Log, Not, PRelu, Reciprocal, Relu, Round, Sigmoid, Sin, Slice, Split, Sqrt, Tanh	Add, Relu
One-to-Many	Elementwise w/ broadcast, Expand, Gather, Resize, Upsample	Expand
Many-to-Many	AveragePool, CONV, ConvTranspose, CumSum, Einsum, GEMM, InstanceNormalization, MaxPool, Reduce (e.g. ReduceProd, ReduceMean), Softmax	Conv, GEMM
Reorganize	Flatten, Reshape, Squeeze, Unsqueeze	Reshape
Shuffle	DepthToSpace, SpaceToDepth, Transpose	Transpose

- **One-to-One:** There is a set of functions F, f_1, \dots, f_n , such that

$$y[d_1, \dots, d_n] = F(x[f_1(d_1), \dots, f_n(d_n)])$$

and there is a 1-1 mapping between each $[d_1, \dots, d_n]$ and the corresponding $[f_1(d_1), \dots, f_n(d_n)]$ used to compute it.

- **One-to-Many:** There is a set of functions F, f_1, \dots, f_n , such that:

$$y[e_1, \dots, e_m] = F(x[f_1(d_1), \dots, f_n(d_n)])$$

where $m > n$, and there is a One-to-Many relationship between $[f_1(d_1), \dots, f_n(d_n)]$ and $[e_1, \dots, e_m]$.

- **Many-to-Many:** There is a set of functions $f_1^1, \dots, f_n^1, \dots, f_1^k, \dots, f_n^k$, such that:

$$y[e_1, \dots, e_m] = F(x^1[f_1^1(d_1), \dots, f_n^1(d_n)], \dots, x^k[f_1^k(d_1), \dots, f_n^k(d_n)]).$$

- **Reorganize:** We have

$$y[e_1, \dots, e_m] = x[f_1(d_1), \dots, f_n(d_n)]$$

and there is a 1-1 relationship between each $[e_1, \dots, e_m]$ and the corresponding $[f_1(d_1), \dots, f_n(d_n)]$.

Table 3.3: Mapping type analysis. The first column and the first row (both without color) show the mapping types of first and second operators, respectively, before fusion, and the colored cells show the mapping type of the operator after fusion. Green implies that these fusion combinations can be fused directly (i.e., they are profitable). Red implies that these fusions are unprofitable. Yellow implies that further profiling is required to determine profitability.

Second op \ First op	One-to-One	One-to-Many	Many-to-Many	Reorganize	Shuffle
One-to-One	One-to-One	One-to-Many	Many-to-Many	Reorganize	Shuffle
One-to-Many	One-to-Many	One-to-Many	×	One-to-Many	One-to-Many
Many-to-Many	Many-to-Many	Many-to-Many	×	Many-to-Many	Many-to-Many
Reorganize	Reorganize	One-to-Many	Many-to-Many	Reorganize	Reorganize
Shuffle	Shuffle	One-to-Many	Many-to-Many	Reorganize	Shuffle

- **Shuffle:** There is a set of functions F, f_1, \dots, f_n , where F is a *permutation* function, such that,

$$y[e_1, \dots, e_n] = x[f_1(d_{F(1)}), \dots, f_n(d_{F(n)})].$$

3.3.2 Fusion Opportunity Analysis

Based on the mapping type of each operator, this work proposes a new fusion analysis. The basic idea is that given two fusion candidate operators with a certain combination of mapping types, it is possible to: 1) infer the mapping type of the resulting fused operation; and 2) simplify the profitability evaluation and correct implementation of this fusion.

Table 3.3 shows the details of this analysis. The first column and the first row (without any color) show the mapping types of the first and the second operator to be fused and the colored cells show the mapping type of the resulting operator. It further classifies the fusion of this combination of mapping types into three groups (shown as green, yellow, and red, respectively). Green implies that these fusions are legal and profitable and no further analysis is required. Red implies that these fusions are known to be either illegal or clearly not profitable. Yellow implies that these fusions are legal; however, further profiling is required to determine profitability. This analysis eliminates the need for anytime runtime

analysis or autotuning for red and green cases. For remaining (yellow) cases, we can further accelerate compilation using a *profiling database* that stores the execution results of various fusion combinations collected offline.

These five mapping types have a range of what we call *transformation impedance* (which we informally define as a metric to qualitatively express the difficulty to fuse), i.e., when they are fused with another type, they have different capability of deciding the fused mapping type. One-to-One has the lowest transformation impedance among all five types, whereas Reorganize and Shuffle’s transformation impedance is in the middle, i.e., they can transform One-to-One to their types while they cannot transform others. One-to-Many and Many-to-Many have the strongest transformation impedance, i.e., the resulted mapping type is decided by them solely when they are fused with other operators. Moreover, One-to-Many and Many-to-Many have the same capability, and Reorganize and Shuffle have the same as well.

We elaborate on the following representative combinations to provide intuition behind the Table 3.3.

- *One-to-One with others.* When a One-to-One operator (Op_1 with the input I and the output O) is fused with an operator of any type (Op_2), i.e., Op_2 takes O as the input, the memory access to each element of O can be mapped to the access to each element of I , as long as this mapping function is known. Unlike general programs where the dependencies can be more complex, the use of tensors and a limited set of operators limits the type of mappings, and DNN operators carry this mapping information. Our analysis leverages this high-level operator information to ensure the correctness of these fusions. Moreover, this fusion usually requires limited number of registers and does not incur extra overhead like data copying or redundant computations, so they are profitable. Take a case that fuses **Add** and **GEMM** in either order. Each element in the output of **Add** can be replaced by two elements in the two inputs of **Add**, ensuring correct and profitable fusion, irrespective of the order of these operations.
- *Reorder or Shuffle with others.* Both types are variants of One-to-One with a special

mapping function between the input and the output. Above reasons for the correctness analysis are also applied here; however, when fusing with One-to-Many or Many-to-Many types operators, profitability needs to be validated with further profiling because of the possibility of introduced data copying, change in data access order, or redundant computations. As an example, consider **Expand** and **Transpose** operators – **Expand** copies the input tensor with a continuous memory access pattern, whereas, **Transpose** transposes the input tensor to the output tensor according to the permutation in operator properties. Thus, the resulting fused operation may not have continuous memory accesses.

- *One-to-Many with Many-to-Many.* Take the case that **Expand** followed by **Conv** – as **Conv** reads the feature map input tensor with continuous access, while a One-to-Many operator can distribute the continuous input tensor elements. As it is very desirable for the (compute-intensive) Many-to-Many operators to read the input tensors in a continuous way, we consider this fusion unprofitable.
- *Many-to-Many with Many-to-Many.* When a Many-to-One mapping operator is followed by a Many-to-One operator, e.g. **Conv** followed by another **Conv**, attempting a combined execution will be too complicated and will likely negatively impact register and cache usage. Thus, we consider them unprofitable.
- *Many-to-Many with One-to-Many.* When a Many-to-One mapping operator is followed by a One-to-Many operator, e.g. **Conv** followed by **Expand** or **Resize**, a combined execution may or may not have a desirable data access pattern. When **Conv** is combined with **Expand**, as **Expand** operator only expands a single dimension of the input, so it will not adversely affect the computation pattern of **Conv**. On the other hand, if **Conv** is combined with a **Resize** that will copy the input tensor along different dimensions, it can negatively impact the computation of **Conv**. Thus, we consider such cases to be requiring further profiling.

Extended Computational Graph. Based on the analysis above and as a background for the methods we will present next, we introduce Extended Computational Graph (ECG)

as our intermediate representation (IR). As the name suggests, this represents builds on top of the (traditional) Computational Graph [25], which captures the data-flow and basic operator information like the operator type and parameters. ECG contains more fusion-related information, including `mapping_type` indicating the mapping type of each operator, `IR_removable` denoting if an intermediate result can be removed completely (which is true only if all its successors can be fused and which is calculated during fusion), and mathematical properties of the operations like whether the associative, commutative, and/or distributed properties hold.

3.4 DNNFusion’s Design

3.4.1 Overview of DNNFusion

Figure 3.1 shows an overview of DNNFusion. It takes the computational graph generated from compiler-based DNN execution frameworks (e.g., TVM [25], and MNN [96]) as the input, and adds key information to create the Extended Computational Graph (ECG). Based on this ECG, the main compiler optimization and code generation stage of DNNFusion consists of three components: ❶ mathematical-property-based graph rewriting (Section 3.4.2), ❷ lightweight profile-driven fusion plan exploration (Section 3.4.3), and ❸ fusion code generation and other advanced fusion-based optimizations (Section 3.4.4).

3.4.2 Mathematical-Property-Based Graph Rewriting

DNNFusion first employs a mathematical-property based graph rewriting pass to optimize the Extended Computational Graph (ECG). With this pass, DNNFusion is able to 1) remove unnecessary operations, 2) eliminate redundant intermediate data copies, and 3) replace costly (combination of) operators with more efficient ones. This graph rewriting carried out here is in the spirit of the classical compiler optimization of strength reduction [29]; however, here it is performed on complicated operators on matrices or tensors rather than on scalar expressions. Moreover, the rules we present are more complex and involved,

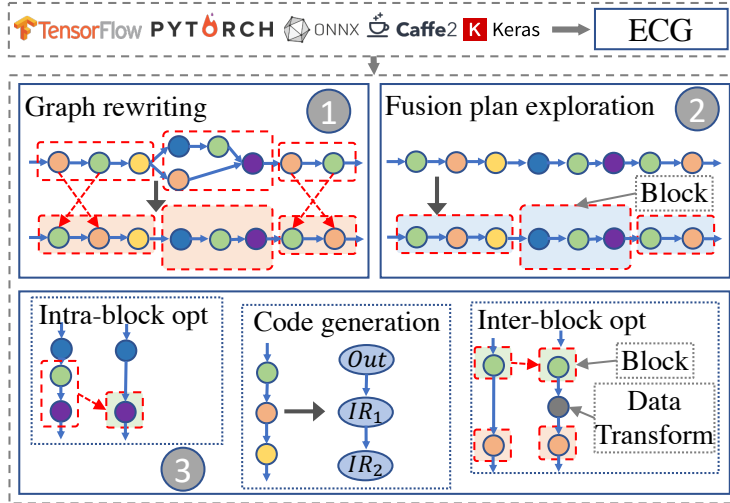


Figure 3.1: DNNFusion overview.

and are based on operations that are common in DNNs. More importantly, compared to existing efforts on computational graph substitution (e.g., TASO [94]), our graph rewriting is designed to work in conjunction with operator fusion and identifies a set of operators and rules for that specific purpose. Our evaluation results (Section 3.5) show that with graph rewriting, there are 18% fewer fused layers left after fusion on GPT-2. We also do an experimental comparison against TASO later in this paper.

Figure 3.2 shows specific examples of leveraged mathematical properties (distributive, communicative, and associative). Table 3.4 shows a more complete set of rules. This table also shows the computation size (in #FLOPS) before and after the rewriting. Our rules mainly focus on operators in the One-to-One mapping type (e.g., element-wise multiplication, addition, reciprocal, square root, and others) and several reduction operators that are in Many-to-Many (e.g., `ReduceSum` and `ReduceProd`) – this is because these operators usually follow our defined mathematical properties. DNNFusion uses #FLOPs (rather than temporary output size or memory footprint) as the metric to drive graph rewriting mainly because of two reasons: first, in most of the applications scenarios of these rules, the temporary output size keeps the same before and after graph rewriting, and second, the size of the temporary output in a majority of other cases becomes a non-issue because fusion

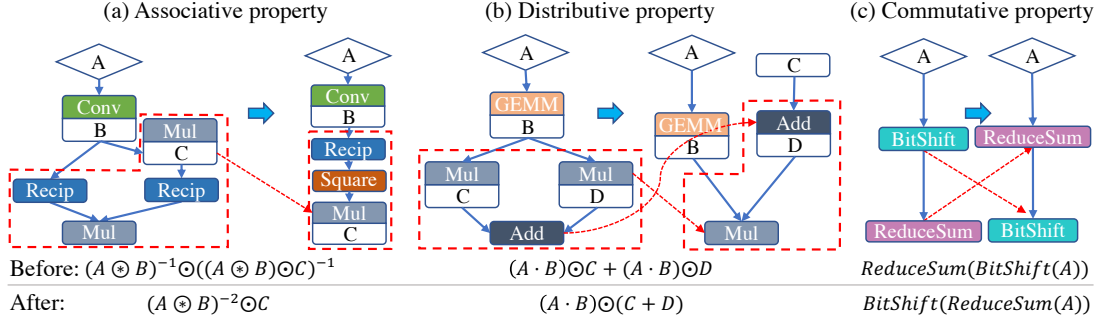


Figure 3.2: Examples of graph rewriting with mathematical properties. Associative property explores the optimal execution order of operators and replaces the expensive combination of operators with a cheaper one. Distributive property explores the common combination of operators and simplifies the computation structure. Commutative property switches the execution order of operators to reduce the overall computation. Note: the letter below each operator (e.g., B below Conv in (a)) or the letter in rectangle (e.g., C in (b)) denotes that this input is from model weights rather than an intermediate result. The letter in diamond (e.g., A) means that this is the input of this operator block, which could be the input of the model or intermediate result from a prior block. The intermediate results within this block are omitted for readability.

is applied after rewriting. For a small number of remaining cases, i.e., where temporary output size changes and the fusion is not applied, more sophisticated methods will be considered in the future.

We now elaborate on some of the rules presented in Table 3.4, which were also depicted in Figure 3.2.

- **Associative:** By leveraging the associative property, the graph rewriting pass can identify an optimized order of operators execution, and hence replace the expensive combination of operators with a new cheaper one. Figure 3.2 (a) shows an example, in which a combination of two Recip operators and two Mul operators is replaced by a combination of a Recip, a Square, and a Mul. The latter is more efficient as it eliminates a Mul operator and the intermediate result size is significantly reduced, leading to reduced register pressure after subsequent fusion.
- **Distributive:** Following the same ideas as above, applying distributive property also enables optimization opportunities. As shown in Figure 3.2 (b), the combination of two Mul operators and an Add can be replaced by an Add followed by a Mul, thus

Table 3.4: Graph rewriting with mathematical properties. Only representative graph rewriting rules are listed due to space limitation. In summary, DNNFusion derives 45, 38, and 66 graph rewriting rules in the category of Associative, Distributive, and Communicative, respectively. We omit unrelated operators for better readability. $\odot, +, -, Abs, Recip, Square, \sqrt{}$ mean element-wise multiplication, addition, subtraction, absolute, reciprocal, square, and square root, respectively. `BitShift` calculates the bit shifted value of elements of a given tensor element-wisely. `ReduceSum` and `ReduceProd` calculate the reduced summation and production of elements of an input tensor along an axis. `Exp` calculates the exponent of elements in a given input tensor element-wisely. #FLOPS denotes the number of floating point operations

Property	Without graph rewriting		With graph rewriting	
	Graph structure in equation	#FLOPS	Graph structure in equation	#FLOPS
Associative	$Recip(A) \odot Recip(A \odot B)$	$4 * m * n$	$Square(Recip(A)) \odot B$	$3 * m * n$
	$(A \odot \sqrt{B}) \odot (\sqrt{B} \odot C)$	$5 * m * n$	$A \odot B \odot C$	$2 * m * n$
	$Abs(A) \odot B \odot Abs(C)^\dagger$	$4 * m * n$	$Abs(A \odot C) \odot B$	$3 * m * n$
	$(A \odot ReduceSum(B)) \odot (ReduceSum(B) \odot C)^\P$	$5 * m * n$	$A \odot Square(ReduceSum(B)) \odot C$	$3 * m * n + m$
Distributive	$A \odot C + A \odot B$	$3 * m * n$	$(A + B) \odot C$	$2 * m * n$
	$A + A \odot B$	$2 * m * n$	$A \odot (B + 1)$	$2 * m * n$ §
	$Square(A + B) - (A + B) \odot C$	$5 * m * n$	$(A + B) \odot (A + B - C)$	$3 * m * n$
Commutative	$A \odot B$	$m * n$	$B \odot A$	$m * n^\ddagger$
	$ReduceSum(BitShift(A))^\P$	$2 * m * n$	$BitShift(ReduceSum(A))$	$m * n + m$
	$ReduceProd(Exp(A))^\P$	$2 * m * n$	$Exp(ReduceSum(A))$	$m * n + m$

§ Although #FLOPS is not reduced, A is loaded once instead of twice.

† First use commutative property to swap B and Abs(C), then apply associative property.

‡ Even though this pattern has no #FLOPS gains, it can enable further optimization, e.g the case of †.

¶ #FLOPS is calculated by assuming the reduction of ReduceSum/ReduceProd is along with the inner-most dimension.

eliminating an unnecessary operator.

- **Commutative:** The property guaranties the legality of swapping the position of two operators, which usually results in computation reduction. As shown in Figure 3.2 (c), `BitShift`² and `ReduceSum`³ satisfy communicative property, thus `ReduceSum` can be scheduled to execute before `BitShift`, reducing the number of elements on which `BitShift` is applied.

DNNFusion employs pattern matching [112, 113] to recognize rewriting candidates. However, associative and commutative matching is NP-complete [9]. Therefore, DNNFusion first partitions the entire Extended Computational Graph into many sub-graphs by considering operators with neither of associative, communicative, or distributive properties as partitioning points within the original graph. Within each sub-graph, DNNFusion

²Calculate the bit shifted value of elements of a given tensor element-wisely.

³Calculate the reduced sum of elements of an input tensor along an axis.

can explore all possible patterns and pattern combinations because these sub-graphs have limited number of operators. More specifically, all matching rules within a partition are considered and the rule leading to the largest reduction in #FLOPS is applied. This process is repeated till there are no additional matching rules within the partition. DNNFusion chooses this greedy scheme to keep the optimization overheads low.

3.4.3 Light-Weight Profile-Driven Fusion Plan Exploration

3.4.3.1 Overall Idea

Optimal fusion plan generation requires a large search space [14, 50] and has been shown to be NP-complete [35, 103]. To keep the process at manageable costs, DNNFusion explores fusion plans by employing a new light-weight (greedy) approach based on our proposed Extended Computational Graph (ECG) IR and our classification of operations into mapping types.

The high-level ideas are as follows. First, DNNFusion selects the starting operators (called *fusion seed operators*) from our ECG to restrict the search space. This is based on a key insight that operators of One-to-One mapping type have the potential to yield more benefits because they a) potentially result in fusion of more layers, including both with their predecessors and successors because of what we refer to as lower transformation impedance, and b) have lower memory requirements and need for fewer registers among all mapping types. Second, starting with these seed operators, DNNFusion explores fusion opportunities along the seed operator’s successors and predecessors, respectively. Third, DNNFusion creates fusion plans based on an approach that combines machine-independent mapping type analysis and a *profiling result database*. The mapping type analysis follows Table 3.3 to check the operators’ mapping type combination (in ECG) to decide if these operators should be fused. Such mapping eliminates unnecessary profile data lookup for most cases.

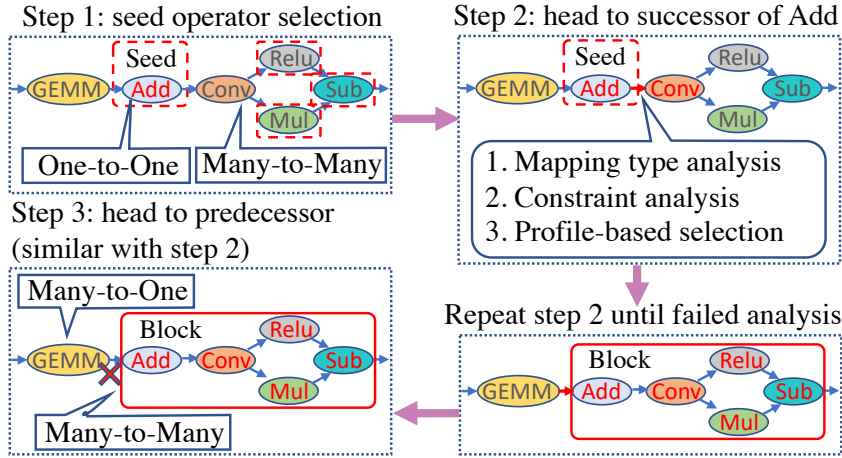


Figure 3.3: An example of fusion plan exploration. Assume Add, Conv, ReLU, Mul, and Sub have identical output shape and IRS size.

3.4.3.2 Fusion Plan Generation Algorithm

List 3.1 shows our detailed fusion plan exploration algorithm. Its goal is to generate candidate fusion blocks that are further optimized by subsequent intra-block optimizations (Section 3.4.4) before fusion code generation. Figure 3.3 illustrates its basic idea with a simplified example. This algorithm consists of three main steps:

Step I: Fusion seed operator(s) selection. DNNFusion selects the One-to-One operator with the minimum intermediate result as the fusion seed operator (as shown in Listing 3.1 lines 1 to 5). This heuristic is used because a smaller intermediate result makes fusion more profitable. This may seem counter-intuitive because fusing the operators with larger intermediate results usually results in more benefits. However, DNNFusion has a different goal, i.e., to ultimately enable more fusions to occur. Starting with a pair of operators with smaller intermediate results creates opportunities to fuse more (smaller) operators together, increase overall computation granularity, and hence enable higher parallelism and better load balance for the entire DNN computation. If multiple seed operators with the same minimum size of intermediate results exist, DNNFusion initiates fusion with them one after another (unless another seed is grouped to the same candidate fusion block). In Figure 3.3, Add, ReLU, Mul, and Sub are in One-to-One type (with an identical intermediate result

size), then Add is selected as the seed for the first round of fusion plan exploration.

```
1 def generate_seed(ops):
2     # find all one_to_one mapping operators
3     oto_ops = find_all_one_to_one(ops)
4     # find the operator with minimum IRS size
5     return min(op.IRS_size for op in oto_ops)
6
7 def fuse_successor(op, successor, block):
8     # Step 2.1: check the mapping relationship
9     relation = mapping_check(op, successor)
10    # return if successor can not be fused
11    if relation == fuse_break: return
12    # Step 2.2: check the constraint requirement
13    if not check_constraint(op, successor, block):
14        return
15    # fuse by profile-based selection
16    if relation == fuse_depend:
17        # Step 2.3: get latency w/ database/runtime
18        temp_latency = latency(block + successor)
19        if temp_latency > latency(block, successor):
20            return
21    block = op + successor
22    # Step 2.4: recursively head to successor
23    for fusing_op in successors(successor):
24        fuse_successor(successor, fusing_op, block)
25
26 # Similar with fuse_successor
27 def fuse_predecessor(op, predecessor, block):
28     # Similar with step 2.1, 2.2, 2.3, 2.4
29
30 # <Algorithm Entry>
31 unfused_ops = all_operators
32 # Step 1: start fuse from the selected seed
33 while(sp = generate_seed(unfused_ops)):
34     block = [sp]
35     # Step 2: head to successor
36     for successor in successors(sp):
37         fuse_successor(sp, successor, block)
38     # Step 3: head to predecessor
39     for predecessor in predecessors(sp):
40         fuse_predecessor(sp, predecessor, block)
41     unfused_ops = unfused_ops - block
```

Listing 3.1: Fusion plan generation

Step II: Propagated exploration along seed’s successors. Each operator may have one or multiple immediate predecessors and successors. DNNFusion first processes the seed operator’s successors one by one (Listing 3.1 Lines 7 to 24). At any stage in this recursive exploration, if a node cannot be fused with any of its immediate successors, fusion is not considered any further. Broadly, this step proceeds as follows. First, *mapping type*

analysis (Listing 3.1 Step 2.1) categorizes the potential fusion result into three types based on Table 3.3: 1) `fuse_break` indicates this is a Red case, and fusion should be aborted; 2) `fuse_through` indicates that this is a Green case, and should be proceeded without any further analysis; 3) `fuse_depend` indicates that this is a Yellow case, requiring a profile data lookup. Second, a *constraints check* (Listing 3.1 Step 2.2) is applied to analyze if further fusion is likely undesirable, i.e, it incurs too many overheads (e.g., can cause excessive register spills). Using an empirically determined threshold, the algorithm can decide to not consider any additional fusion candidates. Otherwise, DNNFusion continues exploring fusion candidates recursively. Figure 3.3 shows an example of fusing `Add` with `Conv` and other operators with this step. After this step, the generated candidate fusion block has a mapping type of Many-to-One, and includes five operators (`Add`, `Conv`, `Relu`, `Mul`, and `Sub`).

Step III: Propagated exploration along seed’s predecessors. After processing along the seed’s successor direction, DNNFusion processes along the seed’s predecessors direction with the same algorithm as Step II (In fact, Step III and Step II can be swapped). However, one difference is that if an operator has multiple immediate predecessors, there is an option of fusing with some, but not all, of these immediate predecessors. In the example in Figure 3.3, the first attempt of fusing current candidate fusion block with `GEMM` fails because both of them are of many-to-one mapping type. Table 3.3 indicates this is a `fuse_break` case, so `GEMM` is not included in this candidate fusion block.

Iterate. DNNFusion completes a round of fusion plan generation with above steps. If more fusion seeds exist, it will iterate from Step II with the next seed until no additional fusion seed is available.

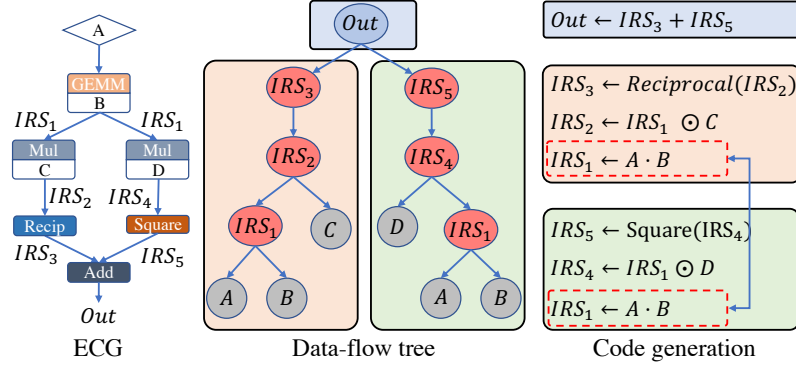


Figure 3.4: Code generation.

3.4.4 Fusion Code Generation and Optimizations

3.4.4.1 Fusion Code Generation

Once fusion blocks have been selected by our algorithm, DNNFusion generates fused code for each fusion block with a *data-flow tree* (DFT) built from the Extended Computational Graph (ECG) and a set of pre-defined code generation rules. DNNFusion generates C++ code for mobile CPU and OpenCL for mobile GPU, respectively. More specifically, DNNFusion traverses DFT and generates fused code for each pair of operators to be fused by leveraging the code generation rules that are based on abstract mapping types (e.g., One-to-One). 23 code generation rules are defined for each of mobile CPU and mobile GPU, with one rule corresponding to a green or yellow cell in Table 3.3. The basic idea is that as long as the operators are of the same type, the same rules lead to efficient code. While fusing more than two operators, these rules are invoked each time two operators are fused. Finally, the subsequent code optimizations (e.g, vectorization, unrolling, tiling, and memory/register optimizations, and auto-tuning of these optimizations) are handled by our existing framework called PatDNN [158], thus not a major contribution of this paper. Note that almost each fusion generates a new operator (and its codes) that is not present in the original operator library; however, once the new operator (and its code) is generated, it can be used for both the current model and future models.

Figure 3.4 shows an example of the code generation. To elaborate, DNNFusion first

generates a data-flow tree (DFT) from the Extended Computational Graph (ECG). This DFT represents the final output (**Out**), all intermediate results (**IRS**), and all inputs (**A**, **B**, **C**, and **D**) with the edges reversed as compared to the ECG (i.e., the parent node depends on the children nodes). During the fused code generation, DNNFusion traverses this DFT to recognize the input/output data dependence (and fuses corresponding ECG operations), recursively. The right-hand side of Figure 3.4, shows an example of this DFT traversal (the fused code generation based on the pre-defined code generation rules is omitted in this Figure for readability and is introduced in the next paragraph). First, DNNFusion recognizes that *Out* depends on $IRS_3 + IRS_5$; next, it recognizes that IRS_3 depends on reciprocal of IRS_2 , and so on, until reaching the input of *A,B,C,D*. It is worth noting DNNFusion can also find *redundant computations* in DFT with a common sub-tree identification and eliminate them during code generation. In our example, both **Mul** operators use IRS_1 , resulting in a common sub-tree in DFT, so the recognition in two red boxes of Figure 3.4 is only taken once.

During this DFT traversal, DNNFusion employs the pre-defined code generation rules to generate the code for each pair of operators to be fused. For the example shown in Figure 3.4, DNNFusion first fuses **Add** with its left input branch **Recip**. Both **Add** and **Recip** belong to One-to-One mapping, and hence the fused operator is also One-to-One. DNNFusion keeps fusing **Mul** (One-to-One) with this newly fused operator, and the result is still One-to-One. Next, this newly generated operator is fused with **GEMM** (Many-to-One), generating a new Many-to-One operator. Similar steps are taken along the right input branch of **Add** until all operators are fused into a single new Many-to-One operator. DNNFusion relies on the DFT traversal introduced in the prior paragraph to figure out the input/output data dependence, and employs the operator mapping type to handle the index mapping relationship and generate proper nested loop structures.

To explain this further, here is an example with more complicated mapping types: **GEMM** (Many-to-Many) + **Div** (One-to-One) + **Transpose** (Shuffle). First, DNNFusion fuses **Transpose** and **Div**, a case of (“Shuffle + One-to-One”) by first permuting the loop in the

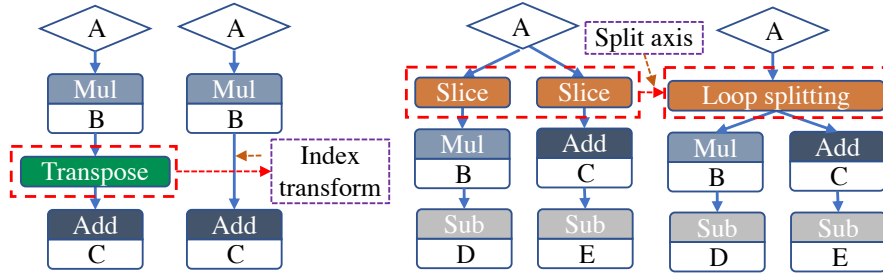


Figure 3.5: Data movement operators optimization.

Transpose operator and then fusing it with the Div operator. It generates a new operator of the type Shuffle. Next, DNNFusion fuses GEMM (Many-to-Many type) with this new operator (Shuffle type), in which DNNFusion maps output elements of GEMM to the destination that is decided by this new operator.

3.4.4.2 Other Fusion-related Optimizations

DNNFusion also includes several advanced optimizations enabled by our fusion analysis and fused code generation. They broadly can be characterized into two groups, *intra-fusion-block optimizations* that are performed on Extended Computational Graph (ECG) immediately before the code generation and *inter-fusion-block optimizations* on the generated fused code. ***Intra-block Optimizations:*** Operators in Shuffle and Reorganize mapping types usually involve intensive data movement. We observed many of these time/memory consuming data operations can be eliminated. In particular, consider the case when the transformed data is used by only one subsequent operator because the data locality improvement brought this data transformation cannot be compensated by the overhead of intermediate results generation and storage. Figure 3.5 shows such examples – particularly, in these, data transpose and data slicing operations bring more overheads than the benefit. Thus, in such cases, DNNFusion replaces them with operations that have a changed data index. These optimizations are performed after graph rewriting and result in an ECG that should have a more efficient implementation.

Inter-block Optimization: Different operators prefer different data formats. Without the proposed graph rewriting optimizations and operator fusion, normally such choices are made at the level of each individual operator – however, this can result in redundant or unnecessary transformations. In contrast, DNNFusion considers the data format choice at a global level, thus avoiding redundant or unnecessary transformations. Currently, DNNFusion employs a heuristic approach to optimize the data format, which is as follows. For a specific fusion block, it identifies one *dominant* operator whose performance is impacted the most by the choice of the layout (e.g., CONV, GEMM, and Softmax are most likely to such operators). The optimal layout for this operation is then used for the entire fusion block. This heuristic approach works based on a key observation that most other *non-dominant* operators can employ any layout/format without their performance being significantly affected. A potential future work will be to consider more sophisticated cost models, including balancing the cost of reformatting the data with reductions in execution because of the optimized layout.

3.5 Evaluation

DNNFusion is implemented on top of an existing end-to-end DNN execution framework called PatDNN [158] that supports both dense and sparse DNN execution. It has been shown in our previous work that PatDNN [158] performs slightly better than TVM, MNN, and TFLITE even without our proposed operator fusion. For readability, we also call this optimized framework DNNFusion. Our evaluation has four objectives: 1) demonstrate that the proposed fusion framework (together with graph rewriting) is effective by showing how DNNFusion outperforms other state-of-the-art frameworks, and no-fusion and fixed-pattern fusion implementations on various DNN models; 2) validating DNNFusion’s generality by showing its efficient execution on both CPU and GPU on a wide spectrum of DNNs (for 5 types of tasks, with varied sizes, and layer counts ranging from relatively shallow to extremely deep); 3) analyzing the impact of different compiler optimizations on both

execution time and compilation time; and 4) demonstrating the effective portability of DNNFusion by evaluating it on three different mobile phones.

More specifically, DNNFusion (also called **DNNF** for short) is compared against four popular state-of-the-art end-to-end DNN execution frameworks: MNN [96], TVM [25], TensorFlow-Lite (TFLite) [1], and Pytorch-Mobile (Pytorch) [165]. Because certain extremely deep neural networks are not supported by any of these existing frameworks (or just supported by their mobile CPU implementation), we also set a baseline by turning off DNNFusion’s all fusion related optimizations (called **OurB**, i.e., our baseline version without fusion) and implement a version that optimizes **OurB** with fixed-pattern fusion (using operator fusion described in TVM [25]) (called **OurB+**), and compare DNNFusion against them.

3.5.1 Evaluation Setup

Models and datasets. DNNFusion is evaluated on 15 mainstream DNN models. Table 3.5 characterizes them with a comparison of their targeted task, number of parameters, total number of layers, and number of floating point operations (FLOPS). Particularly, we have 1) two image classification 2D CNNs (EfficientNet-B0 [213] and VGG-16 [206]), 2) two object detection two-dimensional (2D) CNNs (MobileNetV1-SSD [142] and YOLO-V4 [13]), 3) two action recognition three-dimensional (3D) CNNs (C3D [218] and S3D [239]), 4) one image segmentation 2D CNN (U-Net [199]) and two image segmentation R-CNNs (Mask R-CNN [76] and FasterRCNN [195]), and 5) six natural language processing (NLP) models (TinyBERT [97], DistilBERT [201], ALBERT [115], BERT_{BASE}, MobileBERT [212], and GPT-2 [188]).

Because the choice of datasets has a negligible impact on the final inference latency or relative execution speeds (and also because of space limitations), we report results from one dataset for each model. EfficientNet-B0 and VGG-16 are trained on ImageNet dataset [41]; MobileNetV1-SSD and YOLO-V4 are trained on MS COCO [136]; C3D and S3D are trained on UCF-101 [209]; U-Net, Faster R-CNN, and Mask R-CNN are trained on PASCAL VOC

2007 [51]; TinyBERT, DistilBERT, ALBERT, BERT_{base}, MobileBERT, and GPT-2 are trained on BooksCorpus [46] and English Wikipedia [46]. Because the model accuracy is identical among all frameworks, and also because of space limitations, we only focus on execution times and do not report accuracy.

Evaluation environment. The evaluations are carried out on a Samsung Galaxy S20 cell phone that has Snapdragon 865 processor [183], which comprises an octa-cores Kryo 585 CPU and Qualcomm Adreno 650 GPU yielding high performance with good power efficiency. For demonstrating portability, we further use a Samsung Galaxy S10 with a Snapdragon 855 [181] (Qualcomm Kryo 485 Octa-core CPU and a Qualcomm Adreno 640 GPU), and an Honor Magic 2 with a Kirin 980 [84] (ARM Octa-core CPU and a Mali-G76 GPU). All executions used 8 threads on mobile CPUs, and similarly all pipelines on mobile GPUs. 16-bit and 32-bit floating points are used for all GPU runs and CPU runs, respectively. All experiments were run 100 times but as the variance was very small, we only report averages.

3.5.2 Overall Mobile Inference Evaluation

Our comparison includes both fusion rate⁴ and execution latency.

Fusion rate. Table 3.5 shows detailed layer counts (including computation-intensive (CIL), memory-intensive (MIL), and all layers), and intermediate result sizes for models before fusion and after fusion with different frameworks. Note that DNNFusion is the only end-to-end framework that can support all of the target models on both mobile CPU and mobile GPU. In Table 3.5, “-” implies that this framework does not support this model. Certain extremely deep neural networks (e.g., Faster R-CNN and Masker R-CNN) are not supported by any other frameworks on mobile devices because these frameworks either lack the support of multiple key operators and/or limited optimization supported in them lead to a large model execution footprint. For transformer-based models, only TFLite can support execution on mobile CPU (without GPU support).

⁴Fusion rate = original layer count/fused layer count.

Table 3.5: Fusion rate evaluation: computation layer count and intermediate result size for all evaluated DNNs. CIL (Compute-Intensive Layer): each input is used more than once, e.g. MatMul, CONV. MIL (Memory-Intensive Layer): each input is used only once, e.g. Activation. IRS: intermediate results. '-' means this framework does not support this model.

Model	Type	Task	Layer counts and IRS sizes before opt.				Layer counts and IRS sizes after opt.					
			#CIL	#MIL	#Total layer	IRS size	MNN	TVM	TFLite	Pytorch	DNNF	IRS
EfficientNet-B0	2D CNN	Image classification	82	227	309	108MB	199	195	201	210	97	20
VGG-16	2D CNN	Image classification	16	35	51	161MB	22	22	22	22	17	52
MobileNetV1-SSD	2D CNN	Object detection	16	48	202	110MB	138	124	138	148	71	37
YOLO-V4	2D CNN	Object detection	106	292	398	329MB	198	192	198	232	135	205
C3D	3D CNN	Action recognition	11	16	27	195MB	27	27	-	-	16	90
S3D	3D CNN	Action recognition	77	195	272	996MB	-	-	-	272	98	356
U-Net	2D CNN	Image segmentation	44	248	292	312MB	241	232	234	-	82	158
Faster R-CNN	R-CNN	Image segmentation	177	3,463	3,640	914MB	-	-	-	-	942	374
Mask R-CNN	R-CNN	Image segmentation	187	3,812	3,999	1,524MB	-	-	-	-	981	543
TinyBERT	Transformer	NLP	37	329	366	183MB	-	304 [†]	322	-	74	58
DistilBERT	Transformer	NLP	55	402	457	540MB	-	416 [†]	431	-	109	197
ALBERT	Transformer	NLP	98	838	936	1,260MB	-	746 [†]	855	-	225	320
BERT _{BASE}	Transformer	NLP	109	867	976	915MB	-	760 [†]	873	-	216	196
MobileBERT	Transformer	NLP	434	1,953	2,387	744MB	-	1,678 [†]	2,128	-	510	258
GPT-2	Transformer	NLP	84	2,449	2,533	1,389MB	-	2,047 [†]	2,223	-	254	356

[†] TVM does not support this model on mobile. This layer count number is collected on a laptop platform for reference.

Table 3.5 shows that compared with the other frameworks, DNNFusion results in better fusion rates, with $1.3\times$ to $2.9\times$, $1.3\times$ to $8.1\times$, $1.3\times$ to $8.8\times$, and $1.3\times$ to $2.8\times$ over MNN, TVM, TFLite, and Pytorch, respectively. Particularly, compared with original models, DNNFusion yields more benefits for R-CNN and Transformer-based models ($3.9\times$ to $10.0\times$ fusion rate) than 2D/3D CNNs ($1.7\times$ to $3.6\times$ fusion rate). This is because 2D/3D CNNs have higher fractions of computation-intensive layers that are in either One-to-Many or Many-to-Many types, while transformer-based models have more memory-intensive layers that are in One-to-One, Shuffle, or Reorganize categories. The latter offers more fusion opportunities according to our mapping type analysis (Table 3.3). Because of the same reason, 3D CNNs have the lowest fusion rate because they are more compute-intensive. Moreover, comparing to TVM (that performs the best among all other frameworks), DNNFusion particularly yields more benefits for transformer-based models. This is because these models have more types of operators, and TVM’s fixed pattern-based fusion cannot capture fusion opportunities among many types of operators while DNNFusion can. This result demonstrates that DNNFusion has a better generality.

Execution latency. Table 3.6 shows the execution latency evaluation results. Comparing with MNN, TVM, TFLite, and Pytorch, with fusion optimization, DNNFusion achieves the

Table 3.6: Inference latency comparison: DNNFusion, MNN, TVM, TFLite, and PyTorch on mobile CPU and GPU. #FLOPS denotes the number of floating point operations. OurB is our baseline implementation by turning off all fusion optimizations and OurB+ is OurB with a fixed-pattern fusion as TVM. DNNF is short for DNNFusion, i.e., our optimized version. '-' denotes this framework does not support this execution.

Model	#Params	#FLOPS	MNN (ms)		TVM (ms)		TFLite (ms)		Pytorch (ms)		OurB (ms)		OurB+ (ms)		DNNF (ms)	
			CPU	GPU	CPU	GPU	CPU	GPU	CPU	GPU	CPU	GPU	CPU	GPU	CPU	GPU
EfficientNet-B0	5.3M	0.8B	41	26	56	27	52	30	76	-	54	35	38	24	16	10
VGG-16	138M	31.0B	242	109	260	127	245	102	273	-	251	121	231	97	171	65
MobileNetV1-SSD	9.5M	3.0B	67	43	74	52	87	68	92	-	79	56	61	39	33	17
YOLO-V4	64M	34.6B	501	290	549	350	560	288	872	-	633	390	426	257	235	117
C3D	78M	77.0B	867	-	1,487	-	-	-	2,541	-	880	551	802	488	582	301
S3D	8.0M	79.6B	-	-	-	-	-	-	6,612	-	1,409	972	1,279	705	710	324
U-Net	2.1M	15.0B	181	106	210	120	302	117	271	-	227	142	168	92	99	52
Faster R-CNN	41M	47.0B	-	-	-	-	-	-	-	-	2,325	3,054	1,462	1,974	862	531
Mask R-CNN	44M	184B	-	-	-	-	-	-	-	-	5,539	6,483	3,907	4,768	2,471	1,680
TinyBERT	15M	4.1B	-	-	-	-	97	-	-	-	114	89	92	65	51	30
DistilBERT	66M	35.5B	-	-	-	-	510	-	-	-	573	504	467	457	224	148
ALBERT	83M	65.7B	-	-	-	-	974	-	-	-	1,033	1,178	923	973	386	312
BERT _{Base}	108M	67.3B	-	-	-	-	985	-	-	-	1,086	1,204	948	1,012	394	293
MobileBERT	25M	17.6B	-	-	-	-	342	-	-	-	448	563	326	397	170	102
GPT-2	125M	69.1B	-	-	-	-	1,102	-	-	-	1,350	1,467	990	1,106	394	292

speedup of $1.4\times$ to $2.6\times$, $1.5\times$ to $3.5\times$, $1.4\times$ to $3.3\times$, and $1.6\times$ to $9.3\times$, respectively, on the mobile CPU. Focusing on mobile GPU, improvements over MNN, TVM, and TFLite are $1.7\times$ to $2.6\times$, $2.0\times$ to $3.1\times$, $1.6\times$ to $4.0\times$, respectively, whereas Pytorch does not support execution on this mobile GPU. The reason for speedups including the fact that our baseline implementation with a fixed-pattern fusion (OurB+) is already faster than other frameworks (with speedup of $1.04\times$ to $5.2\times$ on mobile CPU and $1.05\times$ to $1.7\times$ on mobile GPU), and with our more advanced fusion, DNNFusion achieves $1.4\times$ to $2.5\times$ speedup over OurB+ on mobile CPU and $1.5\times$ to $3.9\times$ speedup on mobile GPU. In addition, comparing DNNFusion (with fusion) and our baseline without fusion (OurB), our advanced fusion brings $1.5\times$ to $5.8\times$ speedup. Moreover, fusion optimization offered by DNNFusion brings more benefits for mobile GPU than CPU, particularly for extremely deep models (e.g., Faster R-CNN and GPT-2). This is because mobile GPU offers more parallelism but smaller cache capacity compared to CPU, and GPU kernel launch incurs certain overheads, so it is more sensitive to intermediate data reduction, kernel launch reduction, and processor utilization increase that are brought by DNNFusion’s fusion.

To further validate DNNFusion’s performance, Figure 3.6 compares it with a state-of-the-art computational graph substitution approach mentioned earlier, TASO [94]. We

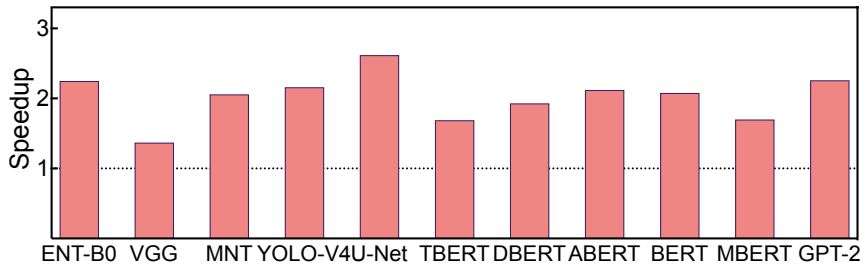


Figure 3.6: Speedup over TASO optimized execution on mobile CPU. The models (computational graphs) are optimized by TASO and then executed on TFLite.

use TASO to optimize all eleven models (computational graphs) supported by TFLite among those listed in Table 3.6, including EfficientNet-B0 (ENT-B0), VGG-16 (VGG), MobileNetV1-SSD (MNT), YOLO-V4, U-Net, TinyBERT (TBERT), DistilBERT (DBERT), ALBERT (ABERT), BERT_{Base} (BERT), MobileBERT (MBT), and GPT-2. Then, for our experiments, these models are executed under TFLite on mobile CPU (not GPU because TFLite lacks GPU support for many of these models). Compared with TASO, DNNFusion yields $1.4\times$ to $2.6\times$ speedup on mobile CPU. The graph rewriting in DNNFusion is designed to work in conjunction with operator fusion and identifies a set of operators and rules for that specific purpose, thus enabling more fusion opportunities. TASO does not emphasize the relationship between graph rewriting and fusion, resulting in less efficient execution as compared to DNNFusion.

3.5.3 Understanding Fusion Optimizations

This section studies the effect of our key optimizations.

Optimization breakdown. Figure 3.7 shows the impact of our proposed optimizations on latency with four models (EfficientNet-B0 (ENT-B0), YOLO-V4, S3D, and GPT-2) on both mobile CPU and GPU. Experiments on other models show a similar trend and are omitted due to space constraints. We evaluate each compiler-based optimization speedup incrementally over the `OurB` version. Compared with `OurB`, graph rewriting brings $1.2\times$ to $1.5\times$ speedup, fusion brings additional $1.6\times$ to $2.2\times$ speedup, and other optimizations (intra-block optimizations like data movement operator optimizations and inter-block

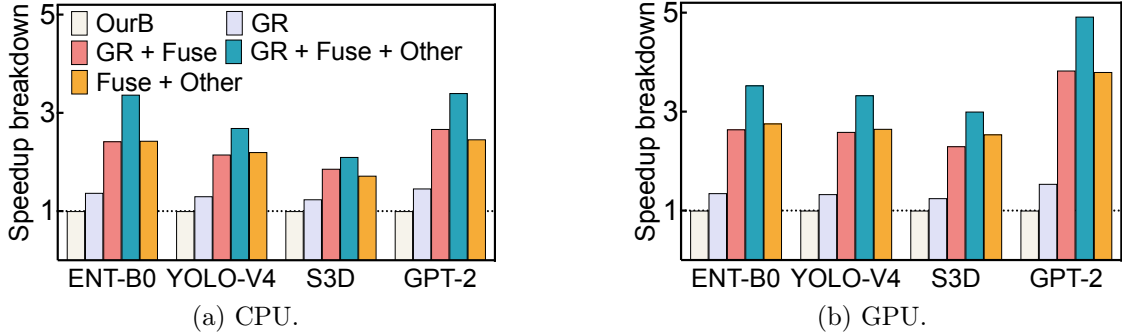


Figure 3.7: Optimization breakdown on y-axis: speedup over OurB, i.e. a version w/o fusion opt. GR, Fuse, and Other denote graph rewriting, fusion, and other fusion-related optimizations, respectively.

optimizations like data format optimizations introduced in Section 5.4.4) bring additional $1.3\times$ to $1.8\times$ speedup on mobile CPU. On mobile GPU, these numbers are $1.3\times$ to $1.5\times$, $2.1\times$ to $3.3\times$, and $1.7\times$ to $2.1\times$, respectively. Again, our fusion brings more benefit for mobile GPU than CPU due to the aforementioned reasons of memory, parallelism, and kernel launch overheads. Although graph rewriting by itself brings fewer benefits than the fusion, its hidden benefit is in enabling more fusion opportunities. Take GPT-2 as an example – without graph rewriting, the generated fused layer count is 310, while after rewriting, it is 254 (18% reduction). This is because graph rewriting simplified the computational graph structure. To further assess this impact of graph rewriting on operator fusion, the last bar (in orange) of Figure 3.7 reports the speedup of fusion with other optimizations only (i.e., without graph rewriting) over the baseline `OurB`. Compared with no graph rewriting, graph rewriting brings an additional $1.4\times$ to $1.9\times$ and $1.6\times$ to $2.0\times$ speedup (over `OurB`) on mobile CPU and GPU, respectively.

Memory and cache performance We compare memory (and cache) performance of DNNFusion with other frameworks on both mobile CPU and GPU. We only report YOLO-V4’s results due to space constraint, and because it is one of the models supported by all frameworks. Figure 3.8 (a) left shows memory performance (measured in total memory accesses (MA) and memory consumption (MC)) – MA and MC are collected from Snapdragon Profiler [176], and Figure 3.8 (a) right shows cache miss count on data cache

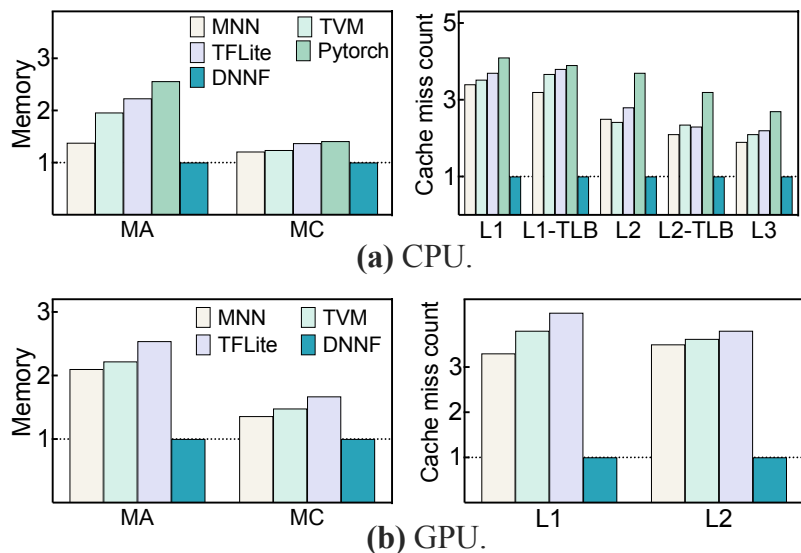


Figure 3.8: Memory (left) and cache miss (right) analysis. MA and MC denote memory access and memory consumption, respectively. Cache miss count is compared on L1/L2/L3 data cache and L1/L2 TLB cache on mobile CPU, and on L1/L2 data cache only on mobile GPU. All values are normalized w.r.t DNNF (the optimal version).

and TLB cache on mobile CPU. All values are normalized with respect to DNNFusion (i.e., our best version) for readability. DNNFusion outperforms other frameworks on both memory access count and memory consumption because it eliminates materialization of more intermediate results. Figure 3.8 (b) shows similar results on mobile GPU (excluding PyTorch because it does not support YOLO-V4 on mobile GPU). Mobile GPU results are generally better than CPU because mobile GPU has smaller cache capacity and simpler hierarchy (GPU only has L1/L2 v.s., CPU has L1/L2/L3). Thus, intermediate results reduction leads to more gains on mobile GPU.

CPU/GPU utilization. Figure 3.9 (a) reports the mobile CPU and GPU utilization on YOLO-V4. This result is collected by Snapdragon Profiler [176]. It shows that DNNFusion results in the highest utilization on both CPU and GPU because its aggressive fusion groups more computation together, resulting in coarser-grained execution with less loop and parallel scheduling (and kernel launch for GPU only) overhead. Particularly, the GPU utilization is slightly higher than CPU because of its higher parallelism.

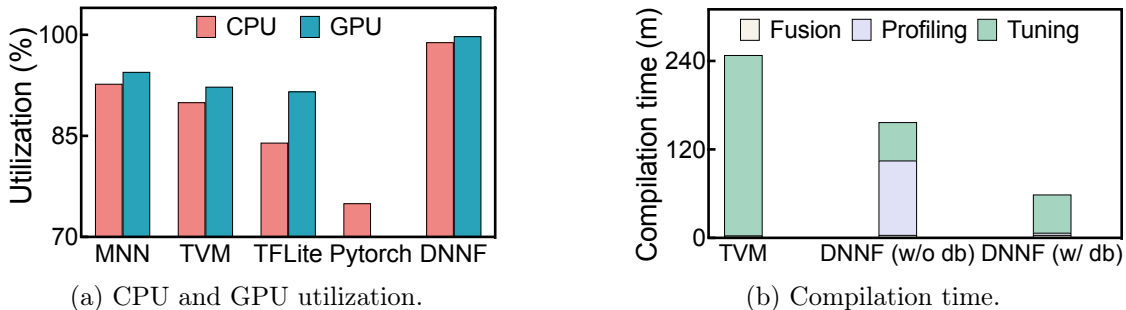


Figure 3.9: (a) Mobile CPU and GPU utilization. CPU utilization is averaged on 8-cores. **(b) Compilation time.** Comparison between TVM and DNNF for YOLO-V4 on mobile CPU. DNNF (w/o db) is without the presence of an existing profiling database; DNNF (w/ db) assumes such a database is pre-computed. **Fusion** is invisible as it spends very little time on both TVM and DNNF.

Compilation time. Figure 3.9 (b) compares the compilation time between TVM and DNNFusion for YOLO-V4 on mobile CPU. TVM’s compilation time consists of operator fusion and other compiler optimizations (**Fusion**), and tuning for performance-critical parameters, e.g., tiling size, unrolling factors, and others (**Tuning**). **Tuning** dominates its compilation time, lasting for around 4 hours for YOLO-V4 on mobile CPU. DNNFusion’s compilation time consists of operator fusion and other compiler optimizations (**Fusion**), profiling to analyze the fusion benefits (**Profiling**), and performance tuning (**Tuning**). DNNFusion’s **Tuning** relies on an auto-tuner based on Genetic Algorithm (reported in our previous publication [158]) to generate the exploration space. Compared with AutoTVM [26], our auto-tuning’s Genetic Algorithm allows parameter search to start with initializing an arbitrary number of chromosomes. Without our pre-existing profiling database, **Profiling** and **Tuning** dominate the compilation, requiring around 3 hours. With a pre-computed database, **Profiling** becomes very fast, and only **Tuning** dominates the compilation, requiring around 1 hour. After evaluating 15 models in Table 3.5, the profiling database consists of around 22K profiling entries with each one including operators’ information (e.g., operator types, shape, and their combinations) and the latency achieved.

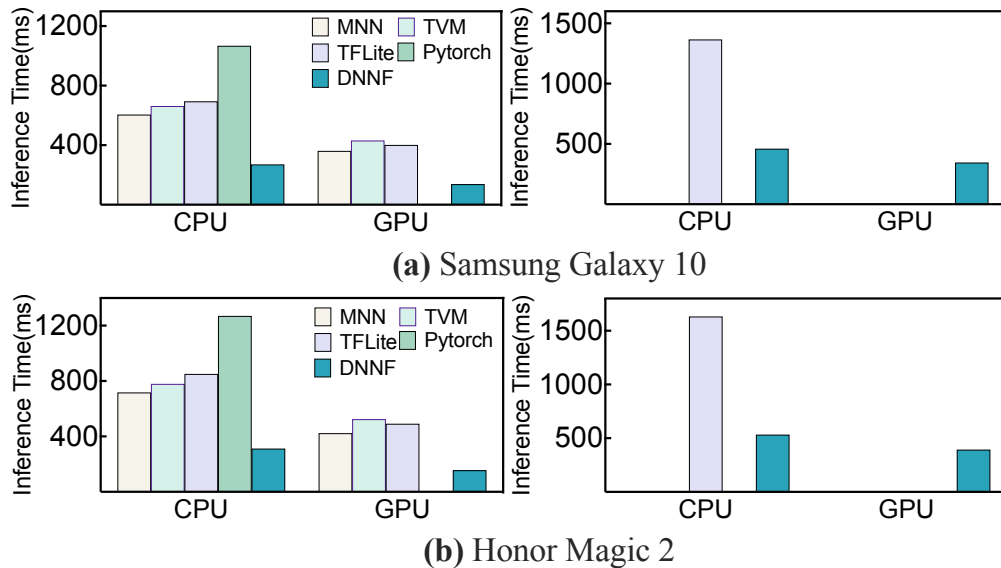


Figure 3.10: Portability evaluation. It is on Samsung Galaxy S10 and Honor Magic 2. Left two figures are YOLO-V4 and right two are GPT-2. Only TFLite supports GPT-2 on mobile CPU (no mobile GPU support).

3.5.4 Portability

Figure 3.10 shows the execution latency on additional cell phones (Samsung Galaxy S10 and Honor Magic 2) to demonstrate effective portability. Only YOLO-V4 and GPT-2 are reported due to limited space. Other models show similar trends. In particular, DNNFusion shows a more stable performance on older generations of mobile devices. This is because our fusion significantly reduces the overall number of layers and intermediate result size, and older cell phones with more restricted resources are more sensitive to these.

3.6 Related Work

Operator fusion in end-to-end mobile DNN frameworks. Operator fusion is an important optimization in many state-of-the-art end-to-end mobile DNN execution frameworks that are based on computational graphs, such as MNN [96], TVM [25], TensorFlow-Lite [1], and Pytorch [165]. However, they all employ fixed-pattern fusion that is too restricted to cover diverse operators and layer connections in deep models like BERT – for example,

`ConvTranspose + ReLU + Concat` cannot be recognized in TVM as it is not one of the specified patterns. Other examples that can be handled by DNNFusion and cannot be recognized by TVM include `MatMul + Reshape + Transpose + Add` in GPT-2, and `Sub + Pow + ReduceMean + Add + Sqrt` in TinyBERT. Comparing to these frameworks, DNNFusion works by classifying both the operators and their combinations, thus enabling a much larger set of optimizations.

Operator fusion on other ML frameworks. There are certain recent frameworks that rely on polyhedral analysis to optimize DNN computations and support operator fusion. R-Stream-TF [172] shows a proof-of-concept adaptation of the R-Stream polyhedral compiler to TensorFlow. Tensor Comprehensions [224] is an end-to-end compilation framework built on a domain-specific polyhedral analysis. These frameworks do not support mobile execution (i.e. ARM architecture), and thus we cannot perform a direct comparison between DNNFusion and them. As we have stated earlier, DNNFusion maintains an operator view but builds a higher-level abstraction on them. In the future, we can combine DNNFusion’s high-level abstraction to existing domain-specific polyhedral analysis. Similarly, another promising direction will be to integrate DNNFusion into other compilation-based DNN frameworks [65, 154] or other popular general tensor/matrix/linear algebra computation frameworks, such as MLIR [121], Tiramisu [8], TACO [109, 110], Halide [189], and LGen [114, 211].

There also exist several other frameworks to optimize machine learning with operator fusion or fusion-based ideas. Closely related to DNNFusion – Rammer [146] relies on fix-pattern operator fusion to further reduce kernel launch overhead of their optimized scheduling, Cortex [53] proposes a set of optimizations based on kernel fusion for dynamic recursive models, TensorFlow XLA [64] offers a more general fusion method than fix-pattern operator fusion by supporting reduce operations and element-wise operations, and TensorFlow Grapper [215] provides an arithmetic optimizer that performs rewrites to achieve both fusion and arithmetic expression simplification (e.g., $a \times b + a \times c = a \times (b + c)$). Comparing with these frameworks, DNNFusion works by classifying the operators and their

combinations into several mapping categories, thus resulting in a more aggressive fusion plan and more performance gains. Elgamal [50] and Boehm [14] presently optimize general machine learning algorithms (e.g., SVM and Kmeans) with operator fusion. These efforts have both different targets and techniques compared to DNNFusion.

Polyhedral-based and other loop fusion methods. Polyhedral analysis [16, 21, 48, 111, 170, 252] is a prominent approach that offers a general and rigorous foundation for loop transformation and optimization.

Many existing efforts [2, 3, 15] rely on a general polyhedral analysis to achieve optimized loop fusion. Pouchet *et al.* [171] have demonstrated that polyhedral analysis can decompose the loop optimization problem into sub-problems that have much lower complexity, enabling optimal selection. The problem arising because of a large number of operators in our target applications (models) is quite different, and thus there does not seem to be a direct application of Pouchet *et al.*'s approach in our context. There have also been other loop fusion efforts targeting general programs [40, 100, 103, 151]. In contrast to these general efforts, DNNFusion is more domain-specific, leveraging the knowledge of DNN computations with a higher-level abstraction to explore more aggressive loop fusion opportunities.

3.7 Summary

This paper has presented a new loop fusion framework called DNNFusion. The key advantages of DNNFusion include: 1) a new high-level abstraction comprising mapping type of operators and their combinations and the Extended Computational Graph, and analyses on these abstractions, 2) a novel mathematical-property-based graph rewriting, and 3) an integrated fusion plan generation. DNNFusion is extensively evaluated on 15 diverse DNN models on multiple mobile devices, and evaluation results show that it outperforms four state-of-the-art DNN execution frameworks by up to $8.8\times$ speedup, and *for the first time* allows many cutting-edge DNN models not supported by prior end-to-end frameworks to execute on mobile devices efficiently (even in real-time). In addition, DNNFusion improves

both cache performance and device utilization, enabling execution on devices with more restricted resources. It also reduces performance tuning time during compilation.

Our future work will enhance DNNFusion by combining it with the latest model pruning advances [49, 158]. Though model pruning is effective, with fusion the dense versions are outperforming these efforts by having fewer layers. Thus, there is an opportunity to combine the two set of approaches to achieve an even better performance.

Chapter 4

GCD²: A Globally Optimizing Compiler for Mapping DNNs to Mobile DSPs

4.1 Introduction

Despite the upcoming end of Moore's law, the last several years have seen a quick increase in transistors density. For example, in going from 22 nm technology to 10 nm, Intel chips saw a nearly $7\times$ increase in transistor density, and the most chip manufacturers are building chips with more than 100 million transistor per square millimeter at the time of writing this paper¹. All processors, but more particularly the specialized ones, have exploited this density by supporting an increasing amount of parallelism, often combined with intricate ways in which this parallelism can be exploited. Even in mainstream processors, the SIMD width has increased and the flexibility of programming API has improved with AVX-512 instruction set that has features like *scatter*, *gather*, and *masks*.

An example of a class of specialized chips that offer a programming interface suited

¹<https://www.techcenturion.com/7nm-10nm-14nm-fabrication>

Table 4.1: Latency and Power Comparisons among Mobile CPU, GPU, and DSP. Experiments are conducted on a Samsung Galaxy S20 with TFLite [1]. CPU, GPU, and DSP uses int8, float16, and int8, respectively. Power is collected by the Android system interface. Results are for each inference.

Model	#MACS [†]	Latency (ms)			Power*		
		CPU	GPU	DSP	CPU	GPU	DSP
EfficientNet-b0 [213]	0.4G	53	11.3	9.1	10.7	1.6	1.0
ResNet [77]	4.1G	62	34.4	13.9	6.2	2.3	1.0
PixOr [245]	8.8G	280	64.6	43	6.7	1.8	1.0
CycleGAN [260]	186G	4320	477	450	5.5	1.2	1.0

[†] #MACS denotes the number of multiply-accumulate operations.

* Normalized by DSP’s power for readability.

for general purpose processing is the Digital Signal Processing (DSP) chips. Particularly, smartphones have invested in sophisticated DSP chips that are also capable of accelerating other highly parallel workloads. To date, however, there is only a limited exploration on the use of DSP chips for other workloads [5, 223, 229, 246].

In recent years, machine learning (ML) or deep learning (DL) workloads, particularly the Deep Neural Networks (DNNs), have emerged as important workloads that have been targeted on a range of hardware – from mainstream processors and accelerators [11, 25, 47, 81, 94, 168, 224, 227] to mobile devices [1, 61, 72, 88, 95, 96, 116, 231, 241, 248, 255] (including mobile DSP [119]) to chips specifically designed for them [80, 102, 204, 257]. A particular requirement, and the driver of our work, is performing inference using complex Deep Neural Network (DNN) models on mobile phones in a time, memory, and power-efficient manner. We observe that DSP chips are a natural candidate for accelerating DNN inference in a mobile setting, not only because mobile phone already have a DSP chip, but also because these chips are optimized for matrix and vector computations on fixed-point values.

This paper reports a compilation system that optimizes Deep Neural Networks (DNNs) for execution on a mobile DSP chip. As a quick motivation for this effort, results from Table 4.1 show that with an existing framework, TFLite [1], execution on a DSP chip outperforms both mobile CPU and GPU in terms of execution time and power. Conceptually, however, it also turns out that compiling for the DSP chip involves dealing with many

advanced features, especially with respect to low-level parallelism exposed through its instruction set, requiring techniques well beyond the ones implemented in current systems or otherwise developed. More specifically, modern (mobile) DSP chips have much more complex SIMD instruction sets with both a larger width and a greater variety of instructions as compared to the mainstream processors, and thus require techniques beyond those explored in current literature [27, 111, 219]. Besides 1024-bit width, there are instructions combining vector operations and reductions in different ways, going even beyond Intel’s additions under VNNIW and FMAPS extensions [60]. In addition, VLIW instructions exist that can combine multiple SIMD instructions for simultaneous execution, and there are other performance characteristics that require new methods for effective mapping of the workload.

This paper develops techniques for exploiting these architectural features. Our contributions include:

- **Methods for Exploiting Disparate SIMD Instructions.** We develop data layouts and execution schemes that use different new instructions for key Deep Learning (DL) kernels. We also investigate the trade-offs between different approaches depending upon the size of the operands.
- **Formulating and Solving a Global Optimization Problem.** We show how the choice of instruction (and their corresponding data layouts) for one operator impacts the choice for their successor and formulate a global optimization problem. We show an optimal linear-time solution for this problem when the operators form a linear chain, and develop useful heuristics for the general case of a computational graph.
- **VLIW Packing (i.e., Scheduling) Problem.** Considering many unique aspects of our target architecture (including the notion of *soft* dependencies, and latency sensitivity), we present a novel Soft Dependencies Aware (SDA) algorithm for instructions packing.

- **Design of an End-to-End Compilation System.** We engineer a system that includes a nuanced code generation design and several additional optimizations.

GCD² is extensively evaluated on 10 real-world large DNNs, with a range of model sizes and operator counts and designed for various ML tasks, targeting popular mobile DSPs. Compared with two state-of-the-art DNN frameworks (TFLite [1] and Qualcomm SNPE [178]) that support end-to-end mobile DSPs execution, GCD² achieves 2.8× and 2.1× speedup (in geometric mean), respectively, reaching *real-time execution* for some of them for the first time. In fact, for two of the models, GCD² implementation supports mobile DSP execution for the first time. While comparing with three established compilers (Halide [189], TVM [25], and RAKE [5]) that support efficient kernels execution on mobile DSPs, GCD² achieves 4.5×, 3.4×, and 4.0× speedup, respectively. GCD² outperforms others primarily because of improved SIMD execution and optimized VLIW instruction scheduling and the evaluation justifies the choices made in GCD²'s algorithms for these optimizations.

4.2 Executing DNNs on Mobile DSPs

Modern mobile DSPs have become increasingly powerful with key features as follows: 1) larger SIMD widths, 2) richer vector instructions with growing computation capabilities, and 3) more flexible instruction pipelines that can tolerate certain data dependencies. Take Qualcomm Hexagon 698 DSP [184]² as an example. Its SIMD width is 1024-bit, twice that of Hexagon 680 [174] and its instruction set includes multiple SIMD/vector instructions (e.g., `vmpy`, `vmpa`, and `vrmpy` elaborated in Section 4.3), and can support complicated MAC (multiply-accumulate) operations. Multiple vector (and scalar) instructions can be packed into a VLIW pipeline, further improving the computational throughput. Finally,

²Qualcomm Snapdragon is one of the most popular SoC and many generations of Snapdragon are equipped with Hexagon DSPs. Although our presentation and evaluation is on Hexagon DSP, the work is generally applicable to other mobile DSPs as well, e.g., Cadence, which is the other major player in the mobile DSP market.

the pipeline offers hardware mechanisms to guarantee execution correctness even in the presence of certain dependencies, thus offering more flexibility.

Mobile DSPs support fix-point operations (8/16/32-bit) with extremely high performance (e.g., the theoretical peak for Hexagon 698 DSP is 15 TOPS [186]). While considering DSP chips for DNN execution, the important context here is that *Quantization*, a well-known technique to convert floating-point values to integer ones, has been very effective in accelerating DNN executions, particularly on resource-constraint devices [32, 173]. The cutting-edge DNN acceleration frameworks, (e.g., TFLite [1] and SNPE [178], and Qualcomm’s built-in library Hexagon NN [182]) aim to combine the benefits of both quantization and mobile DSPs to accelerate DNN execution, achieving both (near) state-of-the-art model accuracy and lower latency as compared to the other parts of the mobile SoC (i.e., CPUs and GPUs). Similarly, MobiSR schedules the super-resolution model over Heterogeneous Mobile Processors (including CPU, GPU, and DSP) [126].

Despite these rapid developments, compilers and libraries built for DSP chips cannot fully exploit the device’s computation power – this applies to, but is not limited to, the compilers and libraries for DNN execution listed above. Specifically, the performance of the mobile DSP is sensitive to 1) the input/output data layout, and 2) the VLIW instruction packing (or scheduling) in view of all hardware resource constraints. This is because first, various SIMD/vector instructions are designed to perform MAC operations in different ways and they are friendly to different input/output data sizes and data layouts. Second, the VLIW pipeline imposes many constraints on the instructions that can be packed together.

In the context of DNN acceleration, complex DNN designs challenge the DSP-oriented implementations in multiple ways. First, modern DNNs usually consist of many operators (e.g., the latest BERT consists of over 1000 operators [46]), and even with the same operator, operands can be of different shapes and sizes. Mapping growing SIMD/vector width and instruction set (variety) to these operators and operands is challenging. Second, as discussed above, the complex opportunities and constraints in VLIW packing need to be considered for implementations of specific operators.

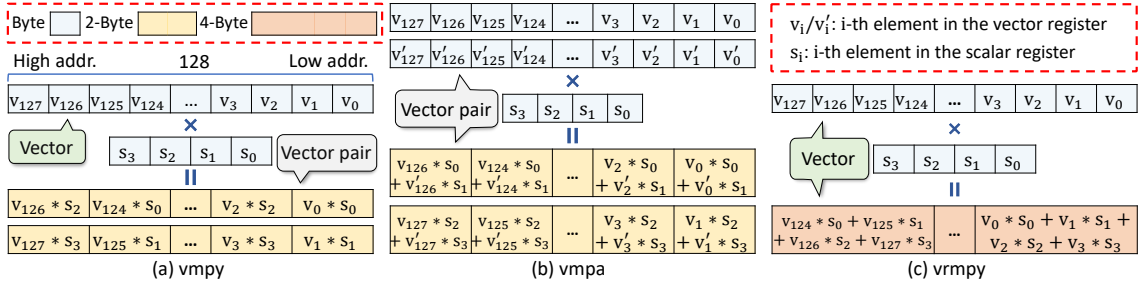


Figure 4.1: SIMD/Vector Multiply Instruction Examples in Mobile DSP Chip

Table 4.2: Execution Latency w/ Different SIMD Instructions (and Layouts) for Matrix Multiplication $C = A \times B$. M , K , and N denote the dimension size of Matrix A ($M \times K$), B ($K \times N$), and C ($M \times N$), respectively. Execution latency and total data size with padding are normalized by `vmpy` for readability. Smaller numbers mean better latency or less padding. Bold ones denote the best case.

M	K	N	Execution Latency			Total Data Size w/ Pad		
			<code>vmpy</code>	<code>vmpa</code>	<code>vrmpy</code>	<code>vmpy</code>	<code>vmpa</code>	<code>vrmpy</code>
32	32	32	1.00	0.79	0.63	1.00	0.56	0.33
64	64	64	1.00	0.69	0.76	1.00	0.60	0.60
96	96	96	1.00	1.06	0.89	1.00	1.00	0.82
128	128	128	1.00	1.10	1.23	1.00	1.00	1.00

4.3 Instructions and Layouts

Our target instruction set includes novel and complex SIMD instructions capable of optimizing computations found in ML (and scientific) workloads. We show three representative instructions in Figure 4.1. While these instructions are used for multiple operators in a DNN (e.g., the convolutions), our presentation here uses matrix multiplication for illustration. Similarly, other instructions like `vtmpy` and `vmpye` can also be used to implement these operators. Our discussion here considers only three instructions. However, as a motivation, we first show the trade-offs between their use.

Table 4.2 shows how the cost of matrix multiplication varies with the three choices when input tensors have different shapes. We can see that the instruction `vmpy` (and the corresponding 1-column layout, both are elaborated later) provides better execution efficiency if the operands have a certain length. However, for other cases, this instruction causes padding overheads, thus making the other instructions more time- and space-efficient.

As additional background, many recent works show that the floating-point representations (and operations) for weights and activations are not necessary to achieve good accuracy for DNNs, but instead fixed point (8-bit or even less) suffices [32,90,91,98,105,238]. However, one caution is that the product between two 8-bit values should be stored in 16-bits to avoid data overflow, and similarly, accumulating several such products requires 32-bits. In either case, a requantization phase is required to generate the 8-bit final output.

With this motivation and background, we explain the existing instructions and associated data layouts we have developed. In Figure 4.1 (a), we show the instruction `vmpy`, whose inputs are a vector with 128 8-bit values and four scalar values. In `vmpy`, four consecutive values in the vector are multiplied by four distinct scalars, with the output being two vectors with 64 16-bit values, each storing alternate results of multiplications.

In Figure 4.1 (b), the input for the instruction `vmpa` are two vectors with 128 8-bit values each. A pair of corresponding values from the two vectors are multiplied by two scalar values and then added together. Specifically, alternate pairs are multiplied with the first two and the last two scalars, respectively, and accumulated to two different output vectors.

Finally, in Figure 4.1 (c), the instruction `vrmpy` is illustrated – here, four consecutive values from the vector are successively multiplied by four distinct scalar values, and accumulated together. The result is a vector with 32 32-bit values.

In this work, we have developed novel dense matrix data layouts that optimize the use of these instructions for multiple key operators in DNN computations (e.g., `MatMul`, `CONV`, `Depthwise CONV`, etc.), and this part takes matrix multiplication (`MatMul`), a critical kernel for our target workload as an example. Developing layouts for implementing arbitrary loop nests using these or similar instructions is an open problem beyond the scope of this paper.

In Figure 4.2 (a), we show the layout that enables the use of `vmpy` instruction shown earlier in Figure 4.1 (a). For efficiency, it is very important that the set of values that are to be loaded to or stored from a vector register are stored in a contiguous fashion. The layout we use is referred to as the `1-column` layout. The numbers shown in the boxes

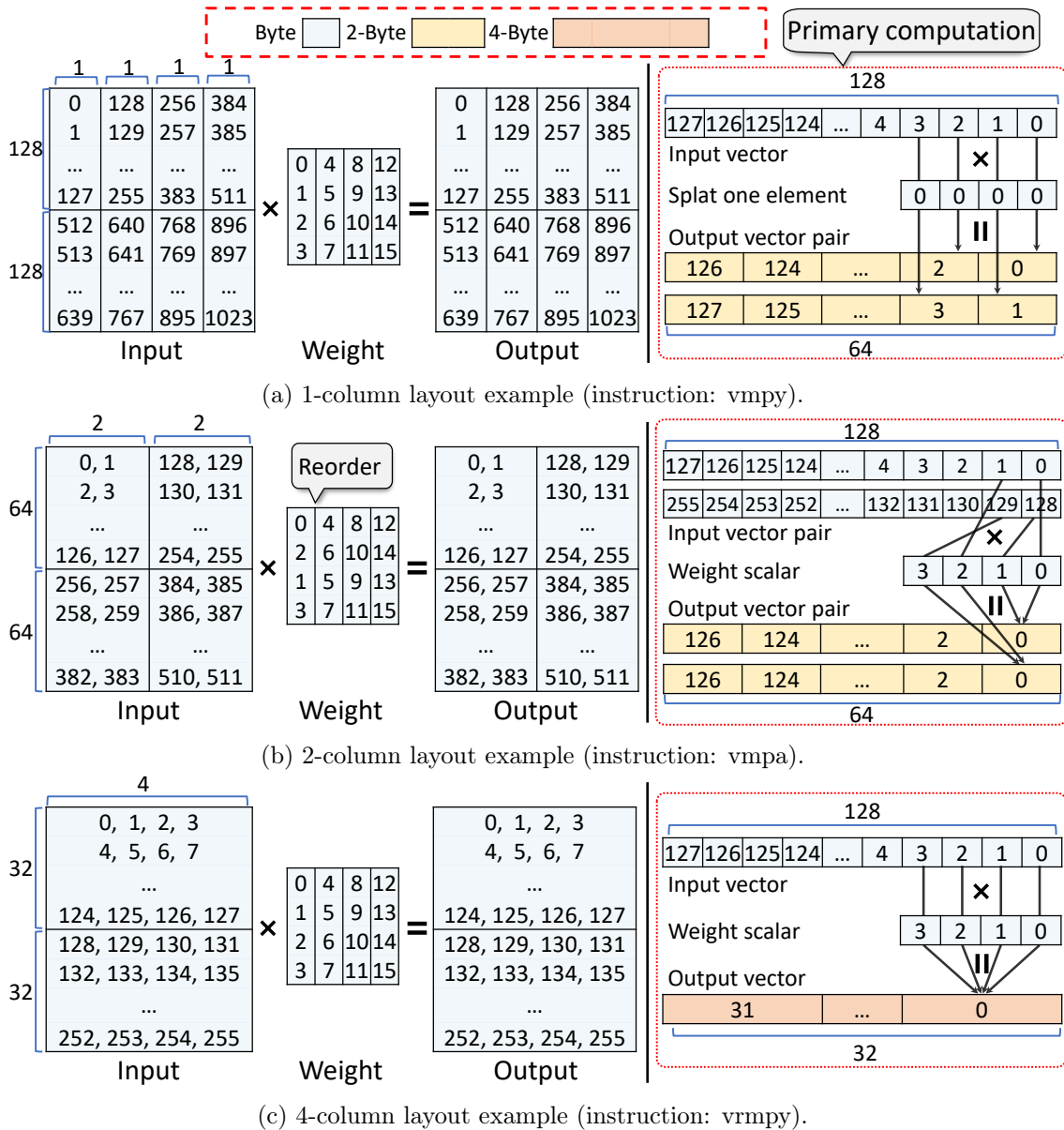


Figure 4.2: Data Layouts to Support Usage of Varied SIMD Instructions for Matrix Multiplication. Each number denotes the linear storage offset of an element. A blue, yellow, and orange cell takes 1, 2, and 4 bytes, respectively. Left shows data storage, and right shows computation.

represent the offset of the location of that element. In `1-column` layout, a set of 128 rows is stored in a column-major way, and this pattern is repeated for the next set of rows. In carrying out the matrix multiplication, the first column is loaded to a vector and all values are multiplied with the first weight (0) stored in the scalar register. The outputs are two vectors storing 64 16-bit elements each, which will eventually be shuffled to obtain an output layout matching the input layout. The process continues by loading the next 128 elements physically stored in our layout, multiplying them with the second weight (1), and reducing the output to the same two vectors.

In Figure 4.2 (b), we show the layout and the key steps of matrix multiplication with the instruction `vmpa`, which was shown earlier in Figure 4.1 (b). The layout we have designed is referred to as `2-column` layout – within the 64-row panels, values for 2 columns are stored adjacent to each other, before following the column-major storage. In applying matrix multiplication, elements 0, 1, 128, and 129, which are from the same rows of the matrix, are multiplied by the output weights 0, 1, 2, and 3, respectively, stored in scalar registers. Note that the two corresponding output elements in the output vectors need to be further added to obtain the results.

In Figure 4.2 (c), we show the matrix multiplication operations and layouts with the use of the instruction `vrmpy` shown in Figure 4.1 (c). The input and weight matrix are of different shapes as compared to the previous examples, in order to illustrate the layout and the computation. Here, panels of 32 rows are used and four elements from each row are stored together. Four elements in a row are multiplied with four weights stored in scalar registers. We also note that while there is an instruction somewhat similar to `vrmpy` in Intel instruction set (`vpdpbusd`), there are no counterparts to `vmpy` or `vmpa` at the current time.

Overall, our work considers a relatively small number of candidate instructions for implementing a single operation, using a “pre-designed” approach for each pair of operator and instruction. Efforts do exist on trying to automate the selection of instruction and code generating using the instruction [5, 222, 223]. We have conducted a brief comparison of our approach against the code generated by the most recent of these efforts (which also targets

Table 4.3: SIMD Instructions Selected and Performance by RAKE [5] and GCD². Representative Conv2d kernels (w/ varied shapes, 7×7 , 1×1 , and 3×3) are from ResNet-50.

Input shape	Conv2d properties		Instruction		Speedup
	Weight shape	Output shape	RAKE	Ours	Ours/RAKE
1x3x224x224	64x3x7x7	1x64x112x112	vrmpy	vmpy	1.63x
1x64x56x56	64x64x1x1	1x64x56x56	vmpy	vmpa	1.98x
1x128x28x28	128x128x3x3	1x128x28x28	vrmpy	vmpy	2.06x

the same instruction set), i.e. RAKE [5]. As shown in Table 4.3, our approach is able to deliver significantly higher performance. Thus, while automation of instruction selection and code generation is valuable, current approaches are not matching the “pre-designed” approach we are taking.

4.4 System Design of GCD²

This section highlights the major optimizations developed in GCD², followed by a brief summary of implementations.

4.4.1 SIMD Global Opt. Problem Formulation

From the discussion earlier in Section 4.3, the important takeaway is that different instructions can be used for the same operation, but with different requirements on input formats, resulting in different output formats, and with different trade-offs (which were summarized earlier in Table 4.2).

With a relatively small number of instructions available to implement a single operation, the instruction and the layout selection can be performed (in isolation) by explicitly considering all choices and choosing the one that requires the fewest cycles for execution. However, it turns out that with distinct input and output layouts for different instructions, choices for each operation cannot be made in isolation. Suppose an operator A can be implemented in the most efficient fashion using the instruction `vmpa`. Let the output of the operation A be the input to the operation B . Without considering the need for

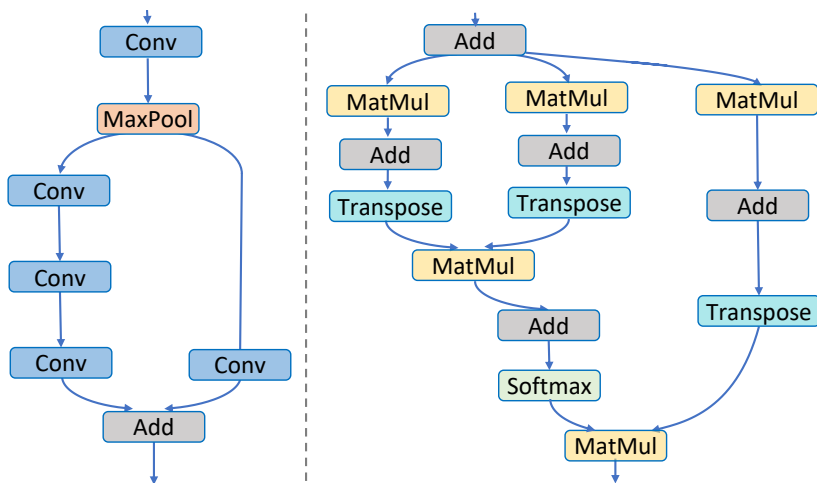


Figure 4.3: Examples of Computational Graphs. Left and right show partial CGs in ResNet [77] and TinyBERT [97].

the formatting of input tensors, let the most efficient implementation of B be using the instruction `vrmpy`. However, the output tensor from the operation A will be in the two-column format (Figure 4.2 (b)), whereas if B is implemented using `vrmpy`, it is expected that the input tensors are in the four-column format (Figure 4.2 (c)). Converting the layout of a tensor itself is a time-consuming step. Thus, if the sequence of two operators A and B are considered, it is possible that the most efficient implementation involves using the same instructor (and thus layouts) for the two operators. In practice, DNN models use many operators (e.g., the model EfficientDet-D0 used in our evaluation has 822 operators), and thus, we have a complex optimization problem.

To formulate this global optimization problem, we use an existing intermediate representation called the Computational Graph (CG) [25], which captures the data-flow and basic operator information like the operator type and parameters. Figure 4.3 shows examples of such graphs. Let V be the set of vertices in a CG and let E be the set of edges. Each vertex is an operation that produces exactly one output tensor. A directed edge (v_i, v_j) denotes that the output of the vertex (operation) v_i is (one of) input(s) to the operation v_j . The source of the edge e is also denoted as $vin(e)$ and similarly, the destination of e is denoted as $vout(e)$.

Now, given an operator (vertex) \mathcal{O} in the CG, let it have a set of immediate predecessors we denote as $Pre(\mathcal{O})$. By each predecessor, we denote interchangeably both the operators and their output. After performing the local analysis of possible implementations and associated layouts for the operator \mathcal{O} we obtain a set of possible execution plans $EP(\mathcal{O})$, comprising execution plans $ep_1(\mathcal{O})$, $ep_2(\mathcal{O})$, and so on. Associated with every execution plan, there is a cost of execution, denoted by $Cost(ep_i(\mathcal{O}))$, which is based on the number of instructions (cycles) required. This cost calculation assumes that all input tensors are already stored in the required layout for the SIMD instruction used.

We consider an execution plan $ep_i(\mathcal{O})$ and a predecessor tensor of \mathcal{O} , which we denote as \mathcal{J} ($\mathcal{J} \in Pre(\mathcal{O})$). If the operator \mathcal{J} is executed with the plan $ep_j(\mathcal{J})$, then there could be a data transformation with the associated cost $TC(ep_j(\mathcal{J}), ep_i(\mathcal{O}))$ (this cost will be 0 when data transformation is not required).

Given this background, the global optimization problem is as follows. For each operator (vertex) v in the CG, we want to select an execution plan ep_v , such that the total cost of execution for the graph G , which is denoted as

$$Agg_Cost(G) = \sum_{v \in V} Cost(ep_v(v)) + \sum_{e \in E} TC(ep_{vin(e)}(vin(e)), ep_{vout(e)}(vout(e))) \quad (4.1)$$

is minimized. In the expression above for $Agg_Cost(G)$, the first term is the cost of execution associated with each operation under the choice of plan made, whereas the second term is the cost of data transformation between the layouts for the source and the sink of the edge, under the choices of implementation plans chosen for the source and sink operators.

4.4.2 Layout & Instruction Select Solution

It is easy to see that a trivial approach for solving this problem will involve comparing $k^{|V|}$ options, where $|V|$ is the number of vertices in the graph and k is number of (assumed fixed for all operators) options available for each operator. Even when k is 2 or 3, this cost can be easily prohibitive for realistic DNN models. Furthermore, the above problem is really

a Partitioned Boolean Quadratic Programming (PBQP) problem, which is known to be NP-hard [7].

If we simply have a linear chain of operations O_1, O_2, \dots, O_n , then the following approach can be used to solve the problem. Let $Sol(i, j)$ denote the lowest possible cost of execution operations O_1, O_2, \dots, O_i such that the output from the operator O_i is the j^{th} available choice ($j \leq k$, where k is the number of choices available for each operator). Then, we have

$$Sol(i, j) = \min_{l=1, \dots, k} (Sol(i-1, l) + TC(ep_l(O_{i-1}), ep_j(O_i))) \quad (4.2)$$

Here, $Sol(i, j)$ is computed by comparing k choices, which are the lowest cost ways of reaching each of the k different output formats for the previous operation in the chain. It is easy to see that this recurrence can be solved in $O(|V| \times k^2)$ time. Moreover, this solution can be easily extended to the cases when either every path from a “source” vertex of a DAG to a given vertex is of the same length, or when every vertex has at most one output. However, this approach does not work for an arbitrary DAG and we focus on an effective heuristic solution. While considering a PBQP solver [7, 71], which is not guaranteed to provide an optimal solution but is in practice close, is an option, we instead focus on exploiting the properties of our target domain. For this, we consider the following definition:

Definition 4.4.1 (Cost Optimal Partitioning). Given a computational graph G , a cost optimal partitioning of G is a disjoint graph partitioning $P = \{G_1, G_2, \dots, G_n\}$, such that for $Agg_Cost(G)$ (as defined in Equation 4.1), we have

$$Agg_Cost(G) = Agg_Cost(G_1) + Agg_Cost(G_2) + \dots + Agg_Cost(G_n) \quad (4.3)$$

Note that we use the popular definition of graph partitioning, where the edges between vertices that are in different partitions are not considered part of either partition. If such partitioning can be found, the optimal plans for all operators within each partition can be determined in isolation, translating to a significant reduction in the complexity of search.

In practice, What we can hope to achieve is to find a set of partitions that can be optimized independently, i.e. where the lowest cost for the entire graph is achieved by

choosing plans within each partition independently. To achieve this, we note that an edge $e = (v_i, v_j)$ is a desirable partitioning edge if 1) the node v_j has only one predecessor (v_i), and 2) The operator v_j is a *layout transformation operator* or the transformation along the edge e is a *profitable* transformation. Typical examples of layout transformation operators include **Reshape** and **Transpose** – they do not perform any computations but change the shape of the operand. A transformation along an edge is considered *profitable* if the reduction in execution time of the successor operator with the transformed layout is higher than the cost of the data transformation itself. The intuition for this definition of desirable partitioning edge is that decisions on nodes leading up to this edge and vertices following this edge can be made in isolation.

However, as next challenge for us, partitioning a graph typically involves many *cut edges*. Now, if we can find a cut edge that is *dominant*, i.e, if every path from the (assumed to be unique) source vertex in the DAG to the (again, assumed unique) sink vertex passes through this edge, then the problem is simplified. When this is not feasible, we can add *complementary edges* to the identified cut edges to create complete partitions.

4.4.3 VLIW Optimization

Instruction packing or scheduling is a long-standing issue in VLIW research that has been proved to be NP-hard [124, 230]. Because of the specific opportunities as well as challenges associated with our target architecture, a new algorithm is developed in this work.

Optimization Foundation: Hard/Soft Dependencies. For our target architecture, dependencies between two instructions can be characterized into two types with respect to their implication on placing them in the same VLIW packet³

- *Hard dependency* denotes a strict dependence relationship where placing such instructions in the same packet likely produces incorrect results.

³This classification is independent of the traditional classification of dependencies into flow/RAW, output/WAW, and anti/WAR, though soft dependencies can only be RAW or WAR.

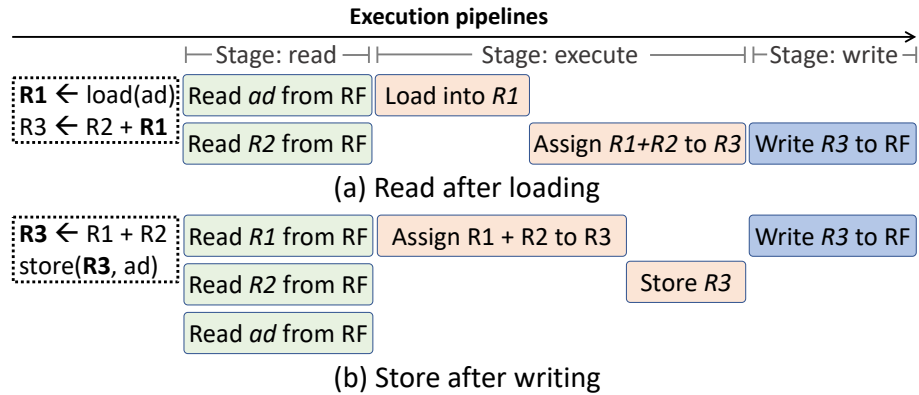


Figure 4.4: Two Examples of Packing Instructions with Soft Dependencies. Different colors show different VLIW execution pipeline stages (**read** in green, **execute** in orange, and **write** in blue). In (a), the second stage (**Assign R1+R2 to R3**) of the second instruction requires to wait for the completion of the first instruction, incurring packing penalty. A similar situation happens to (b) between **Assign** and **Store**.

- *Soft dependency* denotes a relaxed dependence relationship, and placing instructions together produces correct results; however, the resulted execution performance is likely degraded to a certain degree. An example of the soft dependency in our target architecture is the one between a scalar addition operation and a consumer of the result of such an addition.

To further illustrate the nuances associated with soft dependencies, we show two examples of packing instructions with soft dependencies in Figure 4.4. Figure 4.4 (a) shows a dependency between a load operation and an arithmetic operation that consumes the loaded value. Each of these two instructions takes 3 clock cycles⁴ individually, however, if they are packed in the same packet, they can execute correctly by taking 4 cycles in total⁵. This example shows that packing instructions with soft dependencies together takes less clock cycles than not packing them together at all (i.e., treating the soft dependency as a hard dependency). However, if sufficient number of instructions are available without any

⁴According to the target microarchitecture design, each VLIW pipeline execution comprises three stages, **read** from Register File (RF), **execute**, and **write** to RF, though some of these stage can be be empty. Our explanations will assume that each stage is 1 clock cycle [177].

⁵Mobile DSP processors (e.g., Hexagon DSPs) execute instructions within each VLIW packet in parallel, but without overlap between packets.

dependencies between them, we will prefer to not pack instructions with soft dependencies together. Figure 4.4 (b) shows a similar example with a soft dependency between an arithmetic operator and a store operation. Which dependencies are soft and which ones are hard depends upon the microarchitecture. This information needs to be obtained from the details of processor implementation (e.g., [177]) and made available to the instruction packing algorithm.

Soft Dependencies Aware (SDA) VLIW Packing Algorithm. Because of the notion of soft dependencies, we have developed a new VLIW instruction packing algorithm. Besides handling the distinction between soft/hard dependencies, the algorithm is cognizant of other constraints. While a packet can have up to 4 instructions, there can be a limited number of slots for each type of instruction. As an example, packing two shift operations together is not allowed. This instruction packing is implemented as an additional optimization step of LLVM’s assembly code generation.

Like much of the previous work, the packing algorithm uses the notion of a *critical path* [207]. and its overall goal of minimizing execution time as two sub-goals: 1) reducing the total number of instruction packets, and 2) packing instructions with identical or similar latency together to minimize VLIW pipeline stalls. The work also has many similarities with algorithms for code generation targeting superscalars, in the sense the goal is to minimize intra-bundle RAW stalls [10].

Our presentation of the algorithm (as shown in Algorithm 1) is supported by the running example from Figure 4.5. Its left hand side shows the pseudo assembly code for a part of an innermost nested loop of a frequently occurring `Add` operator in deep neural networks ($R = A + B + C$), where A , B , and C are two-dimensional uint8 arrays and R is a two-dimensional int16 array. Take the instruction of `v2 : 1 = vadd(v1, v2)` in this pseudo assembly code as an example. $v1$ and $v2$ are two 8-bit registers. `v2:1` denotes a 16-bit register combining 2 8-bit registers ($v2$ and $v1$) to store the addition result.

Returning to our algorithm, it first builds a Control-Flow Graph (CFG) on assembly for each operator, and finds the basic block corresponding to the computation kernel of

Algorithm 1: Soft-dependency-aware VLIW Packing

```
Func packing: instructions  $\leftarrow$  [Packet]
1  cfg  $\leftarrow$  build_cfg(instructions)
2  all_packets  $\leftarrow$  Stack()
3  foreach block in cfg.block do
4      idg  $\leftarrow$  build_IDG(block)
5      free_insts  $\leftarrow$  Set()
6      find_free_instruction(idg, free_insts)
7      while free_insts is not empty do
8          /* Build critical path from IDG */
9          critical_path  $\leftarrow$  get_critical_path(idg)
10         cur_packet  $\leftarrow$  critical_path[-1]
11         /* Iterate all the free instruction */
12         while len(cur_packet) < 4 do
13             /* Select the most profitable instruction */
14             inst  $\leftarrow$  select_instruction(free_insts, cur_packet)
15             find_free_instruction(idg, free_insts)
16             if inst is None then
17                 | break
18             else
19                 | cur_packet.add(inst)
20                 | idg.remove(inst)
21         all_packet.add(cur_packet)
22 return all_packets

Func select_instruction: free_insts, packet  $\leftarrow$  Instruction
23 all_insts  $\leftarrow$  resource_constraint(free_insts, packet)
24 if all_insts is empty then
25     | return NULL
26 hi_lat  $\leftarrow$  highest_latency(packet)
27 best  $\leftarrow$  NULL
28 foreach i in all_insts do
29     /* The criteria of profitability */
30     i.score  $\leftarrow$  (i.order + i.pred)  $\times$  w - abs(hi_lat - i.lat)  $\times$  (1 - w)
31     if soft_dependency(i, packet) then
32         | i.score  $\leftarrow$  i.score - p(i, packet)
33     if best is NULL or best.score < i.score then
34         | best  $\leftarrow$  i
35 return best
```

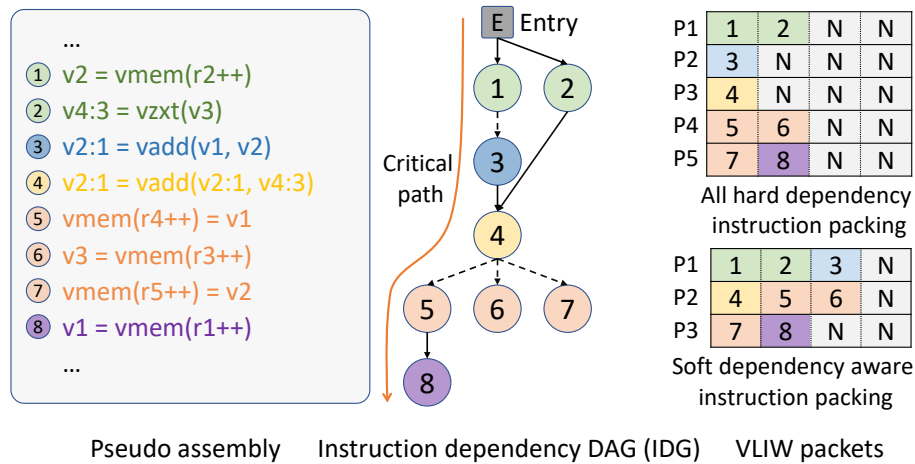


Figure 4.5: An Instruction Packing Example. The left part shows part of the pseudo assembly code for the innermost nested loop performing 2D Element-wise Addition: $R = A + B + C$, where A , B , and C are two-dimensional uint8 arrays and R is a two-dimensional int16 array. $v2:1$ denotes a 16-bit register combining 2 8-bit registers $v2$ and $v1$. The middle part shows an IDG, in which, solid edges denote hard dependency, dot edges denote soft, and critical path is colored in red. Right shows the packing results from our solution and an sub-optimal solution that treats all soft dependencies as hard (*soft to hard*). N denotes an empty instruction slot.

each operator (usually the largest basic block). Next, it builds an instruction dependency DAG (called IDG) based on the hard/soft dependency information, and finds the critical path with the longest execution latency. The middle part of Figure 4.5 shows the IDG – here, a vertex represents an instruction, and an edge represents the dependence between two instructions. A solid edge represents a hard dependency and a dotted edge represents a soft dependency. Take the instructions (or vertices) 4, 5, 6, and 7 in this figure as an example. The dependencies between the instructions 4 and 5, 4 and 6, and 4 and 7 are all soft dependencies. IDG also contains an artificial entry vertex. The number shown with the vertex corresponds to the assembly instruction in the left. The critical path is colored in red. The vertices with identical colors have the same rank (distance to the entry).

Based on the IDG and the critical path, the algorithm now packs instructions. When creating a new packet, the algorithm always uses the last (unpacked) instruction in the critical path as a seed (line 9). Next, such an instruction is packed with other instructions

that either do not have any outgoing edges or have only *soft-dependence edges* to an instruction to be packed (all of these instructions are called *free* instructions). This step consists of three major sub-steps: 1) iterating through all free instructions (line 7), 2) finding a candidate instruction from the set of free instructions (line 11), and 3) grouping the candidate instruction into the current packet. Particularly, the key second sub-step (i.e., finding a candidate instruction) comprises of two steps: first, for the current working packet, the algorithm finds all instructions that can be packed while meeting the hardware constraints (line 20), and also determines the highest latency (hi_lat) among the instructions that are already in the current packet; second, it iterates these available instructions to pick up the *best* instruction and returns it (lines 25 to 30). Note that, the *best* instruction selection is based on this instruction’s *score* ($i.score$) that is calculated as follows:

$$i.score = (i.order + i.pred) \times w - abs(hi_lat - i.lat) \times (1-w) \quad (4.4)$$

According to Equation 4.4, the score of an instruction is decided by its three attributes, its distance from the entry node ($i.order$), its predecessor instruction count ($i.pred$), and its latency ($i.lat$). The first two have positive impacts on the score, while the absolute difference between this instruction’s latency ($i.lat$) and the latency of the longest instruction already in the current packet (hi_lat) has a negative impact on the score. The former is because it is desirable to include instructions that have a longer chain of dependencies and/or a total large number of instructions that it is dependent on. The latter, on the other hand, wants to create more efficiency by packing instructions of the same (or very similar) latency values together. This algorithm introduces two new parameters (w is short of weight, and p is short for penalty) that are empirically decided. w aims to control the weight of the three factors’ impact (line 26), while p aims to control the impact of *soft dependency* on this packing (line 28). Specifically, the value p depends both on the instruction i under consideration and the instructions already placed in the *packet*, and captures the stall that the soft dependence will cause. For comparison in our experiments, we also create a version of our algorithm that reduces all soft dependencies to ‘none’ or no

dependence – this version of the algorithm will ignore the calculation of this penalty. Next, to complete the description of our algorithm, after one packet is created, the algorithm repeats by finding the critical path of the remaining sub-graph.

Returning to Figure 4.5, the right part shows the packets after scheduling (N denotes an empty slot). This example compares our Soft Dependencies Aware (SDA) packing algorithm (bottom) with a sub-optimal algorithm (called *soft to hard*) that treats all soft dependencies as hard ones (top). Taking the first seed (vertex 8, i.e., the last instruction in the critical path) as an example. 8 and 6 cannot be packed together (because of hardware constraints) and 8 can be packed with 7. Our packing algorithm can continue to explore the packing opportunity between 4 and 5/6 because 5 and 6 only have soft dependencies to 4, and the soft dependencies allow the packing for 1, 2, and 3; however, these opportunities do not exist in *soft to hard* version of the algorithm. In summary, our algorithm delivers a schedule with only three packets, while the sub-optimal *soft to hard* version generates a schedule with two additional packets. Evaluation results in Section 4.5 further validate our algorithm’s efficacy.

Impact of Unrolling. Loop unrolling plays an important role in the schedule quality by affecting the scheduling scope and the register pressure. Different from previous work like [196], GCD² employs a low-cost heuristic solution specifically designed for DNN operators. The basic idea is to perform a fast *adaptive* unrolling setting selection according to the shape of output tensors, for example, for GEMM, different unrolling settings are designed for varied output shapes (skinny, near-square, and fat). Our empirical study in Section 4.5 proves that this approach outperforms some simple selections while also yielding comparable performance gains to a much more expensive exhaustive search.

4.4.4 Putting Everything Together

GCD² is implemented on top of an existing end-to-end DNN execution framework, PatDNN [67, 157, 158] to support efficient mobile DSP execution. Figure 4.6 shows the system workflow of GCD². First, it converts the post-training quantized model to a computational graph

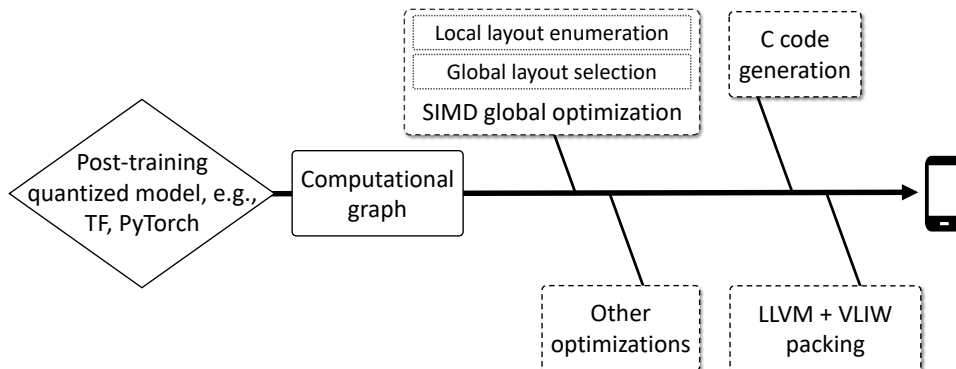


Figure 4.6: System Workflow of GCD².

(and optimizes it with various techniques, e.g., constant folding) by leveraging the existing framework. Second, it feeds the (optimized) computational graph to the SIMD global optimization module to conduct the local layout (instruction) enumeration and the global layout (instruction) selection. The result here is an optimized SIMD code generation plan including the data layout for each operator and corresponding SIMD instructions to use. This is followed by a pass where other optimizations are applied, e.g., replacing an expensive division operation with a database lookup operation. As the next step, the existing framework and the optimized SIMD code generation plan lead to a “low-level” C code with input/output tensor storage details and optimized SIMD intrinsics. Finally, it employs LLVM [143] with our VLIW packing optimization to generate the optimized executable code on the mobile DSP.

4.5 Evaluation

This section evaluates the performance of GCD² by comparing it with five state-of-the-art frameworks, TFLite [1] (V2.6.0), SNPE [178] (V1.55), Halide [189] (V12.0.1), TVM [25] (V0.8.0), and RAKE [5] (V1f99df1). More specifically, TFLite, SNPE, and TVM are the state-of-the-art production-level DNN execution frameworks that can support (or partially support) our target mobile DSP. Both TFLite and SNPE call Hexagon NN, an expert-written hand-tuned library designed by Qualcomm. However, as end-to-end DNN execution

Table 4.4: Overall Performance Comparison among TFLite, SNPE, and GCD² on Mobile DSP. “-” means this model is not supported by the framework yet. OverT and OverS are the speedup of GCD² over TFLite, and SNPE, respectively. GCD²’s overall *compilation time* for these models ranges from 5 minutes (WDSR-b) to 25 minutes (EfficientDet-d0).

Model	Type	Task	#MACS	#Params	#Operators	TFLite (ms)	SNPE (ms)	GCD ² (ms)	OverT	OverS
MobileNet-V3	2D CNN	Classification	0.22G	5.5M	193	7.5	6.2	4.0	1.9	1.6
EfficientNet-b0	2D CNN	Classification	0.40G	4M	254	9.1	9.2	6.0	1.5	1.5
ResNet-50	2D CNN	Classification	4.1G	25.5M	140	13.9	11.6	7.1	2.0	1.6
FST	2D CNN	Style transfer	161G	1.7M	64	935	870	211	4.4	4.1
CycleGAN	GAN	Image translation	186G	11M	84	450	366	181	2.5	2.0
WDSR-b	2D CNN	Super resolution	11.5G	22.2K	32	400	137	66.7	6.0	2.1
EfficientDet-d0	2D CNN	2D object detection	2.6G	4.3M	822	62.8	-	26	2.4	-
PixOr	2D CNN	3D object detection	8.8G	2.1M	150	43	26.4	11.7	3.7	2.3
TinyBERT	Transformer	NLP	1.4G	4.7M	211	-	-	12.2	-	-
Conformer	Transformer	Speech recognition	5.6G	1.2M	675	-	-	65	-	-
Speedup (geometric mean)									2.8	2.1

frameworks, their computational graph optimizations (graph rewriting, operator fusion, etc.) are different, thus resulting in very different execution performance (as shown in Table 4.4). Halide, TVM, and RAKE use LLVM as their back-end to generate DSP instructions. They perform packet generation without distinguishing between soft and hard dependencies (i.e., they treat each soft dependency as a hard dependency). It should be noted that Halide, TVM, and RAKE are tensor compilers, while GCD² comprises both tensor compiler optimizations (e.g., global data layout optimization) and language compiler optimization (instruction packing). We introduce a version of GCD² to facilitate a comparison of tensor compiler aspect of our work with these systems, as we will describe later. Our evaluation has four main objectives: 1) to demonstrate that GCD² outperforms all of these state-of-the-art frameworks on mobile DSP (Section 4.5.2); 2) to identify the benefits of specific optimizations and the choices made in our algorithms (Section 4.5.3); 3) to study the power consumption and energy efficiency of GCD² against alternative implementations on the same chip (Section 4.5.4); 4) to compare the inference speed and energy efficiency of our mobile DSP-based solution with other embedded DNN accelerators (Section 4.5.5).

4.5.1 Evaluation Setup

Models and Datasets. GCD² is evaluated on 10 state-of-the-art neural networks (see Table 4.4) that are categorized into seven groups according to the tasks they perform. Particularly, they include 1) three image classification two-dimensional CNNs (MobileNet-V3 [83], EfficientNet-b0 [213], and ResNet-50 [77]); 2) one image style transfer two-dimensional CNN (FST [99]); 3) one image-to-image translation GAN (CycleGAN [260]); 4) one super resolution two-dimensional CNN (WDSR-b [251]); 5) two object detection two-dimensional CNNs (EfficientDet-d0 [214], and PixOr [245]); 6) one trans-former-based NLP model (TinyBERT [97]); and finally, 7) one transformer-based speech recognition model (Conformer [68]). All the evaluated models in this section are quantized by a standard approach used by well-known TFlite [216](with identical post-training quantization across all frameworks) with 8-bit integers being used for weights and feature maps (activations).

It should be noted that the choice of datasets has a negligible impact on the final inference latency or relative execution speeds, which are the primary metrics in our evaluation. Therefore, and also because of space limitations, we report results from one dataset for each model. MobileNet-V3, EfficientNet-B0, ResNet-50, and CycleGAN are trained on the ImageNet dataset [41], WDSR-b is trained on DIV2K [4], EfficientDet-d0 and FST are trained on COCO [136], PixOr is trained on KITTI [59], TinyBERT is trained on BooksCorpus [46] and English Wikipedia [46], and Conformer is trained on [162]. Because all frameworks employ the identical model quantization approach, they achieve the same accuracy on all models and datasets, and thus accuracy is not reported.

Test Bed. Most of the experiments described in this section are conducted on a Samsung Galaxy S20 (with Snapdragon 865 SoC [184]) that consists of an octa-core Kryo 585 CPU, Adreno 650 GPU, and Hexagon 698 DSP (with Vector eXtensions support). We also tested our framework on older series Snapdragon platforms, which show the similar performance gains against other baseline frameworks. We omit the results due to the space constraints. We note that our optimization designs are general, potentially applicable

to other mobile DSP architectures (e.g., Cadence DSPs with increasingly complex SIMD and VLIW supports). All models are executed with their best configurations while the same parameters are used for all execution platforms. Each data involves inferences on 50 different inputs. After excluding the highest/lowest time, an average is taken and reported. As the variation is negligible, ranges are not reported.

4.5.2 Comparison with Other Frameworks

This part evaluates the overall performance of GCD² by comparing it against five state-of-the-art frameworks, TFLite, SNPE, Halide, TVM, and RAKE. We compare the performance of GCD² with TFLite and SNPE over 10 models. While Halide, TVM, and RAKE have the capability to generate code for the DSP chip, they currently cannot execute full DNN models on this platform. Thus, a `Conv2d` kernel is used for comparison against Halide, TVM, and RAKE.

Execution Latency. Table 4.4 shows the overall performance comparison for all 10 models. TFLite and SNPE do not support Transformer-based models. For the other 8 models, GCD² achieves 1.5× to 6.0×, and 1.5× to 4.1× speedup over TFLite and SNPE, respectively. Table 4.4 shows that

GCD² outperforms TFLite and SNPE mainly because of 1) optimized SIMD instruction selection and layout transformation, and 2) optimized SDA VLIW packing by taking soft dependencies into account. TFLite and SNPE employ a uniform SIMD implementation for each operator type to support mobile DSP execution, and their VLIW packing does not consider soft dependencies as GCD². It turns out that GCD² achieves the most speedup (6.0× over TFLite) on WDSR-b. The reason is that feature map shapes in WDSR vary significantly among different operators, and our instruction selection and layout transformation optimizations deliver much better performance over others.

We also note that GCD² *for the first time* enables mobile DSP execution of two DNNs (TinyBERT and Conformer) because it supports more operators than TFLite and SNPE,

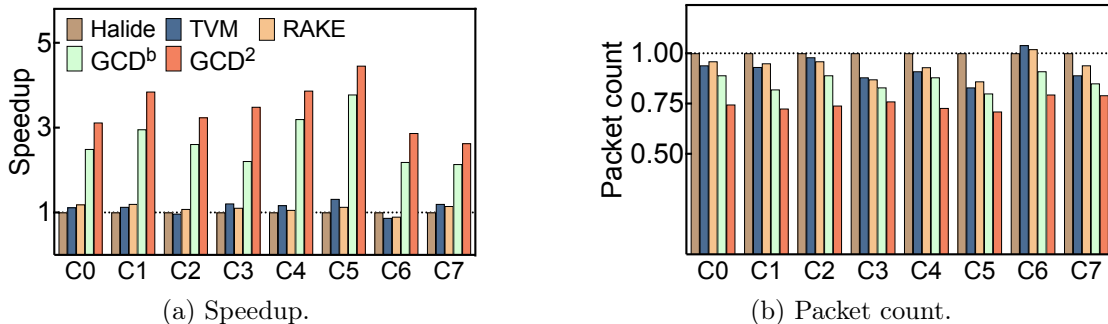


Figure 4.7: Performance Comparison of GCD^2 , Halide, TVM, and RAKE with Individual Kernels. Left shows the speedup and right shows the packet counts, both normalizing Halide as 1. Conv2D operators (from ResNet-50) are used. GCD^b is a sub-optimal version of GCD^2 that contains tensor optimizations only without VLIW packing.

e.g., more variants of `MatMul`, and `Pow`. It also *the first time* supports real-time mobile DSP execution of another (EfficientDet-d0).

Next, we compare several individual convolutional computation kernels with Halide, TVM, and RAKE. Because our native compiler optimizations (SDA VLIW instruction packing) built on LLVM can be applied to all other frameworks as well to further improve their performance, we separate tensor compiler optimizations (e.g., our data layout and instruction selection) and native/language compiler optimizations (e.g., SDA VLIW instruction packing) in this comparison by introducing a new version of GCD^2 called GCD^b . GCD^b only contains tensor compiler optimizations, and can be viewed as a more fair comparison against these three tensor compilers. In this comparison, the first 8 unique Conv2D operators in ResNet-50 are used. Figure 4.7 (a) and (b) show the speedup and the packet count for these 8 Conv2D kernels, respectively, and all results are normalized by Halide. It turns out that GCD^2 outperforms Halide, TVM, and RAKE with significant speedups due to both its layout optimizations and VLIW instruction packing. In comparing GCD^b with other tensor compilers, GCD^b achieves up to $3.8\times$, $2.7\times$, and $3.3\times$ over Halide, TVM, and RAKE due to tensor compiler optimizations like layout and instruction selection. In addition, our instruction packing algorithm results in fewer numbers of packets ($25\% <$ Halide, $19\% <$ TVM, and $21\% <$ RAKE on average, respectively). Please also refer to Section 4.5.3 for a

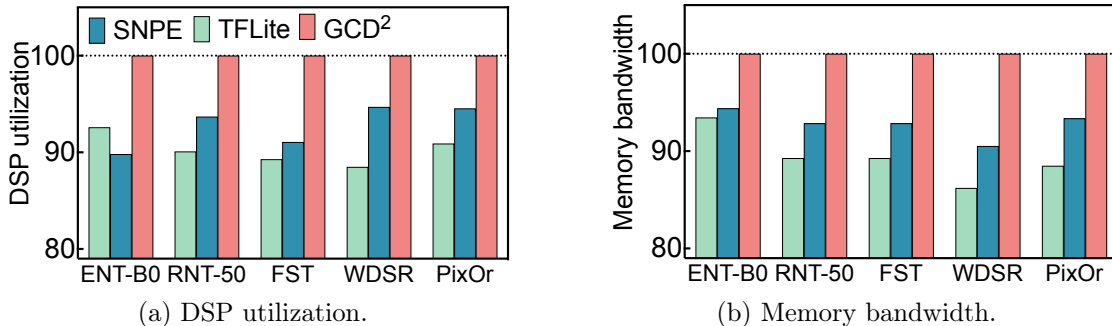


Figure 4.8: DSP Utilization and Memory Bandwidth Comparison. These results are as reported by Snapdragon Profiler [176], and normalized with GCD².

more detailed performance breakdown study.

Overall Performance Analysis. To further understand the performance difference among above frameworks, Figure 4.8 compares DSP utilization and memory bandwidth. This experiment uses 5 representative models out of 8 supported by both TFLite and SNPE, including EfficientNet-B0 (ENT-B0), ResNet-50 (RNT-50), FST, WDSR, and PixOr. Experiments on other models show similar trends and are excluded because of space limits. The data is collected from Snapdragon Profiler [176]. For DSP utilization, TFLite and SNPE can only achieve 88% to 93%, and 89% to 95% of GCD²'s utilization, respectively. For memory bandwidth, TFLite and SNPE can only utilize 86% to 93% and 90% to 94% of GCD²'s, respectively. These results show GCD² better utilizes mobile DSP's computing and memory resources with better VLIW instruction pipeline execution and higher SIMD parallelism.

It should be noted that the theoretical peak performance of Hexagon 698 reported by Qualcomm is 15 TOPS [186]. However, this number includes its Neural Processing Unit that is not publicly programmable yet. To get the peak performance of the publicly available vector processing unit (HVX), we test the highly optimized matrix multiplication kernel in the Qualcomm Hexagon SDK with small inputs that can fit into the L-1 cache, and achieve the performance of 3.7 TOPS. Our evaluation shows GCD² achieves up to 1.51 TOPS for an individual layer in DNN inference. Considering the necessary data loading and

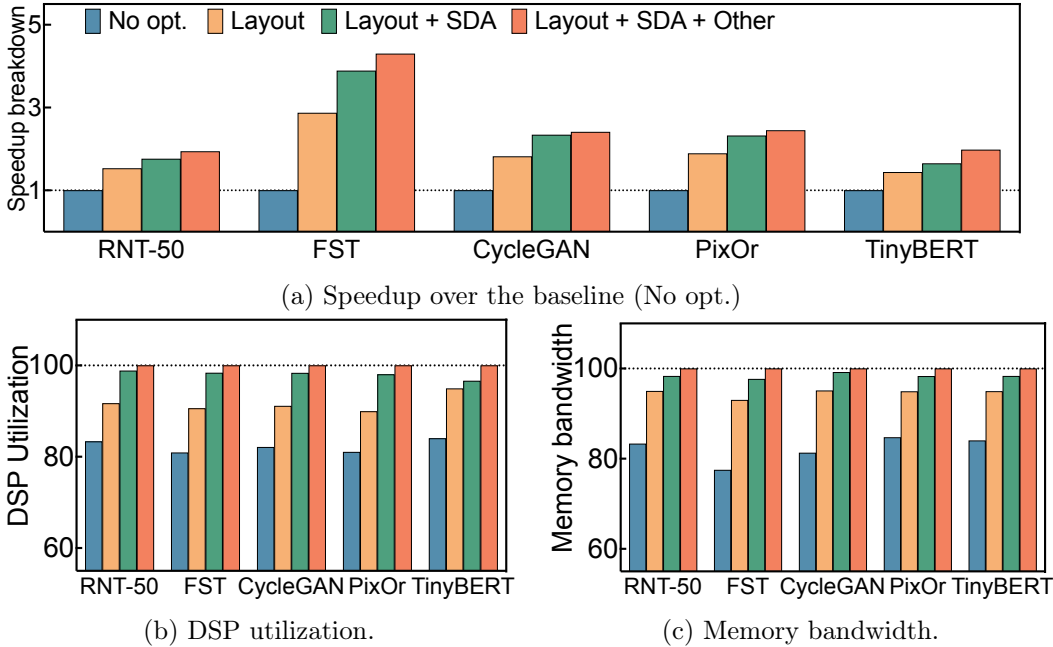


Figure 4.9: Performance Breakdown Analysis. Speedup over the baseline (normalized with the no-opt version). DSP utilization and memory bandwidth analysis (both normalized with the GCD² optimal version as 100%). The results are collected from Snapdragon Profiler [176].

memory latency costs involved, this value shows effective practical use of the hardware.

4.5.3 Impact of Opt. and Algorithmic Features

Impact of Different Optimizations. To understand how different optimizations (instruction and layout selection, VLIW packing, and other optimizations) contribute towards performance speedups, Figure 4.9 (a) studies the impact of these optimizations with 5 representative models that cover 2D CNN, GAN, and Transformer (EfficientNet-B0 (ENT-B0), ResNet-50 (RNT-50), FST, WDSR, and PixOr). We evaluate each compiler-based optimization speedup incrementally over our baseline (w/o proposed optimizations). Compared with `No opt`, instruction and layout selection brings $1.4\times$ to $2.9\times$ gains, VLIW scheduling achieves additional $1.2\times$ to $2.0\times$ speedup, and finally, other optimizations (e.g., replacing an expensive division operation with a database lookup) add $1.1\times$ to $1.4\times$ speedup. Figure 4.9 (b) and Figure 4.9 (c) further reveal that instruction and layout selection also has the

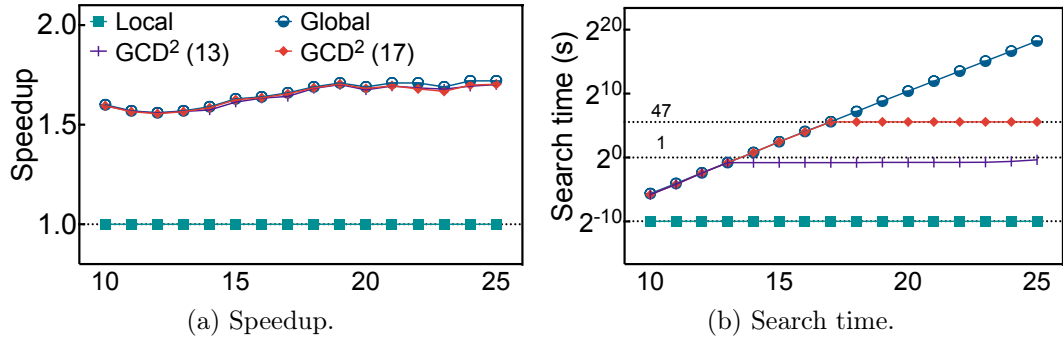


Figure 4.10: Layout Optimization Analysis. X-axis denotes the number of operators in the computational graph. The left figure shows the speedup over local optimal with different numbers of operators. The right figure shows the search time, and its y-axis is logarithmically scaled.

largest impact on DSP utilization and memory bandwidth.

Instruction (and Layout) Selection Analysis. This section justifies the choice we have made in performing global layout selection. Specifically, we compare the algorithm used in GCD² with two baselines - local optimal and exhaustive search based global optimal solutions. The local optimal solution selects the layout with the best performance independently for each operator, whereas the global optimal always conducts an (expensive) exhaustive search on the entire computational graph to find the optimal solution.

For the purpose of these experiments, partial computational graphs are extracted from ResNet-50 using contiguous operators. Figure 4.10 (a) compares the model execution performance among local optimal, global optimal, and our two versions – GCD² (13) and GCD² (17) mean the maximum number of operators within each sub-graph is 13, and 17, respectively. Compared with local optimal, GCD² brings 1.55× to 1.7× speedup, while global optimal brings 1.56× to 1.72× speedup. This validates the design choice we have made – specifically, the performance of GCD² (13) is almost identical to global optimal. At the same time, it is clear that local-only decisions impose large data transformation overheads and do not achieve good performance.

Figure 4.10 (b) compares the search time for the four solutions. Obviously, the search time in global optimal solution increases exponentially, making it impracticable even when

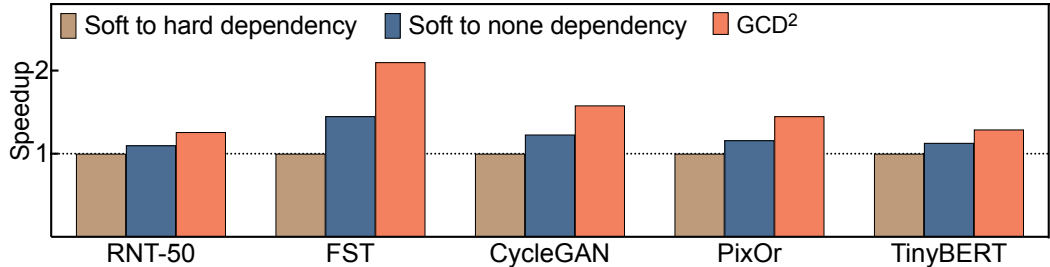


Figure 4.11: VLIW Scheduling Analysis. The version treating all soft dependencies as hard ones is used as the baseline.

there are 25 operators (complete models have more operators, see Table 4.4). The search time is over 80 hours with only 25 operators in the graph, while GCD^2 (13) and GCD^2 (17) need less than 2 seconds and 1 minute, respectively.

VLIW Packing Analysis. One of the unique aspects of our SDA VLIW instruction packing is the treatment of *soft* dependencies. We evaluate this by comparing our method against two versions: 1) all soft dependencies are treated as hard dependencies, i.e., separating all instructions with soft dependencies into different packets (**soft to hard**; 2) all soft dependencies are treated as no dependencies **soft to none** (i.e., removing lines 27, 28 in Algorithm 1 and thus not associating with penalty with packing an instruction with a soft dependency). Figure 4.11 reports the effectiveness of our optimization using 5 models and establishes our current algorithm does better than either of these choices. GCD^2 achieves up to $2.1\times$, and $1.4\times$ speedup compared with **soft to hard** and **soft to none**, respectively because of better packing efficiency as compared to **soft to hard** and fewer runtime stalls as compared to **soft to none**.

Unrolling Analysis. Figure 4.12 (a) shows the performance comparison of different unrolling strategies for a matrix multiplication kernel (three loop-levels): **Out** (only unroll the outer-most-level loop), **Mid** (only unroll the mid-level loop), and **Exhaustive** (unroll the loops by an exhaustive search). We omit the inner-most-level loop as a possibility as vectorization is performed at that level. The x-axis denotes the unrolling factor, while the speedup is normalized by no unrolling, i.e., when the unrolling factor is 1. The unrolling

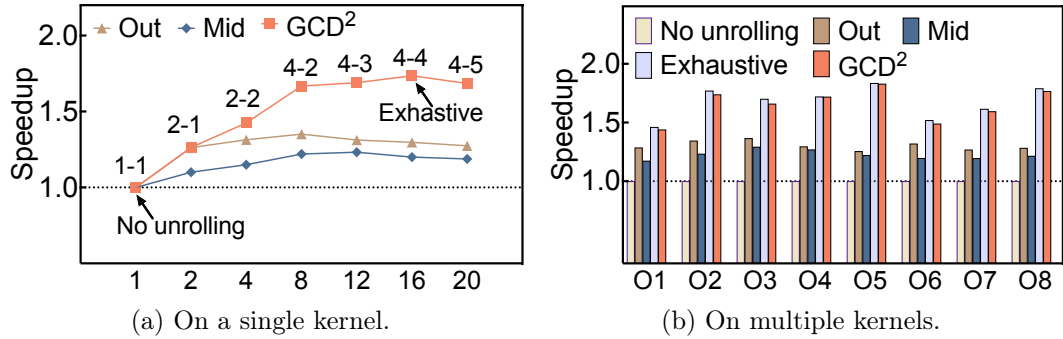


Figure 4.12: Unrolling Factor Analysis on a Single MatMul Kernel and on Multiple MatMul Kernels. The x-axis in the left figure denotes the unrolling factors. The right figure shows the performance comparison among the best settings of three unrolling strategies (Out, Mid, and GCD^2) on 8 operators (from O1 to O8). For comparison, it also shows versions w/o unrolling and w/ exhaustive search.

settings of GCD^2 for both loop levels are also labeled in this figure. The best configuration by exhaustive search is 4 – 4. GCD^2 achieves higher performance compared with the other two options. For all options, we see the expected result that the performance drops if unrolling factor is too large due to increasing register spilling. Figure 4.12 (b) compares the performance of Out, Mid, Exhaustive search, and GCD^2 under different matrix multiplication kernels – here again the y-axis is normalized by No unrolling in each kernel. Unrolling factor in No unrolling is 1, while Out and Mid both use the best unrolling factor obtained from Figure 4.12 (a). Compared with exhaustive search (Exhaustive that searches the best unroll plan for a loop structure in some common unrolling configurations), GCD^2 achieves very comparable performance while saving significant search time (exhaustive search generally takes over 3 minutes for each kernel). GCD^2 unrolling achieves much higher performance compared with the other two strategies across all kernels.

4.5.4 Power Consumption and Energy Efficiency

Figure 4.13 compares the total power consumption and energy efficiency of GCD^2 against TFLite and SNPE also executed on DSP (*-DSP) on four representative DNN models (EfficientNet-b0, ResNet-50, PixOr, and CycleGAN). As additional baseline, TFLite on a

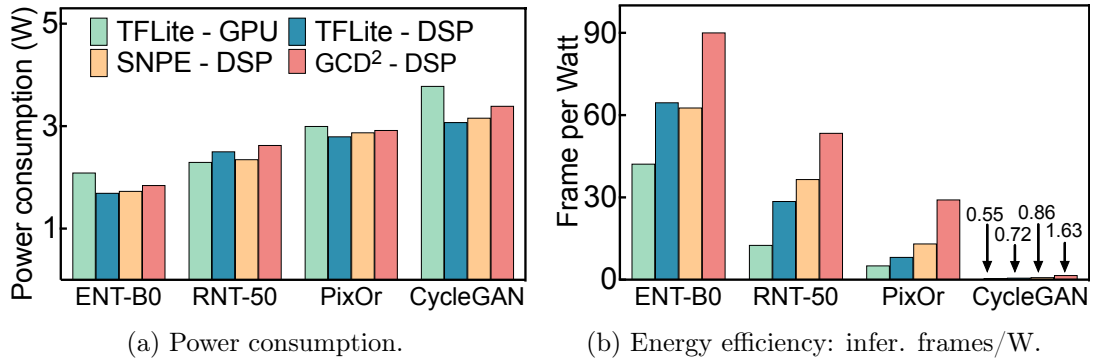


Figure 4.13: Comparison of Total Power Consumption (left) and Energy Efficiency in Inference Frames/Watt (right). Three DSP frameworks and TFLite with GPU back-end on 4 representative DNNs.

mobile GPU, Qualcomm Adreno 650 GPU on the same Snapdragon 865 SoC (TFLite-GPU) is also included. Figure 4.13 (a) shows the total power consumption of each solution, where we see that TFLite-GPU consumes the most power (ranging from 2.1 Watt to 3.8 Watt), and three DSP-based solutions consume less power. GCD²-DSP consumes less power than TFLite-GPU (by around 3.6% on average) while consuming slightly higher power than TFLite-DSP and SNPE-DSP (7.2% and 6.7% on average, respectively). GCD²-DSP consumes more power than other DSP solutions mainly because of its better DSP and memory utilization. As this results in reduced execution times, GCD²-DSP achieves much better energy efficiency as measured in inference frames per Watt – specifically improving on TFLite-DSP and SNPE-DSP by around 1.7 \times and 1.5 \times on average, respectively (Figure 4.13). Figure 4.13 also shows that all mobile DSP-based solutions result in better energy efficiency than the state-of-the-art mobile GPU-based solution, TFLite-GPU. Specifically, GCD² outperforms it by 2.9 \times in energy efficiency.

4.5.5 Comparison with Other DNN Accelerators

To better understand the inference speed and energy efficiency of mobile DSP, we also compare GCD² with two popular embedded DNN accelerator-based solutions, EdgeTPU [63] and Jetson Xavier [217] using a representative DNN (ResNet-50). EdgeTPU is a

Table 4.5: Inference Speed and Energy Efficiency Comparison with ResNet-50 on EdgeTPU [63] and NVIDIA Jetson Xavier [217]. FPS is short for frames per second, and FPW represents for inference frames per Watt.

Platform	Device	FPS	Power	FPW
EdgeTPU [63]	Edge TPU (int8)	17.8	2 W	8.9
Jetson Xavier [217]	GPU + DLA (fp16)	291	≈30 W	9.7
Jetson Xavier [217]	GPU + DLA (int8)	1100	≈30 W	36.7
GCD ²	DSP (int8)	141	2.6 W	54.2

low-power embedded platform with an edge TPU aiming to accelerate integer computations. Jetson Xavier utilizes both a GPU and DLA (deep learning accelerator), with operators not supported by DLA executed by the GPU. In this evaluation, EdgeTPU and Jetson Xavier use TFLite, and TensorRT, respectively, as their inference engine. The evaluation results are presented in Table 4.5. Jetson Xavier with int8 results in the highest FPS (frames per second) though with more power consumption. Our mobile DSP solution, GCD² achieves $6.1\times$ and $1.48\times$ better energy efficiency (FPW) with the same data type (int8) over EdgeTPU and Jetson Xavier, respectively.

4.6 Related Work

This section discusses efforts related to DNN acceleration and compilation, SIMD optimizations, VLIW instruction packing, and other compilation work targeting DSP chips.

DNN Acceleration and ML/DL Compilers. There are many recent efforts on accelerating DNN inference on edge and mobile devices including DeepX [116], TFLite [1], TVM [25], MNN [96], DeepCache [241], DeepMon [88], DeepSense [248], MCDNN [72], and MobiSR [126]. Some of them (e.g., TVM, and TFLite) rely extensively on compiler techniques, and hence are called ML or DL compilers. Most of these efforts do not target DSP, except TVM, TFLite, and MobiSR that offer options to call certain versions of Hexagon NN [182]. They do not focus on SIMD/VLIW optimizations as GCD². TASO [94] and AccPar [208] are two recent DNN acceleration efforts with some similarities to GCD². TASO’s computation-graph-level optimization is restricted to a sub-graph with a limited

number of operators, aiming to assist in their proposed effective operator substitution; while GCD² focuses on a global optimization aiming to find a data layout solution that can result in the optimized execution of the entire DNN. The partitioning problems considered by AccPar have similarities with the data layout (and instruction) selection problem GCD² considered. However, AccPar’s formulation is different and can always be solved by dynamic programming, while GCD²’s problem maps to an NP-complete problem, PBQP [7], and thus requires a different solution.

Compiling for DSP Chips. Digital Signal Processing chips have been around for several decades and there have been multiple systems developed for compiling for them [37, 138, 240, 262], including considering SIMD features [145] and exploring VLIW instructions [22, 190]. However, the DSP chip instructions set targeted in this earlier work do not have much correspondence to a modern mobile DSP chip like the one considered in this work. The techniques presented in this work are all related to advances in SIMD instruction sets and properties of VLIW instruction execution. Recently, Ahmad *et al.* [5] have reported a system that does instruction selection and code generation for the same instruction set as the one we have targeted. Their work is more general in considering arbitrary loop nests but does not address the global optimization problem. Moreover, their approach has a high compilation cost, and they report results on small kernels only – our experimental comparison shows better results for our system even on individual operators. The work from Vanhattum *et al.* [222, 223] also has similar focus (and limitations) but their target backend is different, making a direct experimental comparison infeasible. Next, Yang *et al.* have mapped a vision-related DNN to a chip that comprises several DSP processors, performing effective mapping to their vector instruction [246]. However, their work has been applied to a single model and does not include a general compiler-based optimization framework. Prior to that, another system (based on Halide system) was extended to support DSP chips [229], but this work did not emphasize data layout issues.

SIMD Optimizations. Compiler-driven code optimization and generation for SIMD [149, 159, 210] goes back several decades. Earlier work was heavily driven by the fact that Intel

SIMD extensions required operands of vector instructions to be contiguous [57, 159, 194]. More advanced techniques in this area used polyhedral models to map arbitrary loop nests for SIMD execution [111, 219] or even consider irregular applications [23]. Because of our target workloads, where there are relatively fewer options for the computations within one operator, but there can be a very long chain of operators, the challenges we address are related to global optimization, and not dealing with arbitrary loop nests. Previous work on global optimizations for SIMD [87, 152] did not consider a comparable instruction set as ours, and therefore, SIMD instruction selection and associated data layout optimizations were not their focus. Recently, Chen *et al.* have developed VeGen [27] that targets the growing diversity in available SIMD and vector instructions. The VeGen compiler extracts what they term as *lane-level parallelism* by finding the instruction most suitable for a loop (nest). This work, however, does not consider the possibility (and costs) of data transformation to use specific instructions, does not target instructions as complicated as the one we have handled, and there are no global optimizations in their work. In another recent work, a JIT compilation system was presented to use Intel SIMD advances for convolution operations [60] – this work, however, does not consider any layout or global optimizations.

VLIW Instruction Packing. VLIW instruction scheduling with timing and resource constraints is a long-standing issue, and many solutions have been proposed for various DSP architectures (that are different from modern mobile DSPs), including advanced software pipelining [38, 128, 220, 221]. Closely related to this work, Six *et al.* [207] discussed a critical path based approach based on a variant of Coffman-Graham list scheduling [36]. This approach is top-down by leveraging the heuristic that instructions with the longest latency path to the exit have priority. However, our scheduling is bottom-up by considering a heuristic of assigning higher priority to instructions that are on a critical path and can enable more instructions packing if they are packed early. More importantly, compared with all existing efforts, GCD² categorizes data dependencies and tolerates *soft* dependencies with advanced hardware support, and focuses on a more domain-specific design for DNN

accelerations on mobile DSP.

4.7 Summary

This paper has presented a compilation system, GCD², for efficiently mapping real-world complex DNN workloads on modern mobile DSP architectures. GCD² consists of three major optimizations including the development of matrix layout formats to support novel advanced SIMD instructions in the mobile DSP, a global SIMD optimization procedure that selects optimal SIMD instructions and associated layouts, and an SDA VLIW instruction packing that considers the effect of soft dependencies. GCD² is extensively evaluated with ten real-world complex DNNs on popular mobile DSPs. The results show that GCD² outperforms two cutting-edge end-to-end DNN execution frameworks supporting mobile DSPs by up to 6.0× and outperforms three established compilers that support efficient computation kernels execution on mobile DSPs by up to 4.5× because of the improved SIMD execution and optimized VLIW instruction scheduling. For certain DNNs, GCD² is unique in supporting the real-time execution of the model. For two of these ten models, GCD² implementation has, for the first, enabled execution on mobile DSPs. The overall compilation time is also justified. In the future, we plan to design and integrate a more advanced (or customized) Quantization approach [32] to GCD², and explore DSP-friendly operator fusion [157] to further improve the performance.

Chapter 5

SOD²: Statically Optimizing Dynamic Deep Neural Network Execution

5.1 Introduction

Deep Neural Networks are enabling several of the most exciting and innovative applications that are executed on a variety of computing devices, ranging from servers to edge and mobile devices. From a systems research viewpoint, this had led to a large set of ongoing projects on optimizing DNN inference (and training) tasks [1, 72, 88, 96, 116, 126, 224, 241, 248] as well as tensor compilers [110, 120, 189].

Most of the work on optimizing DNNs considers *static models* that are characterized by the following two properties: 1) input and output shapes and sizes for each layer are known a priori, and 2) the execution path is fixed, i.e., independent of the input. In *dynamic models*, in contrast, one or both of the above two properties are no longer true, and such models are now becoming prevalent. For example, Skipnet [232] decides, based on the input, whether to include or exclude certain operators (or layers). A different form of dynamism seen in transformers for NLP like BERT [46] or cutting-edge computer vision models [107, 191, 198] can take inputs with different shapes and/or apply variable portions of filter kernels during the execution. At least three factors have contributed to the popularity of dynamic models

and this trend is expected to continue: the need for adapting to computational capacities of different devices, the need for supporting different types of input (e.g. images of different resolutions), and the need for achieving high accuracy for different scenarios.

Dynamic shapes, sizes, and control flow in these models pose many challenges for the optimizations that have been key to obtaining high efficiency. For example, loop fusion [64, 146, 157, 215] cannot be applied [203, 258, 261] if we do not know that the index space of two loops (which likely is the same as the dimensions of respective input tensors) is identical. Planning the execution order [6] to reduce memory requirements or otherwise planning memory allocation [169] is, similarly, not possible if tensor sizes are not statically known.

While many of the existing systems for DNN execution can support dynamic models, they do with high overheads due to very conservative assumptions and/or expensive analyses at the runtime. For example, TFLite [1] and MNN [96] perform re-initialization (equivalent of recompilation) when the input shape to the model changes.

This paper presents the first nuanced approach for optimizing DNN inference in the presence of dynamic features. Our approach emphasizes reducing inference latency as well as memory requirements – the latter being quite important on the mobile devices we target. The foundation of our approach is an in-depth study of operators that form the basis for modern DNNs. These operators are classified into several groups on the basis of how the output shapes relate to input shapes and values. Based on such a classification, we present a data-flow analysis framework, called Rank and Dimension Propagation (RDP) that infers shapes and dimensions of intermediate tensors. RDP analysis considers known constants, symbolic constants, and expressions involving these. RDP analysis results are then used for enabling a number of optimizations, which includes operator fusion and fused code generation, static execution planning, runtime memory allocation, and multi-version code generation.

This work integrates RDP and optimizations enabled by it together and builds a comprehensive framework for optimizing Dynamic DNNs, called SOD². SOD² is extensively

evaluated on 10 cutting-edge DNN models with shape dynamism and/or control-flow dynamism. Specifically, these models include the ones for emerging Artificial General Intelligence (AGI) [62] such as StableDiffusion [198] and SegmentAnything [107]. Our evaluation results show that SOD² saves 27% to 88% memory consumption and results in 1.7× to 3.9× execution speedup compared with four state-of-the-art product-level DNN execution frameworks (such as ONNX Runtime [45], MNN [96], TVM [25] with Nimble extension [203], and TensorFlow Lite [1]) that support dynamic DNNs.

In all, this paper makes the following contributions.

- **DNN Operator Classification.** We classify the operators used for modern DNNs (specifically 150 operators used in ONNX (Open Neural Network Exchange)) into 4 categories, which are *Input Shape Determined Output*, *Input Shape Determined Output Shape*, *Input Shape & Value Determined Output Shape*, *Execution Determined Output*. We formally define these operators and explain their significance for inferring ranks and dimensions for the DNNs where the input can be of different sizes and the execution is data dependent.
- **Data-Flow Analysis for Rank and Dimension Propagation.** Building on the operator classification, we have developed a static analysis framework for propagating shape and size information through a computational graph. This framework, called RDP, considers both known and symbolic constants as well as expressions involving these values. Though somewhat similar to the well-known constant propagation analysis [20], our work is different in having transfer functions specific to the operator (types), supporting both backward and forward analyses, and considering not only known and symbolic constants but also expressions involving them.
- **Comprehensive Set of Static and Dynamic Optimizations.** Using results from RDP analysis, we enable a series of optimizations. First, we enable code fusion, including generating multiple versions when sufficient static information is not available. Next, we perform execution planning, using the results of RDP to partition

Table 5.1: Inference overhead for shape dynamism w/ execution re-initialization. SL: shape propagation and layout selection. ST: schedule and tuning. Alloc: memory allocation. Infer: inference time. Experiments are conducted on a Samsung Galaxy S21 w/ MNN [96].

Model	CPU latency (ms)				GPU latency (ms)			
	SL	ST	Alloc	Infer	SL	ST	Alloc	Infer
YOLO-V6 [131]	6.9	1,155	2.2	476	0.8	1,678	30,605	102
Conformer [68]	3.8	127	7.8	926	3	1,021	73,170	1193
CodeBERT [56]	2.3	253	2.8	370	1	856	4,568	498

the original graph, and further using several heuristics based on RDP output. Finally, we enable runtime plan generation for memory allocation and also generate multiple versions of optimized implementations for individual operators.

5.2 Existing Frameworks and Limitations

Existing DNN inference engines on mobile devices use two common approaches when handling dynamic DNNs.

Static Solutions. Many existing DNN inference engines for mobile platforms (specifically, TFLite [1] and MNN [96]) support dynamic features by extending their static model execution. For handling dynamic input shapes, this involves either execution re-initialization when the input shape changes or, alternatively, conservative (maximum) memory allocation when the input shapes are unknown. To handle dynamic control flow, it typically requires the execution of all possible paths, and stripping out invalid results. Not surprisingly, such simplistic handling of dynamic features incurs significant execution and/or memory overhead. To further illustrate, Table 5.1 shows a performance study of three models (YOLO-V6 [131], Conformer [68], and CodeBERT [56]) that can take input with dynamic shapes. MNN [96] runs these models on a Samsung Galaxy S21 with execution re-initialization to handle varied input shapes. These results show that the re-initialization usually takes even significantly longer time than the inference itself. This approach might be acceptable for cases where the overhead of re-initialization can be amortized over a number of inference tasks (e.g.,

certain video processing scenarios). However, many application scenarios (across the image, audio, and language processing) involve continuously changing inputs. An alternative way, as also indicated above, is to conservatively allocate large memory spaces. However, it incurs significant memory wastage, which can limit the ability to execute large models or to do so efficiently, especially on mobile (or edge) devices with limited memory.

Runtime Solutions. TVM (with Nimble extension) [25, 203] improves on the limitations of static solutions by providing a set of optimizations within a virtual machine. An example of this functionality is a *shape function* to infer the output tensor shape and use this information for dynamic memory allocation. However, such functions and the subsequent dynamic memory allocation introduces significant execution overhead.

5.3 Operator Classification based on Dynamism

Our observation is that DNN operators have different dynamism degrees, leading to distinct levels of challenges and opportunities in optimizing them. More specifically, this work categorizes DNN operators into four types: *Input Shape Determined Output*, *Input Shape Determined Output Shape*, *Input Shape & Value Determined Output Shape*, and *Execution Determined Output*. This section gives a formal definition.

Background and Notation. It is common to represent a DNN as a Computational Graph, which happens to be a Directed Acyclic Graph (DAG). Each tensor (which can be an input and/or output) can be categorized by a shape (including dimensions) and the contents or values. Each operator is denoted as L^l , where l is the operator index. Assume L^l has m input tensors (of which, $[1, k]$ are constant tensors while $[k, m]$ are output tensors from previous operators) and n output tensors. The shape of the input tensor i for the l^{th} operator is denoted as IS_i^l and the corresponding tensor value can be denoted as IV_i^l . Similarly, each output tensor's shape and value are OS_i^l and OV_i^l , respectively. Now, intuitively, a class of functions relates the output shapes and values to the input shapes

and values – F^{fs} for the shapes and F^v for the values.

- **Input Shape Determined Output:** The output (tensor), which is characterized by both its shape and value, has the following dependence on the input. The output tensor shapes are dependent on the input tensor shapes, whereas the output tensor values are determined by the input tensor shapes and possibly some of the constant tensors – input values do not impact the output. Examples include **Shape** and **EyeLike**. Formally, there is a pair of functions (F^{fs}, F^v) , such that:

$$OS_i^l \xleftarrow{(F^{fs})} (IS_1^l, \dots, IS_m^l)$$

$$OV_i^l \xleftarrow{(F^v)} (IS_1^l, \dots, IS_m^l), (IV_1^l, \dots, IV_{k-1}^l)$$

where $1 \leq k \leq m$.

- **Input Shape Determined Output Shape:** Similar to the previous category, the output shapes depend on the input shapes. However, what is different is that the output values rely on all the input values (including intermediate and constant input values). Examples include **Conv**, **Add**, and **Pooling**. The significance of this category, as compared to the next set of categories, is that if the input shape of this operator is known, compiler optimizations (e.g., operator fusion, execution/memory optimizations) are enabled. Formally, there is a pair of functions (F^{fs}, F^v) , such that:

$$OS_i^l \xleftarrow{(F^{fs})} (IS_1^l, \dots, IS_m^l)$$

$$OV_i^l \xleftarrow{(F^v)} (IS_1^l, \dots, IS_m^l), (IV_1^l, \dots, IV_m^l).$$

- **Input Shape & Value Determined Output Shape:** Similar to the previous category, the output values rely on the input shapes and all the input values. The difference is that the output shapes also rely on partial set of input values. Examples include **Extend** and **Range**). Formally, there is a pair of functions (F^{fs}, F^v) and a subset of input tensors (p, \dots, q) whose values specify the output shape, such that:

$$OS_i^l \xleftarrow{(F^{fs})} (IS_1^l, \dots, IS_m^l), (IV_p^l, \dots, IV_q^l)$$

$$OV_i^l \xleftarrow{(F^v)} (IS_1^l, \dots, IS_m^l), (IV_1^l, \dots, IV_m^l)$$

Table 5.2: Classification of DNN operators based on dynamism degrees. Operators are from ONNX (Open Neural Network Exchange) [160].

Operator type	Operators	Representative
Input Shape Determined Output	Shape, ConstantOfShape, Eyelike	Shape
Input Shape Determined Output Shape	Add, AveragePool, Cast, Concat, Conv, Elementwise w/ broadcast, Gather, MatMul, MaxPool, Reduce, Relu, Round, Sigmoid, Softmax	Conv, MatMul
Input Shape & Value Determined Output Shape	Expand, GroupNormalization, MaxUnpool, Onehot, Range, Reshape, Resize, Slice, TopK, Upsample	Reshape, Range
Execution Determined Output	If, Loop, NMS, Nonzero, <Switch, Combine> [†]	If, Loop

[†] <Switch, Combine> is a pair of customized operators for dynamic control flow that is not defined in ONNX.

, where $1 \leq p \leq q \leq m$. If $p \leq q \leq k$, which is identical to *Input Shape Determined Output Shape*, and all the dependent input tensors are constant. In such cases, the input shapes can be calculated without knowing other intermediate input tensors. If only the input shape of this operator is known, only partial compiler optimizations with conservative analysis can be applied to it, and full optimizations need dynamic execution results.

- **Execution Determined Output:** Similar to the previous two categories, the output values rely on the input shapes and all the input values. Examples include `Nonzero` and `If`. Formally, there is a function F^v , such that:

$$OV_i^l \xleftarrow{(F^v)} (IS_1^l, \dots, IS_m^l), (IV_1^l, \dots, IV_m^l)$$

, and the shape of i -th output tensor can only be measured after materializing its value:

$$OS_i^l \leftarrow \text{SHAPE_OF}(OV_i^l)$$

, which means it is not able to know the output shapes until materializing the output tensors (i.e., after executing the layer). Only partial optimization with conservative analysis can be applied to this operator, and full optimizations need dynamic execution results.

Although these operator types are defined according to *forward transfer*, i.e. an output tensor shape and value are related to the input tensor shape and/or value. In practice, *Backward transfer* is also used, i.e., we can (and need to) backward propagate the known output shapes (either rank or dimension or both) to the unknown input shapes. For instance, if we know the output shape of `Add`, its input dimension might be 1 or identical

to the corresponding output dimension due to broadcasting rules [44]. We define backward transfer functions as:

$$IS_i^l \xleftarrow{(F^{bs})} (OS_1^l, \dots, OS_n^l).$$

Table 5.2 shows typical operators in ONNX [160] categorized by the above classification. As further illustration, Figure 5.1 shows four sub-graphs that represent operators with different dynamism degrees (marked with red boundary) and their connections. Figure 5.1 (a) shows an *Input Shape Determined Output Shape* operator **Shape**. Once its input shape is known, its value result can be directly inferred (and in fact, this value can be propagated from **Shape** to **BiasAdd** because all following operators belong to the *Input Shape Determined Output Shape* group). Similarly, Figure 5.1 (b) implies that if the input shape to **Conv** is known, this shape information could be propagated to the entire sub-graph because all operators in this sub-graph belong to the *Input Shape Determined Output Shape* group. For the cases represented in both (a) and (b), even if the exact shape is unknown, it is still possible for us to perform compiler optimizations such as operator fusion and fused code generation, execution order optimization, and memory optimization, which will be elaborated in the next Section. In Figure 5.1 (c), the output shape of **TopK** depends on its input value (which is the left predecessor’s branch in the example), i.e., the output shape of **TopK** (and its successors) is unknown until its left predecessor branch is executed. Figure 5.1 (d) represents a sub-graph involving a dynamic control flow. **Switch** results decide if path ①, ②, and/or ③ will be taken, and **Combine** merges the results from executed paths. Both (c) and (d) require dynamic execution, thus is more difficult to optimize statically.

Discussion. Although the examples in Table 5.2 and Figure 5.1 mentioned above simply classify each operator into one category, there are additional considerations. For example, an **Upsample** operator may belong to either *Input Shape Determined Output Shape* or *Input Shape & Value Determined Output Shape* depending on whether some of the input tensors are constant or not. Therefore, with constant propagation, an operator may transform from a more dynamic classification to a less dynamic one, offering us more aggressive optimization opportunities. This has motivated certain aspects of SOD².

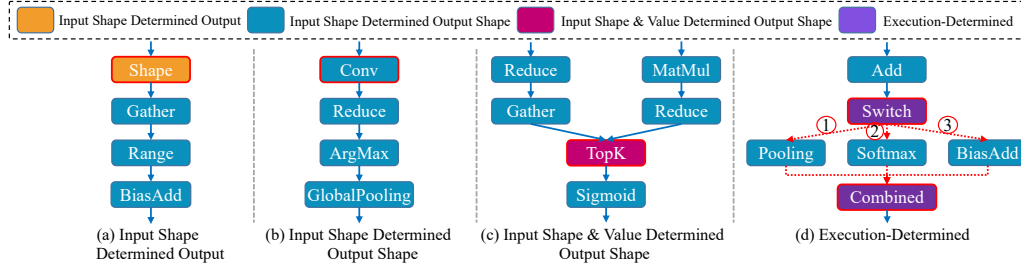


Figure 5.1: Different degrees of dynamism. Each node is a DNN operator. Yellow, blue, red, and purple mean *Input Shape Determined Output*, *Input Shape Determined Output Shape*, *Input Shape & Value Determined Output Shape*, and *Execution Determined Output*, respectively. In (d), **Switch**’s execution path is decided dynamically during runtime and red dot edges represent both the computation dependency and control flow.

5.4 Design of SOD²

Based on the DNN operator classification introduced above, SOD² introduces a new static data-flow analysis framework to infer the intermediate result tensor shape. Such an analysis is the enabler of several optimizations, which are dynamic DNN operator fusion, execution path planning, memory planning, and multi-version code generation. At a high level, our approach does not require conservative static assumptions or runtime overheads, thus providing significant improvement over the existing state-of-the-art.

5.4.1 Pre-Deployment Data-Flow Analysis

To facilitate static optimizations for dynamic DNNs, a critical requirement is knowing (possibly symbolically) the intermediate result tensor shape (i.e., rank and dimension). Our key observation is that *for many operators and operator combinations (e.g., an Input Shape Determined Output operator and an Input Shape Determined Output Shape operator), even without knowing the input tensor shape, it is still possible to infer the shape of the intermediate result tensor to a certain degree.* Our framework is based on this observation and is called operator Rank and Dimension Propagation, or RDP. While RDP has certain similarities with the classical (symbolic) constant propagation frameworks [20], it needs to deal with nuances of the DNN operations and the computational graph. RDP also considers operations over multiple (symbolic) constants as a possibility in its lattice and requires

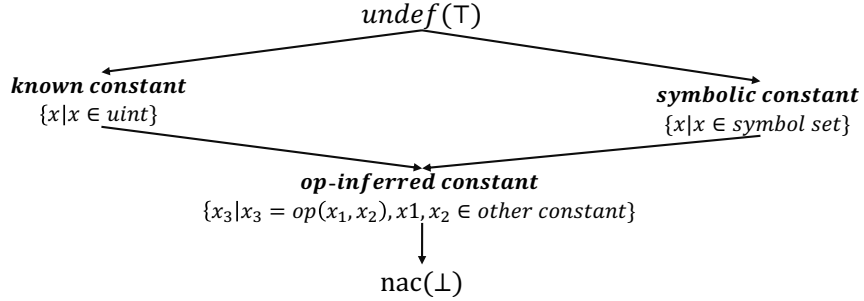


Figure 5.2: Domain of RDP dataflow analysis. It includes known, symbolic, and operation-inferred constants that form a lattice.

iterative forward and backward analysis.

Formal Definition of Operator Rank and Dimension Propagation (RDP). The entire RDP algorithm is expressed as a four-tuple $\langle G, D, L', F \rangle$.

- G is an *extended computational graph* (a DAG), with control-flow operators $\langle \text{Switch}, \text{Combine} \rangle$. If this extended computational graph involves multiple branches that all need to be executed, we assume the execution order is always from left to right. It is easy to prove that G is equivalent to a control-flow graph on operators, which serves as the foundation of this data-flow analysis.
- D is the direction of the data flow, which can be FORWARD and BACKWARD. Unlike most classic data-flow formulations, e.g., constant propagation or reaching definitions, RDP iteratively processes G in forward and backward directions until the results converge. This is because the shape of a tensor could be inferred from its producing operator and/or consuming operator, and their inference results should be the same to guarantee the correctness of this DNN execution.
- L' itself is a three-tuple $\langle V', \wedge, m \rangle$. V' is the domain of values (also shown in Figure 5.2) and includes known constants, symbolic constants, and operation-inferred constants that form a lattice. The lattice also includes undefined (*undef*) as the top (\top) of the lattice and Bottom (\perp) which is not-a-constant (*nac*). \wedge is a meet operator,

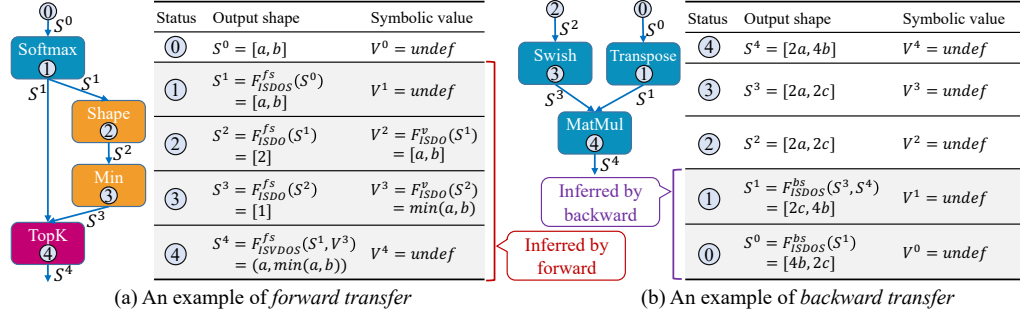


Figure 5.3: Examples of forward and backward transfer. Each node is an operator. Yellow, blue, and red mean *Input Shape Determined Output*, *Input Shape Determined Output Shape*, and *Input Shape & Value Determined Output Shape*, respectively. Ids (e.g., ①) indicate the location where transfer functions apply and their applying orders for a forward transfer (a backward transfer reverses this order). S and V equations map values in the RDP domain to the shape and value of each tensor, in which, F denotes the transfer function. *fs* and *bs* of F denote forward and backward, and F's subscript is a short form of its type (e.g., *ISDOS* means *Input Shape Determined Output Shape*).

which follows the common definition for product lattice. m is a map function mapping values in lattice to two variables, Shape (S) and Value (V), representing these for the intermediate tensor.

- $F : V' \rightarrow V'$ is the domain for transfer functions. F is designed for each operator (type) and transfers the Shape (S) and Value (V) from the input tensor to the output tensor based on the operator type. Similar to data-flow analysis for *constant propagation* RDP has two kinds of transfer functions, **Update**, and **Merge**. **Update** transfers from the input tensor to the output tensor for an individual operator; while **Merge** operates on branch control flow and merges (output) tensors from multiple possible execution paths. Because RDP has both FORWARD and BACKWARD directions, F also contains transfer functions for both directions.

Transfer Function Examples. SOD² contains 16 types of **Update** transfer functions based on the operator dynamism degree classification¹. Figure 5.3 illustrates several common ones. The left-hand side (Figure 5.3 (a)) shows an example with four forward transfers that

¹Due to space constraints, we cannot list all of our update functions here.

Table 5.3: Definition of Rank and Dimensions Propagation (RDP).

Notation	Definition	Notation	Definition
Domain	Tensor Rank and Dimensions	Direction	<i>Forward, Backward</i>
Forward	$OUT(L) = F_{P \in pred(L)}^{fs}(P)$	Backward	$IN(L) := F^{bs}(OUT(L))$
Initial	$OUT[L] = undef$	Terminate	No more changes

employ three types of **Update** transfer functions. Similarly, the right-hand side (Figure 5.3 (b)) shows an example with two backward transfer functions that belong to the same type. A point worth noting is that the appropriate transfer function to apply to an operator depends not only on the computational graph but also on the constants inferred during the RDP analysis process, which determines the dynamism classification of the operator. The **Merge** transfer function is straightforward – it merges the S-map and V-map from multiple control-flow branches based on the lattice in Figure 5.2. Table 5.3 summarizes the key components of RDP.

RDP Solution. The method is shown as Alg. 2 and involves applying the transfer functions (F) to the extended computational graph (G) along the two directions iteratively. Elaborating on Alg. 2, it first sorts the nodes (i.e., operators) in the computational graph G with the dept-first order and initializes the output shape- and value-maps of each node as *undef* (Line 1 to Line 2). It next processes each node (n) by applying forward transfer functions to n 's predecessors' output shape- and value-maps (i.e., n 's input shape- and value-maps) (Line 13). Moreover, it propagates n 's output shape- and value-maps to n 's predecessors' output shape- and value-maps by backward transfer functions if any predecessors have *undef* analysis results (Line 14 to Line 15). These forward and backward transfer functions are defined based on the dynamism classification of DNN operators (as shown in Line 20 to Line 32). Alg. 2 needs to process two specific types of nodes (operators): i) control-flow nodes (like **Combine** or **Switch**), for which, it needs to call the **Merge** function to merge analysis results from multiple control-flow paths (Line 9 to Line 10), and ii) Input Shape Determined Output nodes, for which, it assigns a symbolic constant to the value

Algorithm 2: RDP's Optimized Chaos Algorithm

```
1 foreach node in ecg.sorted_node do
2   | mark_as_undef(node) /* Initialize as undef */
3 set_model_input_shape(ecg)
4 do
5   | changed ← false
6   | /* Traverse the Depth-first sorted nodes */
7   | foreach node in ecg.sorted_node do
8     | predecessors ← predecessor_of(node)
9     | successors ← successor_of(node)
10    | if node.type is Combine then
11      | /* Merge the rank and dims for Combine */
12      | | changed |= node.merge(predecessors)
13    | if node.type is Switch or Combine then
14      | | continue /* Transit to all successors */
15    | /* 1 Forward transfer to current node */
16    | changed |= forward_transfer(node, predecessors)
17    | /* 2 Backward transfer to predecessors */
18    | foreach pred in predecessors do
19      | | changed |= backward_transfer(node, pred)
20    | /* 3 Update for Input Shape Determined Output */
21    | if node.type ∈ Input Shape Determined Output then
22      | | if node.shape ∉ (undef, nac) then
23        | | | node.value ← get_symbolic_value(node.shape)
24    | while changed
25      | Func forward_transfer: node, preds
26      | if all(node.outputs.shape ∉ undef) then
27        | | return False /* Outputs are not in undef */
28      | pred_shapes, pred_values ← shape_of(preds), value_of(preds) switch node.op_type do
29      | | case 'Input Shape Determined Output' do
30        | | | /* Only depends on the first input shape */
31        | | | return FT_ISDO(node, pred_values[0])
32      | | case 'Input Shape Determined Output Shape' do
33        | | | return FT_ISDOS(node, pred_shapes)
34      | | case 'Input Shape & Value Dependent Output Shape' do
35        | | | return FT_ISVDOS(node, pred_shapes, pred_values)
36      | | case 'Execution-Determined' do
37        | | | /* Assign nac */
38      | | return False
39    | return False
40
41    | Func backward_transfer: node, pred
42    | if all(pred.outputs.shape ∉ undef) then
43      | | return False /* Outputs are not in undef */
44    | /* Similar to forward_transfer */
```

map to facilitate subsequent analysis (Lines 16 to 18). Alg. 2 continues processing nodes in G until no updates happen on any node’s shape-/value-maps. Similarly to other data-flow analysis, RDP follows *Lattice Theory* [104], so an optimized chaos implementation (based on worklist) is guaranteed to converge.

5.4.2 Operator Fusion for Dynamic DNN based on RDP

Though fusion has been a successful optimization on DNNs [157], it is also known to be very hard to implement on dynamic DNNs [203]. A frequent issue is that without knowing the tensor shape of two operators, the DNN compiler either cannot fuse them at all or has to generate a large number of code versions, each for a possible combination of shapes for the two operators. In fact, as often more than two operators are merged, the possible combinations for which separate code should be generated increase rapidly. Our proposed RDP analysis can address this issue by using (possibly symbolic) shape information. Information such as the two operators having tensors of the same shape can enable and/or simplify fusion, even if the exact dimensions are not known till runtime.

Figure 5.4 shows a simplified example with two common DNN operators (**Sigmoid** and **Add**) on tensors with shapes not known till runtime. **Sigmoid** takes an input tensor A with a dynamic shape of $[I', J', K']$. **Add** performs an element-wise addition on **Sigmoid**’s output and another input tensor B , whose shape happens to be $[I, J, K]$. Now, if A and B are of different shapes, a shape broadcast operation on the output tensor of **Sigmoid** needs to be conducted immediately before the element-wise addition. Without our RDP analysis, the dynamic shape of A and B (and the possible shape broadcast operation) prevents the DNN compiler from fusing these two operators in an efficient way, i.e., the compiler either generates code without fusion (as shown in the blue box of Figure 5.4), or generates multiple code versions (8 versions for this example) and selects a version during the runtime. Assuming our RDP analysis result is $I' = I$, $J' = 1$, and $K' = 1$, i.e., a mix of symbolic constant (I) and known constant, the DNN compiler can further generate a unique version of fused code (as shown in the green box of Figure 5.4). SOD² incorporates RDP and the

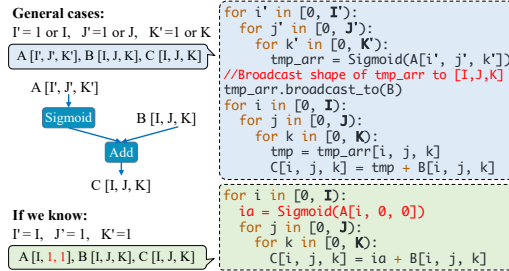


Figure 5.4: Operator fusion with dynamic shapes. The top code snippet shows that fusion is not feasible because of broadcasting [44]. Specifically, *Add* requires *A*’s indices I' , J' , and K' to be either 1 or I , J , and K , resulting in 8 fusion scenarios. With RDP, such fusion is feasible (shown in the below code snippet). This fusion significantly reduces intermediate result materialization requirements.

above operator fusion based on RDP into a state-of-the-art operator fusion for static DNNs (DNNFusion [157]) to generate the fusion plan and optimized fused code.

5.4.3 Static Execution Planning based on RDP

A computational graph (DAG) typically allows for several different orderings for the execution of operators. The choice of ordering has an impact on the peak memory usage (for intermediate results), which further has consequences for cache performance and the execution latency. There has been previous work on this problem, which has in fact shown that generating an *optimal* execution plan (by a metric like memory consumption) is an NP-complete problem [6]. Thus, choosing an optimal plan can be difficult for modern large DNNs with hundreds or even thousands of operators.

The dynamic properties (e.g., dynamic shapes and control flow) further complicate this problem. In SOD², we develop a series of heuristics driven by the use of proposed RDP analysis. The overall idea is that since a globally optimal solution is almost infeasible, an approach based on *graph partitioning* is justified. It turns out that the results of RDP are able to guide both graph partitioning and choice of solution within each sub-graph. Particularly, we observe that known constants, symbolic constants, op-inferred constants, and \perp or *nac* progressively increase the impediment on the generation of an optimal

execution plan. More specifically, for a sub-graph sg with a limited number of operators:

First, if the shape of all tensors in sg are known constants, the optimal execution plan for sg can be obtained statically by an exhaustive search – a limited size of sg can further make such a search feasible. Second, if the shape of tensors in sg are mixed known constants, symbolic constants, and op-inferred constants, it is still possible to compare the memory requirements and thus generate a (close to) optimal execution plan. This is especially true if these shapes are derived from the same set of symbolic constants. Third, if an operator has an *nac* output tensor shape, it disables further analysis and execution planning. Such operators, it turns out, provide an opportunity to partition the original graph into sub-graphs that can be independently analyzed.

5.4.4 Other Optimizations

5.4.4.1 Memory Allocation Plan

Besides execution (order) planning, *memory planning* of DNNs is also a critical step [6, 169]. A *memory allocation plan*, which decides where in a linear memory space each intermediate tensor is allocated, and when it is deallocated, can restrict peak memory usage and improve execution performance – the latter by reducing memory fragmentation, avoiding memory movement, and limiting memory allocation/de-allocation. In contrast to execution planning that (even for dynamic DNNs) can be carried out at compilation time, memory planning for dynamic DNNs can only be performed at execution time when all tensor sizes are known. Memory planning of static DNN execution has also been proved NP-complete [6], while DNN model dynamism further complicates memory planning.

Existing memory planning methods for dynamic DNN execution (e.g., Nimble [203]) have addressed this. Without knowing the exact tensor shapes, the methods usually rely on a greedy strategy [169], (e.g., finding the minimal memory slot currently available that can hold the new tensor). In comparison, we use RDP results and the following two key insights. First, we base our approach on sub-graphs generated by our static execution

planning method. It turns out that for sub-graphs with known constant shapes, as well as those with symbolic/op-inferred constant shapes that are defined solely by the input tensor of the sub-graph, the peak memory requirement can be inferred from static RDP analysis results and subsequent execution plan generation.

Second, we have observed that for most sub-graphs, the memory requirement decreases monotonically in both forward and backward directions from the location in the graph with peak memory usage. Therefore, initiating memory planning from the peak memory consumption location and traversing in the forward and backward directions, and picking the available memory slots for reuse works as a good strategy, and does not lead to the need for extra memory space.

Based on these insights, a lightweight greedy approach that starts from the peak memory requirement location can help to find optimal memory usage for many/most sub-graphs. Our evaluation (details omitted because of space limits) on ConvNet-AIG [225] shows that our RDP-based memory allocation plan requires $1.05\times$ of optimal peak memory consumption (that results from an exhaustive search); while the one based on the greedy strategy mentioned above (MNN) requires $1.16\times$ of optimal peak memory consumption.

5.4.4.2 RDP-based Multi-Version Code Generation

As we discussed in Section 5.4.2, RDP analysis enables and/or simplifies operator fusion by revealing (possibly symbolically) tensor shapes. In cases where a single (fused) version is not feasible, one of the advantages of the information obtained through RDP is that the number of different versions of the fused code generated can be reduced significantly.

SOD² further benefits from this property of RDP by generating *multi-version code* to optimize *hotspot operators* (e.g., CONV and GEMM) that dominate the DNN execution. Prior efforts [1, 96] have shown that the optimization opportunities for these operators depend on the shapes and sizes of the input/output tensors. Therefore, for static DNN executions, existing frameworks (such as TensorFlow Lite [1] and MNN [96]) usually employ multi-version codes that involve different optimizations (e.g., tiling, unrolling, choice of

the number of thread blocks, etc.). However, this optimization is challenging for dynamic DNNs because an unknown tensor shape and/or tensor size implies that too many versions will be needed. The tensor shape (or shape relations) provided by RDP help to generate code for more specific tensor shapes only, thus resulting in fewer code versions.

5.5 Evaluation

SOD² is implemented by extending an existing DNN execution framework (DNNFusion [157]) that supports static DNN execution only. This section evaluates the performance of SOD² by comparing it with four state-of-the-art frameworks. These frameworks are ONNX Runtime (ORT) [45] (V1.14.1), MNN [96] (Vdcb080c), TVM [25] w/ Nimble extension (TVM-N) [203] (V7831a79), and TFLite [1] (V2.11.1). ORT, MNN, and TVM-N support shape dynamism, while for DNNs with control flow, they execute all possible branches and strip out invalid ones. For fairness, this section also shows a performance comparison between SOD² and MNN by disabling SOD²'s *<Combine, Switch>* control-flow support and adopting the same “execute-all, strip-out-invalid” strategy. TFLite supports dynamic input shapes with memory re-initialization; however, it cannot run most of our dynamic models properly because it usually fails on some input shapes. It does not support dynamic control flow either as required by most of the models we target. Thus, we use TFLite as a baseline for comparing DNN executions with fixed inputs and paths only.

Our evaluation has four objectives: 1) demonstrating that SOD² outperforms other frameworks with respect to both memory requirements and execution latency (Section 5.5.2), 2) studying the performance effect of our key optimizations based on RDP (Section 5.5.3), 3) further confirming the performance advantage of SOD² by evaluating it under different situations (Section 5.5.4), and 4) showing that SOD² performs well on different mobile platforms (i.e., SOD² has good portability).

5.5.1 Evaluation Setup

Dynamic Models and Datasets. Our evaluation is conducted on three types of dynamic models: 1) models with shape dynamism, 2) models with control-flow dynamism, and 3) models with both shape and control-flow dynamism. The first category comprises five cutting-edge DNN models, which are StableDiffusion [198] (covering the Encoder part, referred to as SDE), SegmentAnything [107], Conformer [68], CodeBERT [56], and YOLO-V6 [131] (referred to as YL-V6). The second category includes DGNet [130]. The third category consists of four models, including SkipNet [232] (referred to as SNet), ConvNet-AIG [225] (referred to as CNet), RaNet [247], and BlockDrop [237] (referred to as BDrop).

Table 5.4 characterizes these models by showing the nature of dynamism, target input types, model size, and the total number of layers. Because the choice of training datasets has a negligible impact on the final inference latency or memory consumption (since the model size and structure are the same), this section reports results from one training dataset for each model. StableDiffusion-Encoder, SkipNet, DGNet, ConvNet-AIG, RaNet, and BlockDrop are trained on ImageNet dataset [41]; YOLO-V6 is trained on MS COCO dataset [137]; SegmentAnything is trained on SA-1B dataset [107]; CodeBERT is pre-trained on [43]; and finally, Conformer is trained on Librispeech dataset [162]. Since the model accuracy is the same across all frameworks, our evaluation focuses only on execution time and memory consumption.

Test Samples and Setup. Our inference performance evaluation randomly selects 50 input samples from the corresponding validation dataset for each model. Specifically, for models that take images as input, i.e., YOLO-V6, SkipNet, ConvNet-AIG, RaNet, and BlockDrop, our evaluation randomly selects 50 input images from the ImageNet dataset, with the size of dimensions ranging from 224 to 640. DGNet does not support dynamic input shapes, but it does support dynamic control flow. Therefore, we only tested images with a dimension of 224 for DGNet. As YOLO-V6 only accepts images with dimensions

Table 5.4: Memory consumption (allocated for intermediate results) for ONNX Runtime, MNN, TVM with Nimble extension (TVM-N), and SOD² on a mobile CPU. “-” means this model is not supported by the framework yet. “S” stands for shape dynamism, and “C” represents for control-flow dynamism.

Model	#Layers	Model Size (MB)	Dynamism	Input Type	ORT (MB)		MNN (MB)		TVM-N(MB)		SOD ² (MB)	
					Min	Max	Min	Max	Min	Max	Min	Max
StableDiffusion [198]	407	137	S	Text + Image	186	342	124	376	-	-	92	271
SegmentAnything [107]	857	17	S	Text + Image	-	-	-	-	-	-	16	22
Conformer [68]	1,703	303	S	Audio	-	-	61	78	-	-	49	58
CodeBERT [56]	985	502	S	Text	32	75	25	54	-	-	21	41
YOLO-V6 [131]	599	239	S	Image	288	430	148	404	964	1,103	89	206
SkipNet [232]	549	103	S + C	Image	168	597	27	124	522	700	18	86
DGNet [130]	847	91	C	Image	37	37	76	76	-	-	23	29
ConvNet-AIG [225]	282	104	S + C	Image	168	423	33	109	557	646	26	77
RaNet [247]	2,617	525	S + C	Image	675	1275	166	675	-	-	86	452
BlockDrop [237]	439	179	S + C	Image	242	460	35	105	523	723	24	69
Geo-mean memory consumption normalized by SOD^{2*}					3.64×		1.37×		8.62×		1	

* This normalized geo-mean memory consumption is calculated by 1) averaging the memory usage of runs with all input samples for each model, 2) calculating the geo-mean of the average memory usage of all models, and 3) normalizing with SOD²'s geo-mean memory usage.

that are multiples of 32, only a subset of inputs could be used. For StableDiffusion-Encoder and SegmentAnything, the 50 randomly selected input images have dimensions ranging from 64 to 224. For CodeBERT and Conformer, our evaluation randomly selects 50 input samples with sequential lengths ranging from 32 to 384.

The experiments are performed on a Samsung Galaxy S21 smartphone powered by a Snapdragon 888 processor [185]. This processor features an octa-core Kryo 680 CPU, comprising one large core, three middle cores, and four small cores, and a Qualcomm Adreno 660 GPU with 1024 ALUs. Additionally, to demonstrate the portability of our approach, SOD² is also tested on an earlier generation of Snapdragon platform with more constrained resources, specifically the Snapdragon 835 [175] equipped with a Qualcomm Kryo 280 octa-core CPU, consisting of four middle cores and four small cores, and a Qualcomm Adreno 540 GPU with 384 ALUs. Our evaluation employs 8 threads on mobile CPUs and pipelined execution on mobile GPUs. The GPU execution uses a 16-bit floating-point representation, while the CPU execution uses a 32-bit floating-point representation. Each experiment is executed 50 times and only the average numbers are reported – as the variance was negligible, it is not reported for readability.

Table 5.5: End-to-end execution latency comparison among ONNX Runtime, MNN, TVM-N, and SOD² on mobile CPU and mobile GPU. “-” means this model is not supported by the framework yet.

Model	ORT (ms)				MNN (ms)				TVM-N (ms)				SOD ² (ms)			
	CPU		GPU		CPU		GPU		CPU		GPU		CPU		GPU	
	Min	Max	Min	Max	Min	Max	Min	Max	Min	Max	Min	Max	Min	Max	Min	Max
StableDiffusion [198]	179	2,115	217	2,076	189	1,287	159	1,252	-	-	-	-	152	733	105	530
SegmentAnything [107]	-	-	-	-	-	-	-	-	-	-	-	-	66	108	42	71
Conformer [68]	-	-	-	-	51	300	265	498	-	-	-	-	40	225	35	150
CodeBERT [56]	141	752	-	-	125	1,265	-	-	-	-	-	-	102	452	72	366
YOLO-V6 [131]	174	1,386	155	733	168	925	47	287	251	2,108	-	-	118	546	33	178
SkipNet [232]	111	841	-	-	92	789	116	363	109	974	-	-	41	633	29	253
DGNet [130]	122	122	127	127	67	67	211	211	-	-	-	-	32	56	23	42
ConvNet-AIG [225]	90	693	-	-	88	731	96	305	98	947	-	-	46	526	22	203
RaNet [247]	102	641	-	-	139	663	114	208	-	-	-	-	63	403	31	150
BlockDrop [237]	153	1,199	-	-	145	1,421	139	468	185	1,622	-	-	79	668	42	295
Geo-mean latency*	2.5×		3.9×		1.7×		2.3×		2.7×		-		1		1	

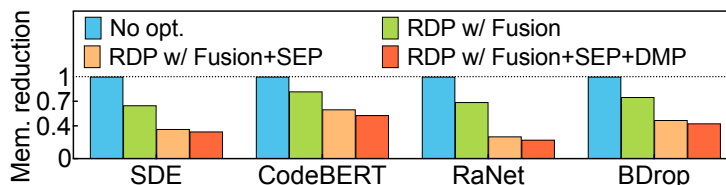
* This normalized geo-mean execution latency is calculated by 1) averaging the execution latency of runs with all input samples for each model, 2) calculating the geo-mean of the average execution latency of all models, and 3) normalizing with SOD²'s geo-mean execution latency.

5.5.2 Overall Comparison

This section focuses on the end-to-end memory reduction and execution latency gains of SOD².

Overall Memory Consumption Comparison. Table 5.5 presents a comparison of end-to-end memory consumption on a mobile CPU using SOD², ONNX Runtime (ORT), MNN, and TVM with Nimble extension (TVM-N). As the results on mobile GPU show a similar trend, they are not included here. ‘-’ implies that a model is not supported by a given framework. The ‘Min’ and ‘Max’ columns indicate the minimum and maximum memory consumption (excluding the memory for holding the model itself because this part is the same for all frameworks). The last row of the table shows the geometric mean memory consumption of each framework normalized by SOD². Its detailed calculation method is shown below the table and is over the cases where execution is possible. Among other frameworks, only MNN can support Conformer. SegmentAnything is not supported by other frameworks as either certain key operators are missing, and/or there are limitations in optimization, leading to large model execution footprints.

Compared with other frameworks, SOD² has significantly lower memory consumption.



(a) Mobile CPU

Figure 5.5: Memory reduction of different optimizations on CPU. Over the baseline w/o any RDP-enabled optimization (No opt.)

Specifically, ORT, MNN, and TVM-N need to use $3.64\times$, $1.37\times$, and $8.62\times$ memory, respectively, over SOD². SOD² results in a greater reduction in memory consumption for image models (compared to other models) because image models generally have larger memory footprints, allowing for more significant optimization opportunities. It is worth noticing that TVM-N executes models as its own Android RPC application, which is one of the causes of higher memory requirements.

Overall Latency Comparison. Table 5.5 presents a comparison of end-to-end latency for SOD² against other frameworks on both mobile CPU and GPU. The table includes the minimum and maximum latency observed across different input samples for each model. On mobile CPU, SOD² achieves an average speedup of $2.5\times$, $1.7\times$, and $2.7\times$ compared to ONNX Runtime, MNN, and TVM-N, respectively. TVM-N does not support dynamic models on a mobile GPU. Compared against the other two frameworks on mobile GPU, SOD² achieves a speedup of $3.9\times$ and $2.3\times$ over ORT and MNN, respectively. Notably, the minimum latency achieved by SOD² on mobile GPU is significantly lower than other frameworks for ConvNet-AIG, RaNet, and BlockDrop models. This is because our optimizations can handle different cases and mitigate the effect of execution path variations.

5.5.3 Optimization Breakdown Analysis

This section studies the individual impact of the key optimizations in SOD² on both memory consumption and latency.

Memory Reduction w/ Different Optimizations. Figure 5.5 evaluates the memory

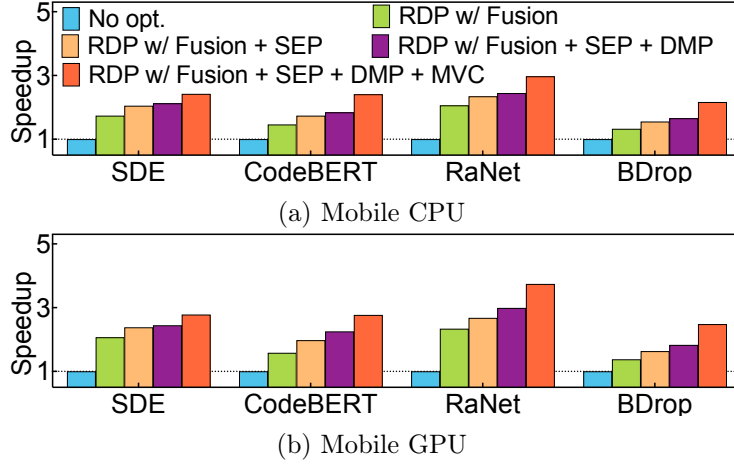


Figure 5.6: Execution speedup of different opt. on CPU and GPU. Over the baseline w/o any RDP-enabled optimization (No opt.)

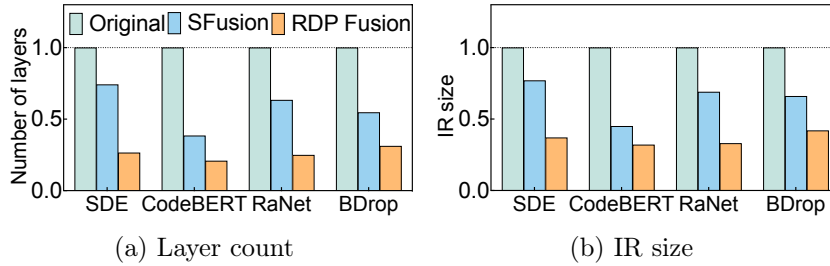


Figure 5.7: Further break down effect of existing static fusion (SFusion) and RDP-based fusion (RDP Fusion). For both layer count and intermediate result size, normalized by no fusion opt.

reduction achieved through different optimizations for 4 models (StableDiffusion-Encoder, CodeBERT, RaNet, and BlockDrop), including RDP-enabled operator fusion (**Fusion**), static execution planning (**SEP**), and dynamic memory planning (**DMP**). The results for other models exhibit a similar trend and are excluded due to space limitations. The baseline version is referred to as **No opt** – despite the name, it includes general static optimizations, such as static operator fusion and constant folding. Building on this version, we study the benefits of optimizations enabled by RDP analysis. On mobile CPU, operator fusion, static execution planning, and dynamic memory planning bring 18% to 30%, an extra 22% to 37%, and another extra 3% to 7% memory reduction, respectively. Multi-version code generation (**MVC**) is primarily designed for latency improvement, its impact on memory

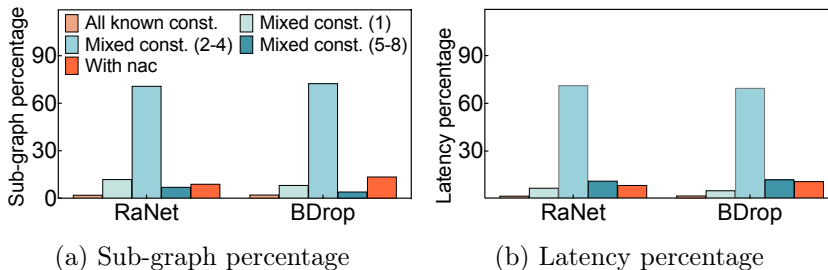


Figure 5.8: The percentage of different types of sub-graph.

reduction is negligible. The memory reduction on mobile GPU is omitted because our optimizations are general to both CPU and GPU, and the results are similar.

Latency Reduction w/ Different Optimizations. Figure 5.6 presents the speedup breakdown of our key optimizations on the same 4 models. On mobile CPU, our RDP-based operator fusion yields $1.3\times$ to $1.9\times$ speedup compared to `No opt.` Additionally, static execution planning provides $1.1\times$ to $1.3\times$ speedup, and dynamic memory planning gains $1.04\times$ to $1.1\times$ speedup, and Multi-version code generation brings an extra $1.3\times$ to $1.6\times$ speedup. On mobile GPU, these numbers are $1.4\times$ to $2.3\times$, $1.2\times$ to $1.3\times$, $1.06\times$ to $1.2\times$, and $1.4\times$ to $1.7\times$, respectively. Our optimizations provide more benefits for mobile GPU since GPU is more sensitive to memory and data movement and supports a higher degree of parallelism. We further study each optimization with more profiling results.

RDP-enabled Operator Fusion. Figure 5.7 further breaks down the effect of existing operator fusion for static DNNs only (`SFusion`) and our RDP-enabled operator fusion (`RDP Fusion`) on these four dynamic DNNs. These results are normalized by the original DNN without fusion (`Original`). `SFusion` reduces the layer counts by 26% to 61%; while `RDP Fusion` further reduces the layer counts by 16% to 46% additionally by leveraging RDP analysis results. In terms of intermediate result (IR) size, `RDP Fusion` saves an additional 13% to 40% on top of `SFusion`.

Subgraph Data. To better understand execution and memory planning, this part studies how many sub-graphs can benefit from RDP analysis results. Figure 5.8 (a) shows the percentage of different sub-graphs, i.e. those with all known constant shapes, with mixed

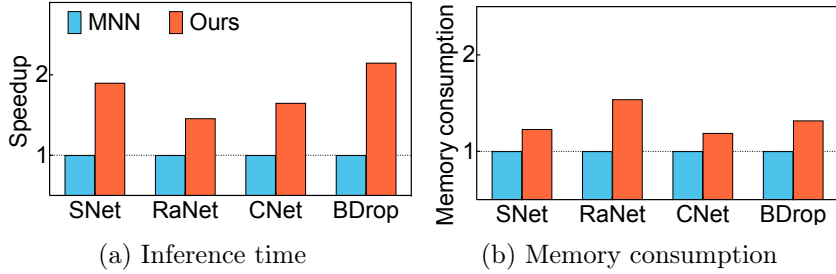


Figure 5.9: Latency and memory consumption comparison between SOD² and MNN with the same execution path.

constant shapes, and with statically unknown (*nac*) only for 2 representative models. The numbers (1, 2-4, and 5-8) after `Mixed const` denote the number of code versions that are required to optimize this sub-graph (the lower the better). This result shows that over 90% of the sub-graphs belong to all known constant or mixed constant categories whose execution plan and memory plan can be optimized by our framework. To further confirm this, Figure 5.8 (b) shows the latency percentage of each kind of sub-graphs.

5.5.4 Further Performance Analysis

This section further studies SOD² under different cases.

Latency Comparison with the Same Execution Path. To provide an apple-to-apple comparison for control-flow dynamism, this test disables the control-flow logic in 4 models (SkipNet, RaNet, ConvNet-AIG, and BlockDrop) that have control-flow dynamism. Our execution included all paths, including all branches in the `<Switch, Combine>` pairs. Figure 5.9 illustrates the performance comparison with MNN because MNN performs the best among all baseline frameworks we compared. SOD² achieves 1.5× to 2.0× speedup and 1.2× to 1.5× memory reduction on the mobile CPU. This result further validates the effect of our RDP analysis and fusion, execution, and memory optimizations based on it even without the dynamic branch selection capability of SOD².

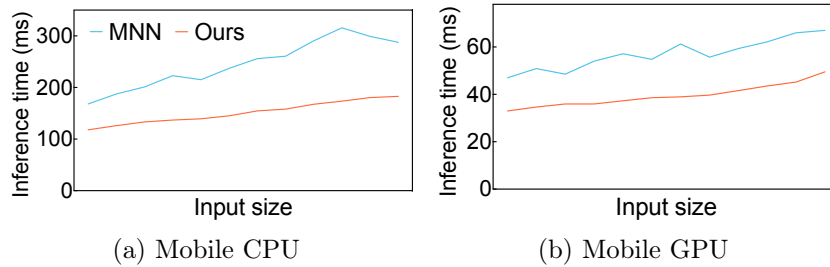


Figure 5.10: Performance variation with different input sizes (shapes). The data is collected from YOLO-V6. A larger input size means more computations.

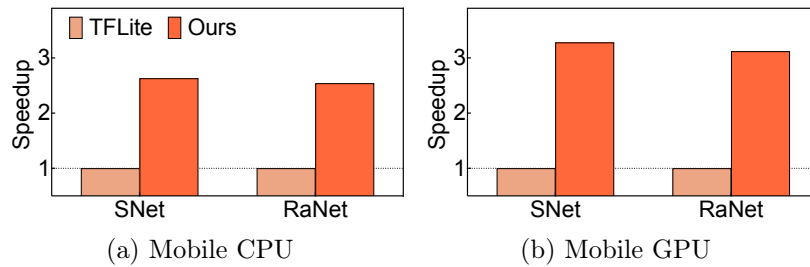


Figure 5.11: Speedup with the same memory consumption.

Latency Comparison with Different Input Sizes. To demonstrate the stability of SOD², this test randomly selects 15 input shapes for YOLO-V6, and Figure 5.10 shows their inference latency with MNN and SOD². These results demonstrate that SOD² outperforms MNN in terms of both latency and stability across increasing input sizes on mobile CPUs and GPUs. Specifically, SOD² exhibits lower and more consistent latency, while MNN exhibits significant variations.

Latency with Fixed Memory Budget. Figure 5.11 presents a latency comparison between SOD² and TFLite with the same memory budget. Specifically, TFLite fixes its memory consumption to match SOD²'s, and uses the XLA rematerialization policy [64] to handle the out-of-memory cases. SOD² outperforms TFLite by an even greater margin. Additionally, SOD² demonstrates a higher speedup on mobile GPU compared to mobile CPU due to the longer time required for mobile GPU to materialize intermediate tensors from its cache into main memory because of memory mapping.

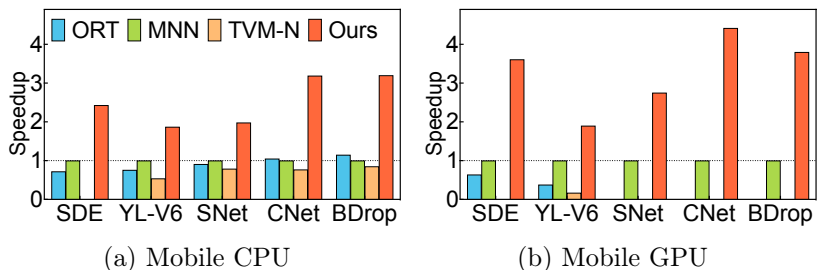


Figure 5.12: Portability evaluation. The results are collected on Snapdragon 835. An empty bar means the model is not supported by the framework. Results are normalized by MNN for readability.

Portability Evaluation To further investigate the effectiveness of portability, Figure 5.12 shows the execution speedup of SOD² over other frameworks on another mobile device – Snapdragon 835, and 5 models (StableDiffusion-Encoder, YOLO-V6, SkipNet, ConvNet-AIG, and BlockDrop). SOD² achieves similar speedup trends, and interestingly, it achieves higher speedups on this earlier generation of SoC because this SoC has more restricted resources (e.g., cache size and memory throughput). The RDP-based optimizations employed in SOD² significantly reduce memory requirements, leading to improved performance on these platforms.

5.6 Related Work

Dynamic Neural Network Optimizations. Type analysis and type inference [33, 75, 120, 153, 205] are widely used to analyze tensor shapes, thus assisting in Dynamic Neural Network optimizations. Nimble [203], which has been integrated into TVM, is a compilation-based Dynamic DNN framework. This framework relies on expensive dynamic functions to interpret dynamic shapes at the runtime. This implementation, which we have extensively compared against, limits the opportunities for optimized code generation, such as performing operator fusion. DISC [261] extends MLIR-HLO [120] and propagates the shape information for operators that have certain constraints, e.g. same dimensions (the case of `Activation`) and same size (the case of `Transpose`). SOD² provides a more

comprehensive operator classification based on dynamism degrees, bringing in significantly enhanced optimization opportunities. Axon [28] is a programming language that allows specification of symbolic shapes for input and output tensors for computational graphs. It uses a constraint solver to find shapes whereas SOD² uses a forward and backward data-flow analysis (RDP), which also alleviates additional programmer involvement. In addition, SOD² includes a set of opts enabled by RDP.

Less closely related to SOD², DietCode [258] proposes an auto-scheduler framework based on TVM for dynamic shapes. The framework builds a cost model to predict runtime performance and reduces the search space to find optimal runtime parameters (e.g., loop tiling). Cortex [54], Cavs [242], and another effort [93] mainly aim to address recursive dynamism of neural networks, different from SOD²'s focus. Other efforts focus on dynamic batching for inference [52, 58, 144, 253] or are designed for dynamic DNN training [155].

DNN Execution and Memory Optimizations. Several studies exist for operator execution order scheduling, such as [6, 133, 135]. Among these efforts [133, 135] focus on minimizing peak memory consumption by reordering operators for resource-constrained devices (e.g., MCUs), and effort [6] proposes an optimized scheduling framework for complex models (irregularly wired neural networks). These approaches rely on static shapes only. There have also been recent efforts on optimizing memory allocation planning and memory management for DNNs. Works such as [129, 169] have designed various heuristic memory planning algorithms for static DNNs only. TelaMalloc [148] performs memory management on the fly for static control-flow graphs with known intermediate tensor shapes and sizes. It does not fully consider the DNN control-flow dynamism and dynamic shapes. A possible future work can be to integrate our RDP analysis and TelaMalloc's combination of heuristics with a solver-based approach to further improve our memory planning. When the available memory is limited, rematerialization [92, 108] and recomputation [19] methods achieve a trade-off between memory consumption and execution latency. These aspects can be considered for dynamic DNNs in the future.

DNN Inference Engines on Mobile. Support for DNN inference on mobile devices has

become an area of active research in recent years. Efforts such as MCDNN [72], DeepX [116], DeepMon [88], DeepSense [248], and DeepCache [241] have primarily concentrated on optimizing the execution of static DNNs with static shapes and control flow. TensorFlow Lite (TFLite) [1], Pytorch-Mobile [166], TVM [25], and MNN [96] provide support for dynamic shapes relying on reinitialization or conservative (maximum) memory allocation. They either do not support dynamic control flow or require executions of all paths with a stripping of invalid results. As shown in our evaluation, these methods introduce high runtime overhead. One of the previous systems for static DNNs, DNNFusion [157], also involved a classification of DNN operators, however, the classification introduced here is orthogonal.

5.7 Summary

This paper has presented a comprehensive framework, SOD², for optimizing DNNs. SOD² classifies common operators of Dynamic DNNs into four types, and comprises a novel static dataflow analysis (RDP). This is followed by a set of optimizations enabled by RDP for Dynamic DNNs, including operator fusion, static execution (order) planning, dynamic memory allocation planning, and multi-version code generation. SOD² is extensively evaluated on a mobile system with 10 emerging dynamic DNNs and the evaluation results show that it saves up to 88% memory consumption and brings up to 3.9× execution speedup over four state-of-the-art DNN execution frameworks. As the underlying techniques are general and applicable to other devices as well, our future work will evaluate SOD²'s efficacy on other devices (e.g., edge GPUs and Raspberry Pi).

Chapter 6

Conclusion and Future Plan

6.1 Conclusion

This dissertation proposes four novel compiler-based frameworks to achieve real-time DNN execution on mobile devices, including mobile CPU, GPU, and DSP. In the first work, we present the sparse neural network compilation system tailored for an innovative hardware-friendly pruning that, for the first time, achieves real-time DNN execution for large-scale models on smartphones. With the increasing number of layers in recent neural networks, we design an advanced flexible operator fusion strategy with our dedicated compiler support, which significantly reduces the data movement among the successive layers. In the third work, we carefully investigate the soft dependencies tolerated by the mobile DSP and propose a heuristic VLIW packing strategy and global instruction selections for DSP architecture. Lastly, we propose a novel dynamic neural network compilation system that can efficiently execute dynamic neural networks on mobile devices.

6.2 Future Plan

In continuation of my existing research on real-time machine learning systems, I will explore a broader scope of embedded platforms and related areas. In particular, I would like to

investigate the following research opportunities:

The opportunity for emerging parallel hardware: Today, a few mobile chip vendors, such as edge TPU and Kirin NPU, offer their own specialized processors, and each mobile platform has a unique combination of such processors. For example, the Kirin NPU has a 3D Cube Tensor Computing Engine that can perform 2,048 FP16 multiplications. This exemplifies an important trend in modern parallel hardware design: SIMD unit expansion to increase computational capacity. Existing computation-intensive workloads, on the other hand, are difficult to map onto such hardware and necessitate significant low-level hardware and software stack expertise. My expertise will provide me with advantageous opportunities to advance the frontiers of this topic in the future. I plan to look into it from two angles: parallelism and data locality. By optimizing parallel algorithms and guiding further hardware optimization, the ultimate goal is to fully unleash the power of new parallel architectures. The interaction of parallel hardware and parallel algorithms is so complex that more research is needed.

Bibliography

- [1] MARTIN ABADI, PAUL BARHAM, JIANMIN CHEN, ZHIFENG CHEN, ANDY DAVIS, JEFFREY DEAN, MATTHIEU DEVIN, SANJAY GHEMAWAT, GEOFFREY IRVING, MICHAEL ISARD, MANJUNATH KUDLUR, JOSH LEVENBERG, RAJAT MONGA, SHERRY MOORE, DEREK G. MURRAY, BENOIT STEINER, PAUL TUCKER, VIJAY VASUDEVAN, PETE WARDEN, MARTIN WICKE, YUAN YU, AND XIAOQIANG ZHENG. Tensorflow: A system for large-scale machine learning. In *OSDI 2016*, pages 265–283, USA, 2016. USENIX Association.
- [2] ARAVIND ACHARYA, UDAY BONDHUGULA, AND ALBERT COHEN. Polyhedral auto-transformation with no integer linear programming. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 529–542, New York, NY, USA, 2018. Association for Computing Machinery.
- [3] ARAVIND ACHARYA, UDAY BONDHUGULA, AND ALBERT COHEN. Effective loop fusion in polyhedral compilation using fusion conflict graphs. *ACM Transactions on Architecture and Code Optimization (TACO)*, 17(4):1–26, 2020.
- [4] EIRIKUR AGUSTSSON AND RADU TIMOFTE. Ntire 2017 challenge on single image super-resolution: Dataset and study. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, July 2017.
- [5] MAAZ BIN SAFEER AHMAD, ALEXANDER J. ROOT, ANDREW ADAMS, SHOAIB KAMIL, AND ALVIN CHEUNG. Vector instruction selection for digital signal processors using program synthesis. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2022*, page 1004–1016, New York, NY, USA, 2022. Association for Computing Machinery.
- [6] BYUNG HOON AHN, JINWON LEE, JAMIE MENJAY LIN, HSIN-PAI CHENG, JILEI HOU, AND HADI ESMAEILZADEH. Ordering chaos: Memory-aware scheduling of irregularly wired neural networks for edge devices. *Proceedings of Machine Learning and Systems*, 2:44–57, 2020.
- [7] ANDREW ANDERSON AND DAVID GREGG. Optimal dnn primitive selection with partitioned boolean quadratic programming. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, pages 340–351, 2018.

- [8] RIYADH BAGHDADI, JESSICA RAY, MALEK BEN ROMDHANE, EMANUELE DEL SOZZO, ABDURRAHMAN AKKAS, YUNMING ZHANG, PATRICIA SURIANA, SHOAIB KAMIL, AND SAMAN AMARASINGHE. Tiramisu: A polyhedral compiler for expressing fast and portable code. In *CGO 2019*, pages 193–205. IEEE, 2019.
- [9] DAN BENANAV, DEEPAK KAPUR, AND PALIATH NARENDRAN. Complexity of matching problems. *Journal of symbolic computation*, 3(1-2):203–216, 1987.
- [10] DAVID BERNSTEIN AND MICHAEL RODEH. Global instruction scheduling for superscalar machines. In *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 241–255, 1991.
- [11] SOMASHEKARACHARYA G BHASKARACHARYA, JULIEN DEMOUTH, AND VINOD GROVER. Automatic kernel generation for volta tensor cores. *arXiv preprint arXiv:2006.12645*, 2020.
- [12] SOURAV BHATTACHARYA AND NICHOLAS D LANE. From smart to deep: Robust activity recognition on smartwatches using deep learning. In *2016 IEEE International Conference on Pervasive Computing and Communication Workshops (PerCom Workshops)*, pages 1–6. IEEE, 2016.
- [13] ALEXEY BOCHKOVSKIY, CHIEN-YAO WANG, AND HONG-YUAN MARK LIAO. Yolov4: Optimal speed and accuracy of object detection. *arXiv preprint arXiv:2004.10934*, 2020.
- [14] MATTHIAS BOEHM, BERTHOLD REINWALD, DYLAN HUTCHISON, PRITHVIRAJ SEN, ALEXANDRE V. EVFIMIEVSKI, AND NIKETAN PANSARE. On optimizing operator fusion plans for large-scale machine learning in systemml. *Proc. VLDB Endow.*, 11(12):1755–1768, August 2018.
- [15] UDAY BONDHUGULA, OKTAY GUNLUK, SANJEEB DASH, AND LAKSHMINARAYANAN RENGANARAYANAN. A model for fusion and code motion in an automatic parallelizing compiler. In *PACT 2010*, page 343–352. ACM, 2010.
- [16] UDAY BONDHUGULA, ALBERT HARTONO, JAGANNATHAN RAMANUJAM, AND PONNUSWAMY SADAYAPPAN. A practical automatic polyhedral parallelizer and locality optimizer. In *PLDI 2008*, pages 101–113, 2008.
- [17] IVICA BOTICKI AND HYO-JEONG SO. Quiet captures: A tool for capturing the evidence of seamless learning with mobile devices. In *Proceedings of the 9th International Conference of the Learning Sciences-Volume 1*, pages 500–507. International Society of the Learning Sciences, 2010.
- [18] STEPHEN BOYD, NEAL PARIKH, ERIC CHU, BORJA PELEATO, AND JONATHAN ECKSTEIN. Distributed optimization and statistical learning via the alternating direction method of multipliers. *Foundations and Trends® in Machine Learning*, 3(1):1–122, 2011.

- [19] SAMUEL ROTA BULO, LORENZO PORZI, AND PETER KONTSCHIEDER. In-place activated batchnorm for memory-optimized training of dnns. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 5639–5647, 2018.
- [20] DAVID CALLAHAN, KEITH D COOPER, KEN KENNEDY, AND LINDA TORCZON. Interprocedural constant propagation. *ACM SIGPLAN Notices*, 21(7):152–161, 1986.
- [21] PRASANTH CHATARASI, JUN SHIRAKO, AND VIVEK SARKAR. Polyhedral optimizations of explicitly parallel programs. In *PACT 2015*, pages 213–226. IEEE, 2015.
- [22] CHUNG-KAI CHEN, LING-HUA TSENG, SHIH-CHANG CHEN, YOUNG-JIA LIN, YI-PING YOU, CHIA-HAN LU, AND JENQ-KUEN LEE. Enabling compiler flow for embedded vliw dsp processors with distributed register files. In *Proceedings of the 2007 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 146–148, 2007.
- [23] LINCHUAN CHEN, PENG JIANG, AND GAGAN AGRAWAL. Exploiting recent simd architectural advances for irregular applications. In *2016 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 47–58. IEEE, 2016.
- [24] LONG CHEN, SHAOBO LIN, XIANKAI LU, DONGPU CAO, HANGBIN WU, CHI GUO, CHUN LIU, AND FEI-YUE WANG. Deep neural network based vehicle and pedestrian detection for autonomous driving: A survey. *IEEE Transactions on Intelligent Transportation Systems*, 22(6):3234–3246, 2021.
- [25] TIANQI CHEN, THIERRY MOREAU, ZIHENG JIANG, LIANMIN ZHENG, EDDIE YAN, HAICHEN SHEN, MEGHAN COWAN, LEYUAN WANG, YUWEI HU, LUIS CEZE, CARLOS GUESTRIN, AND ARVIND KRISHNAMURTHY. Tvm: An automated end-to-end optimizing compiler for deep learning. In *OSDI 2018*, pages 578–594, 2018.
- [26] TIANQI CHEN, LIANMIN ZHENG, EDDIE YAN, ZIHENG JIANG, THIERRY MOREAU, LUIS CEZE, CARLOS GUESTRIN, AND ARVIND KRISHNAMURTHY. Learning to optimize tensor programs. *arXiv preprint arXiv:1805.08166*, 2018.
- [27] YISHEN CHEN, CHARITH MENDIS, MICHAEL CARBIN, AND SAMAN AMARASINGHE. Vegen: a vectorizer generator for simd and beyond. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 902–914, 2021.
- [28] ALEXANDER COLLINS AND VINOD GROVER. Axon: A language for dynamic shapes in deep learning graphs. *arXiv preprint arXiv:2210.02374*, 2022.
- [29] KEITH D COOPER, L TAYLOR SIMPSON, AND CHRISTOPHER A VICK. Operator strength reduction. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 23(5):603–625, 2001.

- [30] MATTHIEU COURBARIAUX, YOSHUA BENGIO, AND JEAN-PIERRE DAVID. Binarconnect: Training deep neural networks with binary weights during propagations. In *Advances in neural information processing systems*, pages 3123–3131, 2015.
- [31] MATTHIEU COURBARIAUX, ITAY HUBARA, DANIEL SOUDRY, RAN EL-YANIV, AND YOSHUA BENGIO. Binarized neural networks: Training deep neural networks with weights and activations constrained to +1 or -1. *arXiv preprint arXiv:1602.02830*, 2016.
- [32] MEGHAN COWAN, THIERRY MOREAU, TIANQI CHEN, JAMES BORNHOLT, AND LUIS CEZE. Automatic generation of high-performance quantized machine learning kernels. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*, pages 305–316, 2020.
- [33] KARL CRARY AND STEPHANIE WEIRICH. Flexible type analysis. In *Proceedings of the fourth ACM SIGPLAN international conference on Functional programming*, pages 233–248, 1999.
- [34] XIAOLIANG DAI, HONGXU YIN, AND NIRAJ K JHA. Nest: a neural network synthesis tool based on a grow-and-prune paradigm. *arXiv preprint arXiv:1711.02017*, 2017.
- [35] ALAIN DARTE. On the complexity of loop fusion. In *1999 International Conference on Parallel Architectures and Compilation Techniques (Cat. No. PR00425)*, pages 149–157. IEEE, 1999.
- [36] B DUPONT DE DINECHIN. From machine scheduling to vliw instruction scheduling. *ST Journal of Research*, 1(2):1–35, 2004.
- [37] HUGO DE MAN, J RABAEY, PAUL SIX, AND LUC CLAESEN. Cathedral-ii: A silicon compiler for digital signal processing. *IEEE Design & Test of Computers*, 3(6):13–25, 1986.
- [38] GIOVANNI DE MICHELI. *Synthesis and optimization of digital circuits*. Number BOOK. McGraw Hill, 1994.
- [39] JEFFREY DEAN, GREG CORRADO, RAJAT MONGA, KAI CHEN, MATTHIEU DEVIN, MARK MAO, MARC’AURELIO RANZATO, ANDREW SENIOR, PAUL TUCKER, KE YANG, ET AL. Large scale distributed deep networks. In *Advances in neural information processing systems*, pages 1223–1231, 2012.
- [40] SAUMYA K DEBRAY. Unfold/fold transformations and loop optimization of logic programs. In *PLDI 1988*, pages 297–307, 1988.
- [41] JIA DENG, WEI DONG, RICHARD SOCHER, LI-JIA LI, KAI LI, AND LI FEI-FEI. Imagenet: A large-scale hierarchical image database. In *CVPR 2009*, pages 248–255. IEEE, 2009.
- [42] YUNBIN DENG. Deep learning on mobile devices – a review. *arXiv preprint arXiv:1904.09274*, 2019.

- [43] MICROSOFT DEVELOPER. Codebert. <https://github.com/microsoft/CodeBERT>, 2023.
- [44] NUMPY DEVELOPERS. Tensor broadcasting. <https://numpy.org/doc/stable/user/basics.broadcasting.html>, 2023. Version: 1.24.
- [45] ONNX RUNTIME DEVELOPERS. Onnx runtime. <https://onnxruntime.ai/>, 2023. Version: 1.14.1.
- [46] JACOB DEVLIN, MING-WEI CHANG, KENTON LEE, AND KRISTINA TOUTANOVA. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.
- [47] YAORYAO DING, LIGENG ZHU, ZHIHAO JIA, GENNADY PEKHIMENKO, AND SONG HAN. Ios: Inter-operator scheduler for cnn acceleration. *Proceedings of Machine Learning and Systems*, 3, 2021.
- [48] JOHANNES DOERFERT, KEVIN STREIT, SEBASTIAN HACK, AND ZINO BENAÏSSA. Polly’s polyhedral scheduling in the presence of reductions. *arXiv preprint arXiv:1505.07716*, 2015.
- [49] PEIYAN DONG, SIYUE WANG, WEI NIU, CHENGMING ZHANG, SHENG LIN, ZHEN-GANG LI, YIFAN GONG, BIN REN, XUE LIN, AND DINGWEN TAO. Rtmobile: Beyond real-time mobile acceleration of rnns for speech recognition. In *57th ACM/IEEE Design Automation Conference*, pages 1–6. IEEE, 2020.
- [50] TAREK ELGAMAL, SHANGYU LUO, MATTHIAS BOEHM, ALEXANDRE V EV-FIMIEVSKI, SHIRISH TATIKONDA, BERTHOLD REINWALD, AND PRITHVIRAJ SEN. Spoof: Sum-product optimization and operator fusion for large-scale machine learning. In *CIDR*, 2017.
- [51] MARK EVERINGHAM, LUC VAN GOOL, CHRISTOPHER KI WILLIAMS, JOHN WINN, AND ANDREW ZISSERMAN. The pascal visual object classes challenge 2007 (voc2007) results. 2007.
- [52] JIARUI FANG, YANG YU, CHENGDUO ZHAO, AND JIE ZHOU. Turbo Transformers: an efficient gpu serving system for transformer models. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 389–402, 2021.
- [53] PRATIK FEGADE, TIANQI CHEN, PHIL GIBBONS, AND TODD MOWRY. Cortex: A compiler for recursive deep learning models. *arXiv preprint arXiv:2011.01383*, 2020.
- [54] PRATIK FEGADE, TIANQI CHEN, PHILLIP GIBBONS, AND TODD MOWRY. Cortex: A compiler for recursive deep learning models. *Proceedings of Machine Learning and Systems*, 3:38–54, 2021.

- [55] DI FENG, LARS ROSENBAUM, AND KLAUS DIETMAYER. Towards safe autonomous driving: Capture uncertainty in the deep neural network for lidar 3d vehicle detection. In *2018 21st international conference on intelligent transportation systems (ITSC)*, pages 3266–3273. IEEE, 2018.
- [56] ZHANGYIN FENG, DAYA GUO, DUYU TANG, NAN DUAN, XIAOCHENG FENG, MING GONG, LINJUN SHOU, BING QIN, TING LIU, DAXIN JIANG, ET AL. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020.
- [57] FRANZ FRANCHETTI, STEFAN KRAL, JUERGEN LORENZ, AND CHRISTOPH W UEBERHUBER. Efficient utilization of simd extensions. *Proceedings of the IEEE*, 93(2):409–425, 2005.
- [58] PIN GAO, LINGFAN YU, YONGWEI WU, AND JINYANG LI. Low latency rnn inference with cellular batching. In *Proceedings of the Thirteenth EuroSys Conference*, pages 1–15, 2018.
- [59] ANDREAS GEIGER, PHILIP LENZ, CHRISTOPH STILLER, AND RAQUEL URTASUN. Vision meets robotics: The kitti dataset. *International Journal of Robotics Research (IJRR)*, 2013.
- [60] EVANGELOS GEORGANAS, SASIKANTH AVANCHA, KUNAL BANERJEE, DHIRAJ KALAMKAR, GREG HENRY, HANS PABST, AND ALEXANDER HEINECKE. Anatomy of high-performance deep learning convolutions on simd architectures. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 830–841. IEEE, 2018.
- [61] PERRY GIBSON, JOSÉ CANO, JACK TURNER, ELLIOT J CROWLEY, MICHAEL O’BOYLE, AND AMOS STORKEY. Optimizing grouped convolutions on edge devices. In *2020 IEEE 31st International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 189–196. IEEE, 2020.
- [62] BEN GOERTZEL. Artificial general intelligence: concept, state of the art, and future prospects. *Journal of Artificial General Intelligence*, 5(1):1, 2014.
- [63] GOOGLE. Snapdragon 820, 2020.
- [64] GOOGLE. Tensorflow xla. <https://www.tensorflow.org/xla>, 2023.
- [65] SRIDHAR GOPINATH, NIKHIL GHANATHE, VIVEK SESHADRI, AND RAHUL SHARMA. Compiling kb-sized machine learning models to tiny iot devices. In *PLDI 2019*, pages 79–95, 2019.
- [66] JOSEPH L GREATHOUSE, KENT KNOX, JAKUB POLA, KIRAN VARAGANTI, AND MAYANK DAGA. clsparse: A vendor-optimized open-source sparse blas library. In *Proceedings of the 4th International Workshop on OpenCL*, page 7. ACM, 2016.

- [67] HUI GUAN, SHAOSHAN LIU, XIAOLONG MA, WEI NIU, BIN REN, XIPENG SHEN, YANZHI WANG, AND PU ZHAO. Cocopie: enabling real-time ai on off-the-shelf mobile devices via compression-compilation co-design. *Communications of the ACM*, 64(6):62–68, 2021.
- [68] ANMOL GULATI, JAMES QIN, CHUNG-CHENG CHIU, NIKI PARMAR, YU ZHANG, JIAHUI YU, WEI HAN, SHIBO WANG, ZHENGDONG ZHANG, YONGHUI WU, ET AL. Conformer: Convolution-augmented transformer for speech recognition. *arXiv preprint arXiv:2005.08100*, 2020.
- [69] YIWEN GUO, ANBANG YAO, AND YURONG CHEN. Dynamic network surgery for efficient dnns. In *Advances In Neural Information Processing Systems*, pages 1379–1387, 2016.
- [70] SUYOG GUPTA, ANKUR AGRAWAL, KAILASH GOPALAKRISHNAN, AND PRITISH NARAYANAN. Deep learning with limited numerical precision. In *International Conference on Machine Learning*, pages 1737–1746, 2015.
- [71] LANG HAMES AND BERNHARD SCHOLZ. Nearly optimal register allocation with pbqp. In *Joint Modular Languages Conference*, pages 346–361. Springer, 2006.
- [72] SEUNGYEOP HAN, HAICHEN SHEN, MATTHAI PHILIPOSE, SHARAD AGARWAL, ALEC WOLMAN, AND ARVIND KRISHNAMURTHY. Mcdnn: An approximation-based execution framework for deep stream processing under resource constraints. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*, pages 123–136. ACM, 2016.
- [73] SONG HAN, HUIZI MAO, AND WILLIAM J DALLY. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.
- [74] SONG HAN, JEFF POOL, JOHN TRAN, AND WILLIAM DALLY. Learning both weights and connections for efficient neural network. In *Advances in Neural Information Processing Systems*, pages 1135–1143, 2015.
- [75] ROBERT HARPER AND GREG MORRISSETT. Compiling polymorphism using intensional type analysis. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 130–141, 1995.
- [76] KAIMING HE, GEORGIA GKIOXARI, PIOTR DOLLÁR, AND ROSS GIRSHICK. Mask r-cnn. In *ICCV 2017*, pages 2961–2969, 2017.
- [77] KAIMING HE, XIANGYU ZHANG, SHAOQING REN, AND JIAN SUN. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.
- [78] YIHUI HE, JI LIN, ZHIJIAN LIU, HANRUI WANG, LI-JIA LI, AND SONG HAN. Amc: Automl for model compression and acceleration on mobile devices. In *European Conference on Computer Vision*, pages 815–832. Springer, 2018.

- [79] YIHUI HE, XIANGYU ZHANG, AND JIAN SUN. Channel pruning for accelerating very deep neural networks. In *Computer Vision (ICCV), 2017 IEEE International Conference on*, pages 1398–1406. IEEE, 2017.
- [80] KARTIK HEGDE, ROHIT AGRAWAL, YULUN YAO, AND CHRISTOPHER W FLETCHER. Morph: Flexible acceleration for 3d cnn-based video understanding. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 933–946. IEEE, 2018.
- [81] PARKER HILL, ANIMESH JAIN, MASON HILL, BABAK ZAMIRAI, CHANG-HONG HSU, MICHAEL A LAURENZANO, SCOTT MAHLKE, LINGJIA TANG, AND JASON MARS. Deftnn: Addressing bottlenecks for dnn execution on gpus via synapse vector elimination and near-compute data fission. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 786–799. ACM, 2017.
- [82] MINGYI HONG, ZHI-QUAN LUO, AND MEISAM RAZAVIYAYN. Convergence analysis of alternating direction method of multipliers for a family of nonconvex problems. *SIAM Journal on Optimization*, 26(1):337–364, 2016.
- [83] ANDREW HOWARD, MARK SANDLER, GRACE CHU, LIANG-CHIEH CHEN, BO CHEN, MINGXING TAN, WEIJUN WANG, YUKUN ZHU, RUOMING PANG, VIJAY VASUDEVAN, ET AL. Searching for mobilenetv3. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 1314–1324, 2019.
- [84] HUAWEI. Kirin 980, 2018.
- [85] ITAY HUBARA, MATTHIEU COURBARIAUX, DANIEL SOUDRY, RAN EL-YANIV, AND YOSHUA BENGIO. Binarized neural networks. In *Advances in neural information processing systems*, pages 4107–4115, 2016.
- [86] ITAY HUBARA, MATTHIEU COURBARIAUX, DANIEL SOUDRY, RAN EL-YANIV, AND YOSHUA BENGIO. Quantized neural networks: Training neural networks with low precision weights and activations. *The Journal of Machine Learning Research*, 18(1):6869–6898, 2017.
- [87] JOONMOO HUH AND JAMES TUCK. Improving the effectiveness of searching for isomorphic chains in superword level parallelism. In *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 718–729. IEEE, 2017.
- [88] LOC N HUYNH, YOUNGKI LEE, AND RAJESH KRISHNA BALAN. Deepmon: Mobile gpu-based deep learning framework for continuous vision applications. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*, pages 82–95. ACM, 2017.
- [89] SERGEY IOFFE AND CHRISTIAN SZEGEDY. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

- [90] BENOIT JACOB, SKIRMANTAS KLIGYS, BO CHEN, MENGLONG ZHU, MATTHEW TANG, ANDREW HOWARD, HARTWIG ADAM, AND DMITRY KALENICHENKO. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2704–2713, 2018.
- [91] ANIMESH JAIN, SHOUBHIK BHATTACHARYA, MASAHIRO MASUDA, VIN SHARMA, AND YIDA WANG. Efficient execution of quantized deep learning models: A compiler approach. *arXiv preprint arXiv:2006.10226*, 2020.
- [92] PARAS JAIN, AJAY JAIN, ANIRUDDHA NRUSIMHA, AMIR GHOLAMI, PIETER ABBEEL, JOSEPH GONZALEZ, KURT KEUTZER, AND ION STOICA. Checkmate: Breaking the memory wall with optimal tensor rematerialization. *Proceedings of Machine Learning and Systems*, 2:497–511, 2020.
- [93] EUNJI JEONG, JOO SEONG JEONG, SOOJEONG KIM, GYEONG-IN YU, AND BYUNG-GON CHUN. Improving the expressiveness of deep learning frameworks with recursion. In *Proceedings of the Thirteenth EuroSys Conference*, pages 1–13, 2018.
- [94] ZHIHAO JIA, ODED PADON, JAMES THOMAS, TODD WARSZAWSKI, MATEI ZAHARIA, AND ALEX AIKEN. Taso: optimizing deep learning computation with automatic generation of graph substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 47–62, 2019.
- [95] SHIQI JIANG, LIHAO RAN, TING CAO, YUSEN XU, AND YUNXIN LIU. Profiling and optimizing deep learning inference on mobile gpus. In *Proceedings of the 11th ACM SIGOPS Asia-Pacific Workshop on Systems*, pages 75–81, 2020.
- [96] XIAOTANG JIANG, HUAN WANG, YILIU CHEN, ZIQI WU, LICHUAN WANG, BIN ZOU, YAFENG YANG, ZONGYANG CUI, YU CAI, TIANHANG YU, CHENGFEI LYU, AND ZHIHUA WU. Mnn: A universal and efficient inference engine. In *Proceedings of Machine Learning and Systems*, I. Dhillon, D. Papailiopoulos, and V. Sze, editors, volume 2, pages 1–13. 2020.
- [97] XIAOQI JIAO, YICHUN YIN, LIFENG SHANG, XIN JIANG, XIAO CHEN, LINLIN LI, FANG WANG, AND QUN LIU. Tinybert: Distilling bert for natural language understanding. *arXiv preprint arXiv:1909.10351*, 2019.
- [98] QING JIN, LINJIE YANG, AND ZHENYU LIAO. Adabits: Neural network quantization with adaptive bit-widths. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 2146–2156, 2020.
- [99] JUSTIN JOHNSON, ALEXANDRE ALAHI, AND LI FEI-FEI. Perceptual losses for real-time style transfer and super-resolution. In *European conference on computer vision*, pages 694–711. Springer, 2016.

- [100] MAHMUT KANDEMIR, A CHOUDHARY, J RAMANUJAM, AND PRITHVIRAJ BANERJEE. Improving locality using loop and data transformations in an integrated framework. In *Proceedings. 31st Annual ACM/IEEE International Symposium on Microarchitecture*, pages 285–296. IEEE, 1998.
- [101] ANDREJ KARPATHY, GEORGE TODERICI, SANKETH SHETTY, THOMAS LEUNG, RAHUL SUKTHANKAR, AND LI FEI-FEI. Large-scale video classification with convolutional neural networks. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, pages 1725–1732, 2014.
- [102] SAM KAUFMAN, PHITCHAYA PHOTHILIMTHANA, YANQI ZHOU, CHARITH MENDIS, SUDIP ROY, AMIT SABNE, AND MIKE BURROWS. A learned performance model for tensor processing units. *Proceedings of Machine Learning and Systems*, 3, 2021.
- [103] KEN KENNEDY AND KATHRYN S MCKINLEY. Maximizing loop parallelism and improving data locality via loop fusion and distribution. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 301–320. Springer, 1993.
- [104] GARY A KILDALL. A unified approach to global program optimization. In *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 194–206, 1973.
- [105] SEHOON KIM, AMIR GHOLAMI, ZHEWEI YAO, MICHAEL W MAHONEY, AND KURT KEUTZER. I-bert: Integer-only bert quantization. *arXiv preprint arXiv:2101.01321*, 2021.
- [106] DIEDERIK P. KINGMA AND JIMMY BA. Adam: A method for stochastic optimization. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2014.
- [107] ALEXANDER KIRILLOV, ERIC MINTUN, NIKHILA RAVI, HANZI MAO, CHLOE ROLLAND, LAURA GUSTAFSON, TETE XIAO, SPENCER WHITEHEAD, ALEXANDER C. BERG, WAN-YEN LO, PIOTR DOLLÁR, AND ROSS GIRSHICK. Segment anything. *arXiv:2304.02643*, 2023.
- [108] MARISA KIRISAME, STEVEN LYUBOMIRSKY, ALTAN HAAN, JENNIFER BRENNAN, MIKE HE, JARED ROESCH, TIANQI CHEN, AND ZACHARY TATLOCK. Dynamic tensor rematerialization. *arXiv preprint arXiv:2006.09616*, 2020.
- [109] FREDRIK KJOLSTAD, PETER AHRENS, SHOAIB KAMIL, AND SAMAN AMARASINGHE. Tensor algebra compilation with workspaces. In *CGO 2019*, pages 180–192. IEEE, 2019.
- [110] FREDRIK KJOLSTAD, SHOAIB KAMIL, STEPHEN CHOU, DAVID LUGATO, AND SAMAN AMARASINGHE. The tensor algebra compiler. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):1–29, 2017.

- [111] MARTIN KONG, RICHARD VERAS, KEVIN STOCK, FRANZ FRANCHETTI, LOUIS-NOËL POUCHET, AND PONNUSWAMY SADAYAPPAN. When polyhedral transformations meet simd code generation. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, pages 127–138, 2013.
- [112] EMMANUEL KOUNALIS AND DENIS LUGIEZ. Compilation of pattern matching with associative-commutative functions. In *Colloquium on Trees in Algebra and Programming*, pages 57–73. Springer, 1991.
- [113] MANUEL KREBBER. Non-linear associative-commutative many-to-one pattern matching with sequence variables. *arXiv preprint arXiv:1705.00907*, 2017.
- [114] NIKOLAOS KYRTATAS, DANIELE G SPAMPINATO, AND MARKUS PÜSCHEL. A basic linear algebra compiler for embedded processors. In *2015 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1054–1059. IEEE, 2015.
- [115] ZHENZHONG LAN, MINGDA CHEN, SEBASTIAN GOODMAN, KEVIN GIMPEL, PIYUSH SHARMA, AND RADU SORICUT. Albert: A lite bert for self-supervised learning of language representations. In *International Conference on Learning Representations*, 2020.
- [116] N. D. LANE, S. BHATTACHARYA, P. GEORGIEV, C. FORLIVESI, L. JIAO, L. QENDRO, AND F. KAWSAR. Deepx: A software accelerator for low-power deep learning inference on mobile devices. In *2016 15th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*, pages 1–12. IEEE Press, 2016.
- [117] NICHOLAS D LANE, SOURAV BHATTACHARYA, PETKO GEORGIEV, CLAUDIO FORLIVESI, AND FAHIM KAWSAR. An early resource characterization of deep learning on wearables, smartphones and internet-of-things devices. In *Proceedings of the 2015 international workshop on internet of things towards applications*, pages 7–12. ACM, 2015.
- [118] NICHOLAS D LANE, SOURAV BHATTACHARYA, AKHIL MATHUR, PETKO GEORGIEV, CLAUDIO FORLIVESI, AND FAHIM KAWSAR. Squeezing deep learning into mobile and embedded devices. *IEEE Pervasive Computing*, 16(3):82–88, 2017.
- [119] NICHOLAS D LANE, PETKO GEORGIEV, AND LORENA QENDRO. Deeppear: robust smartphone audio sensing in unconstrained acoustic environments using deep learning. In *Proceedings of the 2015 ACM international joint conference on pervasive and ubiquitous computing*, pages 283–294. ACM, 2015.
- [120] CHRIS LATTNER, MEHDI AMINI, UDAY BONDHUGULA, ALBERT COHEN, ANDY DAVIS, JACQUES PIENAAR, RIVER RIDDLE, TATIANA SHPEISMAN, NICOLAS VASILACHE, AND OLEKSANDR ZINENKO. Mlir: Scaling compiler infrastructure for domain specific computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 2–14. IEEE, 2021.

- [121] CHRIS LATTNER, JACQUES PIENAAR, MEHDI AMINI, UDAY BONDHUGULA, RIVER RIDDLE, ALBERT COHEN, TATIANA SHPEISMAN, ANDY DAVIS, NICOLAS VASILACHE, AND OLEKSANDR ZINENKO. Mlir: A compiler infrastructure for the end of moore’s law. *arXiv preprint arXiv:2002.11054*, 2020.
- [122] ANDREW LAVIN AND SCOTT GRAY. Fast algorithms for convolutional neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4013–4021, 2016.
- [123] VADIM LEBEDEV AND VICTOR LEMPITSKY. Fast convnets using group-wise brain damage. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2554–2564, 2016.
- [124] CHINGREN LEE, JENQ KUEN LEE, TINGTING HWANG, AND SHI-CHUN TSAI. Compiler optimization on vliw instruction scheduling for low power. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 8(2):252–268, 2003.
- [125] HONGLAK LEE, ROGER GROSSE, RAJESH RANGANATH, AND ANDREW Y NG. Convolutional deep belief networks for scalable unsupervised learning of hierarchical representations. In *Proceedings of the 26th annual international conference on machine learning*, pages 609–616. ACM, 2009.
- [126] ROYSON LEE, STYLIANOS I VENIERIS, LUKASZ DUDZIAK, SOURAV BHATTACHARYA, AND NICHOLAS D LANE. Mobisr: Efficient on-device super-resolution through heterogeneous mobile processors. In *The 25th Annual International Conference on Mobile Computing and Networking*, pages 1–16, 2019.
- [127] CONG LENG, HAO LI, SHENGHUO ZHU, AND RONG JIN. Extremely low bit neural network: Squeeze the last bit out with admm. *arXiv preprint arXiv:1707.09870*, 2017.
- [128] RAINER LEUPERS. Instruction scheduling for clustered vliw dsps. In *Proceedings 2000 International Conference on Parallel Architectures and Compilation Techniques (Cat. No. PR00622)*, pages 291–300. IEEE, 2000.
- [129] MAKSIM LEVENTAL. Memory planning for deep neural networks. *arXiv preprint arXiv:2203.00448*, 2022.
- [130] CHANGLIN LI, GUANRUN WANG, BING WANG, XIAODAN LIANG, ZHIHUI LI, AND XIAOJUN CHANG. Dynamic slimmable network. In *CVPR*, 2021.
- [131] CHUYI LI, LULU LI, HONGLIANG JIANG, KAIHENG WENG, YIFEI GENG, LIANG LI, ZAIDAN KE, QINGYUAN LI, MENG CHENG, WEIQIANG NIE, ET AL. Yolov6: A single-stage object detection framework for industrial applications. *arXiv preprint arXiv:2209.02976*, 2022.
- [132] HAO LI, ASIM KADAV, IGOR DURDANOVIC, HANAN SAMET, AND HANS PETER GRAF. Pruning filters for efficient convnets. In *International Conference on Learning Representations (ICLR)*, 2017.

- [133] EDGAR LIBERIS AND NICHOLAS D LANE. Neural networks on microcontrollers: saving memory at inference via operator reordering. *arXiv preprint arXiv:1910.05110*, 2019.
- [134] DARRYL LIN, SACHIN TALATHI, AND SREEKANTH ANNAPOUREDDY. Fixed point quantization of deep convolutional networks. In *International Conference on Machine Learning*, pages 2849–2858, 2016.
- [135] JI LIN, WEI-MING CHEN, YUJUN LIN, CHUANG GAN, SONG HAN, ET AL. Mcunet: Tiny deep learning on iot devices. *Advances in Neural Information Processing Systems*, 33:11711–11722, 2020.
- [136] TSUNG-YI LIN, MICHAEL MAIRE, SERGE BELONGIE, JAMES HAYS, PIETRO PERONA, DEVA RAMANAN, PIOTR DOLLÁR, AND C LAWRENCE ZITNICK. Microsoft coco: Common objects in context. In *European conference on computer vision*, pages 740–755. Springer, 2014.
- [137] TSUNG-YI LIN, MICHAEL MAIRE, SERGE J. BELONGIE, LUBOMIR D. BOURDEV, ROSS B. GIRSHICK, JAMES HAYS, PIETRO PERONA, DEVA RAMANAN, PIOTR DOLL’A R, AND C. LAWRENCE ZITNICK. Microsoft COCO: common objects in context. *CoRR*, abs/1405.0312, 2014.
- [138] W LIN, CORINNA G LEE, AND PAUL CHOW. an optimizing compiler for the tms320c25 dsp chip. In *Proc. Int. Conf. Signal Processing Applicat. Technol.*, number 5, pages I–689. Citeseer, 1994.
- [139] BAOYUAN LIU, MIN WANG, HASSAN FOROOSH, MARSHALL TAPPEN, AND MARIANNA PENSKY. Sparse convolutional neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 806–814, 2015.
- [140] SICONG LIU, YINGYAN LIN, ZIMU ZHOU, KAIMING NAN, HUI LIU, AND JUNZHAO DU. On-demand deep model compression for mobile devices: A usage-driven model selection framework. In *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services*, pages 389–400. ACM, 2018.
- [141] SIJIA LIU, JIE CHEN, PIN-YU CHEN, AND ALFRED HERO. Zeroth-order online alternating direction method of multipliers: Convergence analysis and applications. In *International Conference on Artificial Intelligence and Statistics*, pages 288–297, 2018.
- [142] WEI LIU, DRAGOMIR ANGUELOV, DUMITRU ERHAN, CHRISTIAN SZEGEDY, SCOTT REED, CHENG-YANG FU, AND ALEXANDER C BERG. Ssd: Single shot multibox detector. In *European conference on computer vision*, pages 21–37. Springer, 2016.
- [143] LLVM. Llm, 2021.
- [144] MOSHE LOOKS, MARCELLO HERRESHOFF, DELESLEY HUTCHINS, AND PETER NORVIG. Deep learning with dynamic computation graphs. *arXiv preprint arXiv:1702.02181*, 2017.

- [145] MARKUS LORENZ, PETER MARWEDEL, THORSTEN DRAGER, GERHARD FETTWEIS, AND RAINER LEUPERS. Compiler based exploration of dsp energy savings by simd operations. In *ASP-DAC 2004: Asia and South Pacific Design Automation Conference 2004 (IEEE Cat. No. 04EX753)*, pages 839–842. IEEE, 2004.
- [146] LINGXIAO MA, ZHIQIANG XIE, ZHI YANG, JILONG XUE, YOUSHAN MIAO, WEI CUI, WENXIANG HU, FAN YANG, LINTAO ZHANG, AND LIDONG ZHOU. Rammer: Enabling holistic deep learning compiler optimizations with rtasks. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 881–897. USENIX Association, November 2020.
- [147] XIAOLONG MA, FU-MING GUO, WEI NIU, XUE LIN, JIAN TANG, KAISHENG MA, BIN REN, AND YANZHI WANG. Pconv: The missing but desirable sparsity in dnn weight pruning for real-time execution on mobile devices. *arXiv preprint arXiv:1909.05073*, 2019.
- [148] MARTIN MAAS, ULYSSE BEAUGNON, ARUN CHAUHAN, AND BERKIN ILBEYI. Tella-alloc: Efficient on-chip memory allocation for production machine learning accelerators. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1, ASPLOS 2023*, page 123–137, New York, NY, USA, 2022. Association for Computing Machinery.
- [149] GEOFFREY MAINLAND, ROMAN LESHCHINSKIY, AND SIMON PEYTON JONES. Exploiting vector instructions with generalized stream fusion. *Communications of the ACM*, 60(5):83–91, 2017.
- [150] HUIZI MAO, SONG HAN, JEFF POOL, WENSHUO LI, XINGYU LIU, YU WANG, AND WILLIAM J DALLY. Exploring the regularity of sparse structure in convolutional neural networks. *arXiv preprint arXiv:1705.08922*, 2017.
- [151] NIMROD MEGIDDO AND VIVEK SARKAR. Optimal weighted loop fusion for parallel programs. In *Proceedings of the ninth annual ACM symposium on Parallel algorithms and architectures*, pages 282–291, 1997.
- [152] CHARITH MENDIS AND SAMAN AMARASINGHE. Goslp: Globally optimized superword level parallelism framework. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–28, 2018.
- [153] ROBIN MILNER. A theory of type polymorphism in programming. *Journal of computer and system sciences*, 17(3):348–375, 1978.
- [154] NAUMS MOGERS, VALENTIN RADU, LU LI, JACK TURNER, MICHAEL O’BOYLE, AND CHRISTOPHE DUBACH. Automatic generation of specialized direct convolutions for mobile gpus. In *Proceedings of the 13th Annual Workshop on General Purpose Processing using Graphics Processing Unit*, pages 41–50, 2020.
- [155] GRAHAM NEUBIG, CHRIS DYER, YOAV GOLDBERG, AUSTIN MATTHEWS, WALEED AMMAR, ANTONIOS ANASTASOPOULOS, MIGUEL BALLESTEROS, DAVID CHIANG,

- DANIEL CLOTHIAUX, TREVOR COHN, ET AL. Dynet: The dynamic neural network toolkit. *arXiv preprint arXiv:1701.03980*, 2017.
- [156] WEI NIU, JIEXIONG GUAN, XIPENG SHEN, YANZHI WANG, GAGAN AGRAWAL, AND BIN REN. Gcd²: A globally optimizing compiler for mapping dnns to mobile dsps. In *Proceedings of the 55th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2022.
- [157] WEI NIU, JIEXIONG GUAN, YANZHI WANG, GAGAN AGRAWAL, AND BIN REN. Dnnfusion: accelerating deep neural networks execution with advanced operator fusion. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 883–898, 2021.
- [158] WEI NIU, XIAOLONG MA, SHENG LIN, SHIHAO WANG, XUEHAI QIAN, XUE LIN, YANZHI WANG, AND BIN REN. Patdnn: Achieving real-time dnn execution on mobile devices with pattern-based weight pruning. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 907–922, 2020.
- [159] DORIT NUZMAN, IRA ROSEN, AND AYAL ZAKS. Auto-vectorization of interleaved data for simd. *ACM SIGPLAN Notices*, 41(6):132–143, 2006.
- [160] ONNX. Open neural network exchange. <https://www.onnx.ai>, 2017.
- [161] KAORU OTA, MINH SON DAO, VASILEIOS MEZARIS, AND FRANCESCO GB DE NATALE. Deep learning for mobile multimedia: A survey. *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)*, 13(3s):1–22, 2017.
- [162] VASSIL PANAYOTOV, GUOGUO CHEN, DANIEL POVEY, AND SANJEEV KHUDANPUR. Librispeech: an asr corpus based on public domain audio books. In *Acoustics, Speech and Signal Processing (ICASSP), 2015 IEEE International Conference on*, pages 5206–5210. IEEE, 2015.
- [163] ANGSHUMAN PARASHAR, MINSOO RHU, ANURAG MUKKARA, ANTONIO PUGLIELLI, RANGHARAJAN VENKATESAN, BRUCEK KHAILANY, JOEL EMER, STEPHEN W KECKLER, AND WILLIAM J DALLY. Scnn: An accelerator for compressed-sparse convolutional neural networks. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pages 27–40. ACM, 2017.
- [164] EUNHYEOK PARK, JUNWHAN AHN, AND SUNGJOO YOO. Weighted-entropy-based quantization for deep neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 7197–7205, 2017.
- [165] ADAM PASZKE, SAM GROSS, FRANCISCO MASSA, ADAM LERER, JAMES BRADBURY, GREGORY CHANAN, TREVOR KILLEEN, ZEMING LIN, NATALIA GIMELSHEIN, LUCA ANTIGA, ET AL. Pytorch: An imperative style, high-performance deep learning library. pages 8024–8035, 2019.

- [166] ADAM PASZKE, SAM GROSS, FRANCISCO MASSA, ADAM LERER, JAMES BRADBURY, GREGORY CHANAN, TREVOR KILLEEN, ZEMING LIN, NATALIA GIMELSHEIN, LUCA ANTIGA, ET AL. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.
- [167] DAMIAN PHILIPP, FRANK DURR, AND KURT ROTHERMEL. A sensor network abstraction for flexible public sensing systems. In *2011 IEEE Eighth International Conference on Mobile Ad-Hoc and Sensor Systems*, pages 460–469. IEEE, 2011.
- [168] PHITCHAYA MANGPO POTHILIMTHANA, AMIT SABNE, NIKHIL SARDA, KARTHIK SRINIVASA MURTHY, YANQI ZHOU, CHRISTOF ANGERMUELLER, MIKE BURROWS, SUDIP ROY, KETAN MANDKE, REZSA FARAHANI, ET AL. A flexible approach to autotuning multi-pass machine learning compilers. In *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 1–16. IEEE, 2021.
- [169] YURY PISARCHYK AND JUHYUN LEE. Efficient memory management for deep neural net inference. *arXiv preprint arXiv:2001.03288*, 2020.
- [170] LOUIS-NOËL POUCHET, CÉDRIC BASTOUL, ALBERT COHEN, AND JOHN CAVAZOS. Iterative optimization in the polyhedral model: Part ii, multidimensional time. *ACM SIGPLAN Notices*, 43(6):90–100, 2008.
- [171] LOUIS-NOËL POUCHET, UDAY BONDHUGULA, CÉDRIC BASTOUL, ALBERT COHEN, JAGANNATHAN RAMANUJAM, PONNUSWAMY SADAYAPPAN, AND NICOLAS VASILACHE. Loop transformations: convexity, pruning and optimization. *ACM SIGPLAN Notices*, 46(1):549–562, 2011.
- [172] BENOÎT PRADELLE, BENOÎT MEISTER, MUTHU BASKARAN, JONATHAN SPRINGER, AND RICHARD LETHIN. Polyhedral optimization of tensorflow computation graphs. In *Programming and Performance Visualization Tools*, pages 74–89. Springer, 2017.
- [173] HAOTONG QIN, ZHONGANG CAI, MINGYUAN ZHANG, YIFU DING, HAIYU ZHAO, SHUAI YI, XIANGLONG LIU, AND HAO SU. Bipointnet: Binary neural network for point clouds. *arXiv preprint arXiv:2010.05501*, 2020.
- [174] QUALCOMM. Snapdragon 820. <https://www.qualcomm.com/products/snapdragon-820-mobile-platform>, 2016.
- [175] QUALCOMM. Snapdragon 835. <https://www.qualcomm.com/products/snapdragon-835-mobile-platform>, 2016.
- [176] QUALCOMM. Snapdragon profiler. <https://developer.qualcomm.com/software/snapdragon-profiler>, 2016.
- [177] QUALCOMM. Hexagon v66 manual, 2017.
- [178] QUALCOMM. Snpe, 2017.

- [179] QUALCOMM. Snapdragon 850. <https://www.qualcomm.com/products/application/mobile-computing/snapdragon-8-series-mobile-compute-platforms/snapdragon-850-mobile-compute-platform>, 2018.
- [180] QUALCOMM. Snapdragon 855. <https://www.qualcomm.com/products/snapdragon-855-mobile-platform>, 2018.
- [181] QUALCOMM. Snapdragon 855, 2018.
- [182] QUALCOMM. Hexagon nn library, 2019.
- [183] QUALCOMM. Snapdragon 865, 2019.
- [184] QUALCOMM. Snapdragon 865, 2019.
- [185] QUALCOMM. Snapdragon 888. <https://www.qualcomm.com/products/snapdragon-888-5g-mobile-platform>, 2020.
- [186] QUALCOMM. Theoretical speed of hexagon dsp, 2021.
- [187] QUALCOMM. Snapdragon 8 gen 1. <https://www.qualcomm.com/products/application/smartphones/snapdragon-8-series-mobile-platforms/snapdragon-8-gen-1-mobile-platform>, 2022.
- [188] ALEC RADFORD, JEFFREY WU, REWON CHILD, DAVID LUAN, DARIO AMODEI, AND ILYA SUTSKEVER. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.
- [189] JONATHAN RAGAN-KELLEY, CONNELLY BARNES, ANDREW ADAMS, SYLVAIN PARIS, FRÉDO DURAND, AND SAMAN AMARASINGHE. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *PLDI 2013*, page 519–530, New York, NY, USA, 2013. Association for Computing Machinery.
- [190] SUBRAMANIAN RAJAGOPALAN, SREERANGA P RAJAN, SHARAD MALIK, SANDRO RIGO, GUIDO ARAUJO, AND KOICHIRO TAKAYAMA. A retargetable vliw compiler framework for dsps with instruction-level parallelism. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20(11):1319–1328, 2001.
- [191] YONGMING RAO, WENLIANG ZHAO, BENLIN LIU, JIWEN LU, JIE ZHOU, AND CHO-JUI HSIEH. Dynamicvit: Efficient vision transformers with dynamic token sparsification. *Advances in neural information processing systems*, 34:13937–13949, 2021.
- [192] MOHAMMAD RASTEGARI, VICENTE ORDONEZ, JOSEPH REDMON, AND ALI FARHADI. Xnor-net: Imagenet classification using binary convolutional neural networks. In *European Conference on Computer Vision*, pages 525–542. Springer, 2016.

- [193] AO REN, TIANYUN ZHANG, SHAOKAI YE, JIAYU LI, WENYAO XU, XUEHAI QIAN, XUE LIN, AND YANZHI WANG. Admm-nn: An algorithm-hardware co-design framework of dnns using alternating direction methods of multipliers. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 925–938, 2019.
- [194] GANG REN, PENG WU, AND DAVID PADUA. Optimizing data permutations for simd devices. *ACM SIGPLAN Notices*, 41(6):118–131, 2006.
- [195] SHAOQING REN, KAIMING HE, ROSS GIRSHICK, AND JIAN SUN. Faster r-cnn: Towards real-time object detection with region proposal networks. In *Advances in neural information processing systems*, pages 91–99, 2015.
- [196] RODRIGO CO ROCHA, VASILEIOS PORPODAS, PAVLOS PETOUMENOS, LUÍS FW GÓES, ZHENG WANG, MURRAY COLE, AND HUGH LEATHER. Vectorization-aware loop unrolling with seed forwarding. In *Proceedings of the 29th International Conference on Compiler Construction*, pages 1–13, 2020.
- [197] MARY M RODGERS, VINAY M PAI, AND RICHARD S CONROY. Recent advances in wearable sensors for health monitoring. *IEEE Sensors Journal*, 15(6):3119–3126, 2014.
- [198] ROBIN ROMBACH, ANDREAS BLATTMANN, DOMINIK LORENZ, PATRICK ESSER, AND BJÖRN OMMER. High-resolution image synthesis with latent diffusion models. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 10684–10695, 2022.
- [199] OLAF RONNEBERGER, PHILIPP FISCHER, AND THOMAS BROX. U-net: Convolutional networks for biomedical image segmentation. In *International Conference on Medical image computing and computer-assisted intervention*, pages 234–241. Springer, 2015.
- [200] MARK SANDLER, ANDREW HOWARD, MENGLONG ZHU, ANDREY ZHMOGINOV, AND LIANG-CHIEH CHEN. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4510–4520, 2018.
- [201] VICTOR SANH, LYSANDRE DEBUT, JULIEN CHAUMOND, AND THOMAS WOLF. Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter. *arXiv preprint arXiv:1910.01108*, 2019.
- [202] MANUEL SELVA, FABIAN GRUBER, DIOGO SAMPAIO, CHRISTOPHE GUILLON, LOUIS-NOËL POUCHET, AND FABRICE RASTELLO. Building a polyhedral representation from an instrumented execution: Making dynamic analyses of nonaffine programs scalable. *ACM Transactions on Architecture and Code Optimization (TACO)*, 16(4):1–26, 2019.
- [203] HAICHEN SHEN, JARED ROESCH, ZHI CHEN, WEI CHEN, YONG WU, MU LI, VIN SHARMA, ZACHARY TATLOCK, AND YIDA WANG. Nimble: Efficiently compiling

- dynamic neural networks for model inference. *Proceedings of Machine Learning and Systems*, 3:208–222, 2021.
- [204] JUNZHONG SHEN, YOU HUANG, ZELONG WANG, YURAN QIAO, MEI WEN, AND CHUNYUAN ZHANG. Towards a uniform template-based architecture for accelerating 2d and 3d cnns on fpga. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 97–106, 2018.
- [205] JEREMY SIEK AND WALID TAHA. Gradual typing for objects. In *ECOOP 2007–Object-Oriented Programming: 21st European Conference, Berlin, Germany, July 30–August 3, 2007. Proceedings 21*, pages 2–27. Springer, 2007.
- [206] KAREN SIMONYAN AND ANDREW ZISSERMAN. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.
- [207] CYRIL SIX, SYLVAIN BOULMÉ, AND DAVID MONNIAUX. Certified and efficient instruction scheduling: application to interlocked vliw processors. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–29, 2020.
- [208] LINGHAO SONG, FAN CHEN, YOUWEI ZHUO, XUEHAI QIAN, HAI LI, AND YIRAN CHEN. Accpar: Tensor partitioning for heterogeneous deep learning accelerators. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 342–355. IEEE, 2020.
- [209] KHURRAM SOOMRO, AMIR ROSHAN ZAMIR, AND MUBARAK SHAH. Ucf101: A dataset of 101 human actions classes from videos in the wild. *arXiv preprint arXiv:1212.0402*, 2012.
- [210] DANIELE G SPAMPINATO, DIEGO FABREGAT-TRAVER, PAOLO BIENTINESI, AND MARKUS PÜSCHEL. Program generation for small-scale linear algebra applications. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, pages 327–339, 2018.
- [211] DANIELE G SPAMPINATO AND MARKUS PÜSCHEL. A basic linear algebra compiler. In *CGO’14*, pages 23–32, 2014.
- [212] ZHIQING SUN, HONGKUN YU, XIAODAN SONG, RENJIE LIU, YIMING YANG, AND DENNY ZHOU. Mobilebert: Task-agnostic compression of bert by progressive knowledge transfer. 2019.
- [213] MINGXING TAN AND QUOC LE. Efficientnet: Rethinking model scaling for convolutional neural networks. In *International Conference on Machine Learning*, pages 6105–6114. PMLR, 2019.
- [214] MINGXING TAN, RUOMING PANG, AND QUOC V LE. Efficientdet: Scalable and efficient object detection. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 10781–10790, 2020.

- [215] TENSORFLOW. Tensorflow grappler. https://www.tensorflow.org/guide/graph_optimization, 2018.
- [216] TENSORFLOW. Post-training quantization, 2021.
- [217] PETER TORELLI AND MOHIT BANGALE. Measuring inference performance of machine-learning frameworks on edge-class devices with the mlmark benchmark. *Technical Report. Available online: <https://www.eembc.org/techlit/articles/MLMARK-WHITEPAPERFINAL-1.pdf> (accessed on 5 April 2021)*, 2021.
- [218] DU TRAN, LUBOMIR BOURDEV, ROB FERGUS, LORENZO TORRESANI, AND MANOHAR PALURI. Learning spatiotemporal features with 3d convolutional networks. In *ICCV 2015*, pages 4489–4497, 2015.
- [219] KONRAD TRIFUNOVIC, DORIT NUZMAN, ALBERT COHEN, AYAL ZAKS, AND IRA ROSEN. Polyhedral-model guided loop-nest auto-vectorization. In *2009 18th International Conference on Parallel Architectures and Compilation Techniques*, pages 327–337. IEEE, 2009.
- [220] JEAN-BAPTISTE TRISTAN AND XAVIER LEROY. Formal verification of translation validators: a case study on instruction scheduling optimizations. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 17–27, 2008.
- [221] JEAN-BAPTISTE TRISTAN AND XAVIER LEROY. A simple, verified validator for software pipelining. *ACM Sigplan Notices*, 45(1):83–92, 2010.
- [222] ALEXA VANHATTUM, RACHIT NIGAM, VINCENT T LEE, JAMES BORNHOLT, AND ADRIAN SAMPSON. A synthesis-aided compiler for dsp architectures (wip paper). In *The 21st ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 131–135, 2020.
- [223] ALEXA VANHATTUM, RACHIT NIGAM, VINCENT T LEE, JAMES BORNHOLT, AND ADRIAN SAMPSON. Vectorization for digital signal processors via equality saturation. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 874–886, 2021.
- [224] NICOLAS VASILACHE, OLEKSANDR ZINENKO, THEODOROS THEODORIDIS, PRIYA GOYAL, ZACHARY DEVITO, WILLIAM S MOSES, SVEN VERDOOLAEGE, ANDREW ADAMS, AND ALBERT COHEN. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *arXiv preprint arXiv:1802.04730*, 2018.
- [225] ANDREAS VEIT AND SERGE BELONGIE. Convolutional networks with adaptive inference graphs. 2018.
- [226] ANAND VENKAT, MARY HALL, AND MICHELLE STROUT. Loop and data transformations for sparse matrix code. *ACM SIGPLAN Notices*, 50(6):521–532, 2015.

- [227] ANAND VENKAT, THARINDU RUSIRA, RAJ BARIK, MARY HALL, AND LEONARD TRUONG. Swirl: High-performance many-core cpu code generation for deep neural networks. *The International Journal of High Performance Computing Applications*, 33(6):1275–1289, 2019.
- [228] ANAND VENKAT, MANU SHANTHARAM, MARY HALL, AND MICHELLE MILLS STROUT. Non-affine extensions to polyhedral code generation. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 185–194, 2014.
- [229] SANDER VOCKE, HENK CORPORAAL, ROEL JORDANS, ROSILDE CORVINO, AND RICK NAS. Extending halide to improve software development for imaging dsps. *ACM Transactions on Architecture and Code Optimization (TACO)*, 14(3):1–25, 2017.
- [230] GANG WANG, WENRUI GONG, AND RYAN KASTNER. Instruction scheduling using max-min ant system optimization. In *Proceedings of the 15th ACM Great Lakes symposium on VLSI*, pages 44–49, 2005.
- [231] MANNI WANG, SHAOHUA DING, TING CAO, YUNXIN LIU, AND FENGYUAN XU. Asymo: scalable and efficient deep-learning inference on asymmetric mobile cpus. In *Proceedings of the 27th Annual International Conference on Mobile Computing and Networking*, pages 215–228, 2021.
- [232] XIN WANG, FISHER YU, ZI-YI DOU, TREVOR DARRELL, AND JOSEPH E GONZALEZ. Skipnet: Learning dynamic routing in convolutional networks. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 409–424, 2018.
- [233] WEI WEN, CHUNPENG WU, YANDAN WANG, YIRAN CHEN, AND HAI LI. Learning structured sparsity in deep neural networks. In *Advances in Neural Information Processing Systems*, pages 2074–2082, 2016.
- [234] SHMUEL WINOGRAD. *Arithmetic complexity of computations*, volume 33. Siam, 1980.
- [235] BICHEN WU, FORREST IANDOLA, PETER H JIN, AND KURT KEUTZER. Squeezedet: Unified, small, low power fully convolutional neural networks for real-time object detection for autonomous driving. In *Proceedings of the IEEE conference on computer vision and pattern recognition workshops*, pages 129–137, 2017.
- [236] JIAXIANG WU, CONG LENG, YUHANG WANG, QINGHAO HU, AND JIAN CHENG. Quantized convolutional neural networks for mobile devices. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4820–4828, 2016.
- [237] ZUXUAN WU, TUSHAR NAGARAJAN, ABHISHEK KUMAR, STEVEN RENNIE, LARRY S DAVIS, KRISTEN GRAUMAN, AND ROGERIO FERIS. Blockdrop: Dynamic inference paths in residual networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 8817–8826, 2018.

- [238] HONGWEI XIE, YAFEI SONG, LING CAI, AND MINGYANG LI. Overflow aware quantization: Accelerating neural network inference by low-bit multiply-accumulate operations. In *IJCAI*, pages 868–875, 2020.
- [239] SAINING XIE, CHEN SUN, JONATHAN HUANG, ZHUOWEN TU, AND KEVIN MURPHY. Rethinking spatiotemporal feature learning: Speed-accuracy trade-offs in video classification. In *ECCV 2018*, pages 305–321, 2018.
- [240] JIANXIN XIONG, JEREMY JOHNSON, ROBERT JOHNSON, AND DAVID PADUA. Spl: A language and compiler for dsp algorithms. *ACM SIGPLAN Notices*, 36(5):298–308, 2001.
- [241] MENGWEI XU, MENGZE ZHU, YUNXIN LIU, FELIX XIAOZHU LIN, AND XUANZHE LIU. Deepcache: Principled cache for mobile deep vision. In *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking, MobiCom '18*, page 129–144, New York, NY, USA, 2018. ACM, Association for Computing Machinery.
- [242] SHIZHEN XU, HAO ZHANG, GRAHAM NEUBIG, WEI DAI, JIN KYU KIM, ZHIJIE DENG, QIRONG HO, GUANGWEN YANG, AND ERIC P XING. Cavs: An efficient runtime system for dynamic neural networks. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, pages 937–950, 2018.
- [243] DANIEL LK YAMINS AND JAMES J DICARLO. Using goal-driven deep learning models to understand sensory cortex. *Nature neuroscience*, 19(3):356, 2016.
- [244] DANIEL LK YAMINS, HA HONG, CHARLES F CADIEU, ETHAN A SOLOMON, DARREN SEIBERT, AND JAMES J DICARLO. Performance-optimized hierarchical models predict neural responses in higher visual cortex. *Proceedings of the National Academy of Sciences*, 111(23):8619–8624, 2014.
- [245] BIN YANG, WENJIE LUO, AND RAQUEL URTASUN. Pixor: Real-time 3d object detection from point clouds. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, pages 7652–7660, 2018.
- [246] CHAO YANG, SHUMING CHEN, YAOHUA WANG, AND JUNYANG ZHANG. The evaluation of dcnn on vector-simd dsp. *IEEE Access*, 7:22301–22309, 2019.
- [247] LE YANG, YIZENG HAN, XI CHEN, SHIJI SONG, JIFENG DAI, AND GAO HUANG. Resolution adaptive networks for efficient inference. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2020.
- [248] SHUOCHAO YAO, SHAOHAN HU, YIRAN ZHAO, ASTON ZHANG, AND TAREK ABDELZAHER. Deepsense: A unified deep learning framework for time-series mobile sensing data processing. In *Proceedings of the 26th International Conference on World Wide Web, WWW '17*, page 351–360, Republic and Canton of Geneva, CHE, 2017. International World Wide Web Conferences Steering Committee, International World Wide Web Conferences Steering Committee.

- [249] SHAOKAI YE, XIAOYU FENG, TIANYUN ZHANG, XIAOLONG MA, SHENG LIN, ZHENGANG LI, KAIDI XU, WUJIE WEN, SIJIA LIU, JIAN TANG, ET AL. Progressive dnn compression: A key to achieve ultra-high weight pruning and quantization rates using admm. *arXiv preprint arXiv:1903.09769*, 2019.
- [250] DONG YU AND LI DENG. Deep learning and its applications to signal and information processing [exploratory dsp]. *IEEE Signal Processing Magazine*, 28(1):145–154, 2011.
- [251] JIAHUI YU, YUCHEN FAN, JIANCHAO YANG, NING XU, XINCHAO WANG, AND THOMAS S HUANG. Wide activation for efficient and accurate image super-resolution. *arXiv preprint arXiv:1808.08718*, 2018.
- [252] TOMOFUMI YUKI, VAMSHI BASUPALLI, GAUTAM GUPTA, GUILLAUME IOOSS, D KIM, TANVEER PATHAN, PRADEEP SRINIVASA, YUN ZOU, AND SANJAY RAJOPADHYE. Alphaz: A system for analysis, transformation, and code generation in the polyhedral equational model. *Colorado State University, Tech. Rep*, 2012.
- [253] YUJIA ZHAI, CHENGQUAN JIANG, LEYUAN WANG, XIAOYING JIA, SHANG ZHANG, ZIZHONG CHEN, XIN LIU, AND YIBO ZHU. Bytetransformer: A high-performance transformer boosted for variable-length inputs. *arXiv preprint arXiv:2210.03052*, 2022.
- [254] CHAOYUN ZHANG, PAUL PATRAS, AND HAMED HADDADI. Deep learning in mobile and wireless networking: A survey. *IEEE Communications Surveys & Tutorials*, 2019.
- [255] LI LYNA ZHANG, SHIHAO HAN, JIANYU WEI, NINGXIN ZHENG, TING CAO, YUQING YANG, AND YUNXIN LIU. nn-meter: towards accurate latency prediction of deep-learning model inference on diverse edge devices. In *Proceedings of the 19th Annual International Conference on Mobile Systems, Applications, and Services*, pages 81–93, 2021.
- [256] TIANYUN ZHANG, SHAOKAI YE, KAIQI ZHANG, JIAN TANG, WUJIE WEN, MAKAN FARDAD, AND YANZHI WANG. A systematic dnn weight pruning framework using alternating direction method of multipliers. *arXiv preprint arXiv:1804.03294*, 2018.
- [257] JIE ZHAO, BOJIE LI, WANG NIE, ZHEN GENG, RENWEI ZHANG, XIONG GAO, BIN CHENG, CHEN WU, YUN CHENG, ZHENG LI, PENG DI, KUN ZHANG, AND XUEFENG JIN. Akg: Automatic kernel generation for neural processing units using polyhedral transformations. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2021*, page 1233–1248, New York, NY, USA, 2021. Association for Computing Machinery.
- [258] BOJIAN ZHENG, ZIHENG JIANG, CODY HAO YU, HAICHEN SHEN, JOSHUA FROMM, YIZHI LIU, YIDA WANG, LUIS CEZE, TIANQI CHEN, AND GENNADY PEKHIMENKO. Dietcode: Automatic optimization for dynamic tensor programs. *Proceedings of Machine Learning and Systems*, 4:848–863, 2022.

- [259] AOJUN ZHOU, ANBANG YAO, YIWEN GUO, LIN XU, AND YURONG CHEN. Incremental network quantization: Towards lossless cnns with low-precision weights. In *International Conference on Learning Representations (ICLR)*, 2017.
- [260] JUN-YAN ZHU, TAESUNG PARK, PHILLIP ISOLA, AND ALEXEI A EFROS. Unpaired image-to-image translation using cycle-consistent adversarial networks. In *Proceedings of the IEEE international conference on computer vision*, pages 2223–2232, 2017.
- [261] KAI ZHU, WY ZHAO, ZHEN ZHENG, TY GUO, PZ ZHAO, JJ BAI, JUN YANG, XY LIU, LS DIAO, AND WEI LIN. Disc: A dynamic shape compiler for machine learning workloads. In *Proceedings of the 1st Workshop on Machine Learning and Systems*, pages 89–95, 2021.
- [262] VOJIN ZIVOJNOVIC, STEFAN PEES, CHRISTIAN SCHLAGER, MARKUS WILLEMS, RAINER SCHOENEN, AND HEINRICH MEYR. Dsp processor/compiler co-design: a quantitative approach. In *Proceedings of 9th International Symposium on Systems Synthesis*, pages 108–113. IEEE, 1996.

VITA

Wei Niu

Aug. 1995	Born - Yiyang, Hunan, China
Aug. 2012 - June. 2016	B.S. in Software Engineering, Beihang University, Beijing, China
Apr. 2016 - July. 2018	Mobile Application Developer, Bytedance, Ltd., Beijing, China
Aug. 2018 - Aug. 2023	Ph.D. in Computer Science, William & Mary, Williamsburg, VA, USA
May. 2021 - Aug. 2021	Full-time Research Intern, Bytedance, Ltd., CA, USA
Oct. 2021 - Apr. 2022	Part-time Research Intern, CoCoPIE, Inc., VA, USA

Wei Niu is a Ph.D. candidate in the Department of Computer Science at William & Mary under the supervision of Professor Bin Ren. Wei's research interests lie in real-time machine learning systems, mobile computing, parallel computing, and compilers. In particular, he focuses on achieving real-time DNN execution on mobile platforms with compiler optimizations. His work has appeared at top conferences (e.g., MICRO, PLDI, ASPLOS, RTAS, ICS, DAC, NeurIPS, CVPR, AAAI, ECCV, ICCV) and top journals (e.g., TPAMI, CACM). He is the recipient of the Stephen K. Park Graduate Research Award at William & Mary. He also won first place in the 2020 ISLPED Design Contest, the CACM Contributed Article Award in 2021, and the Best Paper Award at an ICLR workshop in 2021.