

Imperial College London
Department of Computing

Tools for Efficient Deep Learning

Ruizhe Zhao

Submitted in part fulfilment of the requirements for the degree of
Doctor of Philosophy in Computing of Imperial College London and
the Diploma of Imperial College London, October 2022

Abstract

In the era of Deep Learning (DL), there is a fast-growing demand for building and deploying Deep Neural Networks (DNNs) on various platforms. This thesis proposes five tools to address the challenges for designing DNNs that are efficient in time, in resources and in power consumption.

We first present Aegis and SPGC to address the challenges in improving the memory efficiency of DL training and inference. Aegis makes mixed precision training (MPT) stabler by layer-wise gradient scaling. Empirical experiments show that Aegis can improve MPT accuracy by at most 4%. SPGC focuses on structured pruning: replacing standard convolution with group convolution (GConv) to avoid irregular sparsity. SPGC formulates GConv pruning as a channel permutation problem and proposes a novel heuristic polynomial-time algorithm. Common DNNs pruned by SPGC have maximally 1% higher accuracy than prior work.

This thesis also addresses the challenges lying in the gap between DNN descriptions and executables by Polygeist for software and POLSCA for hardware. Many novel techniques, e.g. statement splitting and memory partitioning, are explored and used to expand polyhedral optimisation. Polygeist can speed up software execution in sequential and parallel by 2.53 and 9.47 times on Polybench/C. POLSCA achieves 1.5 times speedup over hardware designs directly generated from high-level synthesis on Polybench/C.

Moreover, this thesis presents Deacon, a framework that generates FPGA-based DNN accelerators of streaming architectures with advanced pipelining techniques to address the challenges from heterogeneous convolution and residual connections. Deacon provides fine-grained pipelining, graph-level optimisation, and heuristic exploration by graph colouring. Compared with prior designs, Deacon shows resource/power consumption efficiency improvement of 1.2x/3.5x for MobileNets and 1.0x/2.8x for SqueezeNets.

All these tools are open source, some of which have already gained public engagement. We believe they can make efficient deep learning applications easier to build and deploy.

Acknowledgements

I would like to express my sincere gratitude to my supervisor, Prof. Wayne Luk. I am lucky to be in his group from the summer of 2015, and have been well supported since then. Without his inspiration, I would not have been interested in hardware acceleration; and without his vision, I would not have the chance of working in the intersection of the most exciting domains of Deep Learning and compiler. Not to mention his overwhelming support in the last 7 years, starting from my summer internship, until the finish of this thesis. Besides these, I also have the most valuable opportunities to explore research ideas myself, some of which succeeded, and some of which failed, but the experiences I have gained from this journey are absolutely precious.

Second, I would like to thank the sponsorship from Corerain Technologies and UKRI. Without their vital financial support, I could not have started my PhD journey, and I would not have the chances to travel to different conferences and gaining industrial experiences. Particularly, I would like to express my gratitude to Dr. Xinyu Niu, the co-founder of Corerain Technologies who has been a role model to me the whole time of my PhD, for his inspiring thoughts on the field of study and his understanding of my situation the whole time. Other colleagues from Corerain Technologies, including Dr. Kuen Hung Tsoi and Dr. Chao Xiong, are also greatly appreciated for their support.

I am also privileged to work with many wonderful colleagues from the world. Thanks to Dr. Brian Vogel and Dr. Tanvir Ahmed for their supervision of my internship at Preferred Networks in 2019. Without their help and instructions, we could not have produced the IJCAI paper together. The collaboration with William Moses from MIT, Lorenzo Chelini from TU Eindhoven, and Alex Zinenko from Google Brain is an exceptional experience for me. Especially Alex, who has organised this cross-continental team and generously shared his expertise in polyhedral compilation with me, even at the time when I almost had no experience at all in the polyhedral domain. It will always be a great memory in my mind that we produce the Polygeist project altogether during the pandemic. Thanks to the colleagues from Imperial College London, including Dr. Ho-Cheung Ng, Nadeen Gebara, Jiuxi Meng, for

being great friends sharing the same office, and their support on various my research projects. I would also like to thank other colleagues who have helped me in any form: Dr. Shengjia Shao, Dr. Tim Todman, Dr. Nils Voss, Dr. Shuanglong Liu, Dr. Paul Grigoras, Dr. Andreea-Ingrid Cross, James Targett, and Dr. Jose Gabriel de Figueiredo Coutinho.

I would also like to thank my parents for their support of me studying abroad.

Last, my warmest thanks go to my wife for her support and understanding.

Declaration of Originality

This thesis is a presentation of my original research work. The contributions of others are involved, every effort is made to indicate this clearly in the references to the literature and acknowledgement of collaborative research.

Copyright Declaration

The copyright of this thesis rests with the author. Unless otherwise indicated, its contents are licensed under a Creative Commons Attribution-Non Commercial 4.0 International Licence (CC BY-NC).

Under this licence, you may copy and redistribute the material in any medium or format. You may also create and distribute modified versions of the work. This is on the condition that: you credit the author and do not use it, or any derivative works, for a commercial purpose.

When reusing or sharing this work, ensure you make the licence terms clear to others by naming the licence and linking to the licence text. Where a work has been adapted, you should indicate that the work has been changed and describe those changes.

Please seek permission from the copyright holder for uses of this work that are not included in this licence or permitted under UK Copyright Law

Contents

Abstract	iii
Acknowledgements	v
Declaration of Originality	vii
Copyright Declaration	ix
Contents	xi
List of Tables	xvii
List of Figures	xxi
Acronyms	xxix
Glossary	xxxix
1 Introduction	1
1.1 Motivation	1
1.2 Challenges	2
1.2.1 C1: Memory-efficient training and inference	3
1.2.2 C2: Polyhedral compilation framework for multi-level representations	4
1.2.3 C3: FPGA accelerator design for efficient DNN architecture .	4
1.3 Contributions	5
1.3.1 Mixed precision training and structured pruning	5

1.3.2	MLIR-based polyhedral compilation framework	6
1.3.3	DNN acceleration by streaming architecture	7
1.3.4	Summary	8
1.4	Publications	9
1.5	Thesis organisation	10
2	Background	11
2.1	Deep Learning	12
2.1.1	Convolutional Neural Network	12
2.1.2	Efficient inference	18
2.1.3	Mixed Precision Training	22
2.2	Reconfigurable computing	28
2.2.1	Reconfigurable devices	29
2.2.2	High-level design methodologies	34
2.2.3	Deep learning on reconfigurable platforms	37
2.3	Polyhedral Modelling	40
2.3.1	Preliminaries	41
2.3.2	Polyhedral representation	42
2.3.3	Polyhedral scheduling	44
2.3.4	Polyhedral code generation	47
2.3.5	Polyhedral compilers and tools	48
2.4	The MLIR framework	49
2.4.1	LLVM and MLIR	50
2.4.2	Dialects	51
2.4.3	Applications	54
2.5	Summary	55

3	Memory-Efficient Deep Learning	56
3.1	Introduction	56
3.2	Mixed Precision Training by Gradient Scaling	59
3.2.1	Estimate Gradient Scale	59
3.2.2	The Gradient Scaling Algorithm	62
3.2.3	Experiments	65
3.2.4	Discussions	70
3.3	SPGC: Structured Pruning by Group Convolution	73
3.3.1	Group Convolution Pruning	73
3.3.2	Prune a Layer	76
3.3.3	Optimise Group Configuration	79
3.3.4	Experiments	80
3.3.5	Discussions	87
3.4	Summary	88
4	Polyhedral Compilation Infrastructure	89
4.1	Introduction	89
4.2	Polygeist: Affine C/C++ in MLIR	91
4.2.1	Frontend	92
4.2.2	Raising to Affine	94
4.2.3	Post-Transformations and Backend	95
4.2.4	Evaluation	96
4.2.5	Discussions	100
4.3	Polymer: Polyhedral Scheduling in MLIR	101
4.3.1	Statement mapping	101
4.3.2	Controlling statement granularity	104
4.3.3	Comparison to Polly	106
4.3.4	Evaluation	107
4.3.5	Discussions	113

4.4	POLSCA	115
4.4.1	Methodology	118
4.4.2	Evaluation	126
4.4.3	Discussions	131
4.5	Summary	133
5	Dataflow DNN accelerator	134
5.1	Introduction	134
5.2	Streaming Architecture Design	137
5.2.1	Design Overview	140
5.2.2	Processing Engines	140
5.2.3	Computation Sequence	141
5.2.4	Inter-Layer Pipelining Techniques	142
5.3	Graph-Level Optimisation	145
5.3.1	Depthwise Separable Convolution	145
5.3.2	Split and Concatenation	147
5.3.3	Residual Connection	149
5.4	Design Space Exploration	150
5.4.1	Design Model	151
5.4.2	Optimisation Algorithm	155
5.5	Experiments	158
5.5.1	Compare with Existing Work	158
5.5.2	Empirical Study on Optimisation Techniques	162
5.6	Discussion	163
5.7	Summary	163
6	Conclusion	165
6.1	Summary of Thesis Achievements	165
6.2	Future Work	167
6.3	Final Thoughts	168

List of Tables

2.1	Comparison on the time and space (weight size) asymptotic complexity among standard, depthwise, and pointwise convolution layers. . .	14
2.2	Time and space complexity for different convolution blocks. C_i is the number of channels of the intermediate feature maps in between layers.	15
2.3	Efficient CNN models statistics. Top-1 (%) indicates the top-1 recognition accuracy on ImageNet ILSVRC 2012 [248].	17
2.4	Comparison among CLP, layer fusion (LF), and SA. Resource Eff. indicates the resource efficiency that different architectures can achieve, and among the listed CLP can be the most efficient; Off-Chip Trans. lists the bandwidth requirements of memory transactions between FPGA and off-chip memory, of which SA demands the least.	39
2.5	Comparison on functionalities with prior frameworks. LF as a design target means layer fusion, and OVL is for CLP equipped with overlay. ILP stands for inter-layer pipeline optimisation in SAs, DWS and RES denote whether a framework targets or optimises for depthwise separable and residual convolutions respectively. The last column specifies the design space to explore, PAR includes parallelism, and SEQ is computation sequence	40
3.1	Notations used for describing Aegis (Section 3.2).	59
3.2	ResNet-18/50 results on the ILSVRC 2012 dataset.	67
3.3	SSD-300/512 training results. BS denotes batch size.	68
3.4	SegNet MPT test results on CamVid.	69
3.5	Evaluation on CIFAR-10 (C10) and CIFAR-100 (C100). Number of parameters (# Param.) and operations (# FLOPS) are measured on CIFAR-100. Note that we consider one FMA as two operations. Pruned ResNet-165 models are labelled by $X\%$ <i>pruned</i> , which indicates the ratio of pruned channels as in [180]. When pruning by our method, since we cannot prune away a whole channel, we only take the number of parameters of the corresponding models from [185] as the optimisation constraint.	81

3.6	ImageNet evaluation results. The first sample in each section is the baseline.	83
3.7	CUB-200 evaluation results.	84
3.8	Inference speed of MobileNet-224. Batch size equals to 1 for all cases. CPU and GPU denotes the Xeon and 1080Ti on the rack server and TX2 refers to the embedded GPU on Jetson TX2.	85
3.9	Comparison of the top-1 test error (%) between using the <i>heuristic</i> algorithm and not using (<i>plain</i>). ResNet-50 results are collected on ImageNet.	85
3.10	Comparison between pruning and training from scratch for ResNet-164 on CIFAR-100 regarding top-1 train and test error.	86
4.1	Type correspondence between C, LLVM IR and MLIR types [214]. . .	94
4.2	Hardware setup.	97
4.3	Evaluation of our approach on a set of benchmarks. NA stands for whether an example contains non-affine loop nests or not. The <i>base</i> column covers the designs directly synthesised using Polygeist, the <i>poly</i> column covers the designs directly synthesised using the MLIR from polyhedral, and <i>ours</i> covers the designs synthesised using the code from Polygeist. Geom. Mean measures the geometric mean of values from the baseline over other corresponding scenarios. . . .	128
5.1	SA PE configurations. This table only shows when a PE corresponds to a single layer. In cases that a PE processes multiple layers together, these parameters will be duplicated and certain constraints apply. . .	139
5.2	Buffer usage of a PE that follows different computation sequences. Table 5.1 lists the definitions of the parameters.	141
5.3	Extra buffer usage and parallelism constraints imposed by different residual connection implementations.	146
5.4	Xilinx UltraScale+ VU9P resource capacity.	156

- 5.5 Comparing with results from related works on MobileNet v1 and v2. Numbers in bold are considered the best among their corresponding groups. (a) and (b) versions of [151] refer to the single instance and the two instances designs. (S) and (L) in [306] target small (ZU2) and large (ZU9) FPGAs. DSPs refer to the number of DSP Slices with 27x18 multipliers for Xilinx boards and the number of 27x27 multipliers for Arria 10 FPGAs. BRAMs are of 36Kb per block on Xilinx FPGAs (BRAM36K) and of 20Kb per block on Arria 10 FPGAs (M20K). Memory bandwidth (Mem. BW) is calculated by multiplying the data transfer amount per frame (GB/frame) with frame rate (FPS). Deacon designs only requires to transfer the initial input and final output, while other designs should transfer the input, output, and weights of each layer. We also normalise (listed as norm.) the FPS/DSP by scaling performance to match 200 MHz clock frequency. 157
- 5.6 SqueezeNet v1.1 performance comparison. The definition of rows DSPs, BRAMs, and Mem. BW are the same as in Table 5.5. 161
- 5.7 ResNet-18 performance comparison. We list three results from Deacon here, (a), (b), and (c), in which (a) is the baseline without parallelism, (b) is constrained by around 2k DSP blocks, and (c) is capped by 5k. DSPs, BRAMs, and Mem. BW are the same as in Table 5.5. 161

List of Figures

1.1	An overview of a deep learning application build and deployment procedure. $\mathcal{D}$ is the prepared dataset; $\mathcal{M}$ and $\mathcal{M}^*$ are the deep learning models, before and after training; $\mathcal{P}$ denotes the compiled program; and $\mathcal{H}$ is the hardware design generated.	1
1.2	Thesis structure. The middle layer lists the three challenges, labelled by C1, C2, C3. The bottom layer demonstrates all the corresponding solutions.	2
2.1	The shape of a single image in a feature map.	13
2.2	An illustration of the 2D convolution operator $*$. Here, $\mathbf{I}$ is a single channel of a single image from $\mathbf{X}$, and $\mathbf{K}$ is a single kernel from the weights $\mathbf{W}$	13
2.3	Illustrations of different convolution blocks.	15
2.4	Illustration of a group convolution with $G = 2$, which processes two groups of the input feature map, split by half along the channel axis.	19
2.5	The steps of network pruning.	20
2.6	The general abstracted procedure of neural architecture search, following the framework proposed in [66].	21
2.7	Illustration of the representable range for single-precision floating-point (fp32) and half-precision (fp16). The range of fp16 is much narrower than fp32. The values out of the representable range can be underflow, if close to zero, or overflow, if too large. Those vertical bars mark the minimum and maximum values for fp32 (above) and fp16 (below). The scale of the axis is roughly logarithmic.	22

2.8	The feedforward and backpropagation computation graphs of the same DNN consisting of one convolution layer and one matrix multiplication operation, both of which are connected to their weights, $\mathbf{W}_1$ and $\mathbf{W}_2$. The feedforward graph propagates the input feature map $\mathbf{X}$, weights, and intermediate outputs, $\mathbf{Y}_1$ and $\mathbf{Y}_2$, top down until reaches the loss function. And the backpropagation graph propagates from the bottom, and the data being propagated are the gradients of the loss value. The ∇ symbol before each operator in the backpropagation graph indicates that it calculates gradients by automatically differentiating the feedforward definition.	25
2.9	The mixed precision version of a matrix multiplication layer. Weights are typecast from fp32 to fp16 in the beginning and then passed to Matmul. Matmul takes fp16 input feature map and weights, and produces fp16 output. During backpropagation, the gradients of the output, ∇ Output, and the original input, are passed to ∇ Matmul as fp16 , to get the gradients updates for weights, also in fp16 . This mixed precision operator finally typecasts the gradient update to fp32 and accumulates it to the weights, which are stored as fp32	26
2.10	An illustration of an FPGA architecture from Kastner et al. [142, Figure 1.3].	31
2.11	Visualizing the roofline model. Operational intensity is on the x-axis (with unit flop/B), and the attainable performance (flop/s) is on the corresponding y-axis. There is a ‘roofline’ that captures both the constraints by memory bandwidth (the sloped line) and peak (floating-point) computation performance (the horizontal line) of a processor. The angle of the sloped line depicting memory-bound performance is the off-chip memory bandwidth (B/s).	34
2.12	Generic MLIR syntax depicting an operation with two operands, one result, one attribute and a single-block region. The types of the operation and the block arguments are annotated as well.	50
3.1	SSD MPT details using different scaling methods [341].	69
3.2	Mean gradient scales for all layers of SSD-300/512 across all training iterations. Layer are labelled from input to output in topological order. Black lines are error bars [341].	70
3.3	Details in the SegNet training procedure [341].	71
3.4	Two variants of GConv that are different in information exchange mechanism [340].	74
3.5	This figure shows $\mathcal{W}$ that is being sorted. It is running at the 3rd block (green). Kernels covered by previous blocks (red) are now frozen for further sorting (gray). When updating the 3rd block, input channels (columns) are sorted by $\sigma_c(c, 3)$ in range ① and output channels (rows) are sorted by $\sigma_f(f, 3)$ in range ② [340].	77

3.6	Evaluation results on the heuristic algorithm. Recovery ratio is the percentage of magnitude, measured by L2 norm, retained in the diagonal blocks after pruning [340].	78
3.7	Change in ResNet-164 on CIFAR-100 accuracy while reducing number of parameters. Each sample point is collected from the model found by our model pruning algorithm under the given number of parameters constraint. Manually configured models either use uniform group size G or set a maximum channel number G_M . The latter approach assigns each layer a group number that ensures $\max(C_{in}/G, C_{out}/G) \leq G_M$. We also list the estimated pruning cost for each explored sparsity configuration. Accuracy numbers listed here are collected from <i>fine-tuning</i> [340].	83
4.1	The Polygeist compilation flow consists of 4 stages. The frontend traverses Clang AST to emit MLIR SCF dialect (Section 4.2.1), which is raised to the Affine dialect and pre-optimized (Section 4.2.2). The IR is then processed by a polyhedral scheduler (Sections 4.3, 4.3.2) before post-optimization and parallelization (Section 4.2.3). Finally, it is translated to LLVM IR for further optimization and binary generation by LLVM [214].	91
4.2	Polygeist detects memory locations accessed in all loop iterations, e.g. reduction accumulators such as <code>%r1[0]</code> and transforms them to loop-carried values (secondary induction variables), except when computed with side-effects, interleaved stores or by non-associative/commutative operations [214].	96
4.3	Mean and 95% confidence intervals (log scale) of program run time across 5 runs of Polybench in CLANG, CLANGSING and MLIR-CLANG configurations, lower is better. The run times of code produced by Polygeist without optimization is comparable to that of Clang. No significant variation is observed between single and double optimization. Short-running <code>jacobi-1d</code> shows high intra-group variation [214].	99
4.4	Median speedup over CLANG for sequential configurations (log scale), higher is better. Polygeist outperforms ($2.53\times$ geomean speedup) both Pluto ($2.34\times$) and Polly ($1.41\times$) on average. Pluto can't process <code>adi</code> , which is therefore excluded from summary statistics [214].	109
4.5	Median speedup over CLANG for parallel configurations (log scale), higher is better. Polygeist outperforms ($9.47\times$ geomean speedup) both Pluto ($7.54\times$) and Polly ($3.26\times$) on average. Pluto can't process <code>adi</code> , which is therefore excluded from summary statistics [214].	110
4.6	Excerpt from the <code>deriche</code> benchmark. The outer loop reuses <code>ym1</code> which makes it appear non-parallel to affine schedulers (left). Polygeist detects parallelism thanks to its mem2reg optimization, reduction-like loop-carried <code>%ym1</code> value detection and late parallelization (right) [214].	111

4.7	Mean and 95% confidence intervals of run time across 5 runs of Polybench where statement splitting is applicable (Section 4.3.2), lower is better. It results in faster run time (geomean $1.28\times$ sequential, $1.39\times$ parallel speedup) except for sequential <code>2mm</code> (-4%) and parallel <code>trmm</code> (-9%) [214].	112
4.8	Reduction parallelization allows POLYGEISTPAR to produce larger speedups in <code>durbin</code> and at smaller sizes than POLLYPAR, POLYGEISTPAR without reduction support. PLUTOPAR fails to parallelize, hence no speedup [214].	113
4.9	Summary of our contributions (right) in polyhedral HLS compared with existing methods (left).	116
4.10	Jacobi-1d algorithm. We can either annotate a big SCoP over the outermost loop <code>L0</code> , or two small SCoPs covering inner loops <code>L1</code> and <code>L2</code> . Memory accesses to arrays <code>A[N]</code> and <code>B[N]</code> make <code>L0</code> and <code>L1</code> inter-dependent.	118
4.11	Demonstrating the SCoP decomposition algorithm. In this particular example, we have 4 consecutive loops at the same depth, and there are interdependences (arrows) among loop 1 and 2, 1 and 3, 2 and 4, 3 and 4. The minimum cuts required for SCoP covering loop 2 and 3 is 0, which will propagate through Eq. (4.1) and lead to the optimal solution as shown that uses 2 cuts (dashed horizontal lines).	119
4.12	Sketch of polyhedral-transformed matrix multiplication. The loop nest initially has a depth of 3, and each loop has been tiled by a factor of 32 as listed here. The 3 outer loops <code>i</code> , <code>j</code> , <code>k</code> are referred to as tile loops, and the inner loops <code>ti</code> , <code>tj</code> , <code>tk</code> are point loops. After PE outlining, the inner 3 point loops are wrapped into a new function <code>PE</code>	121
4.13	A memory partitioning example that applies partition factor reconciliation and PE interface extension.	123
4.14	The overall workflow of POLSCA. Our contributions are highlighted in the dashed region.	127
5.1	Key concepts in Deacon: the Deacon framework is listed in the left, and the core graph-level optimisations are demonstrated in the right.	135

- 5.2 These two figures illustrate how computation sequences affect pipelining performance. Each rectangle indicates a channel to process in each layer, and its text indicates the output and the input channels (0-indexed). The shaded rectangles show when the corresponding output channel is ready, and the arrows indicate the producer-consumer dependency. For instance, the rectangle with (0, 2) means the PE is processing the third channel of the input image for the first output channel. When channel-major follows filter-major (top-left), there is no pipeline stall. Whenever the 2nd layer needs an input, it will be ready from the 1st layer. If two filter-major layers are adjacent, the 2nd layer will be stalled to wait for the 1st layer to produce output. 138
- 5.3 SA for a DNN. There are three types of convolution PEs: the left processes a single convolution layer, the middle one can process two layers that can be unified (e.g., depthwise separable convolution), and the right one demonstrates the layer with a residual connection. . . . 139
- 5.4 A base convolution layer design. The definition of the symbols are listed in Table 5.1. On-chip memory resources are used by all the buffers and weights ROM (read-only memory). DSPs are consumed by the arithmetic units. The two FIFOs indicate input and output streams. 139
- 5.5 Comparison among different CompSeq configurations for two (left) or three (right) consecutive PEs. Every PE consumes and produces feature maps of the same shape. Among x-axis labels, F stands for filter-major, C denotes channel-major, and a sequence of {F, C} indicates the computation sequences of all PEs, from top to bottom. . . 143
- 5.6 Pipeline improvement by interleaving computation sequences. With interleaved computation sequences, the 2nd image can be processed immediately after the 1st image. T_{old} and T_{new} indicate the latency between the 1st image and the 2nd image before and after the interleaving computation sequence technique is applied. 143
- 5.7 A special case that interleaved computation sequence fails. The DNN structure is shown on the left, in which the 5-tuple indicates the (H, W, C, F, S) configuration of a layer. CONV 1 and POOL 1 both have 2-stride, which scales the spatial dimensions by 16 times when reaching CONV 2. CONV 2 expands the filters by 16 times to have the same processing time as CONV 1 ($T_1 = T_2$). While CONV 2 is running, the output of CONV 1 is halted since the FIFO is full. . . . 144

5.8	Optimised DWS convolution PE. When counter $f = 0$, this PE processes depthwise convolution, taking weights from the corresponding ROM and reading from the line buffer; its results are accumulated horizontally from the 2D multiplier array and feeds into the input buffer, which will be processed in the following iterations when $f > 0$ to produce pointwise convolution results. Final results from pointwise are vertically reduced from those element-wise multiplications.	146
5.9	Split and concatenation with buffer sharing. The input buffer is being shared by the two split convolution layers. There is a FIFO that delays the processing of the pointwise (1x1) convolution since it runs faster than the 3x3 convolution, and we need to make sure they process at the same speed to ensure we concatenate the right pairs in the end.	147
5.10	Graph-level optimisation for residual connections.	148
5.11	Pipeline balancing results of the best Deacon designs. From left to right are the latency of pipeline stages from the top to the bottom.	159
5.12	Empirical study on Deacon optimisation techniques.	162

List of source codes

- 1 An example code to Vitis HLS. This function implements the dot-product calculation among 10 different vectors of length 32. Vitis HLS directives are represented using the C compiler pragmas, specifically, `ARRAY_PARTITION` instructs the compiler to expand the first dimension of every input array into separated 32-length partitions so that they can be accessed in parallel, `UNROLL` replicates the inner loop (`Loop2`) by 10 copies, and `PIPELINE` indicates the innermost loop should be implemented by pipelines. The resulting design has 10 replicates of a pipelined dot-product processing unit that operate in parallel on different partitions of the input arrays. 36
- 2 An example SCoP. All the loops have affine boundaries, the if-conditional describes a quasi-affine constraints, and the statements `S1`, `S2` contains only affine addresses. `pragma scop` and `pragma endsco` mark the begin and the end of a SCoP when using PET [289]. 43
- 3 An example that cannot be rectangularly tiled. An intuitive explanation is that, every row `t` in each rectangular tile depends on the results from the previous row `t − 1` in adjacent tiles, e.g., the leftmost column in a tile depends on the result from the neighbour tile on the left `A[t − 1][i − 1]`, and the rightmost column depends on elements in the right tile `A[t − 1][i + 1]`, which suggests that we cannot finish the computation of a tile as a whole without interfering with other tiles, and therefore, the rectangular tiling is invalid. 45
- 4 Code transformed by Pluto from Listing 3. The original loops are skewed in this case. The `pragma omp` is added to annotate the tile loop is parallelisable. 46
- 5 Polynomial multiplication in MLIR using Affine and MemRef dialects. 52
- 6 This GConv is implemented as a PyTorch [225] `Module`, which performs computation in the `forward` method. We permute (`index_select`) the input `x` by `I` (`ind_in`) along the channel axis and the output by `Jout` (`ind_out`). Convolution groups (`conv`) takes input groups (`split`) and concatenate (`cat`) their results. 73
- 7 Example demonstrating Polygeist frontend. For functions expected to be compiled with Polygeist such as `setArray`, pointer arguments are replaced with `memref`'s [214]. 93

8	Polygeist breaks region-spanning use-def chains and handles multi-use values by introducing scratchpad storage when operation duplication is illegal. In absence of <code>motion-barrier</code> statement, the <code>%0</code> load would be duplicated and sunk. Pseudo-MLIR with types and braces omitted for brevity [214].	103
9	Outlining makes polyhedral “statements” visible in code from Listing 8 [214].	104
10	Splitting a nested reduction statement (top) into a fully parallel compute statement and a trivial reduction statement (bottom left) makes Pluto generate different schedules (bottom right). Further scratchpad array expansion may enable loop fission and give scheduler even more liberty [214].	105
11	A motivating example for POLSCA. It is modified from <code>2mm</code> [330] with additional indirect memory addresses that create non-SCoPs. There are four separated loop nests below the outermost L0: L1 gathers data from <code>array</code> into <code>D</code> through a hash table <code>addrIn</code> , L2 and L3 run two GEMMs (generalised matrix multiplication) <code>tmp = alpha * A * B</code> and <code>D = beta * D + tmp * C</code> , and L4 distributes <code>D</code> back to <code>array</code> using another hash table <code>addrOut</code>	117

Acronyms

ALM Adaptive Logic Module. 30

ASIC Application-Specific Integrated Circuit. 28

BRAM Block Random Access Memory. 30

CLB Configurable Logic Block. 30

CLP Convolution Layer Processor. 37

CNN Convolutional Neural Network. 12

CPU Central Processing Unit. 1, 28

DNN Deep Neural Network. 2, 28, 89

DRAM Dynamic Random Access Memory, which is used interchangeably with the term ‘off-chip memory’ in this thesis. 34

DSE Design Space Exploration. 34, 35

DSP Digital Signal Processing. 30

EDA Electronic Design Automation. 2, 28

FF Flip-Flop. 29, 32

FIR Finite Impulse Response. 17

FPGA Field-Programmable Gate Array. 2, 28

FSM Finite State Machine. 29

GPU Graphics Processing Unit. 1, 26, 28

HDL Hardware Description Language. 32

HLS High-Level Synthesis. 7, 28, 34, 36, 114

IR Intermediate Representation. 50

LUT Look-Up Table. 29, 32

MPT Mixed Precision Training. 22

MUX Multiplexer. 29

NAS Neural Architecture Search. 21

PE Processing Element. 33

RTL Register-Transfer Level. 7, 28, 32, 33, 36

SA Streaming Architecture. 37

SCoP Static Control Part. 51, 91

SGD Stochastic Gradient Descent. 24

SIMD Single-Instruction Multiple-Data. 30, 33

SRAM Static Random Access Memory. 29

SSA Static Single Assignment. 50

URAM UltraRAM. 30

Glossary

- automatic differentiation** An algorithm that generates the process to evaluate gradients from an input program based on the chain rule. 25
- backpropagation** An algorithm that trains a DNN by propagating gradients from the back, i.e., the loss function, using the automatic differentiation technique. 5, 25
- bottleneck block** A convolution block that inserts a standard convolution layer between two pointwise convolution. 16
- convolution block** A composition of multiple convolution layers. 5, 15
- Convolutional Neural Network** A special type of neural network builds upon convolution layers and most commonly used for computer vision tasks, e.g., image recognition and object detection. 12
- depthwise convolution layer** A standard convolution layer without cross-channel correlation, i.e., there is no accumulation across input channels. 14
- depthwise separable block** A convolution block that has a depthwise convolution followed by a pointwise convolution. 16
- feature map** A 4-D tensor representing a collection of images with the same number of channels, height, and width. 12
- gradient descent** An optimisation algorithm that iteratively updates parameters by the gradient of the objective function to be minimised. 24
- group convolution layer** A set of standard convolution layers, each of which processes a non-intersecting group of the channels of the input feature map. 6, 15, 18
- latency** The time taken to complete a single operation. 33
- layer fusion** A DNN performance optimisation technique that fuses multiple operators into a single operator, a major benefit of which is to reduce inter-operation memory transactions. 38
- loss scaling** A technique that scales up loss value before backpropagation to enlarge the magnitude of gradient values, in order to reduce the underflow rate. 27

mixed precision training A model training methodology that uses a mixture of different data precisions to store the parameters and carry out computation. 22

network pruning A technique to remove unimportant parameters from a pre-trained model without hurting much its accuracy. 3, 20

neural architecture search A searching procedure for an optimal DNN design under certain requirements. xxi, 21

overlay An FPGA technique that instantiates a virtual reconfigurable architecture above the physical FPGA to enable fast compilation and reconfiguration. 38

pipeline A design pattern that interleaves combinational logic with registers to improve throughput. 33, 36

pointwise convolution layer A standard convolution layer without spatial correlation, i.e., the kernel size is 1-by-1. 14

polyhedral model . 4, 37

replication A design optimisation technique that replicates the same processing element multiple times to increase the throughput. 33, 36

Roofline model A technique to model the peak performance of a hardware design given computation capability and memory bandwidth. 34, 35, 37, 131

separable bottleneck block A variant of the bottleneck block, which replaces the inner standard convolution with a depthwise convolution layer. 16

stacked convolution block A convolution block with multiple (typically 2) standard convolution layers. 16

standard convolution layer A layer that performs 2D convolution between every channel of the input feature map and every weight kernel and accumulates the results per channel of the output feature map. 12

Static Control Parts Loop nests that can be represented by polyhedral models. 40, 42

stochastic gradient descent A gradient descent algorithm using gradients calculated on stochastically selected subset of training data. 24

swamping An error in floating-point arithmetic that adding a big number with a small number significantly truncates the smaller one. 26, 28

throughput The number of operations can be done within a unit of time. 33

Chapter 1

Introduction

This chapter gives a general introduction to this thesis. We describe our motivations, corresponding challenges, and our contributions that aim to resolve such challenges.

1.1 Motivation

Deep learning [85] is being pervasively used in many applications nowadays. It powers smart cameras that can recognize objects in real-time [82, 184], translates hundreds of human languages under a unified framework [308], and game agents that can reach [211] and even exceed the best human player [257]. Despite all these merits, the huge demand for computation resources hinders broader usage of deep learning methods: people cannot develop their deep learning applications without access to large data centres with numerous GPUs, not to mention deploying them on small, hand-held devices. Being motivated by this situation, this thesis aims to provide practical solutions to make deep learning more accessible.

Even though CPUs and GPUs are the most common deployment platforms for deep

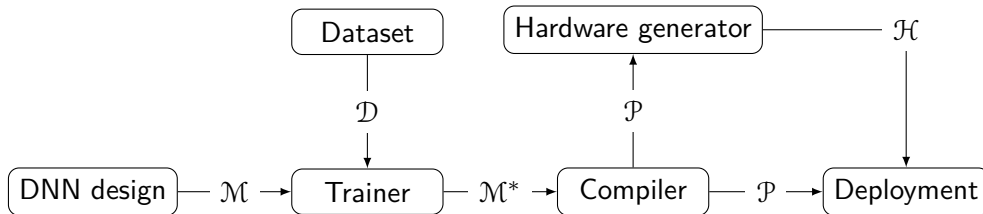


Figure 1.1: An overview of a deep learning application build and deployment procedure. $\mathcal{D}$ is the prepared dataset; $\mathcal{M}$ and $\mathcal{M}^*$ are the deep learning models, before and after training; $\mathcal{P}$ denotes the compiled program; and $\mathcal{H}$ is the hardware design generated.

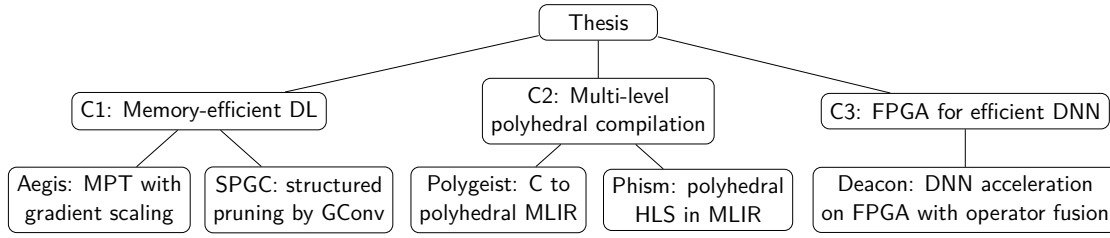


Figure 1.2: Thesis structure. The middle layer lists the three challenges, labelled by C1, C2, C3. The bottom layer demonstrates all the corresponding solutions.

learning applications, this thesis focuses on an alternative target: reconfigurable devices, or FPGAs. CPUs and GPUs are general-purpose processors, and they sacrifice time and energy for their generality. Empowered by the reconfigurability, people can build customised circuitry on FPGA that is just used for one specific deep learning application to push further the boundary of efficiency. Nevertheless, the EDA-flavoured toolchains for FPGA development make it a less favourable platform than CPU and GPU for deep learning engineers that come with a software background.

There are three steps to take to build and deploy a deep learning application. First, after understanding the application requirements, an engineer should design a model based on Deep Neural Network (DNN), the fundamental architecture for deep learning. This DNN should be properly trained on a dataset and improved until its evaluation performance meets the expectation. Second, once a trained model is produced, we need to compile it into an executable program on the target platform. It is desirable if the compiled program can run fast and fully leverage the functionality that the target platform provides. Finally, an extra step should be taken to leverage the customisability if the target platform is FPGA: we should generate a specific hardware design that is dedicated to executing the provided DNN in an optimised, efficient manner. Figure 1.1 illustrates the whole procedure.

1.2 Challenges

There are three specific challenges that this thesis aims to address following the aforementioned motivation. An overview of the challenges and their corresponding contributions (Section 1.3) can be found in Figure 1.2.

1.2.1 C1: Memory-efficient training and inference

The most capable DNNs typically require a substantial memory footprint during training due to the need to store a remarkable quantity of parameters. This creates a barrier for those who do not have access to multiple, high-end GPUs, thus limiting the potential benefits. An effective method for diminishing the memory footprint is mixed-precision training. This technique employs low-precision data types with shorter bit-widths to represent parameters that are not sensitive to precision, thereby significantly reducing memory usage compared to default single-precision floating-point representations, without detrimentally impacting performance. Nevertheless, this approach has one primary challenge: low-precision data types exhibit a limited representation range, which can lead to unexpected numerical issues during training. These issues can either interrupt the training procedure or lead to a degradation in accuracy. One instance, highlighted by [209], reveals that when training a DNN using half-precision floating-point, approximately 50% of non-zero activation gradients become zero due to the constrained representation range, adversely affecting training outcomes. Current methodologies [209, 159, 208] can partially address this issue using meticulously adjusted heuristics. However, a more desirable solution would be one that can be generalized across various DNN models without extensive hyperparameter tuning.

In addition to training, DNN inference performance can also deteriorate when confronted with an extensive memory footprint. This issue is amplified on deployment platforms, such as mobile phones and embedded devices, which typically have resource constraints. A frequently utilised solution is the network pruning, which eliminates parameters that contribute negligibly to model accuracy. This pruning strategy reduces the memory footprint while ensuring minimal variations in the model output. Nonetheless, the removal of parameters in an arbitrary manner could result in models exhibiting irregular sparsity, thereby impinging on execution efficiency. This irregularity stems from an uneven workload distribution among multiple processors and irregular memory references. As depicted in Mao et al. [198], given the same compression level, a model utilizing coarse-grained, structured pruning only requires 65.3% of the memory references compared to one using fine-grained, irregular pruning. However, the challenge lies in devising a method that can simultaneously remove a significant number of parameters while maintaining a structured model architecture.

1.2.2 C2: Polyhedral compilation framework for multi-level representations

A DNN is a specific type of program that supports a restricted set of operations, interconnected via a *computational graph* [2]. Each node in this graph represents a distinct operation such as matrix multiplication or convolution, while each edge signifies dataflow between nodes. This high-level abstraction can encapsulate a wide range of DNN architectures. However, it also leaves a substantial optimisation gap that the compiler must address to generate an efficient executable program.

A compiler mechanically transforms the input program from one representation to another, implying the need to automate the optimisation of the DNN computational graph. As the majority of operations in DNNs are grounded in nested loops with static control-flow, the polyhedral model can be adapted to automate loop transformations, enhancing parallelism and locality.

Several studies [231, 207, 281, 280] have integrated the polyhedral approach into their comprehensive, deep learning-oriented frameworks. These have yielded promising results, but have not yet addressed FPGA deployment needs. Moreover, owing to confidentiality or internal integrated design, these frameworks cannot be simply extended to support FPGAs.

For instance, our experiments revealed that connecting an existing polyhedral compiler, such as Pluto [30], with a high-level synthesis tool led to significant performance degradation. On the Polybench/C benchmark [330], average wall-time inflated by a factor of 4.08 compared to not using polyhedral transformations.

It remains a challenge to create a compiler framework that can effectively interface the polyhedral representation with high-level DNN graphs, and further transform them into executable programs, including FPGA designs, with adequate extensibility.

1.2.3 C3: FPGA accelerator design for efficient DNN architecture

The FPGA has emerged as a potential platform for executing DNN inference tasks on resource-constrained devices. Leveraging its reconfigurability, an FPGA can offer a solution that ideally balances the trade-off among processing speed, power consumption, and resource utilization. However, as DNN architectures evolve towards higher efficiency, the emergence of fragmented structures using interconnected

convolution blocks [48, 118] has increased design complexities for custom FPGA accelerators, in comparison to past works [331, 332, 44, 136, 277] that focused on more generalized DNN models.

One major challenge in FPGA deployment pertains to the design of convolution blocks within the desired accelerator. Rather than accelerating an isolated type of layer, processing an entire block concurrently appears to be a more efficient approach. This can minimise data transfer overhead between the FPGA and host, i.e., only twice per block instead of multiple instances for each sub-layer within the block. Furthermore, a deeper pipeline can enhance throughput.

This methodology, termed as *operator fusion*, has found applications not only in FPGA [197, 10, 182] but also GPU domains [187, 220]. However, the potential combinations of layers within a block can be vast, indicating that a universal template for operator fusion on FPGA may not exist. Thus, our focus should be on discovering a general strategy that can navigate the search space for an optimally fused design.

1.3 Contributions

This thesis contributes novel solutions to address the three previously mentioned challenges.

1.3.1 Mixed precision training and structured pruning

We propose two methods that address the problems with memory-efficient deep learning: Aegis for the numerical issues in mixed precision training, and SPGC for the imbalanced workload in pruned inference models.

Aegis approaches the numerical issues caused by the narrow representation range of low-precision data types by scaling the gradient per layer during backpropagation. Gradients have a rather wide range of values, and by scaling them we can control them within the range that our target data type can represent. In contrast to existing methods [209, 159, 208] that scale the gradients using a singular factor, Aegis calculates the scale factor for each gradient per layer during training, which is more versatile and stabler since the data distributions of gradients can vary significantly, both *spatially* (across different layers) and *temporally* (across different training iterations). To achieve this, Aegis modifies the backpropagation algorithm

to use gradient statistics to estimate the optimal scale factor leveraging probabilistic models. Experiments have shown that Aegis can produce better mixed precision training results over existing methods on image recognition (ResNet [110]), object detection (SSD [184]), and semantic segmentation (SegNet [12]) tasks.

SPGC addresses the imbalanced workload issue from pre-trained DNNs with irregular sparsity by adding structural constraints, targetting group convolution layers. Handling the same size of the input, a group convolution layer consumes $1/G$ of the run-time and parameters of its standard convolution counterpart, in which G is the number of groups. A group convolution layer contains G number of small convolution layers, each of which takes a $1/G$ fraction of the overall workload. More importantly, all these sub-convolution can run in parallel, which is ideal for efficient deployment. Given a determined number of groups, SPGC proposes a heuristic algorithm that can iteratively resolve important parameters for each group. Moreover, to avoid searching within the large hyperparameter space that contains G for all layers, SPGC leverages a local search algorithm to efficiently explore different options. Empirical evaluation on CIFAR-10/100 has shown that SPGC performs on par with the state-of-the-art approaches [122], while SPGC is more efficient since it can directly process pre-trained models instead of training from scratch.

1.3.2 MLIR-based polyhedral compilation framework

Given the multi-level representations involved when compiling from high-level DNNs to low-level programs or hardware with polyhedral optimisation, this thesis presents two polyhedral compilation frameworks, Polygeist and POLSCA, which target software and hardware respectively, and novelly integrate MLIR [167]. MLIR, a recent compiler infrastructure originated from LLVM [165], features a uniform interface for designing multi-level intermediate representations and their transformations. MLIR is proven successful in many domain applications [104, 203, 195, 28], especially, deep learning [86, 228, 134]. Therefore, the support of polyhedral compilation on MLIR can be extended and benefit all these domains.

Polygeist focuses on raising C/C++ code into MLIR representations and applying polyhedral transformations within the MLIR ecosystem. Initially, the input C/C++ code is translated into low-level MLIR *dialects*, collections of operations and transformations that targetting specific domains, e.g., Standard for LLVM-like representation, and SCF for structured control flow. Then, Polygeist tries to raise these representations into Affine [210], a dialect for polyhedral representations. However, Affine is not connected to polyhedral scheduling tools like Pluto [30]. One major

contribution from Polygeist is enabling the conversion between MLIR and polyhedral tools using an intermediate format, OpenScop [20], by establishing the mapping between arbitrary MLIR operations and C-like statements. This mapping algorithm is robust and can deal with different hazardous scenarios. A core component of Polygeist that handles polyhedral scheduling, named Polymer, proposes a *statement splitting* heuristic, which showcases that using MLIR can enlarge the optimisation space that a polyhedral tool can explore. Experiments show that Polygeist is capable of processing on the whole Polybench/C [330], a benchmark suite that contains typical polyhedral workloads, and providing the same or even better performance compared with existing tools.

POLSCA is a compiler framework aiming to connect polyhedral methods and hardware design tools better. POLSCA targets High-Level Synthesis (HLS) designs, which describe hardware as software programs to alleviate the heavy labour when using Register-Transfer Level (RTL). However, even HLS is good at processing input with nested loops, existing polyhedral tools are not well prepared for targeting HLS and may generate bad designs without careful tuning and manual intervention. After entering the MLIR realm from C/C++ using Polygeist [214], POLSCA preprocesses and automatically annotates SCoPs for polyhedral transformation, and post-transforms the potentially complicated code generated from polyhedral optimisers by loop transformations and memory partitioning to ease the burden from the back-end HLS tools. Finally, POLSCA emits legit and simple LLVM IR as the ultimate hardware description, which will later be consumed by Xilinx Vitis [319]. Performance evaluation on Polybench/C and MachSuite [241] demonstrate that POLSCA is capable of carrying out polyhedral transformations on pure and hybrid polyhedral C/C++ code.

It is worth noting that both Polygeist and POLSCA have publicly available codebases^{1,2,3} and are receiving external contributions.

1.3.3 DNN acceleration by streaming architecture

Generating a hardware accelerator design for a given DNN is a challenging task: even if we have a functionally correct design, to achieve an optimal performance, we still need to explore multiple design choices under certain resource constraints, especially when the DNN architecture is much finer-grained with smaller layers for

¹Polygeist main repository: <https://github.com/wsmoses/Polygeist>

²Polyhedral module in Polygeist: <https://github.com/kumasento/polymer>

³POLSCA: <https://github.com/kumasento/polsca>.

efficiency purposes than those early pioneers that basically stack multiple large layers, as stated in C3. Streaming architecture that interconnects components by data pipelines is a promising design choice in such scenario, since all the data stays on-chip by being immediately consumed after production. However, it is challenging to achieve optimal performance with a streaming architecture by orchestrating the large amount of pipelines and exploring the induced large design space. This thesis proposes Deacon, a novel framework that generates FPGA-based DNN accelerators of streaming architectures with advanced pipelining techniques, to address the pipelining challenges that emerged from recent DNN topologies equipped with heterogeneous convolution and residual connections. The key contributions from Deacon are new fine-grained pipelining based on computation sequences, graph-level optimisation on the input DNNs, and heuristic design space exploration empowered by graph colouring and simulated annealing. Compared with prior designs for MobileNets and SqueezeNet, Deacon shows resource/power consumption efficiency improvement of 1.2x/3.5x, and 1.0x/2.8x averaged by geomean. Deacon can optimise large DNNs such as ResNet-18, with 2.7x/35.6x efficiency improvement over the baseline.

1.3.4 Summary

This section presents five tools that are the major contributions from this thesis. These tools cover the three key stages to have a DNN model efficiently working on a reconfigurable device, from training and exporting the model for inference, to transforming the model for more efficient execution on software and hardware, and finally, land into a hardware accelerator on FPGA. These tools are therefore connected and can benefit each other.

We can combine these tools in various application domains. For instance:

- Efficient training: Aegis (Section 3.2) is mainly for numerical stability and model accuracy, which involves the creation of multiple computation graphs for the gradient scales calculation. These new computation graphs are not optimised, and we can leverage the Polygeist tool to automatically find a better schedule of all the nested loops to further improve the speed of mixed-precision training.
- Efficient inference: Similarly, we can combine SPGC (Section 3.3) with the Polygeist framework to create GConv-based DNN models that execute efficiently after being optimised by automatic polyhedral transformations.

- Polyhedral streaming architecture: The Deacon framework (Chapter 5) uses a dataflow representation of the hardware accelerator, which can be translated to corresponding polyhedral representations [59]. We can leverage POLSCA (Section 4.4) to directly produce Deacon designs from high-level programming languages.

Making the connections is beyond the scope of this thesis. We leave the invention of novel hybrid tools a future work.

1.4 Publications

There are various papers being published or under review that form the major parts of this thesis, listed in the order of appearance in each chapter:

- Ruizhe Zhao, Brian Vogel, Tanvir Ahmed, and Wayne Luk, Reducing Underflow in Mixed Precision Training by Gradient Scaling, in *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI 2020*.
- Ruizhe Zhao and Wayne Luk, Efficient Structured Pruning and Architecture Searching for Group Convolution, in *2019 IEEE/CVF International Conference on Computer Vision Workshops*.
- William S. Moses, Lorenzo Chelini, Ruizhe Zhao, and Oleksandr Zinenko, Polygeist: Raising C to Polyhedral MLIR, in *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)* ⁴.
- Ruizhe Zhao, Jianyi Cheng, Wayne Luk, George A. Constantinides, POLSCA: Polyhedral High-Level Synthesis with Compiler Transformations, *2022 32nd International Conference on Field Programmable Logic and Applications (FPL)*.
- Ruizhe Zhao, Ho-Cheung Ng, Wayne Luk, Xinyu Niu, Towards Efficient Convolutional Neural Network for Domain-Specific Applications on FPGA, in *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*.
- Ruizhe Zhao, Ho-Cheung Ng, Wayne Luk, Xinyu Niu, Deacon: Mapping Deep Neural Network to Streaming Architecture with Advanced Pipelining, a journal paper submitted for review.

⁴William S. Moses, Lorenzo Chelini, and Ruizhe Zhao have equal contributions towards this paper

1.5 Thesis organisation

The remaining chapters of thesis are listed as follows:

- Chapter 2 reviews the essential background related to our proposed methods.
- Chapter 3 presents the memory-efficient deep learning methods, specifically, Aegis (Section 3.2) and SPGC (Section 3.3).
- Chapter 4 focuses on the multi-level polyhedral compilation infrastructure, which includes the detailed description on Polygeist (Section 4.2), Polymer (Section 4.3), and POLSCA (Section 4.4).
- Chapter 5 demonstrates the Deacon DNN accelerator library.
- Chapter 6 summaries this thesis and ideates future work.

Chapter 2

Background

In this chapter, we review the preliminaries and related work in the following four topics:

- Deep Learning (Section 2.1): covers the concepts of convolutional neural network, including variants of the convolution layer, and efficient training and inference techniques, e.g., mixed precision training, pruning, etc.
- Reconfigurable computing (Section 2.2): explains the FPGA architecture, design methodology, and the application of Deep Learning accelerator.
- Polyhedral compilation (Section 2.3): describes the core idea of the polyhedral modelling techniques, including the mathematical formulation, the development of polyhedral scheduling and code generation approaches, and relevant tools.
- The MLIR framework (Section 2.4): illustrates the MLIR compilation framework, and how it is related to the major topics of this thesis.

Within these sections, we also demonstrate existing efforts to address the aforementioned challenges, specifically:

- Challenge C1 (Section 1.2.1): efficient Deep Learning training approaches based on mixed-precision training are described in Section 2.1.3, and Section 2.1.2 describes the papers related to efficient inference using pruning and group convolution.
- Challenge C2 (Section 1.2.2): existing polyhedral toolchains and frameworks are mentioned in Section 4.3.

- Challenge C3 (Section 1.2.3): we cover the recent efforts to design a DNN accelerator on reconfigurable platforms in Section 2.2.3.

2.1 Deep Learning

Deep Learning is a broad topic. This section mainly covers the essential aspects that this thesis focuses on, including Convolutional Neural Network (Section 2.1.1), mixed-precision training (Section 2.1.3), and efficient inference (Section 2.1.2).

2.1.1 Convolutional Neural Network

Convolutional Neural Network (CNN) is one of the most successful inventions in Deep Learning. CNN has broad applications in computer vision, e.g., image recognition [258, 110], object detection [82, 245, 244], and segmentation [133], while recent study also shows the potential of CNN in natural language processing [148, 140]. The major component of CNN is the *convolution layer*, which is a three-dimensional (height, width, and depth) convolution filter with trainable kernels. A CNN conventionally stacks many convolution layers together, e.g., 50 as in [110], and interleaves them with auxiliary layers to carry out activation (e.g., ReLU [109]) and normalisation (e.g., batch normalisation [129]) tasks to make training easier. However, even this typical CNN structure is sufficiently powerful, there are still space for execution performance improvement: this thesis mainly looks at the more efficient variants of the standard convolution layer.

Convolution layer

The standard convolution layer correlates an input *feature map* $\mathbf{X}$ and *weights* $\mathbf{W}$ together and produces another feature map $\mathbf{Y}$. The feature map is a four-dimensional tensor¹: suppose the shape is $N \times C \times H \times W$, then the feature map contains N images, each of height H and width W and has C channels (Figure 2.1). N is also known as the batch size. A convolution layer process each image within the feature map independently, therefore, given an input feature map of shape $N \times C \times H \times W$, there will be an output of shape $N \times C' \times H' \times W'$, in which C' , H' , W' are the number of channels, height, and width of each output image. The weights are also represented as a four-dimensional tensor of shape $C' \times C \times K_h \times K_w$. For each output

¹Tensor in deep learning literature is a term interchangeable with *multi-dimensional array*.

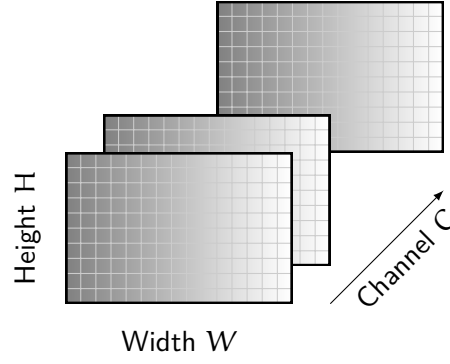


Figure 2.1: The shape of a single image in a feature map.

$$\begin{pmatrix} 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} * \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 4 & 3 & 4 & 1 \\ 1 & 2 & 4 & 3 & 3 \\ 1 & 2 & 3 & 4 & 1 \\ 1 & 3 & 3 & 1 & 1 \\ 3 & 3 & 1 & 1 & 0 \end{pmatrix}$$

I K I * K

Figure 2.2: An illustration of the 2D convolution operator *. Here, I is a single channel of a single image from $\mathbf{X}$, and K is a single kernel from the weights $\mathbf{W}$.

and input channel combination, $0 \leq c' < C'$ and $0 \leq c < C$, there is a kernel $\mathbf{W}_{c',c}$ of shape $K_h \times K_w$.

$$\mathbf{Y}_{i,c'} = \sum_{c=1}^C \mathbf{X}_{i,c} * \mathbf{W}_{c',c} \quad (2.1)$$

Algorithm 1: 2D convolution * and accumulate.

```

1 for  $c' \leftarrow 0$  to  $C'$  do
2   for  $c \leftarrow 0$  to  $C$  do
3     for  $x' \leftarrow 0$  to  $H'$  do /* Spatial */
4       for  $y' \leftarrow 0$  to  $W'$  do
5         for  $k_h \leftarrow 0$  to  $K_h$  do /* Kernel */
6           for  $k_w \leftarrow 0$  to  $K_w$  do
7              $\mathbf{Y}_{i,c',x',y'} \leftarrow \mathbf{X}_{i,c,x'+k_h,y'+k_w} \times \mathbf{W}_{c',c,k_h,k_w} + \mathbf{Y}_{i,c',x',y'}$ 

```

Equation 2.1 demonstrates how each kernel is correlated by each channel of the input feature map to get the output. We use * to represent the 2D convolution operation, which applies element-wise multiplication between the kernel and each $K_h \times K_w$ of the input feature map (Figure 2.2). The algorithm of this 2D convolution operation, together with the across-channel accumulation, is listed under Algorithm 1.

Type	Time complexity	Weights size
Standard	$\mathcal{O}(\text{NHW} \times \text{C}^2 \text{K}^2)$	$\mathcal{O}(\text{C}^2 \text{K}^2)$
Depthwise	$\mathcal{O}(\text{NHW} \times \text{CK}^2)$	$\mathcal{O}(\text{CK}^2)$
Pointwise	$\mathcal{O}(\text{NHW} \times \text{C}^2)$	$\mathcal{O}(\text{C}^2)$

Table 2.1: Comparison on the time and space (weight size) asymptotic complexity among standard, depthwise, and pointwise convolution layers.

As shown in Algorithm 1, we can describe a convolution layer as a set of perfectly-nested, interchangeable loops. We refer to a specific sequence of loops as a *computation sequence* (CompSeq). This thesis focuses on two computation sequences: filter-major, as what Algorithm 1 demonstrates; channel-major, which we can get by swapping the outermost $\mathbf{c}'$ and $\mathbf{c}$ loops in Algorithm 1.

$$\text{Depthwise convolution: } \mathbf{Y}_{\mathbf{i},\mathbf{c}} = \mathbf{X}_{\mathbf{i},\mathbf{c}} * \mathbf{W}_{\mathbf{c}} \quad (2.2)$$

$$\text{Pointwise convolution: } \mathbf{Y}_{\mathbf{i},\mathbf{c}'} = \sum_{\mathbf{c}=1}^{\mathbf{C}} \mathbf{X}_{\mathbf{i},\mathbf{c}} \times \mathbf{W}_{\mathbf{c}',\mathbf{c}} \quad (2.3)$$

Depthwise and pointwise convolution

There are two popular variants to the standard convolution layer [48, 118, 334]:

1. Depthwise convolution layer (Equation 2.2) only applies 2D convolution to each channel of the input feature map without accumulation, which can only learn spatial correlations.
2. Pointwise convolution layer (Equation 2.3) sets the kernel size to 1×1 , which results in a convolution layer that only explores cross-channel correlations, i.e., no spatial correlation can be learned.

These two variants are computationally more efficient than the standard convolution. Considering a standard convolution layer that produces a feature map that has the same shape as the input, e.g., both are $\text{N} \times \text{C} \times \text{H} \times \text{W}$, and assuming the weight kernels are square, i.e., $\text{C} \times \text{C} \times \text{K} \times \text{K}$, we can get the time and space complexity for each type of convolution layer as shown in Table 2.1. Regardless of the impact on model accuracy, depthwise and pointwise convolution layers are obviously better choices to build faster and smaller models.

Type	Time complexity	Weights size
Standard	$\mathcal{O}(\text{NHW} \times \text{C}^2 \text{K}^2)$	$\mathcal{O}(\text{C}^2 \text{K}^2)$
Stacked	$\mathcal{O}(\text{NHW} \times 2\text{CC}_i \text{K}^2)$	$\mathcal{O}(2\text{CC}_i \text{K}^2)$
Depthwise separable	$\mathcal{O}(\text{NHW} \times (\text{CK}^2 + \text{C}^2))$	$\mathcal{O}(\text{CK}^2 + \text{C}^2)$
Bottleneck	$\mathcal{O}(\text{NHW} \times (\text{C}_i^2 \text{K}^2 + 2\text{CC}_i))$	$\mathcal{O}(\text{C}_i^2 \text{K}^2 + 2\text{CC}_i)$
Separable bottleneck	$\mathcal{O}(\text{NHW} \times (\text{C}_i \text{K}^2 + 2\text{CC}_i))$	$\mathcal{O}(\text{C}_i \text{K}^2 + 2\text{CC}_i)$

Table 2.2: Time and space complexity for different convolution blocks. C_i is the number of channels of the intermediate feature maps in between layers.

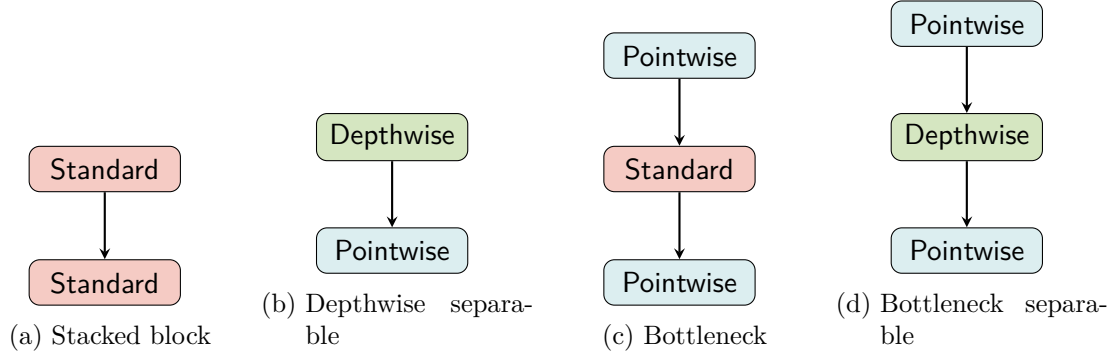


Figure 2.3: Illustrations of different convolution blocks.

Another variant, group convolution layer, which separates the input feature map into groups by the channel axis and processes each of them in parallel by standard convolution layers, can be seen as a generalized version of depthwise convolution. If a channel group only contains a single channel, then the corresponding group convolution becomes depthwise convolution. More details related to group convolution are covered in Section 2.1.2.

Convolution block

Simply replacing a single standard convolution layer with an individual depthwise or pointwise convolution layer can inevitably degrade the model accuracy. One reason is the significant reduction of model parameters that results from layer replacement, which can decrease the learning capacity of a model [85, p. 423]. The other reason is that depthwise or pointwise convolution can only learn either of the cross-channel or spatial correlations, both of which the original standard convolution is capable to learn. A common practice is to compose convolution layers into a *convolution block*, which is a building block that connects multiple relatively small convolution layers together. Compared with the standard convolution layer counterpart (with the same input and output shape), empirically, a convolution block can have at least the same learning capacity but runs faster with less memory footprint. This thesis

mainly explores the following four types of convolution block (Figure 2.3):

1. A *stacked convolution block* stacks two standard convolution layers together. The number of channels C_i from the inner feature map, an intermediate result from the first layer, is normally smaller than the channels of the block input and output feature maps, so that the computational cost can be reduced compared to using a single standard convolution. Based on Table 2.2, if $2C_i$ is less than C , the stacked block is more efficient than the standard convolution counterpart. ResNet-34 [110] extensively uses stacked convolution blocks.
2. A *depthwise separable block* [77, 48, 118] connects a depthwise convolution layer with a pointwise convolution, aiming to learn both the cross-channel and the spatial correlations, which can further reduce parameters compared to the stacked block.
3. A *bottleneck block* consists of a stack of pointwise convolution, standard convolution, and another pointwise convolution. Suppose the two intermediate results are feature maps of the same number of channels, C_i , and its value is a fraction to C , e.g., $\frac{1}{4}$, the overall complexity can still be significantly reduced from the standard convolution counterpart (Table 2.2). Bottleneck block is first demonstrated in ResNet-50 ([110]) and is gaining prevailing use in recent days.
4. A *separable bottleneck block* is a variant to the bottleneck block, which uses depthwise convolution as the inner layer, that is proposed and used in MobileNet V2 [251]. Its promise is to combine the efficiency of depthwise convolution with the effectiveness of the bottleneck structure.

Note that there can be *residual* connections between the input and the output of the blocks, e.g., ResNet-50 adds up the result from the final pointwise convolution with the input to the first pointwise layer as the result of a whole bottleneck block. These residual connections are critical to successfully training very deep neural networks, but since it is unlikely that they will affect much the computational performance, this thesis will not explicitly discuss them.

Winograd algorithm

Previous optimisations have no improvement on the 2D convolution (Algorithm 1) implementation, which implies that the time complexity for each pixel is still $\mathcal{O}(K_h \times K_w)$. Recent research [168] adapts the *Winograd's minimal filtering algorithm* [305,

Model	Block	Ops (GOPS)	Params (M)	Top-1 (%)
VGG-16	—	30.95	138.3	71.5
ResNet-50	Bottleneck	7.72	24.3	75.2
MobileNet V1	Depthwise separable	1.14	4.01	70.9
MobileNet V2	Separable bottleneck	0.61	3.31	71.9

Table 2.3: Efficient CNN models statistics. Top-1 (%) indicates the top-1 recognition accuracy on ImageNet ILSVRC 2012 [248].

p. 39], originally for efficient processing of FIR filters, to further push the performance boundary of 2D convolution.

$$\mathbf{Y} = \mathbf{A}^T \left[[\mathbf{G}\mathbf{w}\mathbf{G}^T] \odot [\mathbf{B}^T \mathbf{x} \mathbf{B}] \right] \mathbf{A} \quad (2.4)$$

Following the notation in [168], an instance of 2D Winograd is denoted by $F(\mathbf{T}_h \times \mathbf{T}_w, \mathbf{K}_h \times \mathbf{K}_w)$, where $\mathbf{T}_h \times \mathbf{T}_w$ is the shape of a 2D *tile* of the output that will be produced simultaneously, and $\mathbf{K}_h \times \mathbf{K}_w$ is the shape of a 2D convolution kernel. When the height and width are identical for both the tile shape and the kernel shape, i.e., $\mathbf{T} = \mathbf{T}_h = \mathbf{T}_w$ and $\mathbf{K} = \mathbf{K}_h = \mathbf{K}_w$, the 2D Winograd convolution can be formulated as equation 2.4. In this formula, $\mathbf{w}$ is a $\mathbf{K} \times \mathbf{K}$ kernel, and $\mathbf{x}$ is a single tile of shape $(\mathbf{T} + \mathbf{K} - 1) \times (\mathbf{T} + \mathbf{K} - 1)$ from the input feature map. $\odot$ is the Hadamard product, i.e., element-wise matrix multiplication. $\mathbf{G}$, $\mathbf{B}$, $\mathbf{A}$ are three *constant transformation matrices* of shape $(\mathbf{T} + \mathbf{K} - 1) \times \mathbf{K}$, $(\mathbf{T} + \mathbf{K} - 1) \times (\mathbf{T} + \mathbf{K} - 1)$, and $(\mathbf{T} + \mathbf{K} - 1) \times \mathbf{T}$ respectively. These matrices contain integer and fractional numbers that are by-products produced while deriving the Winograd convolution using the Chinese Remainder Theorem [143]. More details of their contents can be found in [168].

A widely used instance is $F(4 \times 4, 3 \times 3)$, due to the popularity of 3×3 kernels in CNN [258, 110]. When using standard convolution, the total amount of floating-point multiplication operations to calculate all the 16 output elements is kernel size $\times$ tile size $= 3 \times 3 \times 4 \times 4 = 144$. Regarding 2D Winograd convolution (Equation 2.4), the Hadamard operation takes $(\mathbf{T} + \mathbf{K} - 1) \times (\mathbf{T} + \mathbf{K} - 1) = 36$ multiplications, which is equivalent to the size of the transformed input tile or the transformed kernel. Given that these constant transformations can be calculated without any floating-point multiplication, and floating-point multiplications take the most of the computation time, it is reasonable to say that the overall computational complexity can be reduced by a factor of $144/36 = 4$ when using 2D Winograd $F(4 \times 4, 3 \times 3)$.

Efficient Models

The efficiency of an individual convolution layer or convolution block is computational: we only consider the time and space complexity when an input feature map is being processed. The efficiency of a CNN model, however, is more complicated, since we should take into account the overall achievable accuracy for specific tasks. A model is more efficient than the other if it can achieve better accuracy with fewer parameters and operations. For example, as listed in Table 2.3, compared with the baseline model VGG-16 [258], ResNet-50 [110], MobileNet V1 [118] and V2 [251] are more efficient since they have much fewer parameters and operations while still able to achieve around the same or even better accuracy. Even though in recent research there are more efficient CNN models [268, 269, 120], in this thesis we only cover the models in Table 2.3 because they are much more popular than the more efficient counterparts.

2.1.2 Efficient inference

To build an efficient CNN model, we can start from scratch by designing convolution layers, exploring their connectivities, and training them with randomly initialized parameters. It is a rather tedious procedure, since training can consume much computational resource [263], e.g., several GPUs running in full workloads for days. This section discusses an alternative approach, which searches for an efficient model after it has been trained, i.e., at the inference phase. We first present group convolution layer, the design of which is variable, determined by the number of groups G . Then we demonstrate the key concepts of network pruning, a technique that produce smaller CNN models from pre-trained, big models. Finally, for comparison, we also mention the recent progress on Neural Architecture Search, a method that finds efficient CNN models from scratch.

This section reflects the related work that aims to address Challenge C1 (Section 1.2.1).

Group convolution

Group convolution was originally proposed by [156] for purposes of better parallelising across multiple GPUs. Since its introduction however, group convolution is predominately used to build efficient CNN models. A group convolution layer partitions

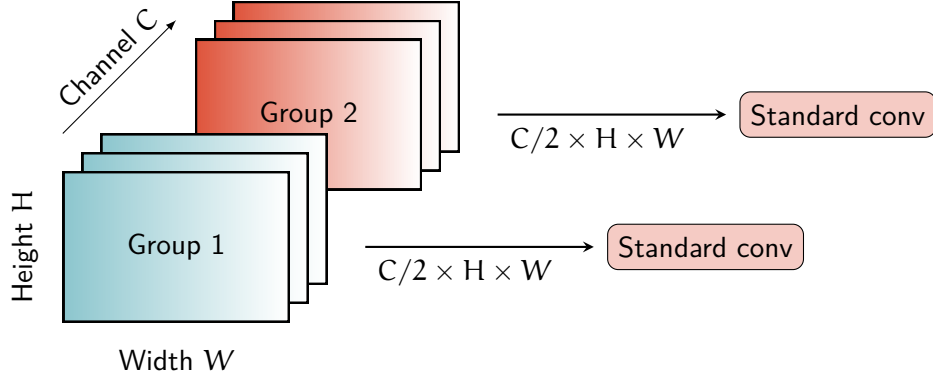


Figure 2.4: Illustration of a group convolution with $G = 2$, which processes two groups of the input feature map, split by half along the channel axis.

its input channels into disjoint groups and separately convolves each with a group-specific set of filters. Concretely, given an input feature map shaped $C \times H \times W$, assuming batch size $N = 1$, a group convolution of group size G runs G convolution layers between each pair of $C/G \times H \times W$ input partition and $C'/G \times C/G \times K_h \times K_w$ weight group. Figure 2.4 illustrates how group convolution works.

Group convolution is computationally more efficient than standard convolution: given the same input and output feature maps, and assuming they have identical shape, a group convolution with group size G has space complexity of $\mathcal{O}(C^2 K^2 / G)$ and time complexity of $\mathcal{O}(NHW \times C^2 K^2 / G)$, which is $1/G$ over a standard convolution counterpart (Table 2.1). And to avoid losing much the learning capacity due to the reduced number of parameters, we can encourage information exchange among groups [334] using *channel permutation*, which shuffles the channels of the input and the output feature maps before and after processing the internal array of standard convolution layers [334, 333, 310]. There are other approaches to enhance information exchange, e.g., adding pointwise convolution [311, 128, 227], using block Hadamard transform [339], etc., but since they need extra floating-point operations, they are less desirable than channel permutation.

To construct a CNN by GConv, one can build and train from scratch [334, 128, 311, 339, 310], or prune from pre-trained models. CondenseNet [122] prunes by a multi-stage, from-scratch training and regularisation procedure. FLGC [297] follows a similar approach to optimise the GConv topology while training from scratch. Peng et al. [227] consider a GConv layer as a low-rank approximation of a convolution layer. This approach produces models with high test accuracy but always needs to add pointwise layers. We also intend to prune convolution into GConv and our method is one-shot and does not add test-time overhead.

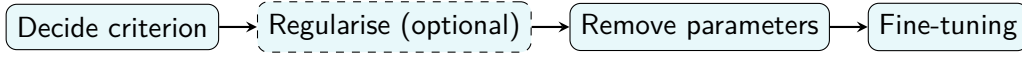


Figure 2.5: The steps of network pruning.

Network pruning

Removing unimportant parameters from a pre-trained model without changing much the accuracy is commonly referred to as *network pruning* [171, 107, 108]. By leveraging pruning, one can get smaller models directly from a pre-trained one without training from scratch, which can save much time and energy budget. The major challenge of pruning is to decide which parameters are unimportant, i.e., what the *importance criterion* is. There are many candidates: magnitude, e.g., L1, L2 norms [198, 105, 174, 190], first-order [212, 74] or second-order [171, 108, 62] gradients, average percentage of zero [190, 119], singular values [227, 200], etc. Each criterion takes different computation budget and embeds variable heuristics of parameter importance: for example, a magnitude-based criterion assumes that if a parameter is close to zero, it can have minor contribution to model accuracy, which is easy to calculate but potentially too basic; while a gradient-based criterion is more sound (e.g., derived from information theory [171]), but gathering the first-order, or even the second-order derivatives uses non-trivial calculation and may need to go through significant amount of the training data until the gradient estimation stabilises.

Figure 2.5 summarizes the steps of network pruning. Once the importance criterion is determined, the rest of the pruning procedure is to remove those parameters seen as unimportant and replace the internal implementation based on dense tensors with sparse tensors to practically reduce computational workload and speed up processing. There can be an extra step of *fine-tuning*, i.e., start training again from the pruned model with small learning rate, to further improve the model accuracy after pruning. In case that the original pre-trained model is hard to be pruned, for example, suppose the target criterion is magnitude-based, there are very few of the parameters are close to zero, i.e., not very sparse. We can apply regularisation before pruning to sparsify the model. L1 norm is studied for regularising CNN models [185, 105] to produce parameters that are zeros, or close to zero. However, the parameters resulted from L1 norm regularisation are typically *unstructured*, i.e., the positions of the zero or close-to-zero parameters cannot form a pattern, which may lead to uneven workload when processing the model. There are some other regularisation methods use *group LASSO* [329] to encode specific sparse structures [223, 301, 170, 122]. In this thesis, we are interested in how to produce structured sparse model that can resemble group convolution.

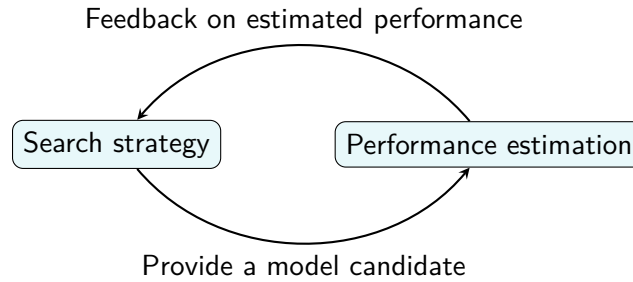


Figure 2.6: The general abstracted procedure of neural architecture search, following the framework proposed in [66].

Neural architecture search

The motivation for pruning is to find a more efficient version of a pre-trained DNN model. Neural architecture search (NAS) [66] also aims for a similar objective of finding the most efficient DNN model, while its candidates are not constrained by a specific pre-trained DNN model: NAS can examine models with arbitrary architectures and train them from scratch. The key procedure of NAS iteratively switches between proposing a candidate architecture from a search space using a *search strategy* and evaluating the performance of that candidate, mostly by *estimation* due to the heavy workload of training from scratch, to provide feedback on its achievable accuracy [66] (Figure 2.6).

This iterative searching procedure can naturally fit into the framework of *Reinforcement Learning* (RL) [267, 303], for example, [349, 350, 270, 268] use RL to find the DNN architecture that are more efficient than their counterparts on certain tasks. Evolutionary algorithm is another solution to the searching problem [242, 45, 178]. To make NAS more efficient, using gradient-based method [179] or reducing the search space [177] is proposed as well. [^R_Z: Add more NAS papers.] This thesis does not adapt the NAS practices due to their resource-heavy nature, however, the iterative searching procedure in our problem setting (Section 3.3) can potentially be improved by NAS techniques.

State-of-the-art related to Section 3.3

Regarding the structured pruning method targeting GConv in Section 3.3, we explicitly compare with CondenseNet [122] that presents a multi-stage, from-scratch training and regularisation procedure to prune into GConv. We also compare with network slimming [185] on pruning performance in general. There is also Peng et al. [227] working on network pruning into group convolution, which is another candidate for our comparison.

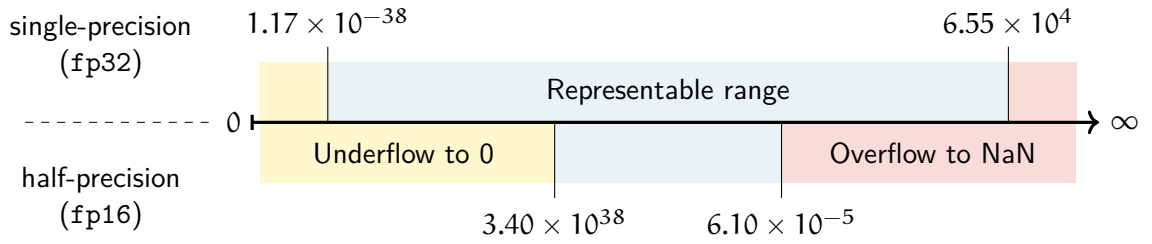


Figure 2.7: Illustration of the representable range for single-precision floating-point (fp32) and half-precision (fp16). The range of fp16 is much narrower than fp32. The values out of the representable range can be underflow, if close to zero, or overflow, if too large. Those vertical bars mark the minimum and maximum values for fp32 (above) and fp16 (below). The scale of the axis is roughly logarithmic.

2.1.3 Mixed Precision Training

Training a DNN model can take much resource and computation budget, and it is largely attributed to the expensive single-precision floating-point data type (fp32): storing a parameter of such type uses 32 bits, and the corresponding floating-point arithmetic is much more complicated than plain integer data types [215]. While switching to lower precision can save space and speed up training, data types with lower precision are error-prone. One example of low precision data type is *half-precision* floating-point (fp16). They can have narrower representation range that can cause either *overflow*, i.e., large values that exceed the boundary become incomputable NaN (not-a-number), or *underflow*, i.e., small values that are close to zero eventually rounded to zero. This phenomenon is shown in Figure 2.7. On the other hand, they can have larger gap between adjacent values that leads to pervasive rounding, which can potentially be problematic for sensitive applications. Mixed precision training (MPT) is proposed to balance the trade-off between high- and low-precision by a mixture of them during training. This section first reviews the key concepts of floating-point arithmetic and dives deeper into numerical issues like overflow and underflow, then covers the elements of backpropagation, the fundamental DNN training algorithm. Finally, this section describes the recent progress in mixed precision training, specifically, reveals the details of *loss scaling* [209], the first and still widely applied MPT algorithm.

The content in this section provides the background and related work for Challenge C1 (Section 1.2.1).

$$\begin{array}{c}
\underbrace{0}_{\text{signed}} \underbrace{01111110}_{\text{exponent}} \underbrace{111000000000000000000000}_{\text{mantissa}} = 0.9375 \\
\\
E = \sum_{i=1}^{N_e} e_i 2^{N_e-i} + 1 - 2^{N_e-1} = 126 + 1 - 128 = -1 \\
\\
M = 1 + \sum_{i=1}^{N_m} m_i 2^{-i} = 1 + 0.875 = 1.875
\end{array} \tag{2.5}$$

Floating-point format and arithmetic

An N -bit floating-point number x has one *signed* bit, N_e *exponent* bits, and N_m *mantissa* bits, that is, $N = 1 + N_e + N_m$. Suppose the bitwise representation of an N -bit floating-point number x is $\{s, e_1, \dots, e_{N_e}, m_1, \dots, m_{N_m}\}$, in which each element is a single bit, and s and m_{N_m} are the most and the least significant bits, then the value that x represents is $(-1)^s \times M \times 2^E$. Here, M is the *significand* equals to $1 + \sum_{i=1}^{N_m} m_i 2^{-i}$, and E is the *exponent* equals to $\sum_{i=1}^{N_e} e_i 2^{N_e-i} + 1 - 2^{N_e-1}$. **fp32** has $N_e = 8$ and $N_m = 23$ and represents magnitude ranging from 1.17×10^{-38} to 3.40×10^{38} . Equation 2.5 shows the interpretation of an example **fp32** number. **fp16** only has 5 exponent bits and 11 mantissa bits, resulting in the representation range of $[6.55 \times 10^4, 6.10 \times 10^{-5}]$, denoted by $[\lambda_{\min}, \lambda_{\max}]$. In the following discussion, we take **fp32** and **fp16** as the default pair of data formats for mixed precision training, and we will present other choices at the end of this section.

This thesis mainly focuses on two problematic aspects of floating-point arithmetic: overflow and underflow, and *swamping*, which appears in the summation of big and small floating-point numbers. More information on general floating-point arithmetic can be found in [215, 84]. The consequence of **fp16** having a much narrower range of representation than **fp32** is the higher probability of inducing overflow and underflow (Figure 2.7). The appearance of overflow is destructive: overflow introduce a special representation, NaN (not-a-number), which irrecoverably forces subsequent arithmetic results to be NaN. Underflow has a relatively milder effect: a value smaller than the minimum representable number just becomes zero. In the context of DNN training, overflow immediately terminates training, while underflow may hinder further training when the updates are too small.

Swamping [112, 246] is a well-studied error from floating-point arithmetic that appears from adding up a pair of big and small numbers. The small number will be shifted to the same scale as the big one, which leads to truncation on the small number, and there will be much larger error if a big number is being accumulated iteratively with many small numbers. There are much literature on swamp-

ing [112, 246, 196, 60], and especially, Wang et al. [296] looks into swamping in the context of low-precision floating-point DNN training.

$$\mathbb{E} = \frac{1}{m} \sum_{i=1}^m L(f(\mathbf{x}_i, \mathbf{W}), \mathbf{y}_i) \quad (2.6)$$

DNN training and backpropagation

Training a DNN is an optimisation problem: the objective is to minimise the *loss* value, a metric measures how the DNN predictions are deviated from the ground-truth. By representing a DNN model as a function $f(\mathbf{x}, \mathbf{W})$, where $\mathbf{x}$ is an input sample (e.g., an image as for CNN), and $\mathbf{W}$ are the model parameters, we can have the optimisation objective as Equation 2.6. In this equation, we denote the ground-truth of prediction (also known as *label*) as $\mathbf{y}$, and the objective is an averaged loss across all m training samples. The loss function, $L(\mathbf{y}, \mathbf{y}')$, can take various forms, e.g., mean squared error $\|\mathbf{y} - \mathbf{y}'\|^2$ if the predictions are continuous, or cross-entropy [216, p. 57] if the predictions are discrete. This formulation of training as optimisation is commonly referred to as *empirical risk minimisation* [85, Chap. 8].

$$\mathbf{W}^{(t+1)} \leftarrow \mathbf{W}^{(t)} - \alpha^{(t)} \frac{\partial \mathbb{E}}{\partial \mathbf{W}}(\mathbf{W}^{(t)}) \quad (2.7)$$

A typical way to solve this empirical risk minimisation is *stochastic gradient descent* [33, 34, 35] (SGD). Gradient descent is an algorithm that iteratively updates the model parameters $\mathbf{W}$ by the gradients of the empirical risk $\mathbb{E}$, which are the partial derivatives of form $\partial \mathbb{E} / \partial \mathbf{w}$. Equation 2.7 demonstrates the basic form of gradient descent: at the optimisation step t , the optimiser subtracts $\mathbf{W}^{(t)}$ by the gradient controlled by the step size α (also known as *learning rate*). Calculating the exact gradient needs to go through the whole training dataset, which is impractical due to its heavy computational budget. SGD estimates the gradient based on a stochastically sampled small batch of the training data, which is more efficient and becomes the default optimisation algorithm for almost all DNN models. Recent studies further improves SGD by changing how the weights are updated by the gradient [149, 188, 63, 114] from the basic form Equation 2.7.

Parameters in a DNN are connected through *computation graph*, in which nodes are DNN operators, e.g., convolution (Algorithm 1), matrix multiplication, etc., and edges representing data blocks and their flow directions. Some nodes in a computation graph just generates data by taking input from a source or reading

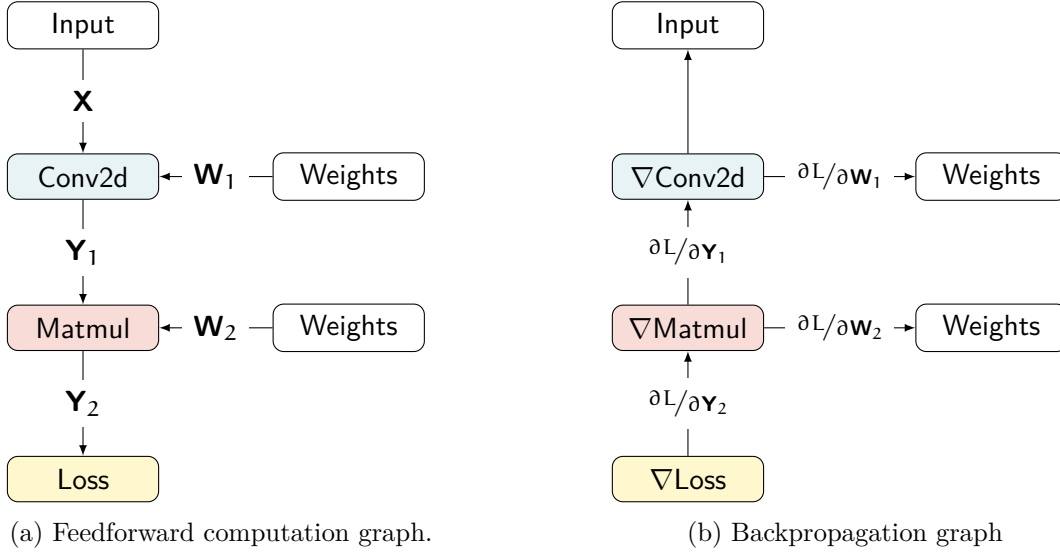


Figure 2.8: The feedforward and backpropagation computation graphs of the same DNN consisting of one convolution layer and one matrix multiplication operation, both of which are connected to their weights, $\mathbf{W}_1$ and $\mathbf{W}_2$. The feedforward graph propagates the input feature map $\mathbf{X}$, weights, and intermediate outputs, $\mathbf{Y}_1$ and $\mathbf{Y}_2$, top down until reaches the loss function. And the backpropagation graph propagates from the bottom, and the data being propagated are the gradients of the loss value. The ∇ symbol before each operator in the backpropagation graph indicates that it calculates gradients by automatically differentiating the feedforward definition.

from memory for parameters. Figure 2.8a provides an example of a computation graph that computes in a *feedforward* direction, i.e., data flow from the input feature map to the output prediction. If we reverse all the edges in the feedforward graph, and replace the data being propagated with their gradients of the loss value, we can get a *backpropagation* graph 2.8b. The key step of the backpropagation algorithm is to generate such a graph, mainly by *automatic differentiation* [21, 65, 25], a technique that evaluates the gradients based on the chain rule [8].

$$\frac{\partial L}{\partial \mathbf{W}_1} = \frac{\partial L}{\partial \mathbf{Y}_1} \frac{\partial \mathbf{Y}_1}{\partial \mathbf{W}_1} = \left(\frac{\partial L}{\partial \mathbf{Y}_2} \frac{\partial \mathbf{Y}_2}{\partial \mathbf{Y}_1} \right) \frac{\partial \mathbf{Y}_1}{\partial \mathbf{W}_1} \quad (2.8)$$

For example, to get the gradient $\partial L / \partial \mathbf{W}_1$ Figure 2.8b, we can use the chain rule to expand it into $\partial L / \partial \mathbf{Y}_1 \times \partial \mathbf{Y}_1 / \partial \mathbf{W}_1$, in which the second term can be evaluated locally, and the first term is propagated from the next layer. It is feasible to evaluate $\partial \mathbf{Y}_1 / \partial \mathbf{W}_1$ *locally* because $\mathbf{Y}_1$ is the direct outcome from applying convolution on both the input feature map and the weights $\mathbf{W}_1$. And as demonstrated in Equation 2.8, we can calculate $\partial L / \partial \mathbf{Y}_1$ by further expansion through the chain rule. In this way, we can build the whole backpropagation graph (Figure 2.8b). In Section 3.2, we present how to modify this backpropagation algorithm for mixed precision training.

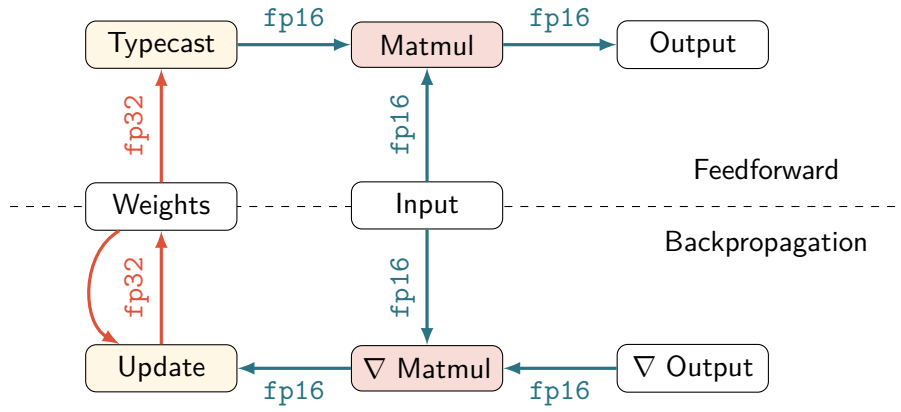


Figure 2.9: The mixed precision version of a matrix multiplication layer. Weights are typecast from fp32 to fp16 in the beginning and then passed to Matmul. Matmul takes fp16 input feature map and weights, and produces fp16 output. During backpropagation, the gradients of the output, ∇ Output, and the original input, are passed to ∇ Matmul as fp16, to get the gradients updates for weights, also in fp16. This mixed precision operator finally typecasts the gradient update to fp32 and accumulates it to the weights, which are stored as fp32.

Mixed precision operator

The presumption of mixed precision training is that some DNN operations are less *precision sensitive* than the others. Empirically, for example, convolution and matrix multiplication, which are operators that multiply and accumulate many long vectors, are less sensitive to the precision of a single element-wise calculation, while operators that use logarithmic and exponential subroutines, e.g., log and exp, requires high-precision data formats. Meanwhile, when using floating-point, the storage format for weights normally takes higher precision, since the whole training procedure needs to accumulate gradient updates onto weights (Equation 2.7), it will be much less accurate if the data precision is low due to swamping. Based on this presumption, Figure 2.9 shows an example of the mixed precision matrix multiplication layer. Although it seems that the mixed precision version is more complicated than plain matrix multiplication, recent GPU architectures can well support such mixed precision computation patterns. For example, NVIDIA GPUs have *Tensor Cores* [221, 222] that accelerate the pattern that computes matrix multiplication in fp16 first, and then adds the result to another input using fp32, which matches the gradient calculation and update scenario as in Figure 2.9.

Loss scaling

One major issue with the mixed precision training is the wide dynamic range of gradient values cannot fit into the representation range of a low precision data type, e.g., `fp16`. Those gradients with values larger than 6.10×10^{-5} immediately terminates training, and those less than 6.55×10^4 trigger underflow and become zero. Underflow seems to be harmless, however, empirical study in [209, Figure 3] shows that there can be 20% of gradient values rounded to zero during training due to underflow, and scaling up the gradients to avoid underflow can significantly improve training result. This phenomenon motivates the invention of *loss scaling* [209].

Loss scaling is based on a straightforward idea: scaling up the loss value before backpropagation can scale up all the gradients. For example, if we scale up the loss value by a factor of 8, then all the gradients will be 8 times larger. This scaling factor is a hyperparameter, which can be a constant value chosen based on heuristics, or a dynamic value tuned during training. One dynamic update strategy is *back-off* [159, 271, 272], which gradually increases the scale factor until it reaches to a maximum value (e.g., 65536), and once overflow occurs, reduces the factor by half and restart the current training iteration. Mellempudi et al. [208] improves back-off with a minimum threshold, i.e., the scaling factor cannot be reduced further below a certain value, in order to make loss scaling more effective at handling underflow with much lower precision data types, e.g., `fp8`. Even though loss scaling can make mixed precision training better and has been widely adapted [271, 272], finding a good loss scale factor, or tuning for a certain dynamic scaling rule, can still be time-consuming and diminish the actual speed-up benefits from using lower precision. Moreover, layers in a DNN model can have different data distribution, which implies that a single scale factor at the back can hardly properly scale every gradient in every layer. Sun et al. [266] propose GradScale that assigns a trainable scale factor to each layer that can be adjusted during training and works well even with ultra-low precision like `fp4`. This thesis aims to address the same challenge, instead, our work [341], published around the same time as [266], uses an alternative approach that leverages the gradient statistics. Detailed discussion can be found in 3.2.

Contents in this section are the state-of-the-art research on using loss scaling to improve mixed precision training performance.

Other mixed precision training methods

Our discussion so far focuses on using `fp16` and `fp32` as the low- and high-precision data types, and leveraging loss scaling to mitigate the underflow issue. Another

major issue from low-precision floating-point is the rounding error (e.g., swamping), and there are several papers target at addressing this problem. Wang et al. [296] devise a chunk-based accumulation mechanism, which splits the accumulation of long vectors into smaller chunks, and consequently, the probability of adding big numbers to smaller ones is reduced. Sakr et al. [250] address the same problem, and they propose to find the optimal bit width of floating point accumulation that minimizes the effect from swamping, by maximizing the variance retention ratio, a statistical property that reflects the quality of computation. Besides accumulation, Hoffer et al. [117] identify the numerical issues in batch normalization that happen when computing the variance, and propose to replace it by numerically stabler alternatives. Outside deep learning, Higham et al. [113] and Blanchard et al. [27] provide theoretical analysis on mixed precision arithmetic units in NVIDIA GPU typically for linear systems, which can inspire further improvement in the mixed precision training field.

Many papers are based on different low precision data types: shorter floating-point (e.g., `fp8` [296], `fp4` [266], `bf16` [139]), as well as other novel data representations, e.g., short integers and fixed-point [58, 17, 325, 103, 307], ternary [47, 124], or even binary formats [52, 53, 54], to speed up DNN training. Notably, Zhang et al. [337] present a novel fixed-point based DNN training method that adapts the fixed-point representation during training for varying gradient distributions across layers, which has similar insights as our approach 3.2 towards loss scaling for floating-point.

2.2 Reconfigurable computing

The deployment target for DNN in this thesis is a reconfigurable platform, specifically, a Field-Programmable Gate Array (FPGA). Unlike CPU and GPU, FPGA can be customised at the circuit-level for specific applications to achieve better performance and power efficiency. Compared with ASIC, another technique to build application-specific circuits, an FPGA takes much shorter time and lower budget to build and iterate, and an FPGA can be easily reconfigured for different applications [132]. Nevertheless, a big obstacle to gain these benefits from FPGA, especially for software developers who intend to improve their applications running on CPU or GPU, are the FPGA design methodologies. FPGA designs are traditionally implemented by hardware engineers using conventional Electronic Design Automation (EDA) tools and Register-Transfer Level (RTL) abstraction, which has a steep learning curve for software developers. High-Level Synthesis (HLS), a methodology that translates high-level software descriptions directly down to low-

level hardware designs, can be a potential solution for the aforementioned problem and has already empowered the development of many applications, including those in high-performance computing and deep learning. This section describes what reconfigurable devices are (Section 2.2.1), typical design methodologies (Section 2.2.2), including the roofline model and high-level synthesis, and example applications (Section 2.2.3). Todman et al. [273] provide more comprehensive review on reconfigurable computing in general.

2.2.1 Reconfigurable devices

A high-level abstraction of a reconfigurable device, typically, an FPGA, is an integrated circuit interconnects massive building blocks, on which the interconnections are reconfigurable. Building an FPGA design is to configure the interconnections among these building blocks, so that the resulting circuit can carry out the desired computation. This section describes common FPGA building blocks, FPGA architectures, design process, and design optimisation methods that are especially useful for high-performance computing and deep learning.

FPGA building blocks

There are four types of building blocks on an FPGA that are commonly identified (e.g., in datasheet [316, 321]):

- A Look-Up Table (LUT) unit conceptually represent a *truth table* [6], a mathematical construction that outputs a 1-bit true-false value for any fixed-width binary input. Internally, a LUT can be implemented by connecting SRAMs by cascaded MUXs. For example, a 4-input LUT has 16 SRAM bits that can be connected by a tree of 15 number of 2:1 MUXs [4, Figure. 2] to produce a 1-bit output. A LUT is versatile and can implement any boolean algebra equation among a fixed number of bits.
- A Flip-Flop (FF) unit, or a *register*, represents state in an FPGA design. An FF connects an 1-bit clock signal and a 1-bit data input, and it produces its *stored* bit as output. When the clock signal rises, i.e., changes from 0 to 1, the FF stores the current data input, which consequently changes the following outputs. We can leverage FFs to implement a Finite State Machine (FSM) [158] or a *pipelined* design [300, 252].

- A Block Random Access Memory (BRAM) block is used to implement *on-chip memory*, which stores larger amount of data that persist longer than FF. It can expose read and/or write ports, which read data from or write data to specific addresses when the connected clock signal rises. A BRAM block is characterised by its *width*, how many bits of data that a certain address can associate, and *depth*, the size of its address space. UltraRAM (URAM) [314] blocks are bigger but less versatile variants of BRAMs that appear in some recent FPGA boards [321].
- A Digital Signal Processing (DSP) block is a special computation unit that is built for efficient arithmetic operations. An arithmetic operation implemented by DSP can run much faster than using other resources, e.g., LUT and FF. A DSP block can be configured to multiple tasks, for example, a DSP48 [317] can carry out multiplication, accumulation, and even Single-Instruction Multiple-Data (SIMD) operations.

Whether these resources are fully and properly utilised determines how good an FPGA design can perform.

FPGA architecture

An FPGA architecture organises these building blocks in a reconfigurable way. A typical FPGA architecture has three components [273, 38, 4] (Figure 2.10):

- A Configurable Logic Block (CLB) wraps LUTs, FFs, arithmetic carry logic (e.g., adders with carry chains), and MUXs into a single processing unit [321]. It is also referred to as Adaptive Logic Module (ALM) [4] or *slice* [315] by different vendors.
- Programmable *interconnects* are switch boxes and routing channels that connect CLBs and other blocks, e.g., DSPs, BRAMs, I/O blocks, etc. These interconnects have a matrix-like placement, and they can be reconfigured to direct signals among blocks in different manners.
- *I/O blocks* handle the input and output requests between the FPGA chip and the external world, e.g., memory and network interfaces.

To leverage an FPGA architecture, we need *place and route* [138], the key FPGA design procedure that determines which blocks to use and how they are interconnected

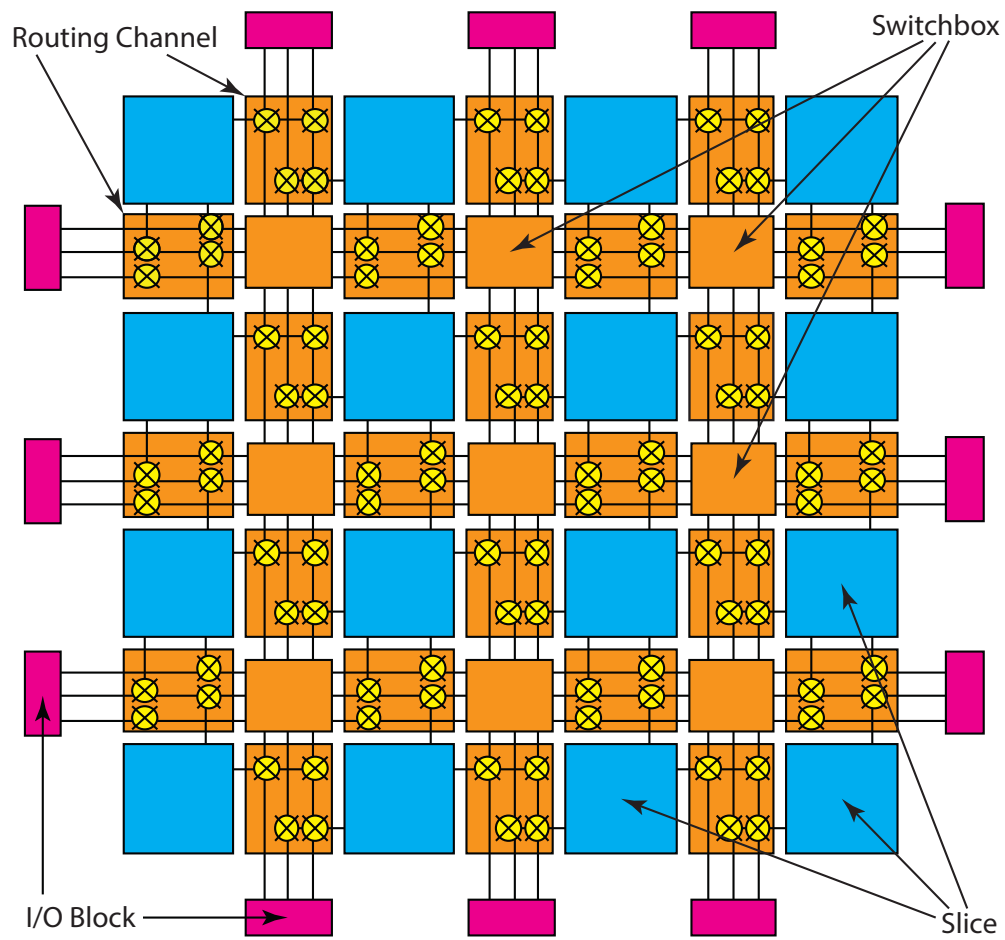


Figure 2.10: An illustration of an FPGA architecture from Kastner et al. [142, Figure 1.3].

for a specific hardware design. If the interconnected blocks are close, the resulting design can have better *timing*, i.e., the delay it takes to propagate signals among adjacent FFs. And the path that has the maximum delay is the *critical path* [115, 205], which longer the critical path is, slower the design will be. Some modern FPGA architectures [321] have multiple dies that are physically more distant, and paths interconnecting them have longer delays, which requires more considerations for place and route [313, 293].

Design process

An FPGA engineer normally starts from describing the desired hardware designs using certain RTL-based HDLs [279]. RTL abstracts synchronous circuits, in which the changing in state is synchronised across the whole design. Engineers working with RTL need to connect wires among registers and other logic operators, which is at a much lower level than normal software descriptions. In Section 2.2.2, we describe ways to build designs beyond RTL.

Once the RTL code is written, a *synthesis* tool [318] takes in the RTL descriptions and produces a *netlist* that contains the connectivity information among LUTs, FFs, and other components. A synthesis tool can optimise the design by simplifying the logic using boolean algebra, or choosing FPGA building blocks with regards to an objective that either minimising the area (resource usage) or maximising the speed (clock frequency).

Starting from the synthesised netlist, an *implementation* [322] tool carries out the previously mentioned place and route process for a specific FPGA chip. The place and route solves another constrained optimisation problem that tries to find the optimal placement under resource (not to overuse existing units) and timing constraints (not to have critical path longer than a threshold). Place and route is computation-intensive, which can take several hours and even days [254].

Finally, we can generate a *bitstream* file from the placement result that can configure the target FPGA chip. We also need to provide the accompanying software that can interact with the design on FPGA to complete an application.

Design optimisation for throughput

There are several objectives to consider when optimising an FPGA design.

- *Latency* measures the time (s) it takes to complete a single operation. It is better to have lower latency.
- *Throughput* denotes the number of operations that within a unit of time the design can process (op/s). It is better to have higher throughput.
- *Area* refers to the hardware resource usage of the target design. It is more efficient if the same design can be implemented with fewer resources.
- *Power* indicates how much energy that the design consumes within a unit of time during processing the input (W). A design is more energy efficient if the power is lower.

It is rare that we can achieve all these objectives at once. A common scenario is to optimise for one or two objectives and balance the trade-off among them all. This thesis focuses on improving the throughput mainly because DNN applications should process inputs in batches, and an increased throughput can reduce the run-time more significantly than the other metrics. There are two typical design optimisation techniques aiming for higher throughput: pipeline and replication.

Pipeline Intuitively, pipelining a hardware design is to insert registers between combinational logic. Pipelining can break the critical path into smaller parts, and therefore, improve the timing. And more importantly, a pipelined design can run the same task for multiple input simultaneously: suppose a pipeline has N stages (there are N cascaded registers), then at most N input elements can be processed at the same time within the pipeline at different stages. Consequently, the throughput of the design is improved by N times given N pipeline stages. There are two drawbacks though: the latency can be increased since now we need N clock cycles to complete the computation of a single input, and there are more resource usage. A systematic way to pipeline a design, typically described in RTL, is *retiming* [173, 172, 259, 194, 50].

Replication A Processing Element (PE) carries out the task of a specific operation. And if we *replicate* the same PE N times, we can get a design that can ideally achieve N times higher throughput than the original version. It is especially suitable for applications that have SIMD structure. The more replications we have, the more resources the design needs to use. The impact on latency is less explicit: in cases that additional map (distributing input across PEs) and reduce (gathering results from all PEs) logic should be deployed, the latency will be higher due to these added components.

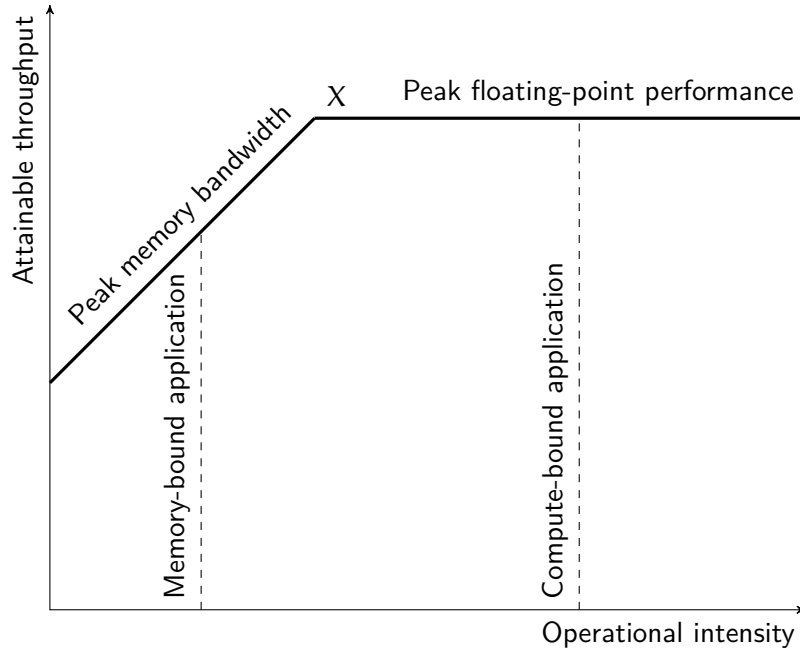


Figure 2.11: Visualizing the roofline model. Operational intensity is on the x-axis (with unit flop/B), and the attainable performance (flop/s) is on the corresponding y-axis. There is a ‘roofline’ that captures both the constraints by memory bandwidth (the sloped line) and peak (floating-point) computation performance (the horizontal line) of a processor. The angle of the sloped line depicting memory-bound performance is the off-chip memory bandwidth (B/s).

2.2.2 High-level design methodologies

This section describes three high-level design methodologies that this thesis leverages: we use the roofline model to understand the peak throughput of a design, we carry out the Design Space Exploration (DSE) process to find the optimal design under certain constraints, and we describe the design using High-Level Synthesis (HLS).

The roofline model

The roofline model [304] provides a valuable insight into how off-chip memory bandwidth can constrain processor performance. The key contribution from the roofline model is the proposal of the *operational intensity* measurement, which means the number of operations executed per byte of DRAM traffic, and it suggests the DRAM bandwidth required by a particular computation kernel. In Figure 2.11, if an application has low operational intensity, the application is memory bound, and its peak attainable performance cannot exceed the product between the operational intensity and the memory bandwidth (the left dashed line); on the other hand, if the operational intensity is high and surpasses the intersection of the roofline, X,

then the peak performance will be constantly the peak floating-point performance provided by the system (the right dashed line).

Closely related to this thesis, Zhang et al. [331] first use the roofline model to analyse and optimise FPGA-based CNN accelerator design. Given all the possible design choices, which can be characterised by their operational intensity and attainable performance, i.e., points in Figure 2.11, we can use the off-chip memory bandwidth and the peak floating-point performance given by target platform to draw the roofline and pick out best design choices within the boundary. Their follow-up work, Caffeine [332], further explores the roofline model for more variants of CNN accelerators.

Design space exploration

An FPGA design normally comes together with configurable parameters, e.g., the number of PE replicates, the data representation to use, etc. The union of the domains from these parameters can be seen as the *design space*, and each point in the space is a design of a specific configuration. Not all of the design points are feasible due to the constraints from the target hardware, e.g., available resources and technology. Therefore, we need to explore the design space to find the design point that is optimal and yet feasible.

$$\mathbf{x}^* = \arg \min_{\mathbf{x} \in \mathbf{X}} f(\mathbf{x}) \quad (2.9)$$

In theory, we can formulate Design Space Exploration (DSE) as a *constrained optimisation* problem, in which the objective is a *cost function* that maps from the design parameters to a single, or multiple cost values that we intend to minimise. A DSE problem can be formulated as Equation 2.9, in which $\mathbf{x}$ is a single design point, $\mathbf{X}$ is the design space, and f is the cost function. Our ultimate goal is to find $\mathbf{x}^*$. One key challenge to DSE is that the cost function is often a *black-box*, i.e., the analytical form between parameters and the cost is unknown. And evaluating this black-box function can be time-consuming, since synthesizing and implementing an FPGA design can take several hours (Section 2.2.1).

There are extensive research on this topic [3, 260, 344, 345, 218, 217, 237]. Nardi et al. [218] propose HyperMapper, a DSE framework for FPGA that leverages the machine learning based hyper-parameter tuning techniques [24, 23, 7]. The roofline model is used for cost function estimation as well [217, 331]. Moreover, DSE is a crucial step to take to get high throughput DNN accelerator designs [237, 331, 332].

```

1  void dot_product(float A[10][32], float B[10][32], float C[10][32]) {
2      #pragma HLS ARRAY_PARTITION variable = A dim = 1 complete
3      #pragma HLS ARRAY_PARTITION variable = B dim = 1 complete
4      #pragma HLS ARRAY_PARTITION variable = C dim = 1 complete
5      Loop1: for (int i = 0; i < 10; ++ i) {
6          #pragma HLS UNROLL
7          Loop2: for (int j = 0; j < 32; ++ j) {
8              #pragma HLS PIPELINE
9              C[i][j] += A[i][j] * B[i][j];
10         }
11     }
12 }

```

Listing 1: An example code to Vitis HLS. This function implements the dot-product calculation among 10 different vectors of length 32. Vitis HLS directives are represented using the C compiler pragmas, specifically, `ARRAY_PARTITION` instructs the compiler to expand the first dimension of every input array into separated 32-length partitions so that they can be accessed in parallel, `UNROLL` replicates the inner loop (Loop2) by 10 copies, and `PIPELINE` indicates the innermost loop should be implemented by pipelines. The resulting design has 10 replicates of a pipelined dot-product processing unit that operate in parallel on different partitions of the input arrays.

High-level synthesis

Building FPGA designs in RTL can be tedious: describing a design based on a given specification in RTL needs engineers attention on how each wire is connected among components, not to mention the extra effort to take if the initial design should be further optimised. Neither building a pipeline nor making replications is trivial to achieve. High-Level Synthesis (HLS) [204, 55] aims to alleviate this burden from hardware engineers: instead of implementing in RTL, they can describe the design at a much higher-level, e.g., using software programming languages, and HLS tools will take care of the rest by automatically generating RTL descriptions. More importantly, just by providing *optimisation directives* to HLS tools, we can control how the RTL designs will be generated for specific objectives (Listing 1). This saves huge amount of time engineers would have devoted to iterate on the RTL designs to make improvement. HLS also provide an alternative way for software developers to program FPGA, which is more familiar and has a much less steep learning curve than RTL.

Many HLS tools starts from imperative languages like C/C++, OpenCL, and Java [319, 320, 126, 127, 338, 40, 9, 98, 202]. Among these, this thesis mainly use Vitis [319] for the compiler study in Chapter 4, and MaxCompiler [202] for building DNN in Chapter 5. Some other works are built upon functional programming lan-

guages, e.g., Lava [26], Ruby [102], Chisel [11], and Spatial [152]. There are papers on building the compiler infrastructure that supports HLS, e.g., CIRCT [49], and Calyx [219].

Even that HLS tools is thriving in many fields, including HPC and deep learning, there are still many challenging problems. For example, loop pipelining and splitting [180, 181], data reusing [230], memory partitioning [299], loop transformations [352], design generation [51, 295], and dynamic scheduling [135, 46]. Many of these solutions leverage the polyhedral model, which is described in Section 2.3, and its interaction with HLS tools is the key field of study in Chapter 4. These works, especially [352], are seen as the state-of-the-art of the problems that POLSCA Section 4.4 aims to solve.

2.2.3 Deep learning on reconfigurable platforms

This section introduces the background on using reconfigurable platforms to deploy DL applications, which is one of the major focuses of this thesis. Given that this thesis only covers CNN, although there are extensive research on accelerating many other DL models, e.g., RNN [100, 233], we mainly review recent progress of CNN on reconfigurable platforms. Not just the accelerator designs are reviewed, this section also discusses the corresponding tools that maps CNN topologies to low-level hardware, which is also an important topic of this thesis.

This section contains the materials that describe the recent study for Challenge C3 (Section 1.2.3).

Accelerator taxonomy

We first review and categorise different approaches to design high-performance accelerators for CNN. A common taxonomy is to divide CNN accelerators into Convolution Layer Processors (CLPs) and Streaming Architectures (SAs).

Convolution Layer Processor A majority of existing CNN accelerators use a single CLP, which executes a single convolution layer at a time and transfers data between FPGA and the host before and after processing. Zhang et al. [331] propose one of the earliest CLP designs since the era of Deep Learning started, which features the usage of the roofline model for design space exploration. Ma et al. [191, 192]

optimise CLP by loop transformations to improve data reuse and save buffer usage. They [193] also customise their designs for residual connections primarily to orchestrate the data flow for the element-wise adder in a generic CLP. Facilitating the overlay technique, Light-OPU [328] provides a CLP design that we can configure to run different DNNs, which has a significantly different design philosophy compared to SA. Another flavour of CLP implements convolution by matrix multiplication [265, 99], which can achieve high throughput, but imposes an overhead to reorder the input and output images before and after applying matrix multiplication. Moreover, exploiting sparsity [189, 224] and custom data formats [277, 294] can also improve CLP performance, which resonates with the discussion in Section 2.1.2.

Beyond single CLP, some recent papers aim to allocate multiple, heterogeneous CLPs to scale up throughput even further. One direction is to leverage layer fusion, which maps repetitive components of a DNN into a single pipelined design, as seen in [197, 309, 183, 323]. A layer-fused design keeps intermediate results on-chip, which improves power efficiency by reducing off-chip memory transactions. Another direction is to place discrete CLPs with different capabilities, e.g., CascadeCNN [154] provides two processors for high- and low-precision, and DNNE Explorer [336] proposes a heterogeneous design that handles different workloads of a CNN by an SA and a CLP, depending on their characteristics.

Streaming Architecture In contrast to CLP, a SA-based CNN accelerator aims to put not just a single layer, but the whole network onto the target hardware. A major benefit from leveraging SA is, similar to layer fusion, that the off-chip memory transactions are significantly reduced. Ideally, only the initial input to the network and the final output should be transferred between the accelerator and the host. Meanwhile, we can improve the throughput even more by exploiting the pipelining opportunities across the whole network, since all operators are executed concurrently in the accelerator, which a CLP-based design can hardly achieve. This thesis focuses on SA techniques.

There are several SA-based DNN accelerators. Venieris et al. [283] propose an SA mapping method for DNN based on the synchronous data-flow model. Their method can partition a DNN into multiple parts, each of which can fit into an FPGA. Shimoda et al. [256] optimise inter-layer pipelining of an SA that supports tiling. Their design is highly customised for a specific network, MobileNet-v1 with PSPNet though. DNNBuilder [335] also targets mapping a whole DNN onto an FPGA using an SA design, while DNNBuilder targets only sequential DNNs, which lack of the topologies in efficient networks (Section 2.1.2). These pieces of literature concentrate

Table 2.4: Comparison among CLP, layer fusion (LF), and SA. Resource Eff. indicates the resource efficiency that different architectures can achieve, and among the listed CLP can be the most efficient; Off-Chip Trans. lists the bandwidth requirements of memory transactions between FPGA and off-chip memory, of which SA demands the least.

	Mapping	Resource Eff.	Off-Chip Trans.
CLP	Single layer	Highest	Highest
LF	Multi-layers	$\leq$ CLP	Moderate
SA	Full	$\leq$ CLP	Lowest

on inference DNNs, and there are other papers on DNN training. Given that DNN training demands efficient inter-layer pipelining, LayerPipe [278] and PipeLayer [261] propose different approaches by constrained optimisation or batch-level computation overlapping. The constrained optimisation techniques from LayerPipe are useful for DNN training and too heavy-weight for DNN inference. PipeLayer designs pipelining techniques for batched input, while inference models often take a single image at a time.

Table 2.4 summarises the major differences among CLPs, layer-fusion designs, and SA. Generally speaking, a CLP uses less on-chip memory and imposes more pressure on off-chip memory bandwidth than an SA. Given a sufficient amount of on-chip memory, which is the assumption of this paper and achievable by many recent DNNs optimised for inference, the throughput of an SA can scale up to a higher peak compared with a CLP due to its much less off-chip memory bandwidth requirement.

It is worth to mention that these works are treated as the state-of-the-art for Deacon evaluation Section 5.5.

Supporting tools

When mapping a DNN to FPGA, we can leverage compiler techniques to transform the input graph. How this mapping works heavily depends on the target accelerator architecture: if it is a CLP-based design, then the hardware itself is relatively fixed, and the mapping is mainly about generating instructions for the processors; if the target architecture is based on SA, then the mapping workload is largely on the hardware end, while the software that drives the accelerator will be relatively straightforward that only streaming in and out the data is required.

DNNVM [323] targets layer-fused CLPs. It explores multiple opportunities to transform a DNN graph, so that we can either find efficient hardware designs or software

Table 2.5: Comparison on functionalities with prior frameworks. LF as a design target means layer fusion, and OVL is for CLP equipped with overlay. ILP stands for inter-layer pipeline optimisation in SAs, DWS and RES denote whether a framework targets or optimises for depthwise separable and residual convolutions respectively. The last column specifies the design space to explore, PAR includes parallelism, and SEQ is computation sequence

	Target	ILP	DWS	RES	DSE
fpgaConvNet [283]	SA	✗	✗	✗	PAR
DNNBuilder [335]	SA	✓	✗	✗	PAR
DNNVM [323]	LF	✓	✗	✓	PAR
Light-OPU [328]	OVL	✗	✓	✓	PAR
Deacon (Chapter 5)	SA	✓	✓	✓	PAR & SEQ

instructions. However, their optimisations focus on high-level, tensor arithmetic-based opportunities, which is a general solution, but may lose optimisation opportunities that only appear at a lower level. Once the reconfigurable design is generated, prior papers that partially map a DNN require extra scheduling work to get software instructions [323, 343]. TVM [43] provides a comprehensive framework that compiles ML models onto various devices through layered intermediate representations, which is also adaptive to support CLP-based accelerators.

As mentioned above, one benefit of mapping DNNs to SAs is that there is no extra work required for software scheduling: we only need to prepare the input and collect the output. A selection of the frameworks is compared in Table 2.5.

2.3 Polyhedral Modelling

This section gives a brief review of the key concepts from polyhedral modelling that are relevant to Chapter 4. We demonstrate the fundamental concepts of the *polyhedral representation*, including the affine expression, Static Control Parts (SCoP), etc. We also describe Pluto [30], a *polyhedral scheduling* algorithm that this thesis relies on, and its variants. Moreover, we discuss *polyhedral code generation*, a topic that has almost the same importance as the polyhedral scheduling algorithm in order to achieve high-performance programs. Finally, on the application side, we dive deeper into the topic of polyhedral HLS, which is closely related to this thesis (Section 4.4). For more information beyond the scope of this thesis, Benabderrahmane et al. [22] present a detailed overview of the polyhedral methodology.

2.3.1 Preliminaries

This section elaborates two important preliminary concepts for polyhedral modelling: affine expression and static control parts.

Affine expression. An affine expression is represented as $f(\mathbf{x}) = \mathbf{A}\mathbf{x} + \mathbf{b}$, which maps an integer vector $\mathbf{x} \in \mathbb{Z}^n$ into $f(\mathbf{x})$ by first multiplying it with a constant integral matrix $\mathbf{A} \in \mathbb{Z}^{m \times n}$ and then adding it with a constant integral vector $\mathbf{b} \in \mathbb{Z}^m$. It is frequently used in polyhedral modelling to represent a mapping from loop induction variables to another induction variable, a loop bound, or a memory address. For instance, we can represent the address accessed by `A[2i, 3j + 1]` using the following affine expression:

$$\begin{bmatrix} 2i \\ 3j + 1 \end{bmatrix} = \begin{bmatrix} 2 & 0 \\ 0 & 3 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix}.$$

Here, `i` and `j` are two loop induction variables that encloses the memory access operation.

The definition of the affine expression innately prohibits applying operations other than addition and subtraction between two variables. For instance, we cannot represent `A[i*j]` using affine expression. The properties associated with affine expressions are essential prerequisites to affine analysis, which is not just a key component of the whole polyhedral modelling methodology [71, 72], but also a core technique in a wider compiler community [175, 87, 176]. Integrating non-affine expressions into the polyhedral framework [285] is a topic beyond the scope of this thesis.

Quasi-affine expression. Compared with the original definition of the affine expression, its extension, *quasi-affine expression*, is more pervasively used in polyhedral modelling, e.g., it is a major constituent of many polyhedral tools [286, 289]. In general, quasi-affine expressions further allow integer divisions using `floordiv` $\lfloor \times / c \rfloor$, `ceildiv` $\lceil \times / c \rceil$, and the modulo operator `mod`, given the right-hand side operand `c` an integer constant. It is an expressive extension, for instance, the induction variable of a loop `i` tiled by 32 can be represented as a quasi-affine expression: $\lfloor i/32 \rfloor$, which is not available as an ordinary affine expression. In the following discussion, if not explicitly mentioned, we refer a quasi-affine expression as an affine expression.

Presburger formula. In polyhedral literature, a quasi-affine expression is commonly used together with a *Presburger formula* [262] to represent constraints and relations. A Presburger formula composites quasi-affine expressions with first-order logic ($\neg$, $\wedge$, $\vee$, $\exists$), equal ($=$), and less-than-or-equal ($\leq$) [287]. We can use a Presburger *set* $S[i]$ to describe the instances of statement S under a loop induction variable i that starts from 0 and ends at N with a step size of 2 in the following form: $\{S[i] : i \leq N \wedge 0 \leq i \wedge i \bmod 2 = 0\}$. Meanwhile, we can use Presburger *relation* $S[i] \rightarrow A[\lfloor i/2 \rfloor]$ to indicate that statement S accesses A at index $\lfloor i/2 \rfloor$, given i a loop induction variable. In the next section, we demonstrate how the Presburger formula becomes a cornerstone of polyhedral representation.

Static control part. A Static Control Parts is normally used to indicate a loop nest that can be described using a polyhedral representation. There are certain conditions that a statement should meet to be included in a SCoP:

- *Loop.* The loop boundaries should be affine expressions of constants or other variables defined within the SCoP, e.g., a loop induction variable of an outer loop that is within the SCoP.
- *If-conditional.* An if-statement describes a set of constraints, and similarly to loops, these constraints should also be affine expressions on constants or internal variables.
- *Memory access.* Any statement that accesses to an array, the address should be an affine expression as well.

Listing 2 demonstrates an example SCoP, which meets all the conditions mentioned. We will explain the purpose of these conditions in the next section.

2.3.2 Polyhedral representation

This section briefly describes how to describe a program in polyhedral representation using Presburger formulas. The polyhedral representation of programs uses three key sets and relations [20] to describe a statement: iteration domain, scattering function, and access function.

```

1 void f(const int N, float A[N], float B[N]) {
2   #pragma scop
3   for (int i = 0; i < N; i++) {
4     S1: A[i] = 0.0f;
5     for (int j = i + 1; j < min(N, 2 * i); j++)
6       if ((i + j) % 2 == 0)
7         S2: A[i] += B[i];
8   }
9   #pragma endscop
10 }

```

Listing 2: An example SCoP. All the loops have affine boundaries, the if-conditional describes a quasi-affine constraints, and the statements **S1**, **S2** contains only affine addresses. `pragma scop` and `pragma endscop` mark the begin and the end of a SCoP when using PET [289].

Iteration domain. An iteration domain is a set of statement instances, which are constrained by loop induction variables and their static affine boundaries. We can infer the iteration domain of each statement by taking the joint of the sets defined by affine constraints as specified under loops and if-conditionals. For instance, **S1** and **S2** in Listing 2 has iteration domains as follows, described in Presburger sets:

$$\begin{aligned}
&\{S1[i] : 0 \leq i < N\} \\
&\{S2[i, j] : 0 \leq i < N \wedge i + 1 \leq j < N \wedge j < 2i \wedge (i + j) \bmod 2 = 0\}
\end{aligned}$$

Scattering function. A scattering function describes how to *schedule* statements. We normally use Presburger relations to map a statement to a timestamp, which relatively determines the execution order. For instance, statement **S1** is executed at the beginning of the outer loop, and statement **S2** is executed subsequently within the inner loop, then we can use the following relations to describe their schedule:

$$\{S1[i] \rightarrow [i, 0, 0]; S2[i, j] \rightarrow [i, 1, j]\}$$

Timestamp $[i, 0, 0]$ is lexicographically ordered before $[i, 1, j]$. This representation, which can be traced back to Kelly et al. [146, 144], is just one schedule representation. There are also schedule matrix as in Girbal et al. [81], or schedule tree as in Verdoolaege et al. [292], which are beyond the scope of this thesis.

Access function. Access functions map from statements to memory addresses they access. For instance, we can use Presburger relations as follows to describe the

access functions of **S1** and **S2** in Listing 2:

$$\begin{aligned} &\{S1[i] \rightarrow A[i]; S2[i, j] \rightarrow A[i]\}(\text{write}) \\ &\{S2[i, j] \rightarrow A[i]; S2[i, j] \rightarrow B[i]\}(\text{read}) \end{aligned}$$

In a summary, if a program satisfies the pre-conditions of a SCoP, then for every statement in it, we can deduce their iteration domains, scattering functions, and access functions. In the next section, we describe how to improve a program based on its polyhedral representation.

2.3.3 Polyhedral scheduling

Early research on automatic parallelisation. Polyhedral scheduling is, to some extent, explored to find solutions for automatic parallelisation in the early days. Automatic parallelisation requires a formalisation of the dependence among statement instances, so that we can either find existing statements that have no dependence on each other and are parallelisable, or we should transform the input program to enhance parallelism. The affine expression and polyhedral representation, after years of research, are proven to be feasible and suitable solutions to automatic parallelisation.

The ideas of polyhedral methodology can be dated back to more than 30 years ago. *Array expansion* [69] aims to relocate accessed array elements to avoid unnecessary dependence, and therefore, improve parallelism, by solving an integer programming problem. It imposes affine constraints on the input program that are similar to those for SCoP. Under the same pre-conditions of programs, Feautrier [70] proposes another analytical framework based on affine expressions to solve the problem of finding references or uses, which are embedded in statement instances, of an array element. Later, Feautrier [71, 72] expands the affine formulation into solving scheduling, which is formulated as a problem of assigning execution dates to statement instances, and instances happen earlier should have dates that are lexicographically ordered before dates of those happen later. *Farkas Lemma* [68] is introduced by an algorithm that produces valid scheduling solutions, and under the same framework, there are solutions that minimize latency or have bounded delay [201]. Not just the Feautrier's algorithm, there are many other important scheduling methodologies emerge, e.g., the *hyperplane method* [163] that works on perfectly nested loop with *uniform dependence* [141], the *affine-by-statement* scheduling [56, 57] that extends the hyperplane method by allowing scheduling per statement, and Lim and Lam [175, 176] propose an algorithm that extends Feautrier's algorithm by an abstraction of *affine*

```

1 L1: for (int t = 0; t < T; t++)
2   L2: for (int i = 1; i < N - 1; i++)
3     S: A[t][i] = A[t-1][i-1] + A[t-1][i+1];

```

Listing 3: An example that cannot be rectangularly tiled. An intuitive explanation is that, every row t in each rectangular tile depends on the results from the previous row $t - 1$ in adjacent tiles, e.g., the leftmost column in a tile depends on the result from the neighbour tile on the left $A[t - 1][i - 1]$, and the rightmost column depends on elements in the right tile $A[t - 1][i + 1]$, which suggests that we cannot finish the computation of a tile as a whole without interfering with other tiles, and therefore, the rectangular tiling is invalid.

partition to maximise parallelism while minimizing synchronisation. Feautrier [73] summarises prior research and names the relevant methodologies as *polytope model*.

Tiling and the Pluto algorithm. Loop tiling is another widely researched topic in the compiler community, which is a key technique to enhance locality and boost data reuse. Tiling is a transformation that turns a single loop into a two-level loop nest, the outer loop of which iterates over the tiles (or partitions) of the iteration domain, and the inner loop traverses the internal of each tile. Irigoin and Triolet [130] propose the *supernode partitioning* method, which is a comprehensive solution to the tiling problem. Many other papers continue this journey [324, 116, 240] to optimise tiling for various objectives, however, they are mostly constrained to a setting of perfectly nested loops with a single statement, which is restrictive and not practical for a wide range of problems.

This thesis focuses on Pluto [30, 29, 32], an polyhedral scheduling algorithm that provides a general solution to the loop tiling and parallelisation problem. One of its key contributions is a *cost function*, which minimises the distance of two statements within a dependence. This distance, on the other hand, can be interpreted as the amount of data to be communicated among tiles. By solving the optimisation problem that minimises this cost function, constrained on a legit tiling condition originated from [130], the Pluto algorithm can find a tiling transformation on SCoPs that minimises inter-tile communication.

When the loops cannot be tiled in the rectangular shape, e.g., in stencil computing applications, Pluto automatically *skews* the iteration space to allow rectangular tiling, which may be a suboptimal solution. For instance, the code in Listing 3 will be transformed into Listing 4 by Pluto². This kind of transformation may be a feasible solution for multicore CPU, however, since it results in a *wavefront* parallel

²Using command: `./polycc <code file>`.

```

1  for (t1 = -1; t1 <= floord(T - 1, 16); t1++) {
2      lbp = max(ceild(t1, 2), ceild(32 * t1 - T + 2, 32));
3      ubp = min(floord(T + N - 3, 32), floord(16 * t1 + N + 13, 32));
4      #pragma omp parallel for private(lbv, ubv, t3, t4)
5      for (t2 = lbp; t2 <= ubp; t2++) {
6          for (t3 = max(max(max(0, 16 * t1), 32 * t1 - 32 * t2 + 1),
7                  32 * t2 - N + 2);
8                  t3 <= min(min(min(T - 1, 16 * t1 + 31), 32 * t2 + 30),
9                      32 * t1 - 32 * t2 + N + 29);
10                 t3++) {
11              lbv = max(max(32 * t2, t3 + 1), -32 * t1 + 32 * t2 + 2 * t3 - 31);
12              ubv = min(min(32 * t2 + 31, -32 * t1 + 32 * t2 + 2 * t3), t3 + N - 2);
13              for (t4 = lbv; t4 <= ubv; t4++) {
14                  A[t3][(-t3 + t4)] =
15                      A[t3 - 1][(-t3 + t4) - 1] + A[t3 - 1][(-t3 + t4) + 1];
16              } } } }

```

Listing 4: Code transformed by Pluto from Listing 3. The original loops are skewed in this case. The `pragma omp` is added to annotate the tile loop is parallelisable.

processing pattern, the level of parallelism it achieves is not balanced (lower at the beginning and the end, higher in the middle), which is not promising for devices like GPU. Grosser et al. [92, 93] propose *split tiling* that results in trapezoidal-shaped tiles and *hybrid hexagonal tiling* that produces hexagonal tiles, which are effective for massively parallelised devices like GPU. Bondhugula et al. [31] also integrate the *diamond tiling* idea into polyhedral scheduling. All these ideas can help mitigate the imbalanced parallelism problem with loop skewing and ensure a concurrent start. Grosser et al. [94] compare hybrid-hexagonal tiling and diamond tiling and propose a unified framework. In Chapter 4 we explore these tiling problems for HLS.

Although the original version of the Pluto algorithm can be improved in various aspects, it is still a powerful method to transform and tile loops in general, and that is why we build our polyhedral tools on Pluto (see Chapter 4). There are other scheduling algorithms that emerge from Pluto, for instance, Verdoolaege et al. [291] propose a scheduling algorithm based on Pluto for GPU, which is implemented by isl [286] and is sometimes referred to as the *isl scheduler*. The isl scheduler is later extended by Verdoolaege [290] and Zinenko [348] for better spatial locality. We will leave these work as a future extension of ours.

Application on high-level synthesis. Even in the early days of polyhedral research, analysing and transforming loops for hardware designs was an active topic. Quinton [235] proposes a systematic approach of mapping a problem to a solution in the form of a *systolic array* [161] with the help from Uniform Recurrent System

of Equations [141]. Rajopadhye and Fujimoto [238, 239] extend the mapping algorithm to support affine recurrent equations, and consequently, support mapping data pipeline architectures and control signals. There is still progress in this field until very recently, Wang et al. [295] propose the AutoSA compiler, extended from PolySA [51], which leverages polyhedral representations to automatically transform programs that describe a systolic array architecture, and its optimisations are more pragmatic and FPGA-oriented, compared with prior research.

The polyhedral methodology, in general, has much wider impact on the high-level synthesis community than systolic array. Zuo et al. [352] demonstrate how polyhedral scheduling can be effective for high-level synthesis, especially to analyse data access patterns and use them to transform loops. Pouchet et al. [230] extend the Pluto algorithm to enhance data reusing on FPGA scratchpad memory. Wang et al. [299, 298] leverage polyhedral models to formulate and solve memory partitioning problems. Since the polyhedral representation can describe exactly the dependence among statements, it is a perfect method to analyse and improve loop pipeline [213, 180, 181]. Nevertheless, as what we will demonstrate later in this section, given all the merits from the polyhedral methodology in high-level synthesis, it is still not very popular due to the lack of compiler support.

2.3.4 Polyhedral code generation

A polyhedral scheduling algorithm transforms polyhedral representations that are extracted from a program, however, to make the transformation effective, we need to generate program back from the transformed polyhedral representation. This is the process of polyhedral code generation. Quilleré et al. [234] bring the polyhedral code generation into the focus of the research community. The Quilleré algorithm formulates the problem of generating nested loops as scanning polyhedra. For instance, when there is a single polyhedron of a single statement, the Quilleré algorithm resembles the Fourier-Motzkin elimination procedure to find loop bounds, which is an improvement over an earlier work of Ancourt and Irigoin [5]. When there are multiple polyhedra belong to multiple statements, Quilleré can separate the union of these polyhedra into distinct parts, each of which will have a corresponding loop nest that scans its domain. Finally, Quilleré can generate code from a union of polyhedra in a recursive manner that can balance the trade-off between efficiency and code size. Bastoul [18, 19] later extend Quilleré by allowing non-unit stride, exploiting loop merge and split opportunities, and reducing code size, which result in the CLooG (Chunky Loop Generator) tool. Note that the description here is much simplified.

Chen [42] proposes CodeGen+, a reworked version of CodeGen from Kelly et al. [145]. These methods first generate an initial AST from the polyhedra and then make improvement, while CodeGen+ explicitly explores the trade-off between loop overhead and code size and simplifies if-conditionals. Grosser et al. [96] further extends the AST-based code generation into a framework, which not only provides various novel optimisation techniques, but also allows users to customise code generation for domain-specific applications. Moreover, Zuo et al. [351] customize CLooG for generate polyhedral code that uses instructions more friendly to hardware than the others and explore parallelogram tiling.

2.3.5 Polyhedral compilers and tools

The major focus of this thesis (Challenge C2, Section 1.2.2) is to design better tools for polyhedral optimisation, and consequently, benefit high-level synthesis and deep learning. This section reviews the development of various polyhedral compilers and tools.

Parametric Integer Programming (PIP) [69] is an algorithm that efficiently solves the integer programming problems arising from polyhedral methodology, and a tool (named PipLib³) that later becomes the cornerstone of many polyhedral scheduling algorithms [70]. Omega⁴, a tool that includes the Omega test [232], a dependence testing system, and a unified framework for reordering-based program transformations [147]. PolyLib [302] is another polyhedral transformation library, which is built on the ALPHA framework/language [169] that is used to design systolic arrays. The Quilleré code generation algorithm also targets ALPHA and uses PolyLib to manipulate polyhedral representation. The Parma Polyhedral Library (PPL) [15] is proposed and used in various hardware and software verification projects. Finally, we have LooPo⁵ [89], a source-to-source polyhedral transformation tool. These tools share similar underlying concepts, however, they still have many differences, e.g., supported operations on the polyhedral representations, how to translate between polyhedra and code, etc. And none of these tools works at the scale of an industrial-level compiler.

New tools emerge to focus on some small, but critical components in polyhedral modelling. Clan⁶ and PET [289] are two polyhedral representation extraction tools. They consumes programs typically written in C/C++ and produces polyhedral rep-

³<http://www.piplib.org/>

⁴<http://www.cs.umd.edu/projects/omega/>

⁵<https://www.infosun.fim.uni-passau.de/cl/loopo/>

⁶<http://icps.u-strasbg.fr/~bastoul/development/clan/docs/clan.html>

representations in various format, e.g., OpenScop [20]. These tools first verify that the input program satisfies the SCoP pre-conditions, and if so, they extract loop structures, including the induction variables and their boundaries and steps, and inner statements. The scatter functions and access functions are resolved as well in this process. Meanwhile, Candl⁷ is dedicated to analyse dependences among statement instances. This task can be done by other more fundamental libraries as well, e.g., isl [286]. We also have CLoog [19] that carries out the polyhedral code generation task. These tools are also integrated into the Pluto compiler⁸, a source-to-source polyhedral transformation framework based on the Pluto algorithm [30].

Recently, polyhedral compiler is much closer to the industrial requirements than the past. Polly⁹ [90] successfully puts polyhedral modelling into LLVM [165], an industrial-level compilation framework. Polly is implemented upon isl and is the foundation of various novel polyhedral scheduling algorithms [93, 97]. Graphite is another polyhedral component inside an industrial compiler GCC¹⁰ [276]. It is built on PPL, and features various pragmatic improvements over other polyhedral compilers [160]. R-Stream [206] is a commercial polyhedral compiler, which can perform source-to-source transformation and automatic parallelisation and support many different platforms including GPU and dataflow FPGA.

When deep learning becomes popular, there are some new polyhedral frameworks come out, which take either the deep learning models or tensor computation formulas as input. Given a computation task described in Einstein notation, Tensor Comprehensions [280] can automatically compile it into executable program using polyhedral transformations. Halide [236] is proposed to be a language to define image processing algorithms that can be automatically parallelised by polyhedral transformations, however, it becomes useful for deep learning optimisation as well. We can also find polyhedral representations in TVM [43]. R-Stream·TF [231] applies the R-Stream compiler on TensorFlow.

2.4 The MLIR framework

This section reviews the essential background on the MLIR compilation framework, which is a key component to our work in Chapter 4.

⁷http://icps.u-strasbg.fr/people/bastoul/public_html/development/candl/

⁸<https://pluto-compiler.sourceforge.net/>

⁹<https://polly.llvm.org/>

¹⁰<https://gcc.gnu.org/wiki/Graphite>

2.4.1 LLVM and MLIR

MLIR is an optimizing compiler infrastructure inspired by LLVM [165], a compiler infrastructure. LLVM includes a set of tools and technologies, covering from the frontend of a compiler that translates source code into Intermediate Representation (IR), to compiler backends that gradually lower IRs into optimised machine instructions. A key property of the LLVM IR is its Static Single Assignment (SSA) [247] form: only one variable can be assigned in an instruction, and it is frozen after the assignment. SSA form makes it substantially simpler to implement compiler optimisation. Another benefit from LLVM is that it organises its optimisation techniques into modularised, composable *passes*, which are reusable in the process of constructing other compilers. However, the core LLVM IR is hard to extend, and any other compiler tasks that are not as low-level as the LLVM IR, e.g., transforming Deep Learning computation graph, cannot reuse LLVM passes.

Compared with LLVM, MLIR has a focus on extensibility and modularity [167]. Its main novelty is its IR supporting a fully extensible set of instructions, which are named *operations* in MLIR, with *types* and *attributes*. Specifically, MLIR combines SSA with nested regions, allowing one to express concepts ranging from machine instructions (e.g., floating-point addition) to structured control flow (e.g., loops), from hardware circuitry [49] to large machine learning graphs [86], which overcomes the drawback in the extensibility of the LLVM IR.

```
%c = "dialect.operation"(%a, %b)
      {attribute = #dialect<"value">} ({
// Inside a nested region.
^basic_block(%block_argument: !dialect.type):
  "another.operation"() : () -> ()
}) : (!dialect.type) -> !dialect.result_type
```

Figure 2.12: Generic MLIR syntax depicting an operation with two operands, one result, one attribute and a single-block region. The types of the operation and the block arguments are annotated as well.

MLIR *operations* define runtime semantics of a program and process immutable values. Compile-time information about values is expressed in *types*, and information about operations is expressed in *attributes*. Operations can have attached regions, which in turn contain basic blocks of further operations. The generic syntax, accepted by all operations, illustrates the structure of MLIR in Fig. 2.12. Additionally, MLIR supports user-defined custom syntax.

Attributes, operations and types are organized in *dialects* that resemble modular libraries. MLIR provides a handful of dialects that define common operations such

as modules, functions, loops, memory or arithmetic instructions, and ubiquitous types such as integers and floats. These dialects can be transformed from one to another using modularised MLIR *transformations*, for instance, from a dialect that describing a high-level concept, we can transform it all the way down to a dialect that directly maps to LLVM, which can be converted to actual LLVM IR to generate the final executable program. Next, we present several dialects that are related to this thesis.

2.4.2 Dialects

There are two dialects that this thesis focuses on, *Affine* and *MemRef*. The Affine dialect resembles polyhedral representation, on which we build out polyhedral scheduling tool. The MemRef dialect operates on values of MemRef, which is the common representation of memory in MLIR.

Affine

The *Affine* dialect [210] aims at representing Static Control Parts (SCoPs) (Section 2.3) with explicit polyhedral-friendly loop and conditional constructs.

Values. The values in the Affine dialect are categorised into *symbols*, *dimensions*, and other *non-affine* values, which are common in polyhedral literature. Specifically,

- a symbol is an integer value that is loop-invariant but unknown at compile time, also referred to as program *parameters*, e.g., the dimensions of the input array or other function arguments;
- a dimension is an extension of symbols that also accepts induction variables of affine loops.

Symbols and dimensions have **index** type, which is a platform-specific integer that fits a pointer.

Attributes. MLIR provides two attributes relevant for the Affine dialect. One is *affine map*, which is a multi-dimensional quasi-affine function that maps a list of dimensions and symbols to a list of results. Dimensions and symbols are separated to allow quasi-affine expressions: symbols are treated as constants, which can therefore

be multiplied with dimensions, whereas a product of two dimensions is invalid (Section 2.3.1). For example, $(d_0, d_1, d_2, s_0) \rightarrow (d_0 + d_1, s_0 \cdot d_2)$ is a quasi-affine map, which can be expressed in MLIR as: `affine_map<(d0,d1,d2)[s0]->(d0+d1, s0*d2)>`.

Another is *integer set*, which is a collections of integer tuples constrained by conjunctions of quasi-affine expressions. For example, a triangular set $\{(d_0, d_1) : 0 \leq d_0 < s_0 \wedge 0 \leq d_1 \leq d_0\}$ is represented as `affine_set<(d0,d1)[s0]: (d0 >= 0, s0-d0-1 >= 0, d1 >= 0, d0-d1 >= 0)>`.

It is worth to mention that, although affine map and integer set are both expressed in Presburger arithmetic, FPL [229] provides a fast and comprehensive Presburger implementation that is also inside the MLIR codebase¹¹. This thesis does not take this path to re-implement Presburger arithmetic in MLIR, instead, we decide to convert Affine code into a representation that can be used by other existing, mature libraries that support Presburger operations and polyhedral scheduling, since we care more about functionality than efficiency.

```
%c0 = constant 0 : index
%0 = memref.dim %A, %c0 : memref<?xf32>
%1 = memref.dim %B, %c0 : memref<?xf32>
affine.for %i = 0 to affine_map<() [s0] -> (s0)>() [%0] {
  affine.for %j = 0 to affine_map<() [s0] -> (s0)>() [%1] {
    %2 = affine.load %A[%i] : memref<?xf32>
    %3 = affine.load %B[%j] : memref<?xf32>
    %4 = mulf %2, %3 : f32
    %5 = affine.load %C[%i + %j] : memref<?xf32>
    %6 = addf %4, %5 : f32
    affine.store %6, %C[%i + %j] : memref<?xf32>
  }
}
```

Listing 5: Polynomial multiplication in MLIR using Affine and MemRef dialects.

Operations. The Affine dialect makes use of the concepts above to define a set of operations. An `affine.for` is a for-loop with loop-invariant lower and upper bounds expressed as affine maps with a constant step. An `affine.if` is a conditional construct, with an optional `else` region, and a condition defined as inclusion of the given values into an integer set. Finally, `affine.load` and `affine.store` express memory accesses where the address computation is expressed as an affine map. Listing 5 illustrates the Affine dialect for a polynomial multiplication, $C[i+j] += A[i] * B[j]$. This simple example highlights the fact that MLIR supports, and encourages, IRs from different dialects to be used together.

¹¹<https://github.com/llvm/llvm-project/tree/main/mlir/lib/Analysis/Presburger>

MemRef

A core MLIR type—`memref`, which stands for **m**emory **r**eference—and the corresponding `memref` dialect are also featured in Listing 5. The `memref` type describes a structured multi-index pointer into memory, e.g., `memref<?xf32>` denotes a 1-d array of floating-point elements; and the `memref` dialect provides memory and type manipulation operations, e.g., `memref.dim` retrieves the dimensionality of a `memref` object.

`memref` does not allow *internal aliasing*, i.e., different indices always point to different addresses. This effectively defines away the delinearization problem that hinders the application of polyhedral techniques at the LLVM IR level, e.g., in Polly [95].

Throughout this thesis, we only consider `memrefs` with the default *layout* that corresponds to contiguous row-major storage. In practice, `memrefs` support arbitrary layouts expressible as affine maps, but these are not necessary in this thesis.

Other Relevant Core Dialects

MLIR provides several dozen dialects. Out of those, only a handful are relevant for our discussion:

- The *Structured Control Flow* (`scf`) dialect defines the control flow operations such as loops and conditionals that are not constrained by affine categorization rules. For example, the `scf.for` loop accepts any integer value as loop bounds, which are not necessarily affine expressions.
- The *Standard* (`std`) dialect contains common operations such as integer and float arithmetic, which is used as a common lowering point from higher-level dialects before fanning out into multiple target dialects and can be seen as a generalization of LLVM IR [165].
- The *LLVM* dialect directly maps from LLVM IR instructions and types to MLIR, primarily to simplify the translation between them.
- The *OpenMP* dialect provides a dialect- and platform-agnostic representation of OpenMP directives, which can be used to transform OpenMP constructs or emit LLVM IR that interacts with the OpenMP runtime.
- The *Math* dialect groups together mathematical operations on integer and floating type beyond simple arithmetic, e.g., `math.pow` or `math.sqrt`.

2.4.3 Applications

This section briefly reviews compiler applications built on MLIR.

Deep Learning. MLIR becomes a part of TensorFlow [86] almost since the release date of MLIR. There are several dialects defined for TensorFlow, e.g., TensorFlow IR that describes the TensorFlow computation graph, XLA HLO IR that resembles the existing IR in the TensorFlow and linear algebra compilation framework XLA ¹², etc. Similarly, Torch-MLIR [275] is another effort that brings MLIR into the deep learning framework PyTorch [226], which aims to convert representations in the current PyTorch compilation flow, e.g., TorchScript, into MLIR and reuse existing MLIR passes to support different backend devices (CPU, GPU, etc.). Vasilache et al. [282] provide a comprehensive study on using MLIR to compile tensor computation programs in a composable and modularised way.

Polyhedral compilation. Even though Affine provides a fundamental polyhedral representation and corresponding transformations, there are much more to be accomplished to have a MLIR-based polyhedral compilation workflow as powerful as existing polyhedral tools. PET-to-MLIR [153] redirects the schedule tree [292] output from PET [289] into MLIR. Bondhugula [28] carries out a detailed comparison between MLIR and other libraries and compilers, including Pluto, on the GEMM application.

High-level synthesis. ScaleHLS [327] is a HLS framework built upon MLIR, which uses MLIR transformations (e.g., tiling, unrolling) on source program and emits C++ code in the end with annotated HLS directives. For the new Versal AIEngine devices, Xilinx releases an MLIR-based tool-chain [312] that produces instructions for them. LLHD [253] focuses at a lower-level of circuit synthesising and proposes a unified IR that represents vendor-agnostic transformations on hardware description languages. Even though LLHD predates MLIR, it is gradually integrated into the MLIR-based CIRCT project [49].

¹²<https://www.tensorflow.org/xla/architecture>

2.5 Summary

This chapter gives a comprehensive description of the background for this thesis, including the preliminary knowledge and the related work. As a recap, here we highlight the gaps between the each of the challenges and the best result that the related work achieves.

- The challenges in mixed precision training (Section 1.2.1) are targetted by [209, 159, 271, 272, 208, 266]. These papers mainly use the idea of loss scaling, which can partially solve the problems, however, they normally require hyperparameter tuning, and they cannot easily adapt to the situation appeared dynamically during training.
- The structured pruning challenges (Section 1.2.1) appear in [334, 128, 311, 339, 310, 122, 297, 227, 223, 301, 170]. They provide different ways to build GConv-based models, either by training from scratch or by pruning pre-trained models; and those papers using pruning may leverage group regularisation. These existing GConv-based structured pruning approaches need long training or fine-tuning time, and it is not mentioned that how we can choose the number of groups per GConv layer.
- Existing polyhedral tools [69, 232, 302, 169, 15, 89, 289, 30, 90, 206, 351], either research-oriented or industrial, allow us to leverage the polyhedral methodology at various stages: polyhedral representation extraction, scheduling, and code generation. Unfortunately, most of these tools (except Polly [90]) are not designed in a modularised, reusable, and extensible way, which limits the optimisation opportunities we can explore from a compiler perspective. Meanwhile, other than Zuo et al. [351], the support for high-level synthesis is also limited from existing tools. This situation leaves a research gap for Challenge C2 (Section 1.2.2).
- There are many papers and projects on accelerating DNN through FPGA [331, 191, 192, 193, 328, 265, 99, 189, 224, 277, 294, 197, 309, 183, 323, 283, 335, 256], and they broadly fall into two categories, Convolution Layer Processor and Streaming Architecture. As the FPGAs are having more resources, we can leverage their potential by putting a whole DNN on the board, which is a topic rarely explored by prior work (e.g., [283]). Even if this challenge (C3, Section 1.2.3) is targetted, there is still a gap between the existing approaches and the new DNN topologies that feature more complicated connectivities, which are not yet properly addressed for streaming architectures.

Chapter 3

Memory-Efficient Deep Learning

3.1 Introduction

Deep Learning is memory-intensive: a powerful deep learning model can have millions, or even billions of parameters. For example, GPT-3 [39], one of the most recent language model, has 175 billion parameters, which demands 700 GB memory space if all parameters are stored in FP32 that can hardly fit in a single GPU. Things can become worse when we try to deploy a trained model: the target environment may have only a few megabytes memory space available. Both cases motivate us to find solutions that can mitigate the pressure on memory from Deep Learning applications, which is the overarching challenge that this chapter aims to address.

The life-cycle of a Deep Learning model consists of two stages: the *training* stage and the *inference* stage, and there are two big categories of memory reduction method: *quantisation* and *pruning*. Generally speaking, quantisation is about adapting lower-precision data types. Lower-precision means shorter bit-width, which can effectively reduce memory footprint. For example, we can use half-precision floating-point in training tasks [209], or use short integer types (e.g., `int8`) to perform inference [131]. Pruning, on the other hand, is a method that aggressively removes “unimportant” weights, which have negligible effect on accuracy if being removed, to save more memory space [171]. These weights can be close to zero, or relatively small compared with others in the same layer. More in-depth introduction on both methods can be found in Chapter 2.

In this chapter, we build upon the established concepts of quantisation and pruning, while introducing innovative solutions to tackle two critical challenges associated with their utilization:

- *Numerical issues in quantised training.* While training with low-precision, it is very likely that the gradient values cannot be properly represented since they have large dynamic ranges. An out-of-range gradient value can cause either overflow or underflow, and affect training results. Mixed precision training (MPT), a technique that jointly uses low- and high-precision data types, has been proposed to address this problem. But it still has practical problems, e.g., tedious hyperparameter searching, less satisfying heuristics, etc.
- *Unstructured pruning result.* If we greedily remove as many as the unimportant weights until we reach the target memory footprint, the resulting model can have an unstructured sparsity pattern, e.g., the distribution of the number of zeros on each line varies much. This can lead to unbalanced workload across parallelised processing units, which is a red flag for bad performance. On the other hand, however, considering both the structure and the weight importance can significantly raise the complexity to finding the optimum.

This chapter features two novel techniques: One is Aegis, **An Efficient Gradient Scaling** based mixed precision training method. It is designed to address the numerical problems in MPT, especially underflow, by calculating a *loss scale* for each gradient tensor during training. Loss scale was first proposed in [209], which was originally carried out by scaling up the loss value by a fixed scale before backpropagation. Its major drawback is the need to search for a good loss scale, which can be time-consuming. Alternatively, Aegis updates the loss scale for each layer on-the-fly using a novel probabilistic model, which makes MPT more efficient and stabler. We assess the advantages of Aegis by considering not only the model accuracy, a prominent metric highly regarded by the ML community, but also the ease of achieving stable training. Specifically, we investigate whether a MPT algorithm necessitates extensive exploration of hyperparameters (e.g., loss scale) to avoid divergence. In Section 3.2.3, we present compelling evidence that Aegis attains comparable, if not superior, model accuracy compared to other MPT methods across various computer vision tasks, all while eliminating the need for hyperparameter search efforts typically required by alternative approaches.

The other technique is SPGC, **Structured Pruning by Group Convolution**, the aim of which is to reduce the computation budget of a trained DL model in a way that can be translated into faster inference speed. This objective requires us to prune the model in a structured pattern, so that the inference workload can be balanced among processors running in parallel. SPGC chooses group convolution (Section 2.1.2) as the pruning target and formulates the pruning process as an optimisation problem that can be heuristically solved. With SPGC, we can more

effectively balance the trade-off between model accuracy and inference speed. The key metric we focus on is the model accuracy achieved by a pruned model while adhering to reduced computational and storage budgets. In this regard, our method SPGC demonstrates comparable performance to various other approaches when applied to CIFAR, ImageNet, and CUB-200 image recognition tasks. Moreover, we recognize the significance of assessing the time required for searching a suitable pruning recipe and obtaining the pruned model. Unlike many existing methodologies that need fine-tuning or regularization during the pre-training stage, SPGC leverages its underlying heuristic algorithm to efficiently explore the Pareto front encompassing model size and accuracy (Figure 3.7).

3.2 Mixed Precision Training by Gradient Scaling

This section illustrates the Aegis method in details. We first demonstrate how we estimate the scale factor for a given gradient tensor during training time (Section 3.2.1). Then we present the gradient scaling algorithm that integrates this estimation model with backpropagation (Section 3.2.2). We list experimental results in the end (Section 3.2.3).

Notation	Explanation
$\lambda_{\min}, \lambda_{\max}$	The representation range of FP16: $[\lambda_{\min}, \lambda_{\max}] = [6 \times 10^{-8}, 65504]$
$\mathbf{X}, \mathbf{W}$	Activation and weight tensors.
$\nabla \mathbf{X}, \nabla \mathbf{W}$	Gradients of activation and weight tensors.
MixedGEMM($\mathbf{X}, \mathbf{Y}$)	The mixed-precision GEMM function: $\mathbf{Z} \leftarrow \alpha \mathbf{X} \mathbf{Y} + \beta \mathbf{Z}$. Here $\mathbf{Z}$ is the matrix being updated with the multiplication result between $\mathbf{X}$ and $\mathbf{Y}$. Coefficients α and β are ignored in the notation without loss of generality. The multiplication between $\mathbf{X}$ and $\mathbf{Y}$ are performed in mixed-precision.
typecast($\mathbf{X}$)	The explicit typecast function.

Table 3.1: Notations used for describing Aegis (Section 3.2).

Table 3.1 summarises all the notations being used in this section.

3.2.1 Estimate Gradient Scale

The preliminaries are reviewed in Section 2.1.3.

Underflow happens when non-zero FP32 values are too small to be represented by FP16, which implies that the FP32-to-FP16 *typecasting* should be the target to address underflow. In MPT, gradient is often typecast at two places:

1. Mixed-precision GEMM that computes gradients during backpropagation implicitly typecasts intermediate results in FP32 to FP16 before output.
2. Gradients can be typecast from FP32 to FP16 during the backward pass of an explicit FP16-to-FP32 operator, which is typically inserted before precision-sensitive operators like softmax.

Typically, if the FP32 input is scaled up before typecasting, the underflow rate of the typecast result will be reduced. This trick, which is often used in other mixed-precision scenarios [27], inspires us to devise gradient scaling for MPT.

The objective of gradient scaling is to adopt scaling before typecasting to reduce the underflow rate. To formulate and then solve this objective, we need to quantify to what extent the underflow rate will be reduced by a given scale on actual gradient data. In this section, we construct a statistical model to estimate the mapping between the underflow rate and gradient scale, which helps us decide the gradient scale for various typecasting operators.

Assumptions about Gradient Distribution

Intuitively, the underflow rate of $\nabla \mathbf{X}$ after casting from FP32 to FP16 can be formulated as $P(|\nabla \mathbf{x}| \leq \lambda_{\min})$, the CDF of the absolute gradient values evaluated at $\lambda_{\min}$. If $\nabla \mathbf{X}$ is scaled by α , the underflow rate γ then becomes:

$$\gamma = P(\alpha |\nabla \mathbf{x}| \leq \lambda_{\min}) \quad (3.1)$$

Equation 3.1 relates the underflow rate and the gradient scale. Our model should characterize $P(|\nabla \mathbf{x}|)$ to reveal the underlying mapping between α and γ . However, precisely characterizing the distribution of all gradients is intractable. Therefore, we adopt the following assumptions that are commonly used in prior papers to derive mathematically tractable models.

Assumption 1. $\nabla \mathbf{X}$, $\mathbf{X}$, and $\mathbf{W}$ are independent of each other, and their elements are mutually independent and identically distributed, which can be characterized by random variables $\nabla \mathbf{x}$, $\mathbf{x}$, and $\mathbf{w}$, respectively.

Assumption 1 is used in recent papers [109, 83] to derive the statistical relationship among variables during forward and backward propagation for scenarios like network initialization.

Assumption 2. Product terms in GEMM are mutually independent and identically distributed in zero-mean normal distribution, i.e., $\mathbf{p} \sim \mathcal{N}(0, \sigma^2)$, where $\mathbf{p}$ is the random variable representing any product term.

A linear layer, e.g., convolution, applies GEMM on $\mathbf{X}$ and $\mathbf{W}$ in the forward pass to produce output $\mathbf{Y} = \text{MixedGEMM}(\mathbf{X}, \mathbf{W})$. During backpropagation, $\nabla \mathbf{Y}$ is multiplied by the Jacobian $\frac{\partial \mathbf{Y}}{\partial \mathbf{X}} = \mathbf{W}$ to propagate the gradient to the previous layer, i.e., $\nabla \mathbf{X} = \text{MixedGEMM}(\nabla \mathbf{Y}, \mathbf{W})$. Assumption 2 is needed to model the distribution of the output from this GEMM routine. Sakr *et al.* [250] assume that product terms are *i.i.d.* with zero mean. Based on our empirical observation, we additionally suppose that these product terms share a normal distribution.

Assumption 3. Gradients propagated from softmax has a log-normal distribution.

We need to typecast softmax input from FP16 to FP32, and therefore, gradients propagated from softmax will be typecast from FP32 to FP16. Assumption 3 is applied to estimate the underflow rate of gradients in such cases. Since softmax gradients are exponentially related¹ to the normally distributed activations (Assumption 2) output from the last linear layer, it is plausible to assume that they are log-normally distributed.

Statistical Model

We first introduce the following lemma, which can be directly derived from the assumptions given above.

Lemma 1. Gradients $\nabla \mathbf{X}$ calculated from $\text{MixedGEMM}(\nabla \mathbf{Y}, \mathbf{W})$ are normally distributed with zero mean and standard deviation

$$\sigma_{\nabla \mathbf{x}} = \sqrt{N} \sigma_{\nabla \mathbf{y}} \sigma_{\mathbf{w}}, \quad (3.2)$$

where N is the accumulation length in this GEMM.

Proof. Under Assumption 2 each element in $\nabla \mathbf{X}$ is an accumulation of N product terms $\{\mathbf{p}_i\}_{i=1}^N$ from $\mathcal{N}(0, \sigma_{\mathbf{p}}^2)$. Given Assumption 1, $\sigma_{\mathbf{p}}$ should be $\sigma_{\nabla \mathbf{y}} \sigma_{\mathbf{w}}$.² We can then derive Equation 3.2 since $\{\mathbf{p}_i\}_{i=1}^N$ are independent of each other. $\square$

Lemma 1 characterizes the distribution of a gradient produced by GEMM. The following theorem adopts this lemma to provide the underflow rate for such scenario.

Theorem 3.1. Given scale α , the underflow rate of $\nabla \mathbf{X} = \text{MixedGEMM}(\alpha \nabla \mathbf{Y}, \mathbf{W})$ when casting results from FP32 to FP16 is

$$\gamma = P(\alpha |\nabla \mathbf{x}| \leq \lambda_{\min}) = \text{erf} \left(\frac{\lambda_{\min}}{\alpha \sqrt{2N} \sigma_{\nabla \mathbf{y}} \sigma_{\mathbf{w}}} \right), \quad (3.3)$$

where erf is the error function.

¹When using cross-entropy loss, softmax gradient is in the form of $e^{\mathbf{x}} / \sum e^{\mathbf{x}} - \mathbf{y}$, where $\mathbf{y}$ is the one-hot label vector. Ignoring the multiplier $\sum e^{\mathbf{x}}$ and bias $\mathbf{y}$, we can get the exponential relationship.

²We ignore the mean terms to simplify the expression.

Proof. $|\nabla \mathbf{x}|$ follows a *half-normal distribution* since $\nabla \mathbf{x}$ is normally distributed (Lemma 1). Equation 3.3 can be straightforwardly derived from the CDF of half-normal distribution and Equation 3.2.³ $\square$

We also propose the following theorem to estimate the underflow rate when type-casting softmax gradients.

Theorem 3.2. Given scale α and gradients $\nabla \mathbf{Y}$ from softmax, the underflow rate of $\nabla \mathbf{X} = \text{typecast}(\alpha \nabla \mathbf{Y})$ is

$$\gamma = \frac{1}{2} + \frac{1}{2} \text{erf} \left(\frac{\ln \lambda_{\min} - \mu - \ln \alpha}{\sqrt{2}\sigma} \right), \quad (3.4)$$

where μ and σ are measured on $\ln \nabla \mathbf{Y}$.

Proof. $\nabla \mathbf{y}$ has a log-normal distribution from Assumption 3, which implies that $\nabla \mathbf{y} > 0$ and $\ln \nabla \mathbf{y} \sim \mathcal{N}(\mu, \sigma^2)$. Scaled $\nabla \mathbf{y}$ in the log space, i.e., $\ln(\alpha \nabla \mathbf{y}) = \ln \nabla \mathbf{y} + \ln \alpha$, has distribution $\mathcal{N}(\mu + \ln \alpha, \sigma^2)$. Meanwhile, since $P(\ln(\alpha \nabla \mathbf{y}) \leq \ln \lambda_{\min}) = P(\alpha \nabla \mathbf{y} \leq \lambda_{\min})$, we can calculate the CDF of $\ln(\alpha \nabla \mathbf{y})$ evaluated at $\ln \lambda_{\min}$ to get $P(\alpha \nabla \mathbf{y} \leq \lambda_{\min})$. Given that $\gamma = P(\alpha \nabla \mathbf{y} \leq \lambda_{\min})$, we can figure out Equation 3.4. $\square$

These equations together form the statistical model for gradient scaling. In the next section, we will present how to incorporate this model with the backpropagation algorithm to implement gradient scaling.

3.2.2 The Gradient Scaling Algorithm

The statistical model from the previous section relates the underflow rate and gradient scale, such that we can estimate the required scale from a given underflow rate. That is the core idea of gradient scaling: this algorithm inserts gradient scale calculation for each type cast on the backpropagation path to reduce the rate of underflow. This section provides more details about this algorithm, including:

- How to scale down gradients before updating weights to keep its original magnitude by *propagating scales*.

³Note that if ReLU is applied to $\nabla \mathbf{Y}$ before MixedGEMM, $\sigma_{\nabla \mathbf{x}}$ becomes $\sqrt{N/2} \sigma_{\nabla \mathbf{y}} \sigma_w$ as argued by [109], and we should update Equation 3.3 accordingly. We will not explicitly mention the difference in the following discussion.

Algorithm 2: The gradient scaling algorithm.

Input: A DAG of backpropagation operators, of which $\text{op}(i)$ denotes the type of the i -th operator; a threshold γ' ; and a list of initial gradients in FP32.

```

1 for  $i \leftarrow$  ID of operators in their reversed topological order of the given computational graph do
2   if  $\text{op}(i)$  is MixedGEMM then
3     variables: scaled input gradient  $\langle \alpha, \nabla Y \rangle$ , weights  $W$ , and the accumulation length  $N$ 
4     statistics:  $\sigma \leftarrow \sqrt{N} \sigma_{\nabla y} \sigma_w$ 
5     scale: local gradient scale  $\beta \leftarrow \min \left( \frac{\lambda_{\min}}{\sqrt{2} \sigma \text{erf}^{-1}(\gamma)}, \frac{\lambda_{\max}}{N \max |\nabla X| \max |W|} \right)$ .
6     update:  $W \leftarrow W + \text{MixedGEMM}(X, \nabla Z) / \alpha$ 
7     propagate:  $\langle \beta \alpha, \text{MixedGEMM}(\beta \nabla Z, W) \rangle$ 
8   else if  $\text{op}(i)$  is typecast applied to softmax then
9     variable: scaled input gradient  $\langle \alpha, \nabla Y \rangle$ 
10    statistics: compute  $\mu$  and  $\sigma$  on  $\ln \nabla y$ 
11    scale:  $\beta \leftarrow \min \left( e^{[\ln \lambda_{\min} - \mu - \sigma \sqrt{2} \text{erf}^{-1}(2\gamma - 1)]}, \frac{\lambda_{\max}}{\max |\nabla Y|} \right)$ 
12    propagate:  $\langle \beta \alpha, \beta \nabla Y \rangle$ 
13   else if  $\text{op}(i)$  is concatenation then
14     Duplicate the gradient scale to each partition.
15   else if  $\text{op}(i)$  is branching or splitting then
16     input: a list of gradients  $\{\langle \alpha_i, \nabla X_i \rangle\}_{i=1}^M$ 
17     scale:  $\beta \leftarrow \max(\{\alpha_j\}_{j=1}^M)$ , such that
18        $\beta \max |\nabla X_i| / \alpha_i < \lambda_{\max}, \forall 1 \leq i \leq M$ 
19     propagate:  $\{\langle \beta, \beta \nabla X_i / \alpha_i \rangle\}_{i=1}^M$ 
20   else
21     Pass through the received scaled gradient.
22   end
23 end

```

- The *optimization problem* to be solved for each scale to reduce underflow rate without causing overflow.
- Besides MixedGEMM and typecast, how other operators, e.g., concatenation and branching, affect the scales being propagated.

Algorithm 2 covers our gradient scaling algorithm. We explain it piece by piece in the following sections.

Propagating Scaled Gradient

We keep track of the scale value during backpropagation in the form of a 2-tuple $\langle \alpha, \nabla X \rangle$, in which α is the product of all the scales applied before propagating to

$\nabla \mathbf{X}$. If we further scale $\langle \alpha, \nabla \mathbf{X} \rangle$ by a local scale β , suppose the operator is `typecast`, the new tuple we will get is $\langle \alpha\beta, \beta \nabla \mathbf{X} \rangle$. $\langle \alpha, \nabla \mathbf{X} \rangle$ also indicates that without gradient scaling, the magnitude of the gradient calculated at the same point will be α times smaller.

Weight updates, which are gradients propagated to weight variables, should have the same magnitude as what we get without gradient scaling. Otherwise, it may affect gradient descent optimizers, e.g., SGD and Adam [149], in the sense that these gradient scales change the *step size* differently for weight update. If the scales are being tracked and updated correctly, then the weight update, in the form of $\langle \alpha, \nabla \mathbf{W} \rangle$, can be correctly applied by using $\nabla \mathbf{W}/\alpha$. Line 6 in Algorithm 2 shows how weights are updated.

Optimizing Gradient Scale

Gradient scale should minimize the underflow rate after type casting without causing overflow. This requirement can be formulated as a constrained optimization problem, in which the objective function can be expanded through Equation 3.3 and Equation 3.4, and the constraint ensures the largest value will not exceed $\lambda_{\max}$. Note that even if the statistical model for γ is biased, the overflow constraint is strong enough to prevent numerical problems.

$$\min_{\alpha} \gamma = P(\alpha |\nabla \mathbf{X}| \leq \lambda_{\min}) \quad \text{s.t.} \quad \alpha \max |\nabla \mathbf{X}| < \lambda_{\max}$$

Because γ monotonically decreases with α , we end up with deciding α by $\lambda_{\max}/\max |\nabla \mathbf{X}|$. In cases that $\max |\nabla \mathbf{X}| \ll \lambda_{\max}$, making such a decision can result in very large scale value, which is not necessary and even harmful, since $\alpha > 1$ will enlarge the variance of gradients, i.e., $\text{Var}(\alpha \nabla \mathbf{X}) = \alpha^2 \text{Var}(\nabla \mathbf{X}) > \text{Var}(\nabla \mathbf{X})$, and may trigger gradient explosion [83, 110, 106].

A safer way to decide the scale is by reducing the underflow rate only below a pre-determined threshold γ' , that is, finding the lower bound on α given by $P(\alpha |\nabla \mathbf{X}| \leq \lambda_{\min}) \leq \gamma'$. By default, we set γ' to 10^{-3} , which indicates our intention to keep the underflow rate below 0.1%. In practice, we constrain the scale value to be a power of two, such that scaling up and down will be simply shifting the exponent bits. It is also worth to mention that α can be smaller than 1, if the type cast will cause overflow for any $\alpha \geq 1$.

Handling Specific Operators

The input to Algorithm 2 is a DAG representing backpropagation, in which each node denotes an operator of different types, e.g., type cast, GEMM, etc., and each edge specifies dependency. This algorithm loops over each node in the order of dependency, and its body is a big switch conditioned on the operator type. For each case, besides executing the operator itself, we need to make extra effort to scale gradients:

- **GEMM and type casting.** Simply solve the optimization problem based on Equation 3.3 and 3.4.
- **Concatenation.** Since the backward pass of a concatenation operator only splits a gradient into multiple parts, we can just distribute the scale of the original gradient.
- **Branching and splitting.** Both operators need to combine multiple gradients, possibly with different scales, together into one scaled gradient. Input gradients should be rescaled to one common value, which can be selected from their current scales in a descending order until one that will not cause overflow to any gradient. Details can be found in Line 17 of Algorithm 2.

Extra computation cost

Although Algorithm 2 introduces additional operations compared to the typical backpropagation algorithm, the computational cost incurred by this modification is minimal, if not negligible. Our experimental observations (Section 3.2.3) have not revealed any noticeable slowdown. This outcome can be attributed to the fact that the supplementary operations involve calculating the variance of gradients and performing element-wise addition and multiplication (with time-complexity $\mathcal{O}(N^2)$), which are typically much faster than the primary computation in backpropagation, namely matrix multiplication (with time-complexity $\mathcal{O}(N^3)$), in DNNs that use large matrices. Moreover, it is worth noting that variance calculation is pervasive in DNNs, such as in various normalization layers [129]. As a result, we have not explicitly accounted for this aspect in the subsequent evaluation sections.

3.2.3 Experiments

This section empirically evaluates the effectiveness of gradient scaling.

Experimental Setting

Software. We implemented gradient scaling in the Chainer [274] framework (version 6.3.0). Chainer provides a dynamic computational graph based programming paradigm similar to PyTorch. We also use ChainerCV [1] to apply our algorithm to concrete training examples. The CUDA library (version 10.1) we use supports FP16 and TensorCore [199].

Hardware. Most of our experiments run on NVIDIA RTX 2080Ti that has the latest Turing architecture [221] and enables TensorCore operations. We also have limited access to a multi-GPU cluster that has several NVIDIA Tesla V100 installed to run distributed training.

Baseline methods. We compare gradient scaling (GS) with the state-of-the-art MPT method with loss scaling, which has two variants: using fixed loss scale (Fixed-LS), or dynamically updating loss scale (Dyn-LS). We also compare with FP32 training results.

Hyperparameters. The only hyperparameter for gradient scaling is the underflow rate threshold γ' , which is set as 10^{-3} for all experiments. The update frequency of the gradient scale is per 100 iterations, such that the overhead of calculating gradient scales can be ignored. Task-specific hyperparameters will be listed in the following sections.

Image Recognition

We first examine ResNet-18 and ResNet-50 [110] on the ILSVRC 2012 dataset [248]. These are convolutional neural networks (CNNs) with ReLU, batch normalization [129], and residual connections, which sum up two outputs from distinct layers. Both networks can properly benchmark MPT since they are typically used for recognition tasks, and their architectures are representative of recently developed CNNs. We train them in the same hyperparameter setting as [88]⁴ with 16 GPUs.

Table 3.2 presents the results. Due to limited computational resources, we only train each scenario once, using 128 as the loss scale for Fixed-LS. Note that among the FP16 results, GS produces the best test accuracy for both networks. However, since

⁴The script we use is revised from <https://github.com/chainer/chainercv/blob/master/examples/classification>.

Model	Type	Method	Test Acc.
ResNet-18	FP32	N/A	70.76%
	FP16	N/A	71.24%
		Fixed-LS	71.39%
		Dyn-LS	71.39%
		GS	71.44%
ResNet-50	FP32	N/A	76.49%
	FP16	N/A	76.07%
		Fixed-LS	76.03%
		Dyn-LS	76.12%
		GS	76.22%

Table 3.2: ResNet-18/50 results on the ILSVRC 2012 dataset.

the residual structure partially mitigates underflow, the benefit from using gradient scaling is not significant.

Object Detection

We choose SSD as the benchmark for measuring the performance of MPT with different scaling methods on object detection tasks. SSD is trained on a joint of the PASCAL VOC 2007 and 2012 training sets, and validated on the 2007 test set. The training method⁵ is the same as the original paper, with the data augmentation in [75].

SSD uses VGG [258] as the backbone, which is initialized by pre-trained weights instead of random values. The two SSD variants we examine, SSD-300 and SSD-512, operate on images of 300×300 and 512×512 resolution, respectively. Multiple outputs from the backbone are branched out to feed into predictors that estimate location and class of possible objects at various scales. The use of pre-trained weights combined with the branching structure of the architecture can make loss scaling hard to use, since they enforce the gradients at different places to have distinct distributions. However, gradient scaling is expected to be better suited at handling such distributions.

Table 3.3 shows the result. We first train SSD-300 and 512 with batch size 8, and the reported accuracy is averaged from multiple runs. Specifically, we use a range of loss scales and choose the best result for Fixed-LS. We see that GS performs similarly as the best result from Fixed-LS, and achieves much higher performance

⁵The original training script can be found at <https://github.com/chainer/chainercv/blob/master/examples/ssd/>.

BS	Model	Type	Method	Test mAP
8	SSD-300	FP16	N/A	Diverged
			Fixed-LS	76.23% (best in 7 runs)
			Dyn-LS	69.26%
			GS	76.35%
	SSD-512	FP16	N/A	Diverged
			Fixed-LS	79.29% (best in 8 runs)
			Dyn-LS	75.33%
			GS	79.24%
32	SSD-512	FP32	N/A	80.50%
		FP16	N/A	Diverged
			Fixed-LS	80.01%
			Dyn-LS	80.17%
			GS	80.31%

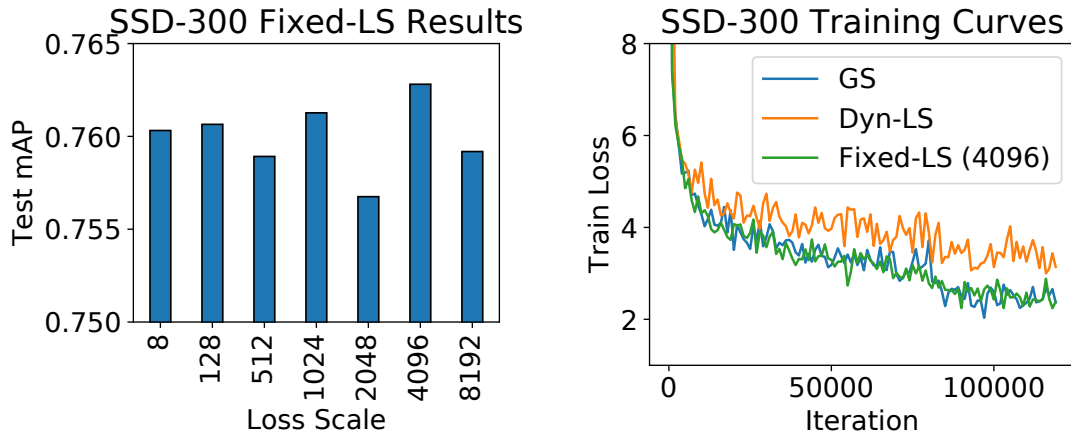
Table 3.3: SSD-300/512 training results. **BS** denotes batch size.

than Dyn-LS. We also trained SSD-512 for the same number of iterations but with batch size 32, and in this scenario, GS also gives the highest test mAP in MPT results, only worse than the FP32 baseline. Note that the results from Fixed-LS are the best among multiple runs, which implies the additional training effort that Fixed-LS requires. The following includes some details for further discussion.

Fixed loss scaling. Fixed-LS requires searching for a loss scale value. Figure 3.1a shows the non-trivial differences in accuracy among loss scale choices. Taking into account the cost for searching such a hyperparameter, Fixed-LS can be multiple times slower than GS to achieve the same performance.

Dynamic loss scaling. Dyn-LS performs poorly especially when the batch size is 8 (Figure 3.1b). The major reason is that Dyn-LS implicitly increases the loss scale value and only decreases it when overflow occurs. The distribution of gradients in SSD is very different, and a large loss scale can easily cause overflow. With the same number of training iterations, these more frequent overflows correspond to wasted computation, such that the convergence rate of Dyn-LS slows down. Since larger batch sizes can reduce the chance of exploding gradient, Dyn-LS performs better when the batch size is 32.

Scale distribution. Figure 3.2 shows the distribution of gradient scales computed for each convolutional layer. Those layers with peak scales are located in network



(a) Fixed-LS performance with various loss scales.

(b) Dyn-LS performs the worst.

Figure 3.1: SSD MPT details using different scaling methods [341].

Model	Type	Method	mIoU	CLS	GLB
SegNet	FP32	N/A	51.1%	68.3%	83.9%
		N/A		Diverged	
	FP16	Dyn-LS	48.3%	64.1%	82.4%
		GS	49.6%	64.6%	83.4%

Table 3.4: SegNet MPT test results on CamVid.

branches that predict object positions and classes at various resolutions. Other layers use smaller scales to further adjust gradients.

Segmentation

SegNet [12], a fully convolutional network with an encoder-decoder architecture, is our benchmark for image segmentation. Its encoder part is a VGG-like CNN with batch normalization, and its decoder takes the output from the encoder and processes it by multiple upsampling and convolution layers. The loss is calculated by pixel-wise softmax. When training it with mixed precision, we can cast the encoder-decoder part into FP16 and cast the output back to FP32 for the pixel-wise softmax.

SegNet is evaluated on the CamVid road scenes dataset, which has 367 training and 233 test images at the resolution of 360×480 . When training it, we use SGD with batch size 12, learning rate 0.1, and momentum 0.9 for 16K iterations. We compare gradient scaling with other loss scaling methods for MPT, as well as the

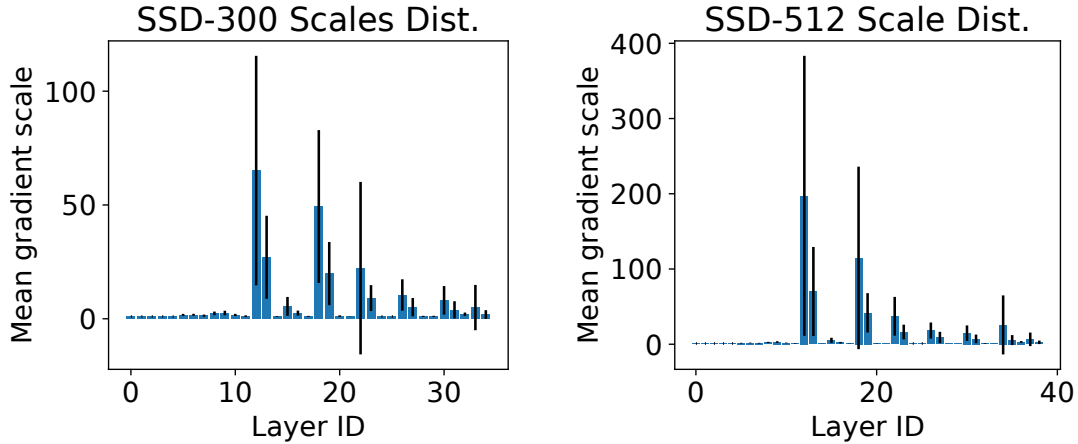


Figure 3.2: Mean gradient scales for all layers of SSD-300/512 across all training iterations. Layer are labelled from input to output in topological order. Black lines are error bars [341].

FP32 training baseline⁶.

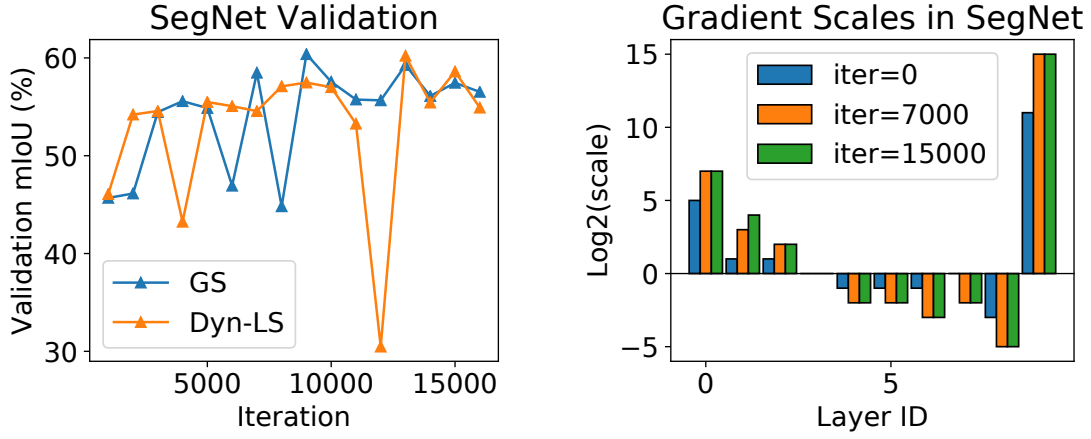
As shown in Table 3.4, GS performs the best among all MPT results and a bit worse than the FP32 baseline. No exhaustive search for Fixed-LS has been performed, and we only use Dyn-LS as a counterpart. Validation results in Figure 3.3a shows that training by Dyn-LS can be less stable than GS.

Explaining GS benefits. Figure 3.3b compares the computed gradient scales in different layers and training iterations. The last layer (ID=8) is the type cast from FP16 to FP32 before pixel-wise softmax, layers 4-7 are the convolution layers in the decoder, and layers 0-3 are in the encoder. The type cast always requires large scales, otherwise the underflow rate will be around 70%. GS can correctly calculate this scale for that type cast. It can also scale down gradients to prevent overflow when ainly because the extremely small pixel-wise softmax gradients, and fully mitigating underflow requires $\alpha > \lambda_{\max}$.

3.2.4 Discussions

This section presents Aegis, a novel method to tackle gradient underflow in MPT. The key idea is to adjust the scale of each gradient tensor during training, which is more effective than the existing loss scaling method. We present the assumption

⁶The original SegNet training script: <https://github.com/chainer/chainercv/tree/master/examples/segnet>. We also collect the FP32 baseline result through this script, which is better than the reported values from the original paper.



(a) mIoU on the validation set, collected every 1k iterations. (b) Gradient scales collected for all layers at different iterations.

Figure 3.3: Details in the SegNet training procedure [341].

on gradient distribution, othe accumulation length is long, specifically, at the intersection of encoder and decoder, which is not available with a single loss scale. The accuracy gap between GS and FP32 is mur gradient scaling algorithm, and details on handling specific operators. Our empirical evaluation on a variety of vision-based model architectures shows improved accuracy compared to existing MPT methods.

The superior performance achieved by Aegis compared to existing methodologies stems from our observation that the distribution of gradients varies across layers and training steps, as evidenced in Figs. 3.2 and 3.3b. To overcome numerical challenges in MPT, our method, Aegis, employs the gradient scaling algorithm (Algorithm 2) to accurately compute the appropriate gradient scale for each layer during training. This novel approach distinguishes Aegis from existing methods and contributes to its ability to effectively mitigate numerical issues and improve overall performance.

While Aegis presents several benefits, it is essential to acknowledge certain limitations that may affect its efficacy:

- The effectiveness of our estimation of the underflow rate equation 3.2 and the subsequent calculation of the gradient scale heavily relies on the assumption that the distribution of gradient values follows a Gaussian distribution [250]. If this assumption does not hold, the estimated underflow rate and calculated gradient scale may not be as accurate or effective as intended.
- Although it is intuitive to associate a high percentage of underflow with unhealthy training progress (indicating numerous zero gradients), accurately determining the exact impact of underflow on the final model accuracy in MPT

is not straightforward.

- In the case of smaller neural networks, the additional computational cost associated with calculating gradient scales can have a more significant impact on the overall computational budget compared to larger networks.

3.3 SPGC: Structured Pruning by Group Convolution

As mentioned in earlier sections, the objective of SPGC is to *prune* a pre-trained CNN model into a GConv-based model. Details about GConv are reviewed in Section 2.1.2. The following sections formulate GConv pruning as an optimisation problem that searches for channel permutations, demonstrate the layer-wise heuristic pruning algorithm that efficiently solves this problem, and show how to explore model sparsity under inference budget constraints.

3.3.1 Group Convolution Pruning

This section first finds the relationship between GConv and block matrix multiplication, based on which we then formulate the pruning problem.

```

1 class GroupConv2d(Module):
2     def forward(self, x):
3         dim = 1 # index of channel
4         gs = index_select(x, dim, self.ind_in).split(self.G, dim=dim)
5         os = [self.conv[i](g)
6               for i, g in enumerate(gs)]
7         return index_select(
8             cat(os), dim, self.ind_out)

```

Listing 6: This GConv is implemented as a PyTorch [225] `Module`, which performs computation in the `forward` method. We permute (`index_select`) the input $\mathbf{x}$ by $\mathbf{I}$ (`ind_in`) along the channel axis and the output by $\mathcal{I}_{\text{out}}$ (`ind_out`). Convolution groups (`conv`) takes input groups (`split`) and concatenate (`cat`) their results.

Group convolution. Eq. (3.5) formulates GConv. $\mathbf{X}$ and $\mathbf{Y}$ are both 3D *tensors* indicating input and output feature maps; $\mathbf{X}_c$ and $\mathbf{Y}_f$ are 2D images belong to input channel c and output channel f respectively; and $\mathbf{W} \in \mathbb{R}^{C_{\text{out}} \times C_{\text{in}} \times K_h \times K_w}$ is a 4D tensor denoting weights. $\mathbf{I}$ and $\mathcal{I}_{\text{out}}$ are two sets of channel indices that specify permutation, e.g., $\mathbf{I}(c)$ is permuted to c . When performing GConv, we permute $\mathbf{X}$ into $\tilde{\mathbf{X}}$ by input channel indices $\mathbf{I}$, partition $\tilde{\mathbf{X}}$ into G groups $\{\tilde{\mathbf{X}}^g \in \mathbb{R}^{C_{\text{in}}/G \times H \times W}\}_{g=1}^G$ along channel, run a convolution layer $(*)$ between $\tilde{\mathbf{X}}^g$ and weight group $\mathbf{W}^g \in \mathbb{R}^{C_{\text{out}}/G \times C_{\text{in}}/G \times K_h \times K_w}$ for each group g , concatenate their output $\{\tilde{\mathbf{Y}}^g\}_{g=1}^G$ into a

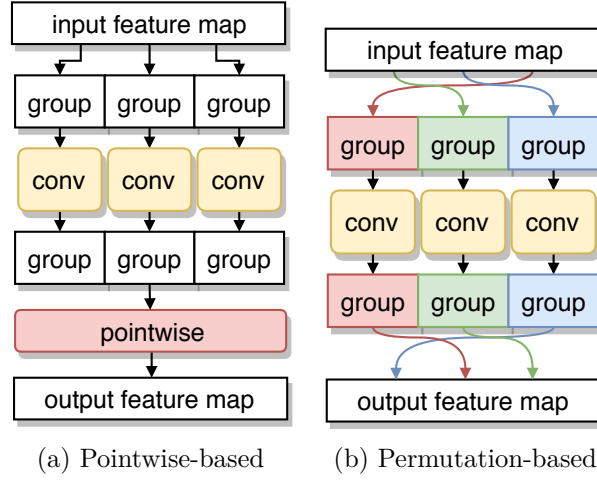


Figure 3.4: Two variants of GConv that are different in information exchange mechanism [340].

tensor $\tilde{\mathbf{Y}}$ by channel, and permute $\tilde{\mathbf{Y}}$ by output channel indices $\mathcal{J}_{\text{out}}$ to produce the result $\mathbf{Y}$.

$$\begin{aligned}
 \tilde{\mathbf{X}}_c &= \mathbf{X}_{\mathcal{I}(c)} \quad \tilde{\mathbf{Y}}^g = \mathbf{W}^g * \tilde{\mathbf{X}}^g \quad \mathbf{Y}_f = \tilde{\mathbf{Y}}_{\mathcal{J}_{\text{out}}(f)} \\
 \forall 1 \leq f \leq C_{\text{out}} \quad 1 \leq c \leq C_{\text{in}} \quad 1 \leq g \leq G
 \end{aligned} \tag{3.5}$$

Alternatively, we can treat GConv as a sparse convolution. To illustrate this idea, we reshape $\mathbf{W}$ into a 6D tensor $\mathcal{W} \in \mathbb{R}^{G \times G \times C_{\text{out}}/G \times C_{\text{in}}/G \times K_h \times K_w}$ by partitioning the input and output channels of $\mathbf{W}$ into G groups. Considering each $K_h \times K_w$ kernel as a single element, this representation can be viewed as a generalised *block matrix* with $G \times G$ number of $C_{\text{out}}/G \times C_{\text{in}}/G$ sized blocks. Similarly, we can partition $\mathbf{X}$ and $\mathbf{Y}$ into 4D tensors $\mathcal{X} \in \mathbb{R}^{G \times C_{\text{in}}/G \times H \times W}$ and $\mathcal{Y} \in \mathbb{R}^{G \times C_{\text{out}}/G \times H \times W}$ and view them as generalised *block vectors*. Note that 2D images are considered as elements in these block matrices.

$$\mathcal{Y}_{g_f} = \sum_{g_c=1}^G \mathcal{W}_{g_f g_c} * \mathcal{X}_{g_c} \quad \forall 1 \leq g_f \leq G \tag{3.6}$$

$$\begin{aligned}
 \mathcal{Y} &= \text{diag}(\{\mathcal{W}_{gg}\}_{g=1}^G) * \mathcal{X} \\
 &= \begin{bmatrix} \mathcal{W}_{11} & 0 & \cdots & 0 \\ 0 & \mathcal{W}_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \mathcal{W}_{GG} \end{bmatrix} \begin{bmatrix} \mathcal{X}_1 \\ \mathcal{X}_2 \\ \vdots \\ \mathcal{X}_G \end{bmatrix} = \begin{bmatrix} \mathcal{X}_1 * \mathcal{W}_{11} \\ \mathcal{X}_2 * \mathcal{W}_{22} \\ \vdots \\ \mathcal{X}_G * \mathcal{W}_{GG} \end{bmatrix}
 \end{aligned} \tag{3.7}$$

We can define convolution among group partitioned tensors $\mathcal{Y} = \mathcal{W} * \mathcal{X}$ as a generalised *block matrix-vector multiplication*. Here, the multiplication between entries in $\mathcal{W}$ and $\mathcal{X}$ is interpreted as convolution. For example, Eq. (3.6) illustrates the dot-product routine that computes the convolution between two blocks in $\mathcal{W}$ and $\mathcal{X}$. Interestingly, if $\mathcal{W}$ is a generalised *block-diagonal* matrix, i.e, only $\mathcal{W}_{gg}$ are non-zero, then this convolution becomes a group convolution Eq. (3.7). Ignoring permutations in Eq. (3.5) and considering $\mathcal{W}_{gg}$ as $\mathbf{W}^g$, $\mathcal{X}_g$ as $\mathbf{X}^g$, and $\mathcal{Y}_g$ as $\mathbf{Y}^g$, it is obvious that Eq. (3.5) is equivalent to Eq. (3.7). This is the basis of the following analysis.

$$\begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \\ w_{31} & w_{32} & w_{33} \end{bmatrix} \xrightarrow[\mathbf{I}=\{1,3,2\}]{\mathcal{J}_{\text{out}}=\{2,1,3\}} \begin{bmatrix} w_{21} & w_{23} & w_{22} \\ w_{11} & w_{13} & w_{12} \\ w_{31} & w_{33} & w_{32} \end{bmatrix} \quad (3.8)$$

Pruning and channel permutation. Pruning means removing weights with low contribution to model accuracy from a trained model based on a specific criterion. Eq. (3.7) shows that removing weights to form a block-diagonal matrix is equivalent to pruning into GConv. A straightforward approach to prune is just removing kernels outside the diagonal. It can hardly perform well since we cannot guarantee that kernels around the diagonal are important to the model accuracy. Since we allow channel permutation on both input and output, we can formulate pruning as an optimisation problem that targets at *finding the channel permutation that can move most of the important kernels to diagonal blocks*. To be specific on permutation, Eq. (3.8) shows an example result after applying a pair of permutation indices $\mathcal{J}_{\text{out}}$ and $\mathbf{I}$ on weights. w_{ij} denotes a single kernel, and rows and columns represent output and input channel axes respectively.

Optimisation problem. Eq. (3.9) formulates the optimisation problem of finding optimal permutations $\mathbf{I}^*$ and $\mathcal{J}_{\text{out}}^*$. We need to find the pair of permutations such that, after applying it on the original weights $\mathbf{W}$, the importance reduction caused after removing weights outside the diagonal will be minimal. $\{\tilde{\mathcal{W}}_{gg}\}_{g=1}^G$ denotes all the diagonal-blocks of permuted weights. $\mathcal{C}$ is the criterion that measures the importance. We choose a magnitude-base criterion that sums the L2 norm of all kernels, based on the assumption that kernels with greater magnitude contribute more to model accuracy [174].

$$\begin{aligned} \arg \min_{\mathbf{I}^*, \mathcal{J}_{\text{out}}^*} \quad & \mathcal{C}(\mathbf{W}) - \mathcal{C}(\text{diag}(\{\tilde{\mathcal{W}}_{gg}\}_{g=1}^G)) \\ \text{s.t.} \quad & \tilde{\mathcal{W}}_{gfgc} \equiv \tilde{\mathbf{W}}^{gfgc} \quad \tilde{\mathbf{W}}_{fc} = \mathbf{W}_{\text{out}(f)\mathbf{I}^*(c)} \end{aligned} \quad (3.9)$$

We notice that solving this problem requires similar efforts as solving the Bottleneck Travelling Salesman Problem (BTSP) [80], which is known to be NP-complete. Since the number of channels can be hundreds or even thousands, directly solving this problem is computationally intractable. Next section presents a heuristic algorithm that produces satisfiable solutions within a limited time.

3.3.2 Prune a Layer

This section focuses on the layer-wise pruning problem: for a given sparsity, indicated by the number of groups G , we aim to find the pair of column-wise and row-wise permutations that minimise the pruning objective defined in Eq. (3.9). Column and row refer to input and output channel axes of weights respectively.

$$\max_{j_{\text{out}}, \mathbf{l}} \sum_{g=1}^G \sum_{f=1}^{C_{\text{out}}/G} \sum_{c=1}^{C_{\text{in}}/G} \|\tilde{\mathcal{W}}_{g g f c}\| \quad (3.10)$$

We start from replacing $\mathcal{C}$ with L2 norm to convert the original problem to an equivalent maximisation problem that *maximises the sum of L2 norm of weight kernels in diagonal blocks* of $\tilde{\mathcal{W}}$, as shown in Eq. (3.10).

$$\begin{aligned} \sigma_c(c, g) &= \sum_{f=\frac{g-1}{G}C_{\text{out}}+1}^{\frac{g}{G}C_{\text{out}}} \|\mathbf{w}_{fc}\| \\ \sigma_f(f, g) &= \sum_{c=\frac{g-1}{G}C_{\text{in}}+1}^{\frac{g}{G}C_{\text{in}}} \|\mathbf{w}_{fc}\| \end{aligned} \quad (3.11)$$

Sub-problem. Suppose we only maximise the sum-of-norm for the G -th block diagonal component $\mathcal{W}_{G,G}$, and we are only allowed to permute input channels, an intuitive solution for this sub-problem is sorting input channels by $\sigma_c(c, G)$ Eq. (3.11). Under the given restrictions, sorting by σ_c in an increasing order practically moves the most important weights to $\mathcal{W}_{G,G}$. Similarly, if only output channels are permitted to permute, we can sort them by $\sigma_f(f, G)$ as well to maximise the importance of $\mathcal{W}_{G,G}$.

Heuristic Algorithm. This intuition leads us to a heuristic algorithm that solves Eq. (3.10). Instead of considering this problem as a whole, we dissect it into sub-problems similar to the example above, which can be solved block by block through sorting with regards to σ_c and σ_f . Specifically:

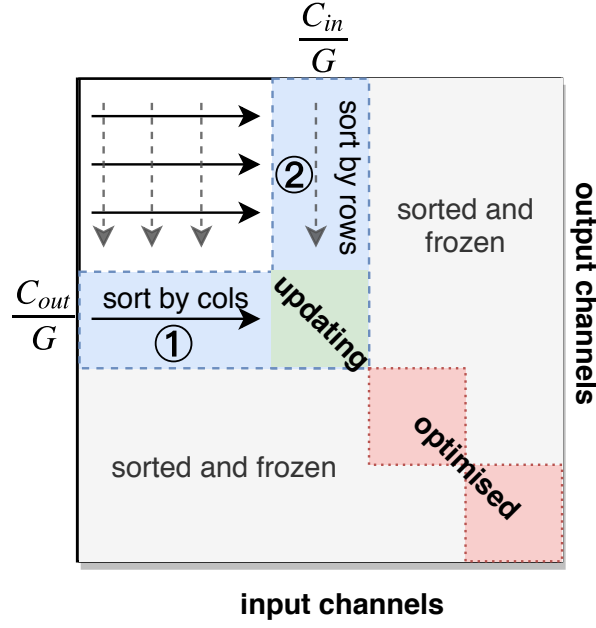


Figure 3.5: This figure shows $\mathcal{W}$ that is being sorted. It is running at the 3rd block (green). Kernels covered by previous blocks (red) are now frozen for further sorting (gray). When updating the 3rd block, input channels (columns) are sorted by $\sigma_c(c, 3)$ in range ① and output channels (rows) are sorted by $\sigma_f(f, 3)$ in range ② [340].

- (i) Our algorithm runs in G iterations and the g -th iteration works on optimising block $\mathcal{W}_{G-g+1, G-g+1}$ only.
- (ii) Each iteration sorts input and output channels by $\sigma_c(c, g)$ and $\sigma_f(f, g)$ respectively for N_S rounds.
- (iii) All channels related to previously resolved blocks remain *frozen*, i.e., the upper bounds for f and c that can be sorted are $(G - g + 1)C_{out}/G$ and $(G - g + 1)C_{in}/G$.

Fig. 3.5 illustrates an intermediate step of this algorithm. N_S is a hyperparameter that denotes the number of sorting rounds for each block. A sorting round means sorting input channels and then output channels. N_S is necessary since we permit sorting both input and output channels: after finishing a sorting round, the increasing order of input channels regarding σ_c may be violated, and running a new round may fix it. For example, as shown in Fig. 3.5, kernels in the white region are not covered in σ_c when sorting input channels at first, and after output channels are sorted, some may enter the blue region and affect the evaluation of σ_c . What we present here is a polynomial time algorithm and its complexity is $\mathcal{O}(GN_S \times (\frac{C_{in}}{G} \log(\frac{C_{in}}{G}) + \frac{C_{out}}{G} \log(\frac{C_{out}}{G})))$.

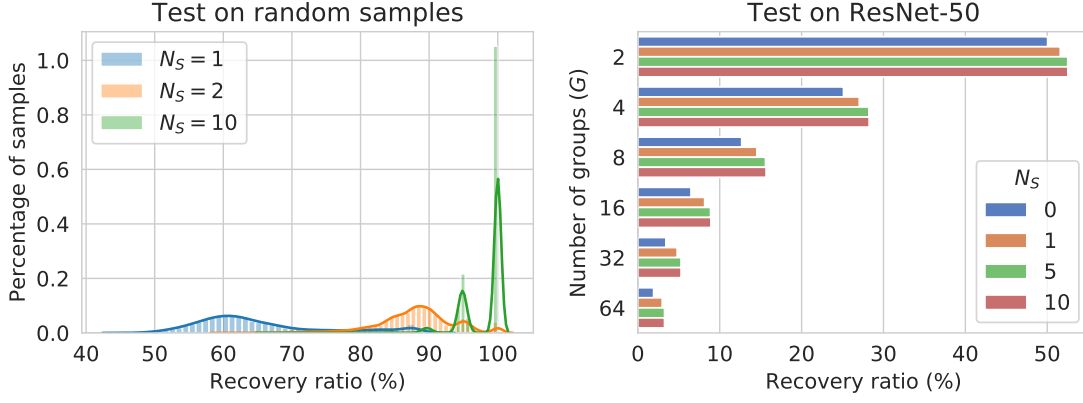


Figure 3.6: Evaluation results on the heuristic algorithm. Recovery ratio is the percentage of magnitude, measured by L2 norm, retained in the diagonal blocks after pruning [340].

$$\text{recovery ratio} = \frac{\sum_{g,f,c} \|\tilde{\mathcal{W}}_{ggfc}^*\|}{\sum_{g,f,c} \|\mathcal{W}_{ggfc}\|} \quad (3.12)$$

Empirical Evaluation. We empirically justify this algorithm by running it on randomly generate sample weights and real-world pre-trained models. We first create a block-diagonal matrix, then permute it by random indices, and try to recover the original permutation as much as possible. We measure the quality by the *recovery ratio* defined by Eq. (3.12), in which $\tilde{\mathcal{W}}^*$ denotes weights permuted by optimal permutations decided by our algorithm. We notice that the recovery ratio becomes higher for more samples when N_S increases, and when $N_S = 10$, most samples can achieve 100% recovery ratio. It shows that by using our heuristic algorithm with $N_S = 10$ we can move most important weights to diagonal blocks, which implicitly guarantees our GConv pruning performance since we remove weights out of diagonal blocks.

We also measure the recovery ratio on weights from pre-trained ResNet-50⁷: as shown on the right of Fig. 3.6, compared with the baseline that does no sorting ($N_S = 0$), our method can recover about 3% more for different numbers of groups. Besides the recovery ratio, we compare the final test accuracy between using and not using our heuristic method in Table 3.9. In the future, we will provide formal proof regarding the performance of this heuristic algorithm.

⁷Pre-trained models downloaded from <https://pytorch.org/docs/stable/torchvision/models.html>

3.3.3 Optimise Group Configuration

To prune a whole CNN model into one that uses GConv, we need to determine the *group configuration*, which is a combination of all layers' numbers of groups. Group configuration can affect both the model accuracy and the inference budget, and therefore, finding an optimal group configuration is an optimisation problem that maximises accuracy under budget constraints. Prior papers use some ad-hoc rules to decide group configuration, e.g., using a uniform group number for all layers [122] or scaling the number of groups by the number of channels [227, 339]. When using these methods, the only way to find the configuration that gives the highest model accuracy is by trial-and-error, i.e., manually picking a group configuration and fine-tuning from it, which is cumbersome. To address this problem, we propose a group configuration optimisation algorithm, which finds near-optimal configuration regarding model accuracy by utilising pre-trained weights through our GConv pruning algorithm.

$$\begin{aligned} \arg \min_{\mathcal{G}} \quad & \sum_{l=1}^L \text{cost}(\mathcal{W}^{(l)}, \mathcal{G}_l) \\ \text{s.t.} \quad & \sum_{l=1}^L \frac{N_{\text{par}}^{(l)}}{\mathcal{G}_l} \leq N_{\text{par}}^{\max} \quad \sum_{l=1}^L \frac{N_{\text{ops}}^{(l)}}{\mathcal{G}_l} \leq N_{\text{ops}}^{\max} \end{aligned} \quad (3.13)$$

This method is formulated as Eq. (3.13). The variable $\mathcal{G}$ is a vector that specifies the number of groups of each layer l . The function **cost** denotes the minimal pruning cost returned from solving Eq. (3.9). We assume that the lower the pruning cost, the higher the accuracy of a pruned model. The objective is to minimise the sum of estimated pruning cost of all layers subject to constraints on the maximum numbers of parameters $N_{\text{par}}^{\max}$ and operations $N_{\text{ops}}^{\max}$.

Our approach is a local search algorithm [249, Chap. 4]. We notice that by adding the number of groups of a layer, the total cost increases and the numbers of parameters and operations decrease. Based on this observation, we devise this local search algorithm by starting with a $\mathcal{G}$ that sets G to 1 for all layers, and in each of the following iterations, selecting one layer that minimally increases the cost when its number of groups is changed to the nearest larger candidate. The whole procedure terminates when the resource constraints are satisfied. To estimate pruning cost more precisely, we can optionally prune and fine-tune the model by the current $\mathcal{G}$ at the end of each iteration. This algorithm is summarised in Algorithm 3.

As a final step, based on the optimised $\mathcal{G}$, we prune and fine-tune the given model

Algorithm 3: The local search algorithm to solve Eq. (3.13)

```

1  $\mathcal{G} \leftarrow$  maximal numbers of groups of all layers;
2 while budget of  $\mathcal{G}$  is under constraints do
3    $i \leftarrow$  layer that reduces cost the most;
4    $\mathcal{G}(i) \leftarrow$  next larger number of groups for  $i$ ;
5   (optional) prune model by  $\mathcal{G}$  and then fine tune;
6 end
7 return  $\mathcal{G}$ ;

```

again to improve its model accuracy as much as possible. Empirically we find this algorithm works well. As shown in the next section (Fig. 3.7), we can explore configurations within given budget and the explored models perform competitively compared to ad-hoc configurations.

3.3.4 Experiments

This section presents various experiments to empirically evaluate our method.

Experiment Setup

Datasets and models. Our method is evaluated on CIFAR-10/100 [155] and ImageNet [248]. [326] is used to build and train CIFAR-10/100 baseline models. Regarding ImageNet, we evaluate on ILSVRC2012 and augment data by random cropping and then random horizontal flipping and the validation accuracy is evaluated by center-cropping.

We choose various CNN models for evaluation: ResNet-110 [110], ResNet-164 [111], and DenseNet-86 [121] for CIFAR-10/100; ResNet-18/34/50/101 [110] for ImageNet. Our models are all implemented by PyTorch [225] v1.1.

Pruning and fine-tuning. When deciding group configuration $\mathcal{G}$, we may use a uniform G value for all layers, a configuration borrowed from a prior work, or one generated by solving Eq. (3.13). Once $\mathcal{G}$ is settled, we run the heuristic layer-wise pruning algorithm (Section 3.3.2) to get channel permutation, which indicates which weights should be pruned and how to permute input and output channels. The hyperparameter N_S is normally set to 10 based on Fig. 3.6.

After pruning the model, in the fine-tuning phase, we normally choose a relatively small learning rate to train the pruned model for a few more epochs. The fine-

Table 3.5: Evaluation on CIFAR-10 (C10) and CIFAR-100 (C100). Number of parameters (# Param.) and operations (# FLOPS) are measured on CIFAR-100. Note that we consider one FMA as two operations. Pruned ResNet-165 models are labelled by $X\%$ pruned, which indicates the ratio of pruned channels as in [180]. When pruning by our method, since we cannot prune away a whole channel, we only take the number of parameters of the corresponding models from [185] as the optimisation constraint.

Model	Test Error (%)		# Param.	Pruned	# FLOPS	Pruned
	C10	C100				
ResNet-164 (Baseline)	4.81	22.83	1.73M	—	504M	—
ResNet-164 (40% pruned)	4.91	22.71	1.46M	15.5%	462M	8.4%
ResNet-164 (60% pruned)	5.08	23.66	1.21M	29.7%	430M	14.7%
ResNet-164 (40% pruned) [185]	5.08	22.87	1.46M	15.5%	333M	33.3%
ResNet-164 (60% pruned) [185]	5.27	23.91	1.21M	29.7%	247M	50.6%
DenseNet-86 (Baseline) [122]	4.44	20.57	2.03M	—	506.1M	—
DenseNet-86 ($G = 4$)	5.94	25.96	0.59M	70.94%	132.7M	73.77%
DenseNet-86 (opt. for 50% budget)	4.98	22.41	1.00M	50.74%	256.5M	49.33%
CondenseNet-86 ($G = 4$) [122]	5.00	23.64	0.59M	70.94%	132.7M	73.77%

tuning period could be around one third of the original training from scratch time. We will tune training hyperparameters further in the future to see at least how much workload is required to recover the accuracy of a pruned model.

Results

ResNet-164 on CIFAR. We list our results on CIFAR-10/100 in Table 3.5. We first compare ResNet-164 with network slimming [185], a state-of-the-art channel-wise structured pruning method based on regularisation. Their models are pruned with respect to the percentage of removed channels, which are quite different from our GConv-based results. To compare our method with them at a similar pruning level, we set the number of parameters of their models as constraints for our group configuration optimisation, and use optimised configurations to prune ResNet-164. As shown in Table 3.5, our models have much smaller test error than their counterparts with the same number of parameters. Considering model topology, our resulting models are also easier to process: convolution layers with arbitrary amount of channels produced by [185] may not be friendly to low-level accelerator, while ours are basically GConv, which runs efficiently on modern hardware.

One drawback of our method on this ResNet-164 case is the relatively higher FLOP number, which is caused by the fact that layers closer to the output normally have lower pruning cost but less contribution to FLOP. Our method tends to give these layers higher pruning priority. This issue will be mitigated by introducing FLOP into the pruning cost measurement in our future work.

Comparison with CondenseNet. CondenseNet [122] introduces a multi-staged, group-lasso regularisation based GConv pruning procedure. With a given group configuration, this paper provides the state-of-the-art GConv pruning results on variants of DenseNet [123]. For the comparison purpose, we select DenseNet-86, a variant of DenseNet and is pruned to CondenseNet-86 in [122], as a baseline model.

Since they put more efforts in training and regularisation, it is hard for our post-training pruning method to surpass their level of accuracy. Table 3.5 shows that our $G = 4$ result is around 1-2% worse on CIFAR-10/100 validation accuracy than the CondenseNet counterpart. This accuracy loss can be explained by the additional regularisation effect introduced while training CondenseNet.

Even though using our method is still beneficial in some scenarios. CondenseNet can only be trained by a fixed, manually picked group configuration, while we can explore the group configuration from pre-trained DenseNet models. We can find more accurate models under different inference budget, e.g., in Table 3.5 we show a better option under a looser budget of 50%. Performing similar exploration in CondenseNet will take much longer time due to the need to train from scratch.

Group Configuration Exploration. One of our major benefits is that we can explore group configuration under given inference budget constraints, as mentioned earlier. To evaluate the quality of the exploration, we gradually reduce the upper bound of number of parameters and run the model pruner. Results for ResNet-164 on CIFAR-100 are in Fig. 3.7. The granularity of exploration here is 0.1M and we use the same training schedule for each sparsity configuration. Results demonstrate that explored models can perform on par with manually selected configurations.

ImageNet. We evaluate various pre-trained ResNet models on ImageNet [110]. The fine-tuning phase has 30 epochs (1/3 of what [110] uses) with learning rate starting at $1e^{-3}$ and being multiplied by 0.1 every 10 epochs. Limited by hardware resources, we only select uniform numbers of groups.

Results are listed in Table 3.6. Since models for ImageNet are rarely sparse, removing many parameters in one-shot normally degrades the accuracy significantly. For ResNet-18, we can reduce 83% parameters and 95% FLOPS with an increase of 14.5% in test error. It is not promising regarding the high test error, but this performance is on par with [339], even if they have higher budget in both training and inference phases: their ResNet-18 example is trained from scratch and the Hadamard transform adds overhead.

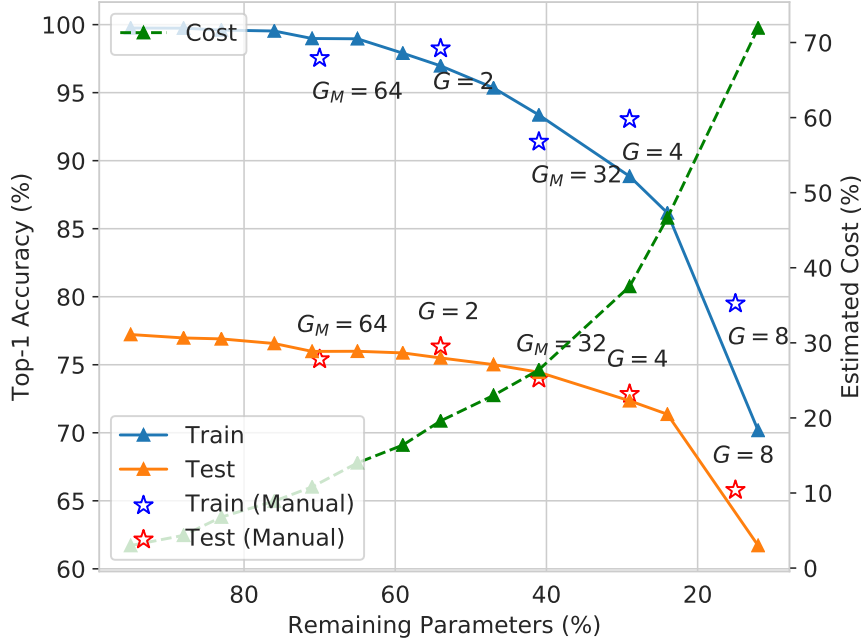


Figure 3.7: Change in ResNet-164 on CIFAR-100 accuracy while reducing number of parameters. Each sample point is collected from the model found by our model pruning algorithm under the given number of parameters constraint. Manually configured models either use uniform group size G or set a maximum channel number G_M . The latter approach assigns each layer a group number that ensures $\max(C_{in}/G, C_{out}/G) \leq G_M$. We also list the estimated pruning cost for each explored sparsity configuration. Accuracy numbers listed here are collected from *fine-tuning* [340].

Table 3.6: ImageNet evaluation results. The first sample in each section is the baseline.

Model	# Params.	# FLOPS	Top-1 Error
ResNet-18	11.69M	3.64G	30.24%
ResNet-18 $G=8$	1.91M	0.20G	44.71%
ResNet-18 $G=8$ [339]	1.91M	0.33G	44.60%
ResNet-34	21.80M	7.34G	26.70%
ResNet-34/A	17.36M	3.48G	28.42%
ResNet-34/B	9.05M	1.89G	32.44%
ResNet-34/A[227]	18.2M	3.98G	27.05%
ResNet-34/B[227]	11.1M	2.62G	27.75%
ResNet-50	25.56M	8.21G	23.85%
ResNet-50 $G=2$	13.82M	3.77G	25.90%
ResNet-101	44.55M	15.7G	22.63%
ResNet-101 $G=2$	23.34M	7.50G	24.22%

We also compare ResNet-34 to [227], which uses low-rank approximation to perform GConv pruning. This method requires adding pointwise 1×1 convolution after each

Table 3.7: CUB-200 evaluation results.

Model	# Params.	# FLOPS	Top-1 Error
ResNet-50	23.92M	8.21G	21.76%
ResNet-50 (G=2)	12.18M	3.77G	23.01%
ResNet-50 (G=4)	6.31M	1.78G	26.49%
MobileNet-224	3.41M	1.15G	27.83%
MobileNet-224 (G=2)	1.97M	0.82G	32.50%
MobileNet-224 (G=4)	1.25M	0.65G	36.81%
ThiNet-GAP [190]	7.91M	2.01G	30.57%
ThiNet-Tiny	1.12M	1.44G	34.55%

GConv. Since they use more parameters that potentially increase learning capacity, their accuracy can be higher. For the two configurations from [227], ResNet-34/A and /B, models produced by us are smaller but less accurate. However, as mentioned before, their GConv is always appended by a pointwise convolution, which means with the same number of groups they use more resources, and they cannot deal with 1×1 group convolution. Specifically, for ResNet-50/101 that heavily uses 1×1 kernels, our method can still reduce around 50% budget without increasing much test error.

CUB-200. We start from ImageNet pre-trained ResNet-50 and MobileNet-224, fine-tune them on CUB-200 first, then run our pruning method (Table 3.7). We still apply uniform G on all layers, however, for MobileNet we only prune its last *seven* 1×1 convolution layers. When pruning to G=4, ResNet-50 adds around 5% test error while MobileNet increases about 10%. It is not surprising since MobileNet is already compact. Comparing with ThiNet [190], which is constructed by filter-level pruning, our results are better in general. However, it is worth to note that ThiNet is pruned from either AlexNet or VGG-16 [258], which are less efficient than ResNet-50 in the first place.

Inference speed. To support our claim on using structured pruning can lead to better inference performance than unstructured pruning, we compare our GConv-based model with others on real-world hardware devices. We evaluate MobileNet on both a rack server that hosts an Intel Xeon X5690 CPU and an NVIDIA 1080Ti GPU, and NVIDIA Jetson TX2, which is an embedded device consists of an integrated 256-core Pascal GPU.

MobileNet is built on depthwise separable convolution, and since depthwise is in fact GConv (G equals to C_{in}), we can only prune the pointwise part into GConv. Due

Table 3.8: Inference speed of MobileNet-224. Batch size equals to 1 for all cases. CPU and GPU denotes the Xeon and 1080Ti on the rack server and TX2 refers to the embedded GPU on Jetson TX2.

Model	Method	Time (ms)		
		CPU	GPU	TX2
MobileNet (baseline)	Dense	34.5	4.75	10.4
MobileNet (G=2)	SpMM	33.6	5.49	18.1
MobileNet ([346])		48.5	5.58	23.5

Table 3.9: Comparison of the top-1 test error (%) between using the *heuristic* algorithm and not using (*plain*). ResNet-50 results are collected on ImageNet.

Model	heuristic		plain	
	C10	C100	C10	C100
ResNet-110 (G = 2)	6.69	27.63	7.13	28.08
ResNet-110 (G = 8)	9.39	33.28	11.17	37.29
ResNet-50 (G = 2)	25.90		28.04	

to the lack of support for sparse convolution from PyTorch, we need to manually implement 1×1 GConv by *sparse matrix multiplication* (SpMM)⁸. Results are listed in Table 3.8. Compared with the highly optimised dense implementation, our sparsified model is normally slower. However, we can achieve notable speed-up compared with the counterpart model [346] pruned to the same level of sparsity ($\sim 50\%$) but in an unstructured way⁹. We will explore more on inference acceleration methods in the future.

In Table 3.8, inference speed of our pruned model is faster than dense model hand-crafted by ourselves on matrix multiplication, but relatively slower than those implemented by heavily tuned DNN libraries. We will explore inference acceleration methods and deploy them to custom hardware in the future.

Ablation Study

This section investigates the effect of different design choices that may appear while using our method.

⁸[torch.sparse](https://pytorch.org/docs/1.0.1/sparse.html) package documentation: <https://pytorch.org/docs/1.0.1/sparse.html>

⁹This pruned model is downloaded from the Model Zoo of the Distiller project: https://nervanasystems.github.io/distiller/model_zoo/index.html

Table 3.10: Comparison between pruning and training from scratch for ResNet-164 on CIFAR-100 regarding top-1 train and test error.

Method	Train Err. (%)		Test Err. (%)	
	G=2	G=4	G=2	G=4
Pruning	1.75	10.50	23.66	27.15
Shuffle	0.81	3.55	23.77	26.60
None	2.58	16.28	26.63	32.00

Effect from heuristic algorithm. We compare our algorithm with a *plain* algorithm, which tries to perform GConv pruning without sorting channels. Referring to Section 3.3.2, this plain algorithm simply sets N_S to 0. This comparison is performed under the same group configuration to see whether our heuristic algorithm can improve the resulting accuracy. Table 3.9 presents the comparison between both approaches on ResNet-110 (CIFAR-10/100) and ResNet-50 (ImageNet) with uniform G . All experiments use the same training scheme (learning rate, number of epochs, etc.).

We notice that by using the heuristic algorithm, the test error of all variants on all datasets is reduced. This result is also in line with the relative order of recovery ratio as shown in Fig. 3.6: $N_S=0$ has less recovery ratio than $N_S=10$ that is used by our heuristic algorithm, and it performs worse regarding the model accuracy. It can provide a concrete evidence that maximising recovery ratio, or equivalently minimising pruned magnitude, is an effective approach to conduct GConv pruning.

Compare with training from scratch. Inspired by [186], we try to investigate that besides the reduction in training budget whether pruning can also achieve higher test accuracy than training from scratch. We focus on training GConv variants of ResNet-164 on CIFAR-100 by the same training from scratch schedule as [326]. These variants use uniform G numbers 2 and 4, and they may use channel shuffle [334] to exchange information among groups or not. We also provide models with the same G number produced by our pruning method.

Table 3.10 shows that overall training from scratch with channel shuffle performs better than other methods. Comparing pruning with channel shuffle, we notice that the difference in train error is much larger than test error, which implies that the worse performance from the pruning method may due to its improper training setup, e.g., the number of training epochs is too limited. But still, pruned models perform much better than their counterparts that are trained from scratch without channel permutation. It shows that the permutation of channels after GConv is indeed an

important architectural choice.

3.3.5 Discussions

This section proposes a novel pruning method that prunes a trained CNN model into one that is built on GConv. We formulate the layer-wise pruning problem as finding optimal permutations to incorporate the structural constraints imposed by GConv, and we efficiently solve it through our heuristic algorithm. We can further explore the best sparsity configuration of a whole model under specific inference budget constraints.

Our empirical findings demonstrate that SPGC can effectively achieve a wide range of pruning levels while maintaining high model accuracy. Figure 3.7 illustrates that even when pruning up to 70% of the parameters, the decrease in accuracy remains below 5%. In comparison to existing approaches, our method consistently achieves nearly identical accuracy levels when pruning to the same degree for ResNet-18 on ImageNet [339] (as shown in Table 3.6).

3.4 Summary

This chapter proposes two novel methods, Aegis and SPGC, for memory-efficient DL training and inference respectively. Aegis addresses the challenges in mixed precision training, specifically, the underflow problems that appear in each gradient tensor during training, through a statistical model and the gradient scaling algorithm. SPGC aims to provide an efficient algorithm that can prune a convolution layer into group convolution, so that the total computation budget can be reduced while the inference speed can retain. We find the relationship between GConv and block matrices and propose a heuristic algorithm that operates on the block representation. Empirically, Aegis results in a stabler mixed precision training algorithm that produces better models, and SPGC can prune pre-trained models into GConv-based ones without losing much accuracy.

Chapter 4

Polyhedral Compilation Infrastructure

4.1 Introduction

Deep Neural Networks (DNNs) are built upon nested loops. The most common DNN operators are matrix multiplication and convolution, which iterate through their multi-dimensional operands by nested loops to carry out computation. Loop bounds, as well as data access addresses in these operators are calculated from affine combinations of their enclosing loop indices and other constant parameters. Moreover, a typical DNN topology has no dynamic control structure, i.e., the control-flow should not change during runtime. Given all these properties, DNN topologies are great targets for polyhedral modelling. Polyhedral modelling (Section 2.3) provides a powerful toolbox for analysing and optimising computation tasks with nested loops, affine loop bounds and data access addresses, and static control-flow. The instances of each statement in the nested loops are abstracted as points with integer values in a polygon. Dependence analysis, scheduling, and loop transformations are formulated as clear mathematical problems using linear algebra and solvable by linear programming. With the help from these features, we can easily reason and concisely specify compilation optimisation strategies, especially for suitable tasks like DNNs. ~~(G1) On modern parallel processors like CPUs and GPUs, there are empirical evidences that justify adapting polyhedral modelling to DNN computation optimisation, e.g., Tensor Comprehensions [280], R-Stream-TF [231], Tiramisu [14], etc.~~

While the polyhedral methodology has been successfully adopted in Deep Learning frameworks [280, 231, 14], there are still gaps to be addressed in the compilation infrastructure domain to enhance the usability of polyhedral methods. In this

chapter, our primary focus is on addressing the key problem of seamlessly integrating the polyhedral methodology as a reusable and modularized component within an industrial-level compiler infrastructure. This integration targets both software compilation and hardware synthesis, aiming to enhance the overall efficiency and effectiveness of the compilation process.

To address this challenge, our novel contributions in this chapter are two projects improving the state-of-the-art modularized compiler infrastructure MLIR [166]. One is called Polygeist, a framework that brings C/C++ code into polyhedral representation in MLIR and performs polyhedral scheduling through interfacing with external polyhedral tools. The other is named POLSCA, a framework that expands the effort of Polygeist into high-level synthesis. When evaluating our contributions, we investigate whether our compiled programs, either software or hardware, can achieve a comparable, or even higher level of speed-up compared to existing polyhedral tools. The speed-up is measured by run-time for sequential or parallelized software programs and by wall clock cycles for hardware designs. We evaluate both work on Polybench/C [330], and it shows that both Polygeist and POLSCA can handle conventional polyhedral workload and generate either efficient software program or hardware designs through an external polyhedral scheduler. Note that we do not aim to improve the core polyhedral scheduling algorithms, instead, we intent to create a better toolchain that is agnostic to a specific polyhedral scheduler.

We demonstrate the details about Polygeist and POLSCA in the following sections, which are originated from our published papers [214, 342]. Polygeist is a collaborative project, and the main contribution from this thesis is Polymer (Section 4.3), a bridge between MLIR and polyhedral schedulers. Section 4.2 mainly presents the work from other authors of [214].

4.2 Polygeist: Affine C/C++ in MLIR

This section describes the Polygeist framework. In general, the purpose of Polygeist is to bring C/C++ code into MLIR and apply transformations within the MLIR scope. We mainly perform polyhedral transformation, which requires that the input C/C++ code have valid polyhedral representation, i.e., we can find Static Control Parts (SCoPs) within them.

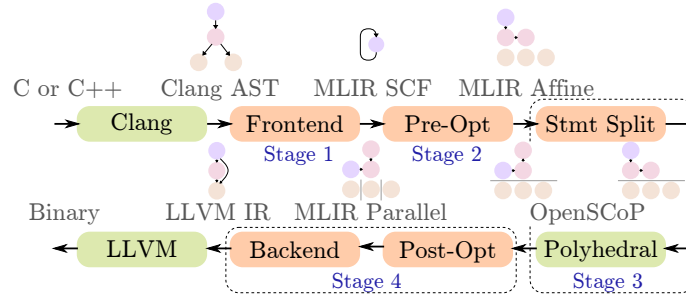


Figure 4.1: The Polygeist compilation flow consists of 4 stages. The frontend traverses Clang AST to emit MLIR SCF dialect (Section 4.2.1), which is raised to the Affine dialect and pre-optimized (Section 4.2.2). The IR is then processed by a polyhedral scheduler (Sections 4.3,4.3.2) before post-optimization and parallelization (Section 4.2.3). Finally, it is translated to LLVM IR for further optimization and binary generation by LLVM [214].

Polygeist consists of 4 components, listed in the order as demonstrated in Fig. 4.1:

1. a *frontend* that allows entering MLIR at the SCF loops level from C or C++ code (Section 4.2.1);
2. a *preprocessing step* in MLIR that raises to the Affine dialect (Section 4.2.2);
3. a *polyhedral scheduler* of the Affine parts of the program *via* a round-trip to and from OpenSCoP (Section 4.3) and running Pluto transformations, controlled by the new statement splitting heuristic (Section 4.3.2);
4. a *backend* that runs postprocessing MLIR optimizations (Section 4.2.3) and final lowering to an executable.

This section covers components 1, 2, and 4, which are mainly about how to transform C/C++ code into MLIR and optimises them for better executable code. We discuss the component 3, which is also named as *Polymer*, in the next section. Polymer focuses on the polyhedral transformation part of Polygeist.

There is a particular challenge associated with each of the frontend, the preprocessing step, and the backend. The frontend needs to address the challenge from the discrepancy between the C/C++ semantics with the MLIR IRs we target, e.g., the control-flow operations in the SCF dialect (Section 2.4.2) are at a higher level than those in C/C++, and representing C/C++ local variables may result in illegal MLIR IR code. The preprocessing step is about converting SCF code to Affine code, so that we can identify which parts of the code are optimisable by polyhedral schedulers. It is challenging to find patterns of Affine code, including loop bounds and conditionals, from what the frontend produces. Finally, the backend that receives polyhedral optimised code should convert them into efficient lower-level dialects, specifically, we need to identify the parallelism opportunities that commonly exist in polyhedral products and turn them into corresponding MLIR parallelised operations.

Most of the contents in this section are published in Moses et al. [214]. The Polygeist framework is open-sourced at <https://github.com/llvm/Polygeist>.

4.2.1 Frontend

Polygeist relies on the Clang AST to emit MLIR IRs. It thus avoids reimplementing parsing and language-level semantic analysis and handles modern C and C++ features. As is typical for compiler frontends, Polygeist creates a recursive symbol table data structure to look up the correct variable for a given scope. Polygeist lazily registers all global variables and functions found in the AST to its symbol table before generating any code. Polygeist then traverses the call graph from a given entry function (`main` by default), creating and defining MLIR functions as necessary.

Control-flow In contrast to traditional compiler pipelines, targeting a branch-based IR, Polygeist leverages the high-level MLIR operations such as `scf.while` (a looping construct) and `scf.if` (a conditional construct) within the SCF dialect to preserve the control flow structure of the source code. Although we can delay the construction of Affine dialect in the preprocessing step, if there exist `#pragmascope` that are used by many polyhedral tools [289] to indicate code regions that are known as SCoPs, Polygeist uses `affine.for` instead of `scf.while` to directly indicate the loop as an affine loop.

Typing and local variables While emitting operations, Polygeist must decide how to represent C/C++ types within MLIR. For primitive types such as `int` or

```

1 void set_array(int N, double val, double* array) {
2     for (int i = 0; i < N; i++)
3         array[i] = val;
4 }
5 int main(int argc, char** argv) {
6     double array[10];
7     set_array(10, array)
8 }

```

⇓

```

func @set_array(%N: i32, %val: f64
               %array: memref<?xf64>) {
    %0 = index_cast %N : i32 to index
    affine.for %i = 0 to %0 {
        affine.store %val, %array[%i] : memref<?xf64>
    }
    return
}

func @main(%argc: i32,
           %argv: !llvm.ptr<ptr<i8>>) -> i32 {
    %array = memref.alloca() : memref<10xf64>
    %arraycst = memref.cast %array : memref<10xf64> to
               memref<?xf64>
    call @set_array(%N, %val, %arraycst) :
               (i32, f64, memref<?xf64>) -> ()
}

```

Listing 7: Example demonstrating Polygeist frontend. For functions expected to be compiled with Polygeist such as `setArray`, pointer arguments are replaced with `memref`'s [214].

`float`, Polygeist emits an MLIR variant of that type with the same width as would be used within LLVM/Clang. On the other hand, for pointer, reference and array types, Polygeist uses `memref` type (Table 4.1). This allows Polygeist to preserve more of the structure available within the original program (e.g., multi-dimensional arrays) and enables interaction with MLIR's high-level memory operations.

We use `memref` to represent local variables stored on stack. These `memref` will be located at the top of each function or block. This permits the desired semantics of C or C++ to be implemented with relative ease. However, as many local variables and arguments contain `memref` types, this immediately results in a `memref` of a `memref`, which is illegal outside of Polygeist. As a remedy, we implement a memory-to-register (`mem2reg`) transformation pass that eliminates unnecessary loads, stores, and allocations within MLIR constructs, which empirically eliminates all `memrefs` of `memref` in the Polybench/C suite [330].

Listing 7 summarises the effect from the Polygeist frontend.

C type	LLVM IR type	MLIR type
int	i32 (on machine X)	i32 (on machine X)
intNN_t	iNN	iNN
uintNN_t	iNN	uiNN
float	float	f32
double	double	f64
ty *	ty *	memref<? x ty>
ty &	ty *	memref<1 x ty>
ty **	ty **	memref<memref<? x ty>>
ty[N][M]	[N x [M x ty]]*	memref<N x M x ty>

Table 4.1: Type correspondence between C, LLVM IR and MLIR types [214].

4.2.2 Raising to Affine

The translation from C or C++ to MLIR directly preserves high-level information about loop structure and n-D arrays, but does not generate other Affine operations. Polygeist subsequently raises memory, conditional, and looping operations into their Affine dialect counterparts if it can prove them to be legal affine operations.

Memory operations and loop bounds We need to convert legally affine operations and loop bounds using the `affine_map` structure. To achieve this, we should fold primitive operations, especially those that calculates arithmetic on loop induction variables, into `affine_map`.

To convert an operation, Polygeist replaces its bound and subscript operands with identity affine maps. It then folds the operations computing the map operands, e.g., `addi` for integer addition, into the map itself. Values that are transitively derived from loop induction variables become map dimensions and other values become symbols. The differences between symbol and dimension are explained in Section 2.4.2.

For example, `affine_map<() [s0] -> (s0) > [%bound]` with `%bound = addi %N, %i`, where `%i` is an induction variable, is folded into `affine_map<(d0) [s0] -> (s0 + d0) > (%i) [%N]`. The process terminates when no operations can be folded or when Affine value categorization rules are satisfied.

Conditionals Conditional operations are emitted by the frontend for two input code patterns: `if` conditions and ternary expressions. The condition is transformed by introducing an *integer set* (Section 2.4.2) and by folding the operands into it similarly to what we apply on the affine maps. Specifically, if there is an `and`

operation, we separate the set constraints; if there is a `not` operation, we invert the constraint. Polygeist processes nested conditionals with C-style short-circuit semantics, in which the subsequent conditions are checked within the body of the preceding conditionals, by hoisting conditions outside the outermost conditional when legal and replacing them with a boolean operation or a `select`.

The *for*-loop in Listing 7 is not annotated within a `#pragmascope` region, but can still be translated to `affine.for` thanks to the Affine raising.

4.2.3 Post-Transformations and Backend

The result from the previous section is a mixture of different MLIR dialects, including Affine. If there are any Affine code, we will pass them to Polymer (Section 4.3) to carry out polyhedral optimisation. Once it is done, we have polyhedral-optimised code described in Affine, and in this section, we describe how to continue the journey and produce efficient executable code. There are two transformations, one is to identify and extract reductions from loops, and the other is to match parallel loop structures.

Transforming Loops with Carried Values (Reductions) Polygeist leverages the first-class support from MLIR for loop-carried values to detect, express and transform reduction-like loops. This support does not require source code annotations, unlike source-level tools [243] that use annotations to enable detection, nor complex modifications for parallel code emission, unlike Polly [61], which suffers from LLVM missing first-class parallel constructs. We do not modify the polyhedral scheduler either, relying on post-processing for reduction parallelization, including outermost parallel reduction loops.

The overall approach follows the definition proposed in [137] with adaptations to MLIR’s region-based IR, and is illustrated in Figure 4.2. Polygeist identifies memory locations modified on each iteration, i.e. `load/store` pairs with loop-invariant subscripts and no interleaving aliasing `stores`, by scanning the single-block body of the loop. These are transformed into *loop-carried values* or secondary induction variables, with the `load/store` pair lifted out of the loop and repurposed for reading the initial and storing the final value. Loop-carried values may be updated by a chain of side effect-free operations in the loop body. If this chain is known to be associative and commutative, the loop is a *reduction*.

```

    affine.for %i = ... {
      // Reduction into r1[0]
      %1 = affine.load %r1[0]
      %5 = addi %1, %2
      affine.store %5, %r1[0]
      // Loop-dependent load
      %10 = affine.load %r2[%i]
      %15 = addi %10, %2
      // Interleaving store
      %20 = affine.load %r2[0]
      affine.store %21, %r2[0]
      %25 = addi %20, %2
      // May have side effects
      %30 = affine.load %r3[0]
      call @f(%30, %2)
    }

    %init = affine.load %r1[0]
    %red = affine.for %i = ...
    iter_args(%arg = %init) {
      // Reduction accumulation
      %5 = addi %arg, %2
      // Loop-dependent load
      %10 = affine.load %r2[%i]
      %15 = addi %10, %2
      // Interleaving store
      %20 = affine.load %r2[0]
      affine.store %21, %r2[0]
      %25 = addi %20, %2
      // May have side effects
      %30 = affine.load %r3[0]
      call @f(%30, %2)
      // Yield accumulated
      affine.yield %5
    }
    affine.store %red, %r1[0]

```

Figure 4.2: Polygeist detects memory locations accessed in all loop iterations, e.g. reduction accumulators such as `%r1[0]` and transforms them to loop-carried values (secondary induction variables), except when computed with side-effects, interleaved stores or by non-associative/commutative operations [214].

Late Parallelization Rather than relying on the dependence distance information obtained by the affine scheduler, Polygeist performs a separate polyhedral analysis to detect loop parallelism in the generated code. The analysis itself is a classical polyhedral dependence analysis [70, 64] implemented on top of MLIR region structure. Performing it after SSA-based optimizations, in particular `mem2reg` and reduction detection, allows parallelizing more loops. In particular, reduction loops and loops with variables whose value is only relevant within a single iteration similar to live-range reordering [288] but without expensive additional polyhedral analyses (live-range of an SSA value defined in a loop never extends beyond the loop).

4.2.4 Evaluation

Our evaluation has two goals:

1. We want to demonstrate that the code produced by Polygeist without additional optimization does not have any inexplicable performance differences than a state-of-the-art compiler like Clang.
2. We explore how Polygeist’s internal representation can support a mix of affine and SSA-based transformation in the same compilation flow, and evaluate the potential benefits compared to existing source and compiler-based polyhedral tools.

We leave polyhedral scheduling related evaluation in Section 4.3.4. The experiment setup will stay the same.

Experimental Setup

We ran our experiments on an AWS `c5.metal` instance with hyper-threading and Turbo Boost disabled (Table 4.2). The system is Ubuntu 20.04 running on a dual-socket Intel Xeon Platinum 8275CL CPU at 3.0 GHz with 24 cores each, with 0.75, 35, 35.75 MB L1, L2, L3 cache per socket, respectively, and 256 GB RAM. We ran all 30 benchmarks from PolyBench [330], using the EXTRALARGE dataset. We ran a total of 5 trials for each benchmark, taking the execution time reported by PolyBench; the median result is taken unless stated otherwise. Every measurement or result reported in the following sections refers to double-precision data. All experiments were run on cores 1-8, which ensured that all threads were on the same socket and did not potentially conflict with processes scheduled on core 0.

Table 4.2: Hardware setup.

CPU	Clock rate	OS	RAM (GB)	L1/L2/L3 (MB)
Intel Xeon Platinum 8275CL	3.0 GHz	Ubuntu 20.04	256	1.5, 48, 71.5

In all cases, we use two-stage compilation: (i) using `clang` at `-O3` excluding unrolling and vectorization; or Polygeist to emit LLVM IR from C; (ii) using `clang` at `-O3` to emit the final binary. As several optimizations are not idempotent, a second round of optimization can potentially significantly boost (and rarely, hinder) performance. This is why we chose to only perform vectorization and unrolling at the last optimization stage. Since Polygeist applies some optimizations at the MLIR level (e.g., `mem2reg`), we compare against the two-stage compilation pipeline as a more fair baseline (CLANG). We also evaluate a single-stage compilation to assess the effect of the two-stage flow (CLANGSING).

Baseline Performance

Polygeist must generate code with runtime *as close as possible* to that of existing compilation flows to establish a solid baseline. In other words, Polygeist should *not introduce overhead nor speedup* unless explicitly instructed otherwise, to allow for measuring the effects of additional optimizations. We evaluate this by comparing the runtime of programs produced by Polygeist with those produced by Clang at

the same commit (Apr 2021)¹. Fig. 4.3 summarizes the results with the following flows:

- **CLANG**: A compilation of the program using Clang, when running two stages of optimization;
- **CLANGSING**: A compilation of the program using Clang, when running one stage of optimization;
- **MLIR-CLANG**: A compilation flow using the Polygeist frontend and preprocessing optimizations within MLIR, but not running polyhedral scheduling nor post-processing.

Performance Analysis

Benchmarking The transformation of reduction loops, in particular parallelization, may result in a different order of partial result accumulation. This is not allowed under IEEE 754 semantics, but is supported by compilers with `-ffast-math` option. We found that Polybench allocation function hinders Clang/LLVM alias analysis, negatively affecting performance in, e.g., `adi`. Therefore, we modified all benchmarks to use `malloc` that is known to produce non-aliasing pointers.

Finally, we found that the custom allocator used by Polybench had a nontrivial impact on the runtime of the kernel, particularly on the `adi` test. This partially stems from the implementation of the allocation itself and LLVM’s optimization pipeline assuming that calls to `malloc` do not alias with existing allocations. To maximize the fairness of the evaluation, all flows were modified to use a standard `malloc` as its memory allocation function.

Baseline Comparison We did not observe a significant difference between the runtimes of **CLANG** and **CLANGSING** configurations, with a geometric mean of 0.43% symmetric difference² across benchmarks. Therefore, we only consider **CLANG** as baseline throughout the remainder of this thesis. We did not observe a significant difference between the runtimes of **CLANG** and **MLIR-CLANG** configurations either, with a geometric mean of 0.24% symmetric difference.

We found a variation in runtimes of short-running benchmarks, in particular, the example named `jacobi-1d`. This can be attributed to the interaction with the

¹LLVM commit 20d5c42e0ef5d252b434bcb610b04f1cb79fe771

²Symmetric difference is computed as $2 \cdot |a - b| / (a + b)$.

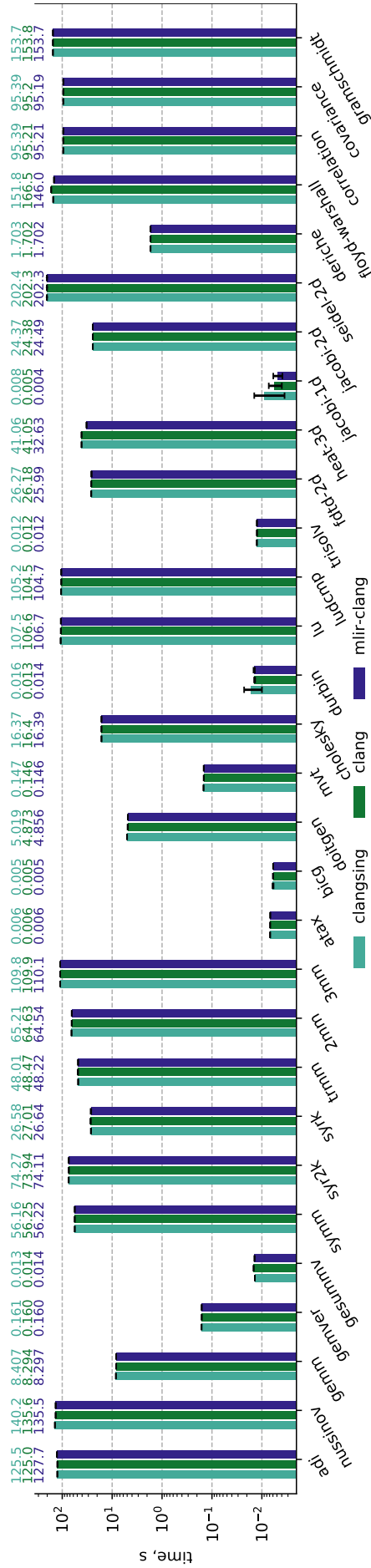


Figure 4.3: Mean and 95% confidence intervals (log scale) of program run time across 5 runs of Polybench in CLANG, CLANGSING and MLIR-CLANG configurations, lower is better. The run times of code produced by Polygeist without optimization is comparable to that of Clang. No significant variation is observed between single and double optimization. Short-running `jacobi-1d` shows high intra-group variation [214].

data initialization and benchmarking code, and with other OS processes. Excluding the benchmarks running in under 0.05s (`jacobi-1d`, `gesummv`, `atax`, `bicg`) from the analysis, we obtain 0.32% and 0.17% geomean symmetric differences respectively for the two comparisons above. These results suggest that our flow has no unexplained (dis)advantages over the baseline.

4.2.5 Discussions

This section presents Polygeist, a framework that connects C/C++ with MLIR in the polyhedral domain. Polygeist addresses the challenges in the frontend that maps C/C++ code to MLIR dialects (SCF, Affine, etc.), in the preprocessing step that raises SCF to Affine, and in the post-processing step that further improves the polyhedral-optimised code by considering reduction and parallelism opportunities. Empirical evaluation shows that on Polybench/C, Polygeist provides almost the same performance as using Clang, which demonstrates the usability of this framework.

Although the implications of this chapter may not directly influence the user interface of a deep learning framework, it subtly enhances the optimisation process of operator kernels conducted by researchers behind the scenes. Traditionally, these kernels, more often described in high-performance languages like C/C++ rather than Python, can be elevated into MLIR via Polygeist. This upgrade allows them to leverage a plethora of MLIR-based, modular compilation optimisation passes. This breakthrough, in turn, considerably alleviates the need for laborious manual performance tuning, paving the way for a more streamlined and efficient research process.

In the next section, we present the core polyhedral scheduling part of Polygeist, i.e., Polymer, and its evaluation results.

4.3 Polymer: Polyhedral Scheduling in MLIR

This section focuses on the polyhedral scheduling part of Polygeist, which is referred to as Polymer. In general, Polymer builds an interface between MLIR Affine and external polyhedral schedulers, so that we can perform fully fledged polyhedral transformation on the polyhedral representations in MLIR.

The key task that Polymer should do is the conversion between MLIR Affine and a representation that polyhedral tools understand. We chose to implement a bi-directional conversion to and from OpenScop [20], an exchange format readily consumable by numerous polyhedral tools, including Pluto [30], and further convertible to `isl` [286] representation. This allows Polymer to seamlessly connect with tools created in polyhedral compilation research without having to amend those tools to support MLIR.

There are two major challenges that Polymer should address in this procedure. First, most polyhedral tools are designed to operate on C or FORTRAN inputs build around *statements*, which do not have a direct equivalent in MLIR. We should design a mechanism that maps MLIR instructions to and from statement-like structure. Second, it is not always clear which instructions should be organised into a statement, or multiple statements. Different mappings may result in different optimisation results from the polyhedral scheduler.

Polymer aims to address these challenges by a creating a statement mapping mechanism, which can tackle the scenario when there are hazardous dependencies. Meanwhile, We further demonstrate that this gives Polymer an ability to favourably affect the behaviour of the polyhedral scheduler by controlling statement granularity (Section 4.3.2), which can improve performance if well utilised. Finally, we empirically evaluate the performance of Polymer, in comparison with Pluto and Polly on the Polybench/C benchmark suite. Like Polygeist, Polymer is also open-sourced as an individual project at: <https://github.com/kumasento/polymer>.

4.3.1 Statement mapping

Simple Statement Formation Observing that C statements amenable to the polyhedral model are (mostly) variable assignments, we can derive a mechanism to identify statements from chains of MLIR operations. A `store` into memory is the last operation of the statement. The backward slice of this operation, i.e., the operations transitively computing its operands, belong to the statement. The slice

extension stops at operations producing a value categorized as affine dimension or symbol, directly usable in affine expressions. Such values are loop induction variables or loop-invariant constants.

Some operations may end up in multiple statements if the value is used more than once. However, we need the mapping between operations and statements to be bidirectional in order to emit MLIR after the scheduler has restructured the program without considering SSA value visibility rules. If an operation with multiple uses is side effect free, Polygeist simply duplicates it. For operations whose duplication is illegal, Polygeist stores their results in stack-allocated `memref`'s and replaces all further uses with memory loads. Figure 8 illustrates the transformation for value `%0` used in operation `%20`. This creates a new statement.

Region-Spanning Dependencies In some cases, a statement may consist of MLIR operations across different (nested) loops, e.g., a load from memory into an SSA register happens in an outer loop while it is used in inner loops. The location of such a statement in the loop hierarchy is unclear. More importantly, it cannot be communicated to the polyhedral scheduler. Polygeist resolves this by storing the value in a stack-allocated `memref` in the defining region and loading it back in the user regions. Figure 8 illustrates this transformation for value `%0` used in operation `%10`. Similarly to the basic case, this creates a new statement in the outer loop that can be scheduled independently.

This approach can be seen as a `reg2mem` conversion, the inverse of `mem2reg` performed in the frontend. It only applies to a subset of values, and may be undone after polyhedral scheduling has completed. Furthermore, to decrease the number of dependencies and memory footprint, Polygeist performs a simple value analysis and avoids creating stack-allocated buffers if the same value is already available in another memory location and can be read from there.

SCoP Formation To define a SCoP, we outline individual statements into functions so that they can be represented as opaque calls with known memory footprints, similarly to Pencil [13]. This process also makes the inter-statement SSA dependencies clear. These dependencies exist between calls that *use* the same SSA value since all use-def chains (except for induction variables that are processed separately) have been encapsulated into statements. We lift all local stack allocations and place them at the entry block of the surrounding function in order to keep them visible after loop restructuring. Figure 9 demonstrates the resulting IR.

```

affine.for %i = ...
  %0 = affine.load %A[%i]
  affine.store %other, %A[%i] // motion-barrier
  affine.for %j = ... {
    %1 = affine.load %B[%j]
    %10 = mulf %0, %1 : f64    // use-1
    store %10, %res[%i, %j]
    %20 = addf %0, %0 : f64    // use-2
  }

↓

%tmp = memref.alloc() : memref<1xf64>
affine.for %i = ...
  %0 = affine.load %A[%i]
  affine.store %0, %tmp[0]    // store to scratchpad
  affine.store %other, %A[%i] // motion-barrier
  affine.for %j = ...
    %1 = affine.load %B[%k, %j]
    %2 = affine.load %tmp[0]  // load back for use-1
    %10 = mulf %2, %1 : f64   // use-1 (%2 instead of %0)
    affine.store %10, %res[%i, %j]
    %19 = affine.load %tmp[0] // load back for use-2
    %20 = addf %0, %0 : f64   // use-2 (%19 instead of %0)
  // ...

```

Listing 8: Polygeist breaks region-spanning use-def chains and handles multi-use values by introducing scratchpad storage when operation duplication is illegal. In absence of `motion-barrier` statement, the `%0` load would be duplicated and sunk. Pseudo-MLIR with types and braces omitted for brevity [214].

The remaining components of the polyhedral representation are derived as follows: the domain of the statement is defined to be the iteration space of its enclosing loops, constrained by their respective lower and upper bounds, and intersected with any “if” conditions. This process leverages the fact that MLIR expresses bounds and conditions directly as affine constructs. The access relations for each statement are obtained as unions of affine maps of the `affine.load` (read) and `affine.store` (must-write) operations, with RHS of the relation annotated by an “array” that corresponds to the SSA value of the accessed `memref`. Initial schedules are assigned using the $(2d + 1)$ formalism, with odd dimensions representing the lexical order of loops in the input program and even dimensions being equal to loop induction variables. Affine constructs in OpenScop are represented as lists of linear equality ($= 0$) or inequality (≥ 0) coefficients, which matches exactly the internal representation in MLIR, making the conversion straightforward.

Code Generation Back to MLIR The Pluto scheduler produces new schedules in OpenScop as a result. Generating loop structure back from affine schedules is a solved, albeit daunting, problem [19, 96]. Polygeist relies on CLooG [19] to generate an initial loop-level AST, which it then converts to Affine dialect loops and

```

func @S1(%A: memref<?xf64>, %tmp: memref<1xf64>,
        %i: index)
    %0 = affine.load %A[%i]
    affine.store %0, %tmp[0]    // store to scratchpad

func @S2(%A: memref<?xf64>, %other: f64, %i: index)
    affine.store %other, %A[%i]

func @S3(%B: memref<?x?xf64>, %tmp: memref<1xf64>,
        %res: memref<?x?xf64>, %i: index, %j: index)
    %1 = affine.load %B[%i, %j]
    %2 = affine.load %tmp[0]    // load back for use-1
    %10 = mul %2, %1 : f64      // use-1
    affine.store %10, %res[%i, %j]

func @S4(%tmp: memref<1xf64>, ...)
    %19 = affine.load %tmp[0]   // load back for use-2
    %20 = add %0, %0 : f64      // use-2
    // ...

%tmp = memref.alloc() : memref<1xf64>
affine.for %i = ...
    call @S1(%A, %tmp, %i)
    call @S2(%A, %other, %i)
affine.for %j = ...
    call @S3(%B, %tmp, %res, %i, %j)
    call @S4(%tmp)

```

Listing 9: Outlining makes polyhedral “statements” visible in code from Listing 8 [214].

conditionals. There is no need to simplify affine expressions at code generation since MLIR accepts them directly and can simplify them at a later stage. Statements are introduced as function calls with rewritten operands and then inlined.

4.3.2 Controlling statement granularity

Recall that Polygeist reconstructs “statements” from sequences of primitive operations (Section 4.3). We initially designed an approach that recovers the statement structure similar to that in the C input, but this is not a requirement. Instead, statements can be formed from any subsets of MLIR operations as long as they can be organized into loops and sorted topologically (i.e., there are no use-def loops between statements). To expose the dependencies between such statements to the affine scheduler, we reuse the idea of going through scratchpad memory: each statement writes the values required by other statements to dedicated memory locations, and the following statements read from those. The scratchpads are subject to partial array expansion [69] to minimize their effect on the affine scheduler as single-element scratchpad arrays create artificial scalar dependencies. This change in *statement*

```

for(i=0; i<NI; i++)
  for(j=0; j<NJ; j++)
    for(k=0; k<NK; k++)
S:   A[i][j] += f(B[k][i], C[k][j]);
      ↓
for(i=0; i<NI; i++)           double M[NK];
  for(j=0; j<NJ; j++)         for(k=0; k<NK; k++)
    double M[NK];             for(i=0; i<NI; i++)
    for(k=0; k<NK; k++)       for(j=0; j<NJ; j++)
S:   M[k] = f(B[k][i], C[k][j]); S:   M[k] = f(B[k][i], C[k][j]);
T:   A[i][j] += M[k];          T:   A[i][j] += M[k];

```

Listing 10: Splitting a nested reduction statement (top) into a fully parallel compute statement and a trivial reduction statement (bottom left) makes Pluto generate different schedules (bottom right). Further scratchpad array expansion may enable loop fission and give scheduler even more liberty [214].

granularity gives the affine scheduler unprecedented flexibility allowing it to choose different schedules for different *parts* of the same C statement.

Consider, for example, the statement **S** in Listing 10(top) surrounded by three loops iterating over **i**, **j** and **k**. Such contraction patterns are common in computational programs (this particular example can be found in the **correlation** benchmark with $B \equiv C$, see Section 4.3.4). The loop order that best exploits the locality is (**k**, **i**, **j**), which results in temporal locality for reads from **B** (the value is reused in all iterations of the now-innermost **j** loop) and in spatial locality for reads from **C** (consecutive values are read by consecutive iterations, increasing the likelihood of L1 cache hits). Yet, Pluto never proposes such an order because of a reduction dependency along the **k** dimension due to repeated read/write access to **A[i][j]** as Pluto tends to pick loops with fewer dependencies as outermost. While the dependency itself is inevitable, it can be moved into a separate statement **T** in Listing 10(bottom left). This approach provides scheduler with more freedom of choice for the first statement at a lesser memory cost than expanding the entire **A** array. It also factors out the reduction into a “canonical” statement that is easier to process for the downstream passes, e.g., vectorization.

Implementing this transformation at the C level would require manipulating C AST and reasoning about C (or even C++) semantics. This is typically out of reach for source-to-source polyhedral optimizers such as Pluto that treat statements as black boxes. While it is possible to implement this transformation at the LLVM IR level, e.g., in Polly, where statements are also reconstructed and injection of temporary allocations is easy, the heuristic driving the transformation is based on the loop structure and multi-dimensional access patterns which are difficult to recover at such a low level [95].

The space of potential splits is huge—each MLIR operation can potentially become a statement. Therefore, we devise a heuristic to address the contraction cases similar to Listing 10. Reduction statement splitting applies to statements:

- surrounded by at least 3 loops;
- with $LHS \neq RHS$, and using all loops but the innermost;
- with two or more different access patterns on the RHS.

This covers statements that could have locality improved by a different loop order and with low risk of undesired fission. This heuristic merely serves as an illustration of the kind of new transformations Polygeist can enable.

4.3.3 Comparison to Polly

Polly [90] is the state-of-the-art polyhedral compilation framework working together with LLVM-IR, which shares various similar features with Polymer. This section compares Polymer to Polly in details.

Translating into Polyhedral Representation In [90, Section 5.1], we can find specific constraints that Polly imposes on the input LLVM-IR to ensure SCoP translation feasibility, which state that the control-flow structure should be for-loops and if-conditions, the boundaries and constraints should be affine, and the only valid statements are assignments that store the result of an expression to an array element. Polly recovers the high-level control-flow structures and statements by employing existing LLVM semantic analysis toolbox. Since Polymer consumes input described in the Affine dialect, most of these constraints are enforced already. For instance, the constraints on and translation steps from LLVM-IR loops described in [90, Section 5.2] are not necessary when the input representation is the Affine dialect, which innately requires the loop bounds to be affine expressions.

Nevertheless, when dealing with IR instructions, Polly and Polymer demand similar validation steps and face alike challenges. Both Polly and Polymer only allow instructions without side effects. Moreover, Polly and Polymer need to carry out memory aliasing analysis.

Additionally, even though Polly exhibits various transformations on LLVM IRs when mapping them to SCoP, the SCoP transformations leveraged by Polymer are still

largely unique. It is unclear if Polly applies similar transformation as our region-spanning dependence analysis, which should be done in the LLVM semantic analysis step instead of Polly based on its design as stated in [90, Section 5.2]. The maximal SCoP detection algorithm [90, Section 5.5] may also workaround this problem by smaller SCoPs that will not incur cross-region def-use chains. DeLICM [157], a work extended from Polly, shares similar insights on the problems we are facing, specifically, in [157, Listing 1-4]. Their approach, however, is designed at a higher level than ours, and it is unclear how much similarity between their low-level implementation and ours. On the other hand, our statement granularity controlling heuristic Section 4.3.2, is novel regarding existing Polly literature as far as we understand.

Translating from Polyhedral Representation Both Polly and Polymer need to translate from a polyhedral representation, namely SCoPs, back to the source representation, which are LLVM IR and MLIR Affine respectively. Polymer’s reverse translation is relatively simpler than Polly, since we only need to ensure the correct mapping of the statements between MLIR and SCoPs, while Polly adds extra optimisation steps to allow generating SIMD operations [90, Section 7.3]. Both frameworks leverage CLooG [19] as the underlying polyhedral code generator.

4.3.4 Evaluation

Compilation Flows

We evaluate Polymer within the Polygeist framework against Pluto and Polly, specifically, we have the following compilation flows:

- **PLUTO**: Pluto compiler auto-transformation [30] using `polycc`³ with flags `--noparallel` and `--tile`;
- **PLUTOPAR**: Same as above but with `--parallel` flag;
- **POLLY**: Polly [91] LLVM passes with affine scheduling and tiling, and no pattern-based optimizations [78];
- **POLLYPAR**: Same as above with auto-parallelization;
- **POLYGEIST**: Our flow with Pluto and extra transforms;

³Pluto commit `dae26e77b94b2624a540c08ec7128f20cd7b7985`

- **POLYGEISTPAR**: Same as above but with `--parallel` Pluto schedule and Polygeist parallelisation and reductions.

Running between source and LLVM IR levels, we expect Polygeist to benefit from both worlds, thus getting code that is on par or better than competitors. When using Pluto, both standalone and within Polygeist, we disable the emission of vectorization hints and loop unrolling to make sure both transformations are fully controlled by the LLVM optimizer, which also runs in Polly flows. We run Polly in the latest stage of Clang compilation, using `--mllvm -polly` and additional flags to enable affine scheduling, tiling and parallelization as required. Polly is taken at the same LLVM commit as Clang. We disable pattern-based optimizations [78] that are not available elsewhere. Fig. 4.4 and Fig. 4.5 summarize the results for sequential and parallel flows, respectively.

Performance Differences in Sequential Code

Overall, Polygeist leads to larger speedups, with $2.53\times$ geometric mean, than both Pluto ($2.34\times$) and Polly ($1.41\times$), although improvements are not systematic. Some difference between Polygeist and Polly is due to the employed polyhedral schedulers, e.g., in `lu` and `mvt`. Polygeist produces code faster than both Pluto and Polly in `2mm`, `3mm` and others thanks to statement splitting, see Section 4.3.4.

Given identical statements and schedules, codegen-level optimization accounts for other performance difference. `seidel-2d` is the clearest example: Pluto executes $2.7 \cdot 10^{11}$ more integer instructions than Polygeist. Assuming these to be index/address computations, a mix of `add` (throughput 1/2 or 1/4) and `imul/shl` (throughput 1), we can expect a ≈ 59 s difference at 3GHz, consistent with experimental observations. Polygeist optimizes away a part of those in its post-optimization phase and emits homogeneous address computation from `memref` with proper machine size type, enabling more aggressive bound analysis and simplification in the downstream compiler. Conversely, `jacobi-2d` has poorer performance because Polygeist gives up on simplifying CLoG code, with up to 75 statement copies in 40 branches, for compiler performance reasons, as opposed to Clang that takes up to 5s to process it but results in better vectorization. Further work is necessary to address this issue by emitting vector instructions directly from Polygeist.



Figure 4.4: Median speedup over CLANG for sequential configurations (log scale), higher is better. Polygeist outperforms ($2.53\times$ geomean speedup) both Pluto ($2.34\times$) and Polly ($1.41\times$) on average. Pluto can't process `adi`, which is therefore excluded from summary statistics [214].

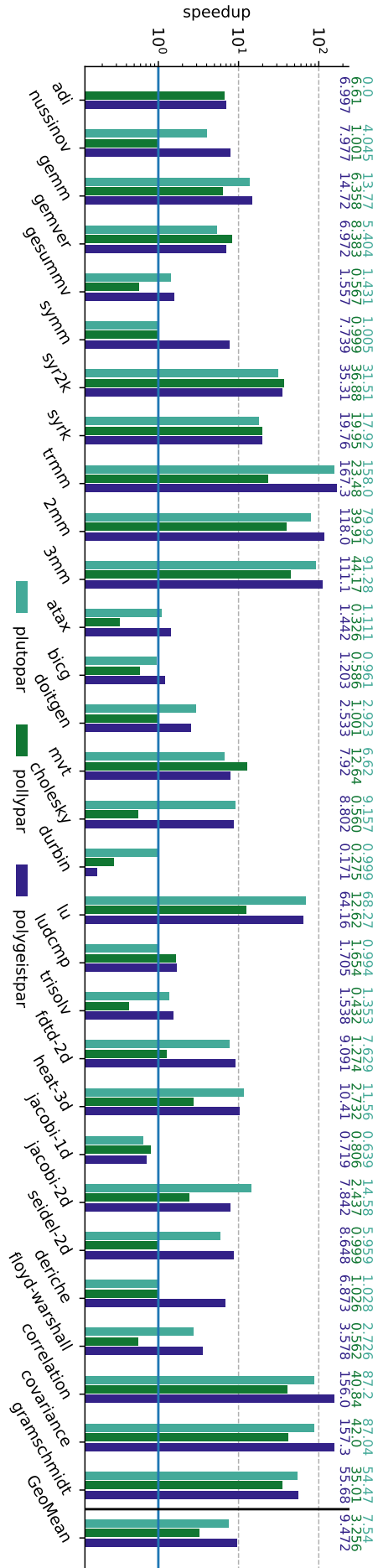


Figure 4.5: Median speedup over CLANG for parallel configurations (log scale), higher is better. Polygeist outperforms ($9.47\times$ geometric speedup) both Pluto ($7.54\times$) and Polly ($3.26\times$) on average. Pluto can't process `adi`, which is therefore excluded from summary statistics [214].

```

for (i=0; i<_PB_W; i++){
  ym1 = SCALAR_VAL(0.0);
  // ...
  for (j=0; j<_PB_H; j++){
    ym1 = y1[i][j];
    /*...*/ }
}

%z = constant 0.0 : f64
affine.parallel %i = ... {
  affine.for %j = ...
    iter_args(%ym1=%z)->f64 {
      %0=affine.load %y1[%i,%j]
      // ...
      affine.yield %0
    }
}

```

Figure 4.6: Excerpt from the **deriche** benchmark. The outer loop reuses `ym1` which makes it appear non-parallel to affine schedulers (left). Polygeist detects parallelism thanks to its `mem2reg` optimization, reduction-like loop-carried `%ym1` value detection and late parallelization (right) [214].

Performance Differences In Parallel Code

Similarly to sequential code, some performance differences are due to different schedulers. For example, in **cholesky** and **lu**, both Pluto and Polygeist outperform Polly, and the remaining gap can be attributed to codegen-level differences. Conversely, in **gemver** and **mvt** Polly has a benefit over both Pluto and Polygeist. On **ludcmp** and **syr(2)k**, SSA-level optimizations let Polygeist produce code which is faster than Pluto and at least as fast as Polly. These results demonstrate that Polygeist indeed leverages the benefits of both the affine and SSA-based optimizations.

Polygeist is the only flow that obtains speedup on **deriche** ($6.9\times$) and **symm** ($7.7\times$). Examining the output code, we observe that only Polygeist manages to parallelize these two benchmarks. Considering the input code in Figure 4.6, one can observe that the `i` loop reuses the `ym1` variable, which is interpreted as parallelism-preventing loop-carried dependency by polyhedral schedulers. Polygeist performs its own parallelism analysis after promoting `ym1` to an SSA register (carried by the `j` loop) whose use-def range does not prevent parallelization.

Similarly, the Polygeist paralleliser identifies two benchmarks with parallel reduction loops that are not contained in other parallel loops: **gramschmidt** and **durbin**. **gramschmidt** benefits from a $56\times$ speedup with Polygeist, compared to $34\times$ with Polly and $54\times$ with Pluto. **durbin** sees a $6\times$ slowdown since the new parallel loop has relatively few iterations and is nested inside a sequential loop, leading to synchronization costs that outweigh the parallelism benefit. Section 4.3.4 explores the **durbin** benchmark in more detail. Polybench is a collection of codes (mostly) known to be parallel and, as such, has little need for reduction parallelization on CPU where one degree of parallelism is sufficient. When targeting inherently target architectures as GPUs, however, exploiting reduction parallelism could be vital for achieving peak performance [164, 243].

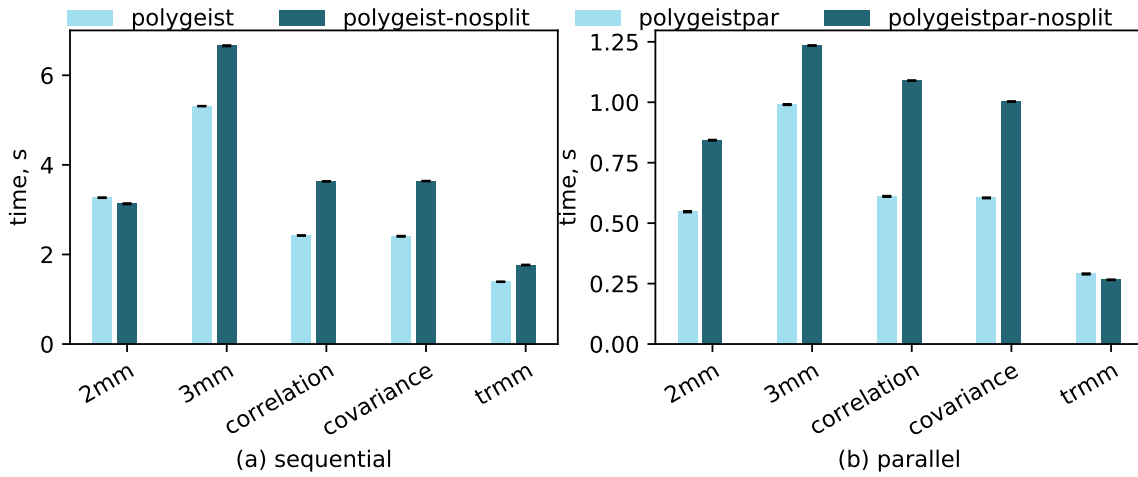


Figure 4.7: Mean and 95% confidence intervals of run time across 5 runs of Polybench where statement splitting is applicable (Section 4.3.2), lower is better. It results in faster run time (geomean $1.28\times$ sequential, $1.39\times$ parallel speedup) except for sequential `2mm` (-4%) and parallel `trmm` (-9%) [214].

Case Study: Statement Splitting

We identified 5 benchmarks where the statement splitting heuristic applied: `2mm`, `3mm`, `correlation`, `covariance` and `trmm`. To assess the effect of the transformation, we executed these benchmarks with statement splitting disabled, suffixed with `-nosplit` in Figure 4.7. In sequential versions, `2mm` is 4.1% slower (3.13s vs 3.26s), but the other benchmarks see speedups of 25% , 50% , 51% and 27% , respectively. For parallel versions, the speedups are of 36% , 20% , 44% , 40% and -9% respectively.

Examination of polyhedral scheduler outputs demonstrates that it indeed produced the desired schedules. For example, in the `correlation` benchmark which had the statement `A[i][j] += B[k][i]*B[k][j]` Polygeist was able to find the (k, i, j) loop order after splitting. Using hardware performance counters on sequential code we confirm that the overall cache miss ratio has indeed decreased by 75% , 50% , 20% , 27% , and -26% , respectively. However, the memory traffic estimated by the number of bus cycles has *increased* by 9% for `2mm`, and decreased by 18% , 32% , 32% , and 21% for the other benchmarks. This metric strongly correlates with the observed performance difference in the same run ($r=0.99$, $p = 3\cdot 10^{-11}$). This behaviour is likely due to the scheduler producing a different fusion structure, e.g., not fusing outermost loops in `2mm`, which also affects locality. Similar results can be observed for parallel code. Further research is necessary to exploit the statement splitting opportunities, created by Polygeist, and interplay with fusion.

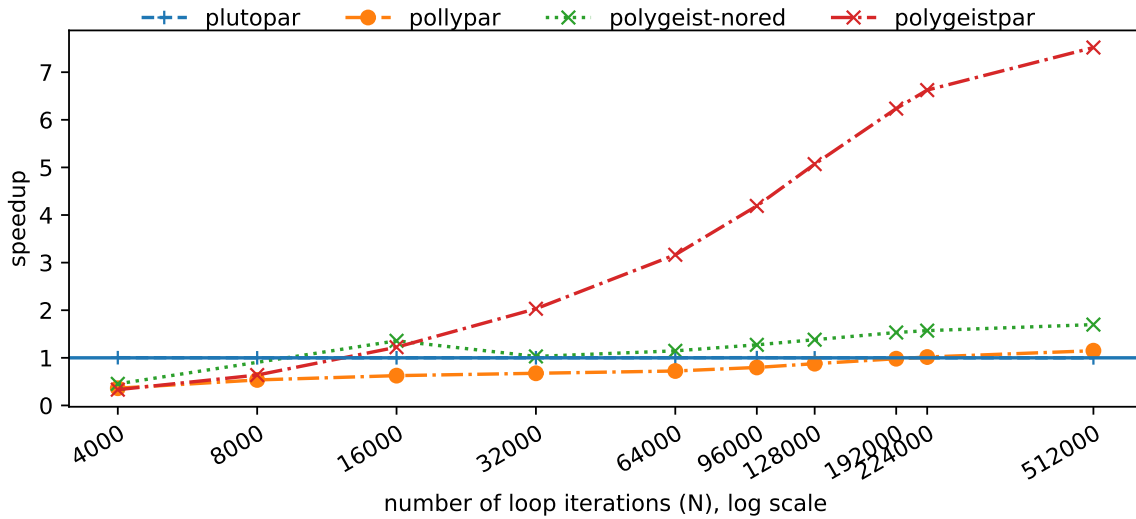


Figure 4.8: Reduction parallelization allows POLYGEISTPAR to produce larger speedups in *durbin* and at smaller sizes than POLLYPAR, POLYGEISTPAR without reduction support. PLUTOPAR fails to parallelize, hence no speedup [214].

Case Study: Reduction Parallelization in *durbin*

In this benchmark, Polygeist uses its reduction optimization to create a parallel loop that other tools cannot. For the relatively small input run by default, $N = 4000$ iterations inside another sequential loop with N iterations, the overall performance decreases. We hypothesize that the cost of creating parallel threads and synchronizing them outweighs the benefit of the additional parallelism and test our hypothesis by increasing N . Considering the results in Figure 4.8, one observes that Polygeist starts yielding speedups (> 1) for $N \geq 16000$ whereas Polly only does so at $N \geq 224000$, and to a much lesser extent: $6.62\times$ vs $1.01\times$. Without reduction parallelization, Polygeist follows the same trajectory as Polly. Pluto fails to parallelize any innermost loop and shows no speedup. This evidences in favour of our hypothesis and highlights the importance of being able to parallelize reductions.

4.3.5 Discussions

This section presents Polymer, the core polyhedral scheduling part to Polygeist, which interacts between external polyhedral schedulers and MLIR. Polymer addresses two challenges arise in this procedure, one is to find a mapping between MLIR instructions and polyhedral statements, which needs concrete analysis on the instruction dependences, the other is to explore different ways of splitting multiple instructions into different statements, which can give us performance boost depending on how the external polyhedral scheduler works. Empirically, we find that Polymer can provide Polygeist better performance than Pluto and Polly, on

average, and we gather evidence that the statement splitting is beneficial.

Next section explores the polyhedral methodology in the domain of High-Level Synthesis (HLS), in which we present the tool called POLSCA that is an extension to Polygeist/Polymer, built upon MLIR.

4.4 POLSCA: Polyhedral High-Level Synthesis

To alleviate the heavy labour in building RTL designs, engineers start to describe designs using abstract, C-like languages and let the *high-level synthesis* (HLS) tools generate low-level representations. After years of development, HLS has shown its potential in many domains, e.g., high-performance computing and deep learning, but it still has many challenging problems, most of which rely on precise dependence analysis among statements within nested loops. If the input code is restricted to static control flow using only affine expressions for bounds and addresses, i.e., *Static Control Parts* (SCoPs), we can leverage polyhedral models to find formal solutions [180, 230, 299]. Using the polyhedral model, which depicts loop statement instances as integral points within polyhedra, we can not just formally analyse dependences, but also automatically transform nested loops for better parallelism [30]. Despite the promising merits, polyhedral transformation is not yet pervasively used in HLS due to the lack of a proper framework.

The current polyhedral HLS workflow has two stages: using existing polyhedral tools (e.g., Pluto [30], isl [286]) to transform C/C++ code and passing its result to backend HLS tools (e.g., Xilinx Vitis) that produce hardware. Unfortunately, polyhedral and HLS tools are independently developed following their own agenda, which consequently leads to unsatisfactory interfacing and poor user experience. There are two challenges:

1. *SCoP annotation*. Users should annotate their code to indicate which parts are SCoPs that polyhedral tools should transform. These SCoPs should satisfy the affine constraints imposed by polyhedral models, which HLS engineers are not familiar with. Moreover, since polyhedral tools are HLS-agnostic, their optimization can be suboptimal for HLS if SCoPs are annotated improperly.
2. *HLS directives*. Code generators [19, 96] scan the transformed polyhedral representation to generate the optimized program. Unfortunately, this optimized program is normally unreadable, on which users can hardly specify HLS directives to produce efficient hardware. Meanwhile, in most cases we may further transform polyhedral optimized code, e.g., to apply memory partitioning, which is challenging due to the code complexity.

Both challenges indicate the lack of automated processes between polyhedral methods and HLS. This thesis presents POLSCA⁴, a compiler framework that aims to

⁴POLSCA = POLyhedral high-level Synthesis using Compiler Automation. Code repository: <https://github.com/kumasento/polsca>.

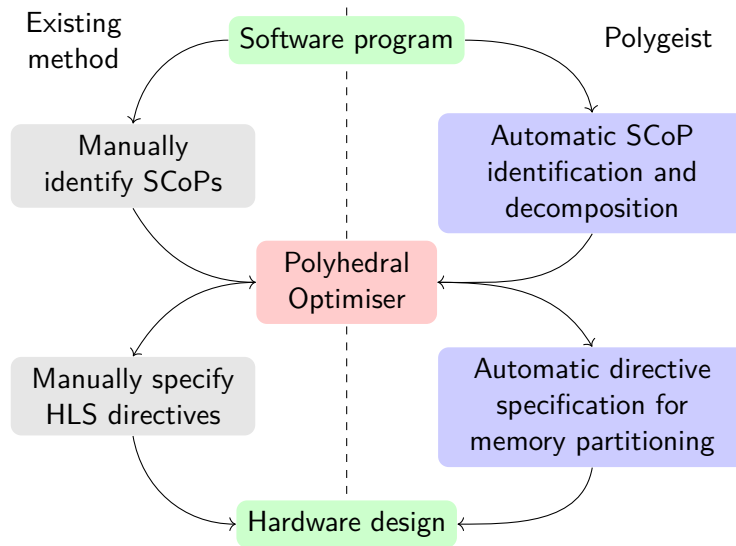


Figure 4.9: Summary of our contributions (right) in polyhedral HLS compared with existing methods (left).

address these challenges by applying pre- and post-polyhedral code transformations. Pre-polyhedral transformations identify and annotate SCoPs by heuristics that balance the code complexity and parallelism, and post-polyhedral transformations inject HLS directives and make certain optimization opportunities, e.g., memory partitioning, explicit for HLS tools. Our prototype POLSCA tool consumes HLS designs in C/C++ and produces optimization results to backend HLS tools using LLVM IR.

To summarise, this thesis presents three major contributions:

1. auto-identification of SCoPs and generation of polyhedral representation from arbitrary HLS code;
2. transformations of polyhedral-generated code for HLS tools to produce optimized designs;
3. performance evaluation on Polybench/C [330] that contains fully polyhedral-optimisable examples demonstrating the benefits of POLSCA by achieving 1.5 times latency reduction compared with applying HLS on C code.

Two steps will illustrate how polyhedral tools can be integrated with HLS as a framework: first, adopt a polyhedral optimizer that can deal with a wide variety of HLS designs to support the proposed approach; second, further optimize the results from the polyhedral optimizer. This thesis focuses on the first step.

```

1  L0: for p = 0 to P
2    // get hashed input
3    L1: for i = 0 to M
4      for l = 0 to L
5        D[i][l] = array[addrIn[p][i][l]]
6    // two consecutive matrix multiplication
7    L2: for i = 0 to M
8      for j = 0 to N
9        T[i][j] = 0.0
10     for k = 0 to K
11       T[i][j] += a * A[p][i][k] * B[p][k][j]
12    L3: for i = 0 to M
13      for l = 0 to L
14        D[i][l] *= b
15     for j = 0 to N
16       D[i][l] += T[i][j] * C[p][j][l]
17    // get hashed output
18    L4: for i = 0 to M
19      for l = 0 to N
20        array[addrOut[p][i][l]] = D[i][l]

```

Listing 11: A motivating example for POLSCA. It is modified from *2mm* [330] with additional indirect memory addresses that create non-SCoPs. There are four separated loop nests below the outermost L0: L1 gathers data from `array` into `D` through a hash table `addrIn`, L2 and L3 run two GEMMs (generalised matrix multiplication) $\text{tmp} = \alpha * A * B$ and $D = \beta * D + \text{tmp} * C$, and L4 distributes `D` back to `array` using another hash table `addrOut`.

A motivating example

Listing 11 shows a motivating example for POLSCA. This example blends typical polyhedral workload (L2 and L3) as in *2mm* from PolyBench/C [330], with common non-SCoP program parts (L1 and L4) as in *AES* from MachSuite [241]. L1 and L4 are not SCoPs due to the existence of non-affine memory accesses to `array`. This example resonates with the two challenges previously mentioned.

First, we need to annotate which parts should be SCoPs. Even though it is trivial for polyhedral experts that L1 and L4 should be disregarded due to non-affine accesses, and only L2 and L3 are SCoPs, HLS engineers can find it difficult. Moreover, we can realize variables like `M` and `N` should be constants, which is a prerequisite that loops here have affine bounds, but there are cases we should leverage static analysis tools to gather such information. We also need to ensure these variables are constants and propagate them into array definitions so that HLS tools can accept this code.

Second, after applying polyhedral transformation on L2 and L3, we can get several tiled nested loops, but the resulting code can be about 10 times longer to handle different combination of conditions on memory sizes. Even these loops are tiled, HLS tools can hardly exploit the parallelisation opportunities since the complicated code hinders accurate dependence analysis. Partitioning the memory into non-intersected parts makes it explicit that each tile has no conflicting accesses with others. HLS

tools provide memory partitioning directives, but they cannot help when a memory size is not divisible by the tiling factor, and there is no automatic procedure to make decisions when there are several valid partitioning tactics and a memory are accessed across SCoPs and non-SCoPs.

We realize that compiler techniques can help address these challenges and overcome the limitations imposed by existing tools. The next section reviews the essential background on polyhedral methods.

4.4.1 Methodology

Pre-polyhedral transformation

POLSCA searches within the user input program for nested loops and discovers those that satisfy the affine constraints—the loop bounds should be affine expressions, and all memory accesses reach affine addresses—imposed by polyhedral methodology. To get valid input for polyhedral tools, POLSCA should annotate certain loop nests as SCoPs that will be transformed later. An annotated SCoP will be in the polyhedral representation that the polyhedral tools can recognize and operate on. This section demonstrates how POLSCA decides which loop nests to annotate, specifically, how to determine the SCoPs when there are multiple affine loop nests.

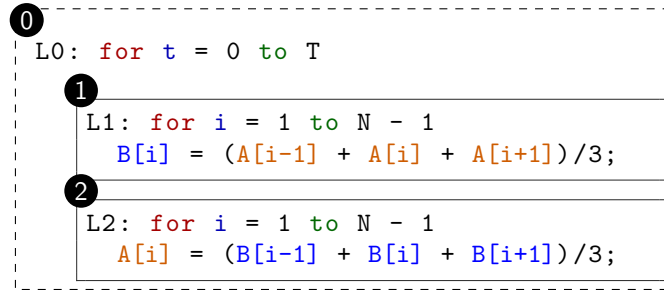


Figure 4.10: Jacobi-1d algorithm. We can either annotate a big SCoP over the outermost loop L0, or two small SCoPs covering inner loops L1 and L2. Memory accesses to arrays $A[N]$ and $B[N]$ make L0 and L1 interdependent.

Hazardous interdependent loops Consecutive loop nests can be covered by a single SCoP, or by multiple separated SCoPs. These two approaches can produce significantly different results if the loop nests are interdependent, i.e., they have hazardous read and write accesses to the same array, at the same address. When provided a large SCoP, polyhedral algorithms intend to schedule interdependent loops closer to improve locality and parallelism. This strategy would be beneficial

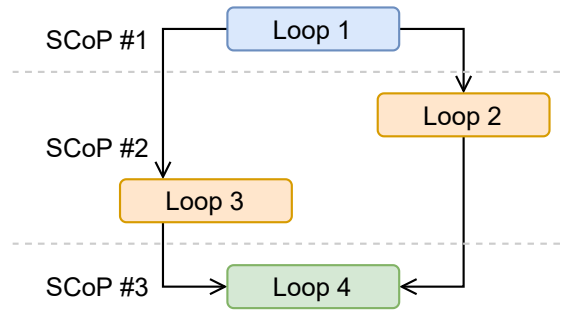


Figure 4.11: Demonstrating the SCoP decomposition algorithm. In this particular example, we have 4 consecutive loops at the same depth, and there are interdependences (arrows) among loop 1 and 2, 1 and 3, 2 and 4, 3 and 4. The minimum cuts required for SCoP covering loop 2 and 3 is 0, which will propagate through Eq. (4.1) and lead to the optimal solution as shown that uses 2 cuts (dashed horizontal lines).

for multi-core CPUs, however, the resulting program can be too complicated to be optimized by HLS tools. For example, a polyhedral optimizer can produce about 30 loops with sophisticated control from the Jacobi-1d code (Fig. 4.10), which originally just have 2 loops that are interdependent on arrays **A** and **B**. This situation also applies to our motivating example (Listing 11), specifically, the two consecutive matrix multiplication loop nests.

SCoP annotation heuristic Instead of modifying the sophisticated internals of Pluto, POLSCA addresses this problem by a heuristic: avoiding SCoPs containing interdependent loops can produce simple and efficient HLS code. This heuristic is based on our observation that SCoPs without interdependent loops will not trigger major loop reconstruction in polyhedral optimizer, and therefore, the optimized code will still close to the simple form of the human-readable user input. The simpler the optimized code is, the more efficient the final design will be, since the backend HLS tools need less effort to resolve statement dependencies, and consequently, better identify parallelisation and pipelining opportunities. Future HLS releases may be capable of handling more complicated code, in which case POLSCA can search for the optimal set of SCoPs that should apply such heuristics.

In Fig. 4.10, annotating two SCoPs on the two inner loops produces a design with just two independently tiled loops, which is sufficient to achieve much higher parallelism than the original code. Moreover, only separately annotating the two inner affine loop nests in Listing 11 produces two tiled matrix multiplication subroutines, which HLS tools are familiar with.

SCoP decomposition To apply this heuristic, we should decompose SCoPs having interdependent internal loops into pieces such that their internal loops only have external dependencies. The simplest solution is to annotate a SCoP per loop, which is suboptimal since loops that are not interdependent cannot be transformed together, e.g., we cannot further simplify loops accessing different arrays through fusion.

We prefer a top-down solution: starting from a big SCoP containing all consecutive affine loop nests, POLSCA tries to find *the minimum number of cuts* on all inter-loop dependencies to remove them. Adapted from graph theory, we define a cut on the inter-loop dependency graph of a single SCoP, in which vertices are loop nests within the SCoP and edges are their dependencies. Each cut decompose a SCoP into two parts, and aiming at minimizing the number of cuts avoid creating excessively small SCoPs that hinder optimization.

This thesis considers the minimum cut problem on a graph consisting of consecutive loop nests at the same depth. Suppose there are N loop nests, we can assign an integer label to each loop following their execution order, e.g., 0 to the first loop nest and $N - 1$ to the last, and any partition of this graph can be represented as an interval. For instance, an interval $[S, E)$ covers the loop nests with labels from S to $E - 1$. Moreover, we can formulate a cut as a partition of intervals.

$$\min_{\text{cut}}[S, E) \leftarrow \mathbf{minimum}_{i \in [S+1, E)} \left\{ \min_{\text{cut}}[S, i) + \min_{\text{cut}}[i, E) + 1 \right\} \quad (4.1)$$

Following the discussion above, we can formulate this minimum cut problem as Eq. (4.1). For a range of loops labeled from S to $E - 1$, the minimum number of cuts ($\min_{\text{cut}}$) equals to the minimum sum of the solutions for two sub-ranges cut at one particular position i . The base case is those ranges without interdependent loops, which requires 0 minimum cut. This equation can lead to a straightforward dynamic programming algorithm with polynomial time complexity. One example solution found by this algorithm is shown in Fig. 4.11.

Fusion options in polyhedral schedulers Our SCoP decomposition algorithm is functionally similar to setting certain loop fusion options in the polyhedral scheduler, for instance, Pluto [30] provides `--nofuse`, `--smartfuse`, and `--maxfuse` options, which decide whether fusion should happen across Strongly-Connected Components (SCCs) in the dependence graph. Our proposed solution works at a higher level than making fusion decisions on SCCs within each SCoP: we instead decide how to

```

// Tile loops
for i = 0 to 2
  for j = 0 to 3
    for k = 0 to 4
      // Point loops
      for ti = i * 32 to (i + 1) * 32
        for tj = j * 32 to (j + 1) * 32
          for tk = k * 32 to (k + 1) * 32
            C[ti][tj] += A[ti][tk] * B[tk][tj];

                                ↓↓ outlining PEs

for i = 0 to 2
  for j = ...
    for k = ...
      PE(i, j, k, A, B, C);

```

Figure 4.12: Sketch of polyhedral-transformed matrix multiplication. The loop nest initially has a depth of 3, and each loop has been tiled by a factor of 32 as listed here. The 3 outer loops *i*, *j*, *k* are referred to as tile loops, and the inner loops *ti*, *tj*, *tk* are point loops. After PE outlining, the inner 3 point loops are wrapped into a new function *PE*.

decompose SCoPs, and which each SCoPs, we give the polyhedral scheduler the freedom to decide whether to fuse or not among SCCs.

Post-polyhedral transformations

This section introduces transformations applied after receiving the optimized code generated from the polyhedral tools, which are not optimal to describe HLS designs due to the lack of directives and their undesirable code structure. Directives help HLS tools find better designs based on user-provided domain knowledge, and well-structured code can ensure HLS tools exploring all dependencies among instructions.

This section mainly describes how POLSCA makes memory partitioning opportunities, which are critical to the achievable level of parallelism, visible to HLS tools. We also demonstrate additional loop transformations that set the boundary between software and hardware. The expected final production is an HLS design that clearly indicates parts that are hardware processing elements (PEs) with properly partitioned memory interface and HLS directives.

Point loops as processing elements A tiled loop nests normally have two separable parts, *tile loops* that iterate over tiles, and *point loops* that iterate within each tile (Fig. 4.12). Heuristically, it is desirable to unroll the tile loops so that the HLS tools can have a better chance to utilize the parallelism that tiling implies. Point

loops are less common for unrolling, and we normally leverage pipelining to improve performance.

POLSCA draws a boundary between tile and point loops by *outlining* point loops into functions (Fig. 4.12). Replacing nested loops with a single caller can help HLS tools realize there is no dependence among tiles and find parallelisation opportunities.

Memory partitioning HLS tools can be too conservative to parallelize PE instances across tiles, mainly because each PE accesses to the same memory object, which does not specify whether they access non-intersected partitions. For instance, in Fig. 4.12, PEs of different i, j values access to different 32×32 partition of array C , but it is not explicit.

A common approach to address this problem is memory partitioning, i.e., breaking a single memory block into smaller parts that are accessed by different PE instances. Here we assume only using the block partition scheme [299]. There are corresponding optimization directives in HLS tools that can transform the input code, e.g., turn $A[64]$ into $A[2][32]$ if partitioned by blocks with a factor of 32, to apply memory partitioning. But it is still challenging for users to identify which memory blocks should be partitioned.

Concretely, there are two major challenges:

1. A single PE can access multiple partitions, e.g., the statement ⁵ in Jacobi-1d (Fig. 4.10) accesses two neighbouring tiles on either boundary ($i = 0$ or 31 supposing the factor is 32).
2. We can decide the partitioning factor by the size of the iteration domain of the memory access, e.g., if i iterates 32 times, we partition $A[i]$ by a factor of 32. But the same array can be accessed in different loop nests of various domain sizes, and we should reconcile them to one value. These dependencies should be explicit to HLS.

POLSCA provides two corresponding solutions, one to analyze how many partitions that each PE instance needs to access given a block partition factor, and the other one reconcile a single partition factor size for each array.

⁵ $A[i-1] + A[i] + A[i+1]$

```

1 void top(float A[70], int B[70])
2   for i = 0 to 70 { A[B[i]] = ... } // Non-SCoP
3   for i = 0 to 3  { PE(i, A) }      // SCoP
4 void PE(int i, float A[70])
5   for ti = i * 32 to (i + 1) * 32
6     A[ti] = A[ti] + A[ti-1]

```

- (a) Before memory partitioning. Array *A* is accessed by both non-SCoP and SCoP loops. The tile factor in the polyhedral-transformed SCoP is 32. Special handling for invalid access *A*[-1] is omitted.

```

1 void top(float A[3][32], int B[70])
2   for i = 0 to 70 { A[B[i] / 32][B[i] % 32] = ... }
3   for i = 0 to 3  { PE(i, A[i], A[i-1]) }
4 void PE(int i, float P[32], float Q[32])
5   for ti = 0 to 32
6     P[ti] = P[ti] + (ti == 0 ? Q[(ti-1) % 32] :
7                       P[(ti-1) % 32])

```

- (b) After memory partitioning. *A* is partitioned by a factor of 32 after reconciling both accesses. Dynamic addresses from *B* are transformed to access the partitioned array, and the PE interface is extended to pass in an extra partition for the boundary accesses.

Figure 4.13: A memory partitioning example that applies partition factor reconciliation and PE interface extension.

Affine analysis on accessed partitions

Figuring out the number of partitions that a single PE instance needs to access is crucial to understanding the profitability by using a specific block partition factor.

Definition 4.1 (Profitable partition factor). A partition factor *T* is more profitable than another *T'* if it can get a PE instance that accesses fewer partitions, or has the same partitions but exposes higher level of parallelism, denoted by $T' \prec T$.

For instance, partitioning *A* in Fig. 4.12 by 32 is more profitable than 33, which requires some PE instances to access more than 1 partition; and 64, the PE instance of which still accesses 1 partition but reduces the potential level of parallelism by half.

As suggested previously, PEs only have point loops resulted from polyhedral transformations, which are SCoPs and satisfy the prerequisites for affine analysis. We limit the access pattern to *scalar* address in the following discussion, while these results can be extended to general cases.

Problem 4.1 (Determine partitions accessed differently). For any two addresses *f*(*i*) and *g*(*i*) being accessed, where *i* is an induction variable, *f* and *g* are affine transformations from *i* to addresses, e.g., $f(i) = ai + b$, they access different partitions if there exists solution for the following:

$$\lfloor f(i)/B \rfloor \neq \lfloor g(i)/B \rfloor, \quad Tk \leq i < Tk + T,$$

in which T is the tile size for the point loop, k is an induction variable of an outer loop that encloses the loop that i belongs, and B is the block partition factor.

By solving Problem 4.1 among all memory accesses for a block partition factor B , we can get the number of partitions that a PE will access, which consequently determines the profitability (Definition 4.1) of B .

Even though Problem 4.1 is solvable by a linear solver, the *candidate set* of B is as large as the target memory size, e.g., B used to partition a 1-D memory of size 1,024 can take at most the value of 1,024. Repeatedly running a linear solver for all B candidates can be computationally intractable. Therefore, we propose the following theorem that directly relates partition factor B and a predetermined parameter, tile size T .

Theorem 4.1. Given block partition factor B equals to a predetermined tile size T , and i has an iteration domain $[Tk, Tk+T)$ where k is an outer loop induction variable, then the partitions accessed for a set of addresses, $\{f_j(i) = a_j i + b_j, j = 1, \dots, M\}$, are included in the following set:

$$\bigcup_{j=1}^M \left[a_j k + \left\lfloor \frac{b_j}{T} \right\rfloor, a_j k + 1 + \left\lfloor \frac{b_j - 1}{T} \right\rfloor \right].$$

Proof. First of all, $\lfloor ax+b/x \rfloor = a + \lfloor b/x \rfloor$. Given that i has a iteration domain $[Tk, Tk + T)$, for each address f_j , we can apply this equation to get the lower and upper bound of $f_j(i)$. Specifically,

- its lower bound is $\lfloor f_j(Tk)/T \rfloor$ when $i = Tk$;
- and its upper bound is $\lfloor f_j(Tk+T)/T \rfloor$ when $i = Tk + T$.

The result can be achieved by taking the union of the lower and upper bounds from all addresses. □

Example 4.1. For statement $A[i-1] + A[i] + A[i+1]$, we have:

$$(a_1, b_1) = (1, -1) \quad (a_2, b_2) = (1, 0) \quad (a_3, b_3) = (1, 1).$$

Using Theorem 4.1, the union of all of their partitions is:

$$\{k-1, k\} \cup \{k\} \cup \{k, k+1\} = \{k-1, k, k+1\}.$$

For tile k , there are three neighbouring partitions accessed.

We leverage Theorem 4.1 to determine the optimal partition factor in the following sections.

Partition factor reconciliation

The optimal partition factor for a memory block is mainly determined by its accesses and their parent loop nests. Given that the amount of valid candidates for a partition factor can be enormous, and finding a global optimum is computationally intractable, we choose to search for a local optimum using an algorithm derived from the following propositions.

Theorem 4.2. If a dimension of an array is accessed by an induction variable i , which has an affine bound of $[Tk, Tk + T)$ where T is a predetermined tile size and k is an outer loop induction variable, then it is more profitable to have a block partition factor B that equals to T than other B candidates.

Proof. Let us consider four possibilities for the value of B : B equals to T ; B is an integer multiples of T , denoted by nT ; B is a factor of T , denoted by T/n ; and none of the above. Here, n is an integer that is larger than 1.

1. If $B = T$, there is a single partition that will be accessed by a PE instance;
2. If $B = nT$ and $n > 1$, there is also a single partition that will be accessed, but within each partition, there are n number of sub-partitions of size T that can potentially be parallelised, which makes nT less profitable than T .
3. If $B = T/n$ and $n > 1$, a PE instance needs to access n partitions, which induces T/n is less profitable than T .
4. When partitioning by any of the rest candidates, which are not an integer multiple or a factor of T , a PE instance must access at least two partitions, which makes it less profitable than T .

Therefore, we conclude that $B = T$ is the most profitable block partition factor under the given presumptions. □

Theorem 4.2 helps determining the partition factor for each access of a memory block. And if it is satisfied, based on Theorem 4.1, we can compute the total number of partitions accessed. In case the determined values are not identical, for every pair of access A and block partition factor B , suppose $N(A, B)$ is the number of accessed partitions per PE instance, and $P(A, B)$ is the achievable level of parallelism, we introduce Definition 4.2 for reconciliation.

Definition 4.2 (Profitability among multiple accesses). Given memory accesses $A_{1\dots n}$, each of which is an address that is an affine mapping from loop induction variables to a scalar, and block partition factors $B_{1\dots n}$, B_p is (local-)optimally profitable if for any other B_q :

- B_p has less number of partitions, i.e., $\sum_i N(A_i, B_p) < \sum_i N(A_i, B_q)$, or
- B_p can provide higher level of parallelism when the number of partitions is the same, i.e., $\sum_i P(A_i, B_p) > \sum_i P(A_i, B_q) \wedge \sum_i N(A_i, B_p) = \sum_i N(A_i, B_q)$.

Notably, we use the properties aggregated by summation over all accesses, which implies that we treat every access of equal importance. This approach is potentially not global-optimal if some PEs have much longer latency than the others, and in those cases, we should properly weight each term before aggregation. These particular cases make no appearance in the benchmark examples that this thesis covers, and therefore, we decide to stick to the local-optimal solution.

Finding the most profitable factor under Definition 4.2 for each memory has computationally tractable time complexity of $\mathcal{O}(n^2)$, imposed by calculating $N(\cdot, \cdot)$ and $P(\cdot, \cdot)$. Suppose there are m memory blocks in the input program, the overall time complexity for the partition factor reconciliation algorithm becomes $\mathcal{O}(mn^2)$. After that, we can transform the input code to accommodate the partition factor, e.g., $A[i]$ to $A[i \text{ floordiv } 32][i \bmod 32]$ if partitioned by a factor of 32.

4.4.2 Evaluation

Implementation details

POLSCA is built upon MLIR [166] (Section 2.4). The polyhedral representation and affine analysis are implemented upon the Affine dialect, a collection of operations and representations that are dedicated to polyhedral modelling. The input code that POLSCA accepts is in C/C++, which are translated into MLIR through

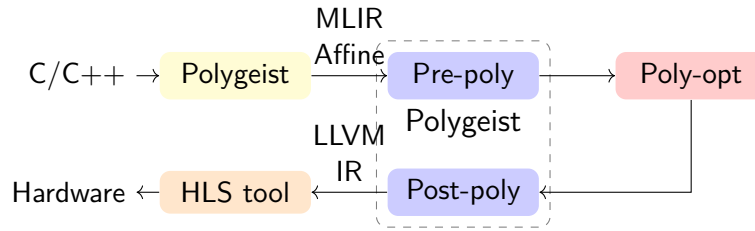


Figure 4.14: The overall workflow of POLSCA. Our contributions are highlighted in the dashed region.

Polygeist [214] (also in Section 4.2). The internal polyhedral optimizer is Pluto [30], and POLSCA translates the polyhedral representation between MLIR and Pluto. The final outcome from POLSCA is in LLVM [165] IR that can be consumed by backend HLS tools. HLS directives are specified in the LLVM IR as well, following the format that the target HLS tool understands. Fig. 4.14 shows how they interconnect.

We experiment POLSCA on Xilinx Vitis HLS (2020.2) and leverage its LLVM frontend following open-sourced instructions⁶. We setup an ideal scenario that the FPGA board is large enough (Zynq UltraScale+ XQZU29DR), since this thesis mainly focuses on pushing the performance boundary by polyhedral transformations without considering design space exploration. We use co-simulation to collect precise cycle numbers of latency and verify the correctness of the generated design. As the top module we synthesize only contains memory interfaces and no memory instances are instantiated, there is no BRAM or URAM usage introduced by our designs.

The baseline of our experiment is using POLSCA without any polyhedral transformation, i.e., the input C/C++ will be translated to MLIR through Polygeist first, then translated to LLVM IR immediately without any optimization, and finally passed to Vitis HLS. By design, there is no optimisations towards throughput carried out by the HLS backend for those baselines, specifically, there is no loop unrolling that would have fundamental impact on the design throughput. The rationale behind this choice is explained at the end of Section 4.4.2.

Polybench/C

POLSCA is evaluated on Polybench/C [330], one of the most popular benchmark suites for high-performance computing that only contains code that is fully polyhedral-optimizable, i.e., their core functions are all affine loop nests. While POLSCA can

⁶<https://github.com/Xilinx/HLS/tree/2020.2>

Table 4.3: Evaluation of our approach on a set of benchmarks. **NA** stands for whether an example contains non-affine loop nests or not. The *base* column covers the designs directly synthesised using Polygeist, the *poly* column covers the designs directly synthesised using the MLIR from polyhedral, and *ours* covers the designs synthesised using the code from Polygeist. **Geom. Mean** measures the geometric mean of values from the baseline over other corresponding scenarios.

Benchmarks	NA	LUTs			DSPs			Cycles (k)			Fmax (MHz)			Wall clock time (μ s)		
		base	poly	ours	base	poly	ours	base	poly	ours	base	poly	ours	base	poly	ours
bicg	✗	1,525	3,259	17,144	15	18	144	960	1,625	580	103	60	60	9,318	27,077	9,661
floyd-warshall	✗	641	1,439	5,326	1	0	24	250,000	250,024	203,192	106	109	103	2,358,490	2,293,798	1,972,737
gemm	✗	1,895	4,116	13,720	26	29	191	10,610	63,580	6,009	103	102	103	103,009	623,335	58,344
gesummv	✗	1,584	3,418	18,610	15	27	137	375	563	132	103	60	60	3,641	9,388	2,191
jacobi-1d	✗	1,463	5,215	7,321	14	14	56	163	1,082	123	103	100	103	1,581	10,818	1,197
jacobi-2d	✗	3,015	13,955	18,169	17	28	150	36,908	19,244	14,297	99	96	103	372,807	3,229,785	138,801
mv	✗	1,635	1,943	8,010	14	16	56	1,290	669	698	60	60	60	21,506	11,146	11,641
radix sort	✓	3,827	2,149	21,817	0	0	0	197	229	148	78	78	76	1,407	1,633	1,025
Geom. Mean		1×	0.5×	0.1×	1×	0.8×	0.2×	1×	0.5×	1.7×	1×	1.2×	1.1×	1×	0.4×	1.5×

process all Polybench examples, we select 7 typical examples covering all categories, including linear algebra (gemm, gesummv, bicg, mvt), stencil (jacobi-1d, jacobi-2d), and medley (floyd-warshall). Even though POLSCA is able to use arbitrary tile sizes, the HLS backend cannot synthesize array partitioned with a large (> 20) factor, therefore, we manually select a set of tile sizes for each example, which can be automated in the future.

Table 4.3 summarizes our results. POLSCA can achieve 1.5 times speed-up averaged by geomean over the baseline directly applying HLS tools on C source code. More importantly, without POLSCA, applying polyhedral optimization directly results in worse performance than the baseline. In general, the performance benefit gained from POLSCA over plain polyhedral designs is mainly from:

1. heuristic SCoP annotation that can avoid generating over-complicated polyhedral-transformed code, especially for stencil examples like jacobi-1d and jacobi-2d, which will result in at least 10 times longer code after polyhedral optimization without conventionally recognizable tiling structure as in Fig. 4.12;
2. memory partitioning that practically generates parallelisable designs, for instance, gemm being polyhedral-transformed into a blocked version is a typical HLS input that can be paralised, however, due to the lack of directives and code transformation, the performance (63,580k cycles) is much slower than the original (10,610k cycles).

Among these examples, linear algebra and stencil applications can be optimized to a level (about 2 times) better than the others. There are two examples that POLSCA does not perform well: bicg and floyd-warshall, which can attribute to the default behaviour of the polyhedral optimizer Pluto.

Although POLSCA optimized bicg has worse wall clock time than the baseline, it is mainly due to the fact that Fmax is reduced more (about 1.71 times) than the level of reduction in the number of cycles (about 1.65 times). This suggests that the improvement in parallelism leads to increased design complexity, which makes the HLS tools less effective in enhancing performance. The major source of complexity is the additional loops generated by the polyhedral optimizer originally targeting better CPU cache locality.

Many polyhedral optimizers such as Pluto are not exceptionally good at optimizing the Floyd-Warshall algorithm [214]. The resulting program can be tiled and partitioned, but the dependencies among the tiles are hard to exploit for enhancing parallelism, so performance improvement is minor.

Resource usage

We notice that the usage of resources is scaled at a higher rate than performance gain, which is mainly due to the general-purpose nature of the Pluto polyhedral optimizer. Pluto optimizes for less communication among tiles, which does not explicitly lead to solutions that are more resource-efficient, e.g., using local buffers to reduce control and save resources.

Also, the memory partition expression we introduce, which involves floordiv and modulo arithmetic, makes it more difficult for Vitis HLS to find expressions like MAC (multiply and accumulate), so leads to higher resource usage. For instance, regarding the core arithmetic statements in gemm, the DSP usage scales from 25 to 103 (4.1 times), while LUT usage increases from 1,197 to 9,733 (8.1 times), when comparing the original design with the POLSCA-optimized one. Our future work will include additional optimizations to reduce resource.

Partially polyhedral-optimizable code

We evaluate POLSCA on a partially polyhedral-optimizable example selected from MachSuite [241], radix sort. Radix sort contains affine loop nests, as well as non-affine ones that have indirect memory accesses, i.e., accessing by addresses looked up from another memory like `A[B[i]]`.

POLSCA can not just recognize affine loop nests and annotate them as SCoPs, but also perform memory partition based on a factor reconciled from all affine and non-affine accesses. We can gain about 37% speed-up over the baseline using POLSCA. We also run just polyhedral optimization on the affine parts of the code without any POLSCA transformations, and the result becomes about 16% worse due to the lack of memory partitioning and the increased code complexity. It is worth mentioning that all these transformations are included in POLSCA, and therefore, easy for application builders to adopt.

Choice of baseline

The baseline of our approach is obtained from applying HLS to the output of Polygeist (Fig. 4.14). Another possible baseline is applying HLS directly on the C/C++ design without going through Polygeist. The reason for our choice is to exclude the benefits of Polygeist in our evaluation, and to minimize the effects due to

the disparity between Clang, the frontend compiler that Vitis HLS uses, and MLIR, which POLSCA uses.

Moreover, we notice that, for the examples that we evaluate, the geomean of LUT and DSP usage from Clang baselines are 1.25 and 1.20 times more than the MLIR baselines, while the geomean of the wall clock time stays the same between both baselines. This suggests that the MLIR baselines are even stronger than Clang baselines, which can support our choice.

Finally, the fundamental workflow of our choice of baseline only marginally diverges from the traditional application of HLS tools. The key differentiation lies in the process of converting C/C++ into LLVM-IR as opposed to the core HLS optimization passes which happens after the creation of LLVM-IR. This implies that our evaluation emphasises the differences specifically at the level of code transformation, the unique and novel part of our approach.

4.4.3 Discussions

This section presents POLSCA, a novel compilation framework that optimises the workflow of applying polyhedral transformations on HLS designs. POLSCA is designed to take arbitrary C/C++ code as input, and produce the corresponding HLS code that are optimised for better consumption by the backend HLS tools to generate high-performance hardware. POLSCA has two major passes, one that happens before polyhedral transformations can auto identify and decompose SCoPs from input HLS code, the other one takes the code generated from the polyhedral scheduler and turns it into a form with PE functions and memory partitions. These pre- and post-polyhedral passes together produce optimised polyhedral HLS designs with simplicity and explicit dependencies.

Our research, while pioneering on the front of polyhedral toolchain, has various limitations. One major limitation is the omission of support for systolic array architectures in our framework. This signifies that when dealing with programs that exhibit regular data-flow patterns, the designs produced by POLSCA may not be able to harness the system’s full memory throughput potential. Taking into consideration the state-of-the-art accelerator design for matrix multiplication in the Tensor Processing Unit (TPU) as delineated by [136], it becomes apparent that their model leverages the efficiency of the systolic array. This design paradigm is considered optimal based on the roofline model of performance analysis, as illustrated in [136, Figure 5]. In contrast, our design, as detailed in Table 4.3, utilizes a row-major, blocked matrix multiplication algorithm, generated automatically by

Pluto [30]. Unfortunately, this configuration results in a memory efficiency that is suboptimal when compared to the wavefront data propagation pattern found in a systolic array. Future work should investigate the integration of systolic arrays into our design framework to enhance memory efficiency.

In the context of our evaluation, it is essential to note that this chapter does not embark on a comparison with methodologies necessitating an in-depth understanding of HLS optimisation capabilities. This can be exemplified by the Polybench case studies, where the meticulous selection and combination of HLS pragmas could exploit system resources to their maximum potential. It is plausible that such an approach would yield more efficient designs than those presented in this chapter: we only use a basic set of pragmas that are reused across all examples, and our focus is on transforming the given code even before the pragma insertion happens. However, this deviation turns away from the central objective of this research, which seeks to lessen the dependency of non-HLS experts on manual optimisation. Instead, we aim to enhance their design through automated polyhedral transformations.

4.5 Summary

This chapter we present three tools for polyhedral compilation: Polygeist, Polymer, and POLSCA:

- Polygeist is a framework that aims to connect MLIR to C/C++, so that existing C/C++ code can benefit from the modularised and reusable MLIR transformations. It addresses the challenges that lie in the semantic differences between MLIR and C/C++, in the target of raising lower-level IRs into the higher-level Affine dialect, and in the postprocessing step that creates reductions and parallelised operations from polyhedral transformed code. Empirically, code compiled by Polygeist performs almost the same as those compiled by Clang for the examples in Polybench/C, which indicates the usability of the Polygeist framework.
- Polymer works together with Polygeist, which provides the core functionality that translates between MLIR Affine and external polyhedral tools, so that MLIR code can be transformed by existing, well-developed polyhedral scheduling algorithms. Polymer tackles the problems around mapping MLIR instructions to polyhedral statements, which are the elements that a polyhedral scheduler operates on. Polymer provides an algorithm that implements the statement mapping, which carefully works with the dependences among instructions, and a heuristic that splits a single polyhedral statement into multiple ones, which is empirically beneficial. Compared with Pluto and Polly on Polybench/C, Polygeist supported by Polymer can achieve $2.34\times$ times and $1.41\times$ times speed-up over Pluto and Polly, respectively, when compiled for sequential execution; and $7.54\times$ times and $3.26\times$ times when in parallel.
- POLSCA further extends our application domain into high-level synthesis. POLSCA targets the problem that existing polyhedral tools do not support high-level synthesis well and provides a solution based on compiler transformations. Around an existing polyhedral scheduler, POLSCA can transform the input C/C++ code into MLIR using Polygeist and identifies and decomposes SCoPs so that the polyhedral scheduler can generate optimised while simple code; POLSCA can also memory partition the polyhedral-transformed code to produce a hardware friendly result. With POLSCA, we can get around 1.5 speed-up of the hardware designs generated from Polybench/C over an existing C-based HLS tool.

Chapter 5

Dataflow DNN accelerator

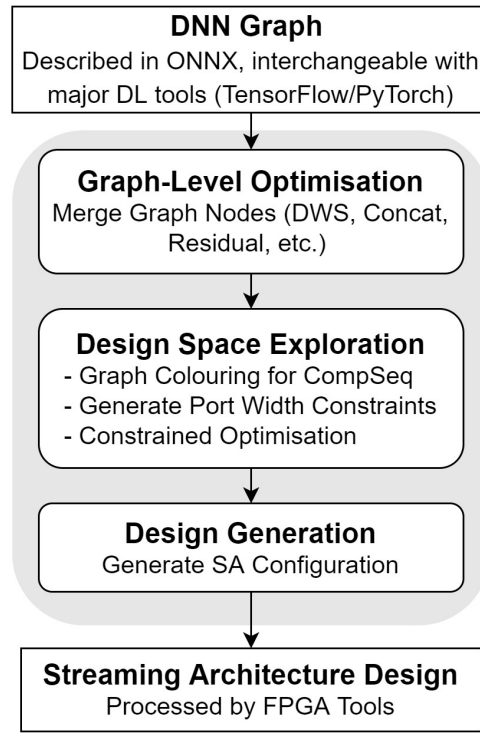
5.1 Introduction

Streaming architectures (SAs) can effectively leverage the benefits of reconfigurable computing: by mapping the most time-consuming parts of an application to a pipelined, control-less hardware design, we can get an efficient accelerator that utilises most of the clock cycles and hardware resources for actual computation, instead of wasting them on control logic. A reconfigurable device, i.e., Field-Programmable Gate Array (FPGA), equipped with SAs can therefore have higher efficiency than its counterparts like CPU and Graphic Processing Unit (GPU), which have to spare a considerable amount of control logic for generic programs. Accelerating Deep Neural Networks (DNNs) presents a promising scenario for SA: the clock cycles of each layer are statically known during synthesis, from which we can customise Processing Engines (PEs) to avoid unexpected stalls and scale throughput up to the whole model; since SAs are designed to store intermediate results on-chip, off-chip memory bandwidth is less likely a bottleneck, and power consumption is reduced due to less off-chip data movement. This chapter aims to provide a tool that can better leverage streaming architectures for DNN acceleration.

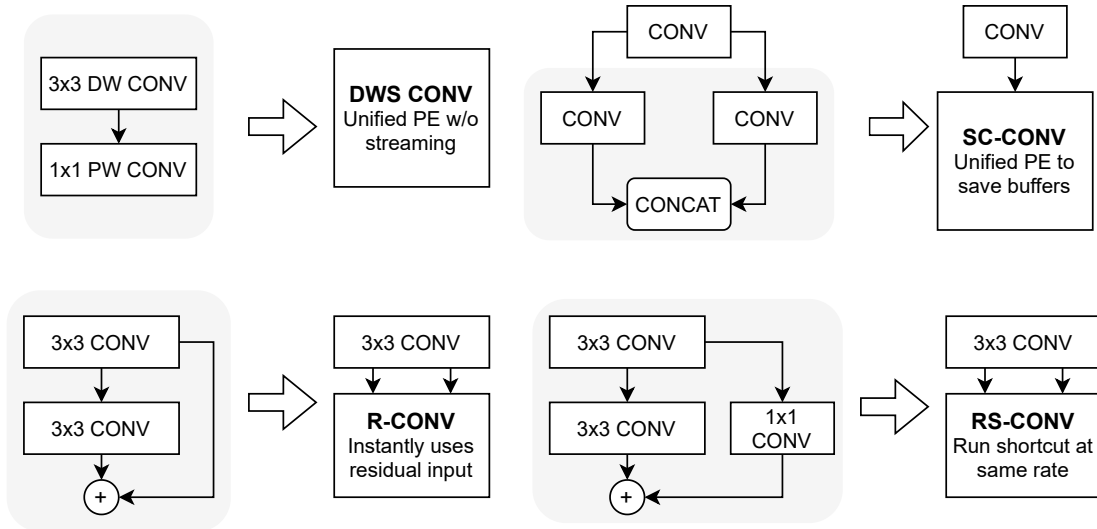
Recent FPGA boards are sufficiently large¹ to accommodate the parameters and intermediate results of DNN models optimised for rapid inference, including MobileNets [118, 251] and SqueezeNet [125]. However, optimally mapping a DNN to a SA on an FPGA presents notable challenges:

1. The performance of inter-layer pipelining and buffer usage is significantly in-

¹For instance, a Xilinx VU9P provides around 9.5 MB of on-chip memory, which exceeds the requirements of models such as MobileNets and SqueezeNet.



(a) The overall workflow of Deacon. Deacon takes a DNN model described in a graph format as input, and processes it in three major steps.



(b) Graph-level optimisation on efficient convolution topologies. Four different convolution blocks can be transformed for more efficient streaming architecture deployment, including depthwise-separable convolution (DWS CONV), split-concat convolution (SC-CONV), residual convolution (R-CONV), and residual convolution with shortcut projection (RS-CONV). The block arrows indicate transformation. Each rectangle indicates a single layer, and the shaded rounded rectangles depict which layers from the original network will be transformed into a new one. Detailed explanations are presented in Section 5.3.

Figure 5.1: Key concepts in Deacon: the Deacon framework is listed in the left, and the core graph-level optimisations are demonstrated in the right.

fluenced by the interaction between *computation sequences* of adjacent layers.

2. Contemporary DNNs incorporate novel topologies with heterogeneous convolution layers, such as depthwise-separable [118, 251] and residual [110] convolution layers, making it complex to balance pipeline stages.
3. The necessity of customising the PE for each DNN layer complicates the design space exploration required to find an optimised solution.

This thesis is committed to addressing these three challenges, each a pivotal aspect of optimizing DNN implementation within the streaming architecture paradigm on FPGA boards. An important point of clarification is that these challenges do not arise from the fundamental design of DNN accelerators in general. For instance, the deployment of ResNets on a device with a single, unified convolution processing unit [101] will not face the same challenges as ours. Instead, they pertain more to the unique situation addressed in this thesis: the mapping of DNN models onto streaming architectures. As will be demonstrated in subsequent chapters, the benefits from using the SA approach justify the efforts taken to overcome these challenges.

To address these challenges, we propose Deacon (DEep Accelerator for CONvolution), a groundbreaking framework that efficiently maps an entire DNN to an SA. We present the following *advanced pipelining* techniques for the first time:

1. We exploit the computation sequences of adjacent Processing Elements (PEs) in an interleaved manner to facilitate fine-grained inter-layer pipelining (Section 5.2).
2. We carry out graph-level optimisations to enhance *resource efficiency*—the throughput per arithmetic resource usage—of pipelined heterogeneous convolution layers, including depthwise separable and split-concatenate convolutions. Furthermore, we enhance data locality for distant residual connections to circumvent stalls or the requirement of large buffers (Section 5.3).
3. We efficiently navigate the vast design space related to the proposed pipelining techniques by leveraging graph colouring for search space pruning, and resort to simulated annealing for optimisation (Section 5.4).

Given our principal intent to generate streaming architectures, primarily due to their power efficiency as discussed in the initial paragraph, our central metric of interest is power efficiency. This is defined as the throughput per power usage (measured in

watts). Concurrently, we also scrutinize resource efficiency, quantified as throughput per DSP, a metric often evaluated in related works. Experimental results in Section 5.5 demonstrate that, in comparison with preceding designs on MobileNet v1, v2, and SqueezeNet, designs incorporating Deacon’s advanced pipelining techniques display a resource/power efficiency improvement of 2.0x/1.7x, 1.2x/3.5x, and 1.0x/2.8x in geomean, respectively. Additionally, Deacon is capable of deploying and optimising large DNNs with residual connections, such as ResNet-18, attaining a 2.7x/35.6x resource/power efficiency improvement over the baseline. Simultaneously, when compared to prior non-streaming designs, Deacon designs require significantly lower memory bandwidth, enabling them to scale the throughput up to a level restricted solely by arithmetic resources. These improvements come from all the novel advanced pipelining techniques listed above, and we provide ablation study on how each point makes its contribution (Figs. 5.5 and 5.12) As Deacon refrains from altering the input DNN models, we can enhance the processing efficiency of a DNN model using a Deacon design while preserving the model’s accuracy.

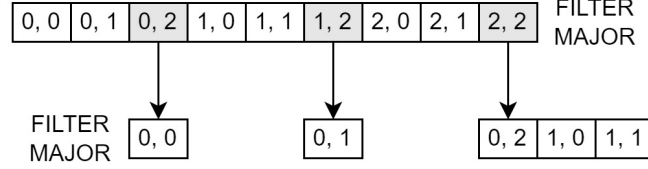
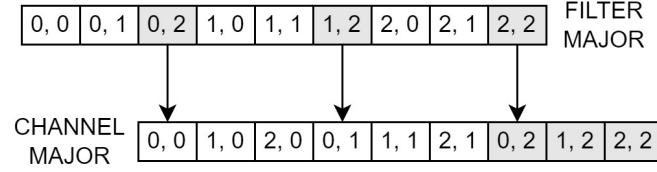
The overall workflow of Deacon is presented as Fig. 5.1a, of which the codebase² is publicly available. Deacon takes three main steps: transforming the DNN graphs, exploring the design space, and generating designs. All these steps are implemented in the Python programming language. Deacon produces designs described in MaxJ, a high-level data-flow hardware description language. We can synthesise such designs by MaxCompiler [202] and deploy the final bitstream on FPGAs.

The initial version of the framework is published in [339]. The Deacon framework is a rework based on that version, with extra improvement of pipelining. This chapter also strips the content about transfer learning that is presented in the original paper.

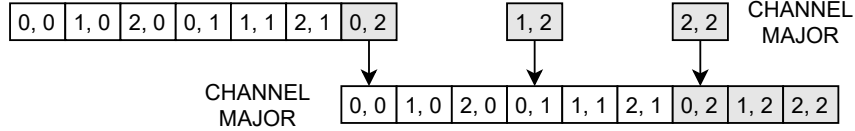
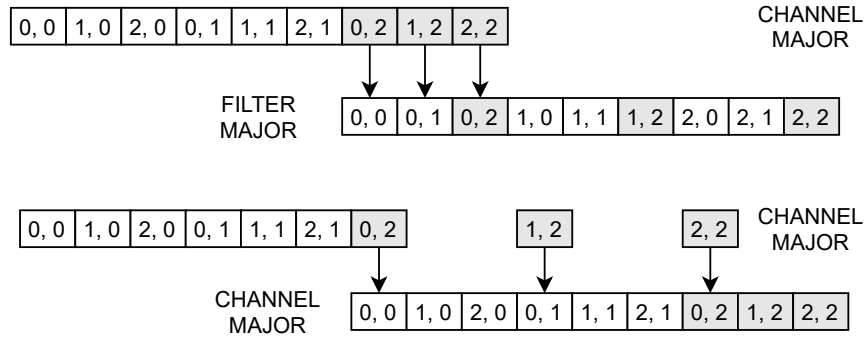
5.2 Streaming Architecture Design

SAs generated by Deacon are Processing Engines (PEs) interconnected by streams. We define PE as a design component that contains arithmetic units and on-chip scratchpad memory. We orchestrate PEs to run *synchronously*, i.e., they share the same state synchronised by a single clock signal. The interfaces of a PE are streams, i.e., FIFO blocks, which allow *asynchronous* processing among PEs. A PE stays idle if its input FIFOs are empty or its output FIFOs are congested. Based on this definition, we do not restrict that one PE should handle one layer; instead, if multiple layers can run synchronously, we can put them together into a single PE.

²Codebase location: <https://github.com/kumasento/deacon>.



(a) Pipelining scenario started with filter-major layers.



(b) Pipelining scenario started with channel-major layers. The first channel-major layer produces the result consecutively, which demands the next layer to consume them immediately (top); otherwise, it will stall (bottom).

Figure 5.2: These two figures illustrate how computation sequences affect pipelining performance. Each rectangle indicates a channel to process in each layer, and its text indicates the output and the input channels (0-indexed). The shaded rectangles show when the corresponding output channel is ready, and the arrows indicate the producer-consumer dependency. For instance, the rectangle with (0, 2) means the PE is processing the third channel of the input image for the first output channel. When channel-major follows filter-major (top-left), there is no pipeline stall. Whenever the 2nd layer needs an input, it will be ready from the 1st layer. If two filter-major layers are adjacent, the 2nd layer will be stalled to wait for the 1st layer to produce output.

For instance, under scenarios in Section 5.3, unifying multiple layers into one PE can reduce inter-PE FIFO resources. In this chapter, we assume that an SA that Deacon generate are in the form of a Directed Acyclic Graph (DAG), which ensures there is no deadlock. Fig. 5.3 demonstrates a typical SA.

Deacon generates parameterised designs. Table 5.1 lists the key design parameters. Each PE has its design parameters. We derive H, W, C, F, K, P, and S from the layer or layers that the PE associates. They are fixed during design space exploration Section 5.4. Only parallelism parameters (P_C and P_F) and computation

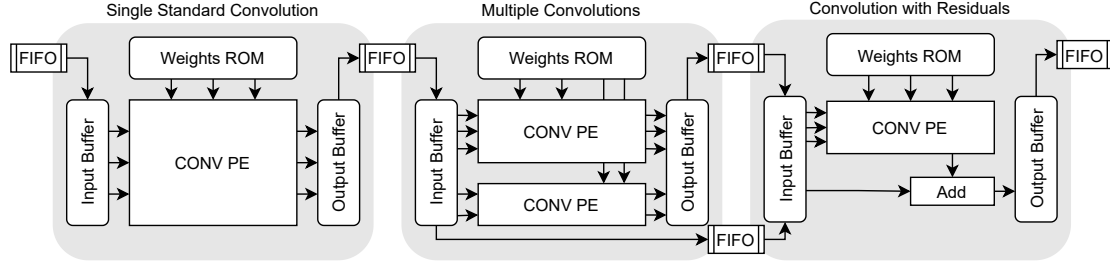


Figure 5.3: SA for a DNN. There are three types of convolution PEs: the left processes a single convolution layer, the middle one can process two layers that can be unified (e.g., depthwise separable convolution), and the right one demonstrates the layer with a residual connection.

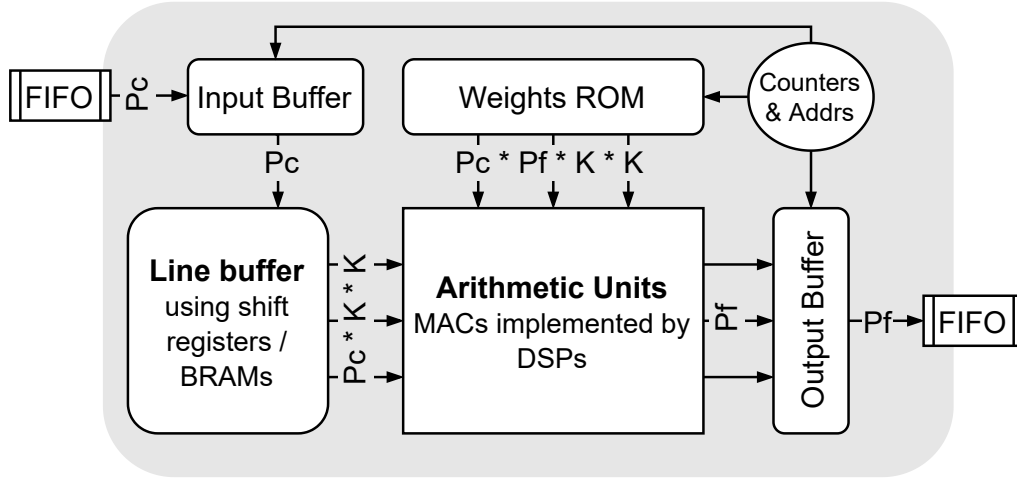


Figure 5.4: A base convolution layer design. The definition of the symbols are listed in Table 5.1. On-chip memory resources are used by all the buffers and weights ROM (read-only memory). DSPs are consumed by the arithmetic units. The two FIFOs indicate input and output streams.

Table 5.1: SA PE configurations. This table only shows when a PE corresponds to a single layer. In cases that a PE processes multiple layers together, these parameters will be duplicated and certain constraints apply.

Param	Description
H, W, C, F	Height, width, channels, filters
K, P, S	Kernel size, padding width, and stride size
P_C , P_F	Parallelism along channel and filter axis
$\mathbf{1}_{fm}$, $\mathbf{1}_{cm}$	Uses filter-/channel-major or not

sequence configurations ($\mathbf{1}_{cm}$ and $\mathbf{1}_{fm}$) are tunable.

$$T_{conv} = (H+2P)/S \times (W+2P)/S \times \lceil C/P_C \rceil \times \lceil F/P_F \rceil \quad (5.1)$$

5.2.1 Design Overview

Before we dives into the details, this section gives a high-level overview on how a DNN is mapped to a SA through the Deacon framework. At the beginning, a network specification is given as input to Deacon, and Deacon will figure out the parameters of each convolution layer, and how these layers are connected together. Deacon will draft an initial design, which resembles the streaming architecture in Fig. 5.3, with a nuance that the leftmost FIFO is connected to the input data stream, and the rightmost FIFO produces the final model predictions.

Deacon then decides the internal implementation of each convolution processing engine, which is sketched in Section 5.2.2. CNNs with a topology no more complicated than a single list do not require additional treatment, Deacon only needs to apply inter-layer pipelining techniques that decide the computation sequences Section 5.2.3 using the idea presented in Section 5.2.4.

If the model has a relatively complicated topology, like ResNet that includes residual connections, or MobileNet that features depthwise-separable convolution, Deacon needs to perform an additional level of optimisation that will change the graph structure Section 5.3. For instance, when there are residual connections, instead of having multiple basic convolution PEs, Deacon will switch to an optimised block Fig. 5.10b, which has more than one FIFOs around the computation units, and the connectivities are improved to avoid dataflow congestion.

Now, we have a design sketch that illustrates the PEs and their connectivities derived from the initial network specification and the intermediate optimisation passes. However, this design is still parameterized, i.e., the level of parallelism within each layer is yet to be instantiated. At the final stage, Deacon runs a hyperparameter search to decide these numbers, with respect to an objective on resource efficiency. Note that we choose resource efficiency instead of power efficiency as the objective, mainly because it can be formulated more concisely.

5.2.2 Processing Engines

Deacon inherits its PE designs from a base convolution layer processor that has the following four major components:

1. *arithmetic* units that carry out multiply-accumulate (MAC) operations in parallel, e.g., to compute the convolution between a kernel and a sliding window from the input image;

Table 5.2: Buffer usage of a PE that follows different computation sequences. Table 5.1 lists the definitions of the parameters.

CompSeq	Input Buffer	Output Buffer
Filter-major	HWC	HW
Channel-major	HW	HWF

2. an *input buffer* that caches the input feature map for local reuse;
3. an *output buffer* that stores and accumulates intermediate results;
4. a *line buffer* that outputs size $K \times K$ sliding windows;
5. a ROM (read-only memory) that stores weights;
6. and a control logic that keeps the counter and generates buffer addresses.

Figure 5.4 illustrates how these components are connected and the width of datapaths. The number of clock cycles to process a single image by a convolution layer is T_{conv} (Eq. (5.1)). This base design (Fig. 5.4) is versatile. We can derive a depthwise convolution processor from this base design by removing adders within the arithmetic units that sum intermediate results across channels. These adders are not required as shown in Eq. (2.2). We can also derive a pointwise convolution processor by setting K to 1 and omitting the line buffer. Moreover, we can extend the base design to implement a pooling layer by replacing the multipliers in the arithmetic units with maximisation/minimisation and discarding the weights ROM.

5.2.3 Computation Sequence

A computation sequence indicates the order of the nested loops that implement a convolution layer (Algorithm 1). This chapter focuses on two computation sequences: filter-major and channel-major. Section 2.1.1 describes them in detail. The computation sequence of a PE has two impacts on its behaviour and resource usage:

- Different computation sequences produce output at different *frequencies*. For instance, as shown in Fig. 5.2, the filter-major PE generates outputs every $\lceil C/P_C \rceil$ cycles, while the channel-major PE only outputs results at the final $\lceil F/P_F \rceil$ cycles;

- The *size* of input and output buffers corresponds to the computation sequence (Table 5.2). A filter-major PE operates on a single output channel at a time, suggesting that its output buffer can be as small as the size of a single channel of an output image. A channel-major PE keeps one input channel at a time, so its input buffer can take only the size of an input channel.

5.2.4 Inter-Layer Pipelining Techniques

Computation sequences affect pipelining performance. Fig. 5.2a demonstrates that two adjacent PEs using filter-major and channel-major can be pipelined without any stall, assuming they have the same number of channels and spatial dimensions. However, if two layers share the same computation sequence, there will be stall. Fig. 5.2a shows that two consecutive filter-major PEs incur pipeline stall, due to that the 2nd PE waiting for its input data from the 1st PE. In Fig. 5.2b, we demonstrate that if both PEs use channel-major, the 1st PE should wait for the 2nd PE to take all the intermediate results, and the 1st PE stalls in the meantime.

We take Fig. 5.2 as an example to dive deeper into our observations. There are two consecutive convolution layers, both of which have identical spatial dimensions and the number of channels and filters. These two layers are mapped to two adjacent PEs. The 1st PE uses filter-major, which generates a frame of size $H \times W$ every three iterations. There are two different scenarios:

- (1) If the 2nd PE uses channel-major, it can consume the output immediately after the 1st PE produces them. Therefore, the 1st PE will not stall and continue processing the next channel, meanwhile, the 2nd PE can finish the computation on the frame it takes.
- (2) On the other hand, If the 2nd PE uses filter-major as well, as shown in Fig. 5.2a, there will be clock cycles that the 2nd PE waits for the output from the 1st PE.

The extra waiting clock cycles make scenario (2) less promising than scenario (1).

From the observations above, we derive that *interleaving computation sequences can avoid pipeline stalls*. Indeed, we can mitigate pipeline stalls by enlarging intermediate buffers, but given that an SA already has much demand on on-chip resources, it is less profitable to increase the buffer size than to interleave computation sequences.

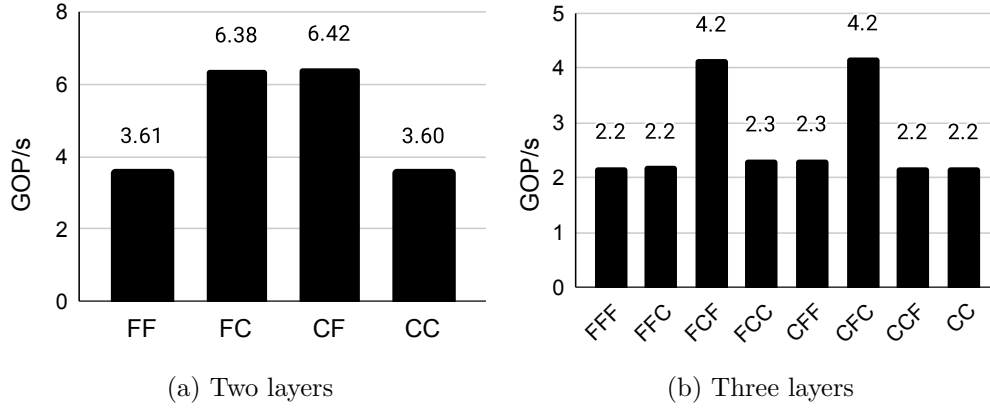


Figure 5.5: Comparison among different CompSeq configurations for two (left) or three (right) consecutive PEs. Every PE consumes and produces feature maps of the same shape. Among x-axis labels, F stands for filter-major, C denotes channel-major, and a sequence of {F, C} indicates the computation sequences of all PEs, from top to bottom.

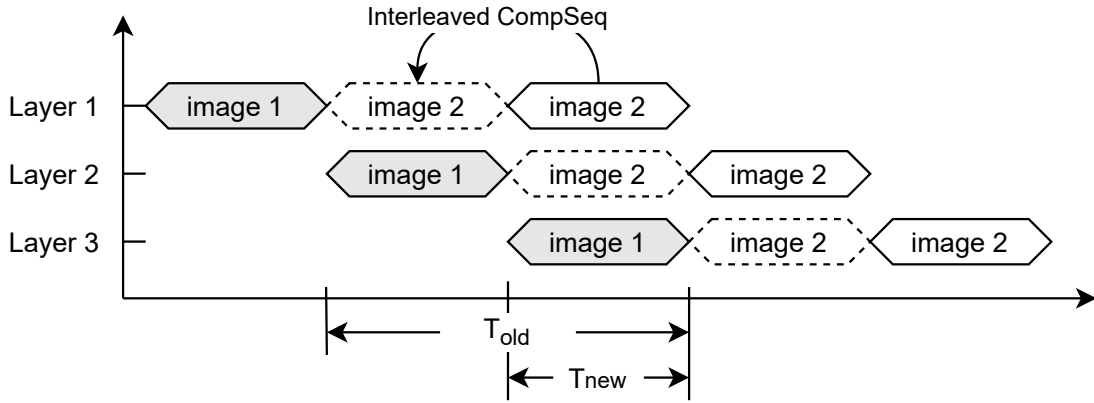


Figure 5.6: Pipeline improvement by interleaving computation sequences. With interleaved computation sequences, the 2nd image can be processed immediately after the 1st image. T_{old} and T_{new} indicate the latency between the 1st image and the 2nd image before and after the interleaving computation sequence technique is applied.

We also evaluate this heuristic on SAs deployed on real hardware, the result of which (Fig. 5.5) shows that designs with interleaved computation sequences have the best throughput. Notably, the best versions have around 2 times higher throughput than the others, which meets our expectation: interleaving computation sequences eliminates pipeline stalls and allows us to process 2 images at a time instead of 1 (Fig. 5.6). We need the following two assumptions:

- adjacent PEs take input images of the *same size*.
- adjacent PEs process images using the *same number of cycles per filter*.

The first assumption is satisfied by many DNN models, e.g., ResNet, MobileNets,

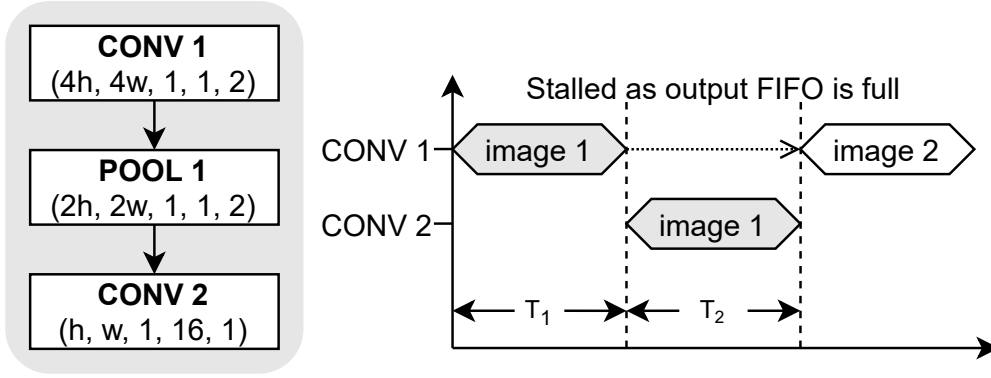


Figure 5.7: A special case that interleaved computation sequence fails. The DNN structure is shown on the left, in which the 5-tuple indicates the (H, W, C, F, S) configuration of a layer. CONV 1 and POOL 1 both have 2-stride, which scales the spatial dimensions by 16 times when reaching CONV 2. CONV 2 expands the filters by 16 times to have the same processing time as CONV 1 ($T_1 = T_2$). While CONV 2 is running, the output of CONV 1 is halted since the FIFO is full.

etc. By padding input images, convolution layers in these models produces output images of the same size as their input. Meanwhile, we can satisfy the second assumption by adding constraints (Section 5.4).

Interleaving computation sequences may not work as expected, especially when there is downsampling. A downsampling operation scales down the spatial size of an image while scales up its output channels, so that the output image has the same number of pixels as the input but a significantly different ratio between spatial dimensions and channels. For instance, if an image has shape (H, W, C) , a downsampling operation will produce an image of shape $(H/n, W/n, n^2C)$, n is a small integer larger than 1. As demonstrated in Figure 5.7, even if we interleave computation sequences, as there are much more output channels to process, the 2nd PE can still stall the 1st PE.

We can employ *double-buffering* to mitigate this issue, that is, we put two output buffers after downsampling. Double-buffering increases buffer usage, but at the whole DNN level, the overall memory efficiency (throughput over on-chip memory usage) can even improve. As shown in Fig. 5.12b, even buffers are doubled, the memory efficiency can still be significantly enhanced. There are two reasons:

- As down-sampling is not common in the tested networks, the increased buffer usage is not significant.
- A single pipeline stall can slow down the whole processor (Section 5.4.1).

5.3 Graph-Level Optimisation

The base SA (Fig. 5.3) is general to map any convolution-based, acyclic DNN models. This section explores different mapping strategies. A straightforward approach is to map a single layer to a single PE (Fig. 5.4), and PEs are connected by FIFOs. Nevertheless, this strategy may abuse resources:

- if two adjacent PEs can never have balanced processing time, pipelining them cannot improve the throughput more than the increased resource usage (Section 5.3.1);
- if multiple layers can run synchronously, there is no need to separate them into PEs that have individual buffers (Section 5.3.2);
- Having imbalance workloads, paths that share the same source and sink can lead to significant FIFO usage and affect timing (Section 5.3.3).

We discuss these scenarios in the following sections, and propose our approaches to optimise the input DNN graph. Figure 5.1b lists all our graph-level optimisation techniques.

5.3.1 Depthwise Separable Convolution

A direct mapping of DWS convolution creates two PEs, one for the depthwise convolution layer, and the other for pointwise convolution layer. A FIFO block connects these two PEs. To avoid congesting the FIFO, the pointwise PE should take input of the same data width as the output of the depthwise PE; otherwise, the producing and consuming rates will not match. We also know that the input and output data width are the same for the depthwise PE based on the design described in Section 5.2.2. Therefore, we can derive that the design parameters (Table 5.1) of these two PEs should satisfy $P_C^{(d)} = P_F^{(d)} = P_C^{(p)}$, in which (d) or (p) indicates depthwise or pointwise PE.

$$\frac{T^{(d)}}{T^{(p)}} = \frac{H \times W \times \lceil C/P_C^{(d)} \rceil}{(H/S) \times (W/S) \times \lceil C/P_C^{(p)} \rceil \lceil F/P_F^{(p)} \rceil} = \frac{S^2}{\lceil F/P_F^{(p)} \rceil} \quad (5.2)$$

As Eq. (5.2) demonstrates, the processing time of that depthwise and pointwise PEs take, $T^{(d)}$ and $T^{(p)}$, are significantly different. Given that S mostly takes value 1 or 2, and F can be as large as 512, $T^{(d)}$ can be much less than $T^{(p)}$. And

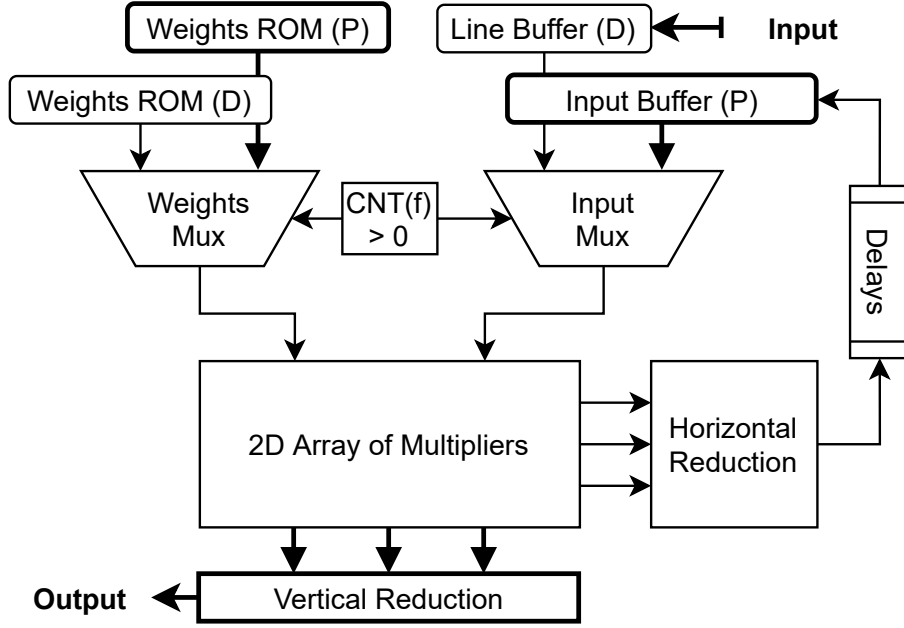


Figure 5.8: Optimised DWS convolution PE. When counter $f = 0$, this PE processes depthwise convolution, taking weights from the corresponding ROM and reading from the line buffer; its results are accumulated horizontally from the 2D multiplier array and feeds into the input buffer, which will be processed in the following iterations when $f > 0$ to produce pointwise convolution results. Final results from pointwise are vertically reduced from those element-wise multiplications.

Table 5.3: Extra buffer usage and parallelism constraints imposed by different residual connection implementations.

Design	Buffer	Constraints
R-CONV	$\text{HW}(\mathbf{1}_{\text{fm}}^{(2)} \times \mathbf{C}^{(2)})$	$P_C^{(1)} = P_F^{(2)}$
RS-CONV-A	$\text{HW}(\mathbf{1}_{\text{cm}}^{(2)} \times \mathbf{C}^{(1)})$	$P_C^{(2)} = P_C^{(1)}(F^{(1)}/C^{(1)})$
RS-CONV-B	$\text{HW}(\mathbf{1}_{\text{cm}}^{(1)} \times F^{(1)})/S^2 + \text{HW}(\mathbf{1}_{\text{fm}}^{(2)} \times \mathbf{C}^{(2)})$	$P_F^{(1)} = P_F^{(2)}$

as Eq. (5.2) shows, the ratio between depthwise and pointwise PEs processing time is solely controlled by $P_F^{(p)}$, which suggests that no depthwise design parameters can reduce the imbalance. In another word, even if we increase the parallelism of the depthwise PE $P_C^{(d)}$, there will be no impact on overall throughput, and the computation resources will be wasted.

Our optimisation, as illustrated in Fig. 5.8, chooses to share the resources between depthwise and pointwise PEs. That is, instead of pipelining the depthwise and pointwise PEs, we time-multiplex them. Specifically, we share the multipliers, which reduces their overall usage from $P_C^{(d)}(P_F^{(p)} + K^2)$ to $P_C^{(d)} \max(P_F^{(d)}, K^2)$. On the other hand, time-multiplexing increases the processing time from $T^{(p)}$ to $T^{(d)} + T^{(p)}$.

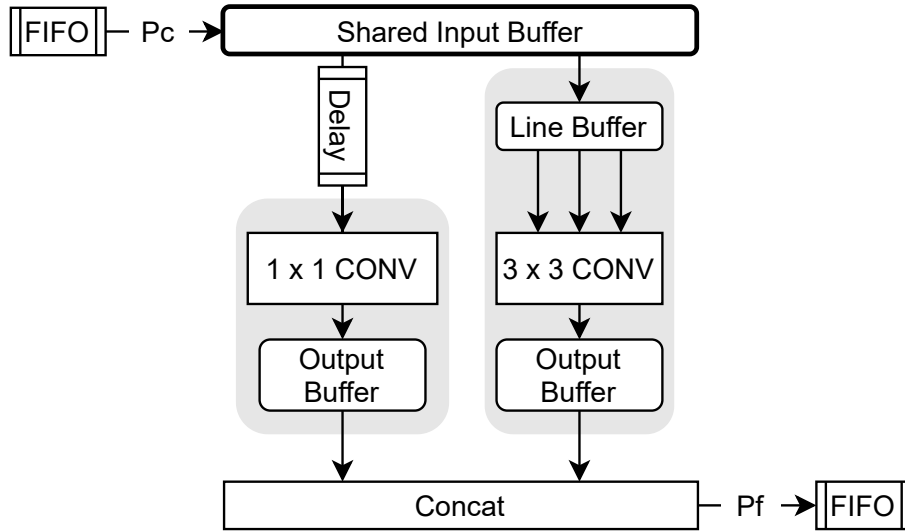


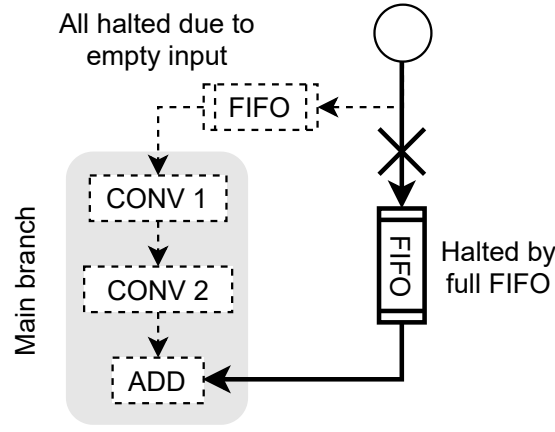
Figure 5.9: Split and concatenation with buffer sharing. The input buffer is being shared by the two split convolution layers. There is a FIFO that delays the processing of the pointwise (1x1) convolution since it runs faster than the 3x3 convolution, and we need to make sure they process at the same speed to ensure we concatenate the right pairs in the end.

Most DWS convolutions use $K = 3$, and by setting $P_F = 8$ we can save almost 50% multipliers. For example, by setting $P_F = 8$ for all layers of MobileNet-v1 [118], in which all DWS has $K = 3$, we can save the DSP usage from 1,840 to 1,008. Meanwhile, Fig. 5.12a empirically justifies that improvement in resource efficiency we can get.

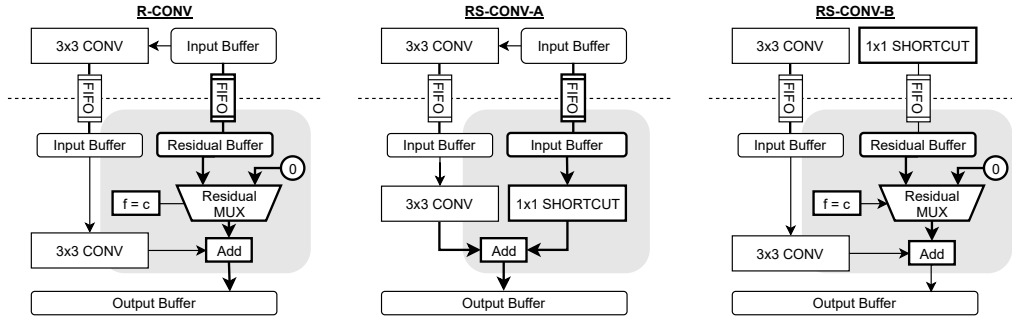
5.3.2 Split and Concatenation

The key component in SqueezeNet [125], the fire module, is based on split and concatenation. A fire module processes an input image by two convolution layers running in parallel and concatenates their output images along their channels. These two convolution layers have the same input and output channels, while one is a standard convolution ($K = 3$), the other is a pointwise convolution ($K = 1$). We can use two different PEs following the base design (Fig. 5.4), but there are two caveats:

- These two PEs take the same image as input, but they have two independent input buffer.
- They should produce output images at the same rate, otherwise, there can be congestion during concatenation. However, the asynchronous behaviour between these two PEs makes it harder to achieve.



(a) The deadlock issue appearing in a directly-mapped design. The input (circle) will be halted by the full FIFO on the right, while the main branch that takes data from the right FIFO cannot run due to the halted input.



(b) Stacked residual block with or without shortcut projection. The dashed horizontal line separates the design into two PEs. The shaded area in each design indicates the residual connection. R-CONV is a design that processes a residual layer without shortcut projection, while RS-CONV-A/B can handle those with shortcuts. This is applicable for both ResNet-18 and the inverted bottleneck in MobileNet v2 [251]. For the latter case, even though there are three layers, the last two form a depthwise separable layer and can be fused into a single PE.

Figure 5.10: Graph-level optimisation for residual connections.

Therefore, a single PE design is a better choice for the fire module. A single PE just has one input buffer, and all its components run synchronously. There are several extensions we should make on the base PE design (Fig. 5.4).

$$P_C^{(i)} = P_C \quad P_F^{(i)} = F^{(i)}/F \times P_F \quad i \in \{\text{pw}, \text{std}\} \quad (5.3)$$

Fig. 5.9 demonstrates the single PE implementation:

- An input buffer is shared between two convolution.
- There are two arithmetic units, one implements the pointwise convolution, the other implements the standard convolution.

- There is a delay FIFO between the input buffer and the pointwise arithmetic units since the standard convolution runs slower, and we need to ensure pointwise and standard components have the same output rate.
- The two convolution components have their own parallelism parameters, but they should satisfy the constraints described in Eq. (5.3). Specifically, the input parallelism should be the same between both PEs, and the output channel parallelism should be distributed based on their numbers of output channels.

5.3.3 Residual Connection

Due to the existence of non-local connections ranging from the 1st layer to the 2nd or even the 3rd, residual connections are particularly challenging for SAs, which become less efficient when the data locality is degraded. Consequently, a direct mapping of a residual connection is problematic: the FIFO that implements the residual connection should have a depth that can buffer one or two input feature maps; otherwise, the processing of the main branch, a sequence of convolution layers that the residual connection attached to will be stalled or even deadlocked. For instance, a stack of two convolution layers in ResNet-18 connects its original input to the final output, and if the residual FIFO has a depth of N and N is smaller than the input feature map size, the FIFO will be full even before the 1st layer finishes processing, and its data cannot be consumed since the 2nd layer will not start unless the 1st finishes. Fig. 5.10a illustrates this issue. Regardless of the lowered memory efficiency imposed by the enlarged residual FIFOs, this direct mapping implementation can still struggle to meet timing constraints: finding storage blocks (either registers or BRAMs) of a large amount and closely connected is already difficult, not to mention the requirements to build the main branch by keeping the two ends of the FIFO physically close.

To improve data locality, Deacon proposes to *pack* the residual data with the values passing through the main branch. Take the 2-layer stack with residual connection as an example, when the 1st layer produces a filter of output, it outputs a channel of the input data as well. How the extra input is handled depending on the channels/filters of the two layers (Fig. 5.10b). If there is no shortcut projection, the extra input will be stored in a residual buffer, the size of which depends on the CompSeq of the 2nd layer:

- if it is channel-major, we can apply residual addition to the filter f equals to the current input channel c , with which a buffer of size $H \times W$ is sufficient;

- if it is filter-major, all the input data should be buffered when processing the first filter, and therefore, the residual buffer should be as large as $F \times H \times W$.

Considering the residual buffer size, we can use it as a heuristic that the 2nd layer in a residual stack should be in channel-major when exploring the design space.

When there exists a shortcut projection, we should process an extra pointwise convolution. Both PEs for the 1st and the 2nd layers can carry out this task, as shown in the middle (RS-CONV-A) and the rightmost (RS-CONV-B) of Fig. 5.10b. They impose different trade-offs:

- putting a shortcut to the 2nd PE needs an extra input buffer, while we can choose the computation sequence more flexibly, i.e., there is no significant disadvantage of choosing one over another;
- leaving shortcut in the 1st PE can save an input buffer by sharing, however, an extra residual buffer is required for the 2nd, and the shortcut convolution needs its own output buffer.

In addition, to match the width of the data-paths, there are different extra constraints imposed by these designs, as shown in Table 5.3, which will be discussed in detail.

5.4 Design Space Exploration

This section discusses Design Space Exploration, a major step we take to find viable and optimised configuration for Deacon design generation.

Although our SA is designed to support any DNN graphs that are DAGs, there are prerequisites on graph properties that should be met to have an accurate design model. First of all, we assume that the DNN graph is *3-colourable*, i.e., suppose there are three colours, then every adjacent node can be coloured with a different colour. This is empirically a well-met condition: given that DNN graphs are commonly neither complete graphs (every pair of nodes is connected) nor cycle graphs (nodes within form a single chained cycle), according to the Brook's theorem [37], any such graph with node that has maximally 3 neighbours can be 3-coloured, while many DNN graphs have such property. Note that this property just needs to be met

on the corresponding graph of PE nodes after finishing graph-level optimisation (Section 5.3). For example, those graphs with multiple split branches are likely to have more than 3 neighbours, however, applying the optimisation (Section 5.3.2) can reduce these branches into a single chain, which still fits into the 3-colourable category.

Based on this 3-colourable prerequisite, we propose the following Lemma:

Lemma 1. *We can always find an assignment of computation sequence on a Deacon SA that at most two adjacent nodes have the same value.*

Proof. A Deacon SA is a 3-colourable graph of PEs. Assuming the three colours are C (channel-major), F (filter-major), and FC (filter-/channel-major), then we can find an assignment that no C, F, or FC are adjacent. We should resolve FC to either C or F in the end, and no matter which value it takes, the final result will never have three connected PEs that have the same colour. $\square$

Lemma 1 is useful for deriving the performance model.

$$R_{\text{dsp}} = \lceil P_C / \mathfrak{x} \rceil \times P_F \times K^2 \quad \mathfrak{x} = \begin{cases} 2 & BW = 8 \\ 1 & BW = 16 \end{cases} \quad (5.4)$$

5.4.1 Design Model

This section describes the design model, which estimates the resource usage and the performance of a design characterised by the parameters (Table 5.1).

Resource Model

A resource model estimates the usage of on-chip memory and DSP from the provided design configurations described by parameters from Table 5.1. The total resource usage is the sum of those consumed by each PE. Depending on the arithmetic bit-width (BW), the DSP usage per PE is either the same as the total MACs if $BW = 16$, or half of the MACs when $BW = 8$ and $P_C > 1$. The latter is possible mainly because our targetting FPGA, Xilinx UltraScale+, provides DSP48E2 blocks that can be configured to carry out two MACs in parallel when the bit-width is 8 or lower, if one of the input operands is shared, e.g., $\mathbf{a}(\mathbf{b} + \mathbf{c})$ [76]. We can share

the input feature map along the parallel input channels, and therefore, the overall DSP usage can be reduced by half.

$$\begin{aligned}
R_{\text{bram}} &= H \times W \times (\mathbf{1}_{\text{fm}} \times C) && \text{(input buffer)} \\
&+ H \times W \times (\mathbf{1}_{\text{cm}} \times F) && \text{(output buffer)} \\
&+ W \times K \times P_C && \text{(line buffer)}
\end{aligned} \tag{5.5}$$

On-chip memory is mainly taken by buffers, as shown in Eq. (5.5), in which $\mathbf{1}_{\text{fm}}$ and $\mathbf{1}_{\text{cm}}$ takes 1 if filter-/channel-major is used respectively. $\mathbf{1}_{\text{fm}}$ will be $1/C$ when using channel-major, and $\mathbf{1}_{\text{cm}}$ will be $1/F$ when using filter-major. Note that here R_{bram} estimates the total number of elements that should be stored on-chip, while the exact BRAM blocks consumed needs a scaling factor of $BW/18 \times 1024$, assuming BRAM18K is targetted. As an alternative resource for on-chip storage, UltraRAM (URAM) can also be leveraged to implement certain buffers. We do not distinguish between these two resources while modelling and use their total capacity as a resource constraint. We also need to further adjust the estimation after considering the optimisations from Section 5.3. Weight ROMs are left out since they are mostly constant for changes in design parameters. Although the storage used by FIFOs scale up by the parallelism as well, they are omitted from our model since their depth are much smaller than PE buffers.

$$P_C^{(e)} = P_F^{(s)} \quad \text{for all FIFOs connecting } s, e \tag{5.6}$$

Constraints

The production and consumption rate should be matched at both ends of a FIFO to avoid congestion. Based on this principle, we first introduce a set of constraints on the parallelism, which determines the processing rate, between any pair of adjacent PEs, as shown in Eq. (5.6). Additional constraints on parallelism are introduced if there is split-and-concatenation, see Eq. (5.3), or residual connections exist (Table 5.3). Besides, there are constraints on resource usage, capping the sum of R_{dsp} and R_{bram} below a certain threshold to avoid depleting available resources.

Performance and Cost Models

A streaming design is pipelined, and the overall throughput is determined by the longest latency (T_{max}) among its stages. Ideally, every PE occupies a single stage

of the pipeline, and $T_{\max}$ equals the maximum number of cycles that a PE takes to produce a single feature map. This scenario is based on a presumption that the computation sequences are interleaved (Section 5.2.4), or in the context of graph, every node (PE) has a computation sequence different from all its neighbours. We discuss how to interleave computation sequences in the next section.

$$T_f = \begin{cases} HW^{\lceil C/P_C \rceil} & (C) \\ HW & (F) \end{cases} \quad T_c = \begin{cases} HW & (C) \\ HW^{\lceil F/P_F \rceil} & (F) \end{cases} \quad (5.7)$$

If there are adjacent PEs that share the same computation sequence, they cannot be fully pipelined, and we will model them as a single pipeline stage with a summed up latency. With Lemma 1, we know that there will not be three nodes in a row that have the same computation sequence, by which there is no need to derive latency formulation.

Moreover, two adjacent PEs with different computation sequences can still stall the pipeline if they have imbalanced processing time, i.e., the data consumption rate (T_c) is less than production rate (T_f), defined in Eq. (5.7). In most cases, the stall effect can be mitigated by the FIFO buffering, however, if there is downsampling (Fig. 5.7) the latency can be amplified. For instance, if a channel-major PE (A) is stalled and there is a pooling layer with a 2-strided one following, then producing a single input for the next PE (B), supposedly in filter-major, requires S^2 number of output from A. Instead of modelling this side-effect, we choose to avoid it by adding a double buffer after each downsampling layer if there exists $T_c < T_f$. The extra buffer size for channel-major equals the whole feature map since a bulk of output will be produced at once (Fig. 5.2b), while for filter-major the size of a buffer can be a single output channel.

$$T_{\max} = \max\{T^{(i_1)}, \dots, T^{(i_n)}, T^{(j_1)} + T^{(k_1)}, \dots, T^{(j_m)} + T^{(k_m)}\} \quad (5.8)$$

As a summary, Eq. (5.8) gives the formula that calculates the longest pipeline stage latency, when there are n PEs $\{i_1, \dots, i_n\}$ that has no neighbour with the same computation sequence, while m pairs of PEs that are both channel-major or filter-major.

Definition 5.1. The resource efficiency of a design is defined by throughput over DSP usage.

Since the throughput of an SA design is determined by the longest-running PE, the

latency of which is defined in Eq. (5.8), we can get the SA resource efficiency as follows:

$$\text{SA Resource Efficiency} = 1/T_{\max} \times R_{\text{dsp}}. \quad (5.9)$$

Resource Efficiency: SA versus CLP

We can derive the resource efficiency of a CLP-based accelerator as follows. Conventionally, a CLP design computes every layer sequentially without any execution overlap. The throughput of a CLP design is therefore the reciprocal of the total execution time. We can further compute the CLP resource efficiency as:

$$\text{CLP Resource Efficiency} = 1/\sum_{i=1}^N T^{(i)} \times R_{\text{dsp}}. \quad (5.10)$$

In Eq. (5.10), N is the total number of layers, and $T^{(i)}$ is the processing time for layer i .

One typical scenario that an SA design has worse resource efficiency than a CLP is when there exists layers that have *much fewer* MAC operations than the others. For instance, suppose there is a network that has N layers with identical configurations. We can build an SA design that achieves the same processing time for all layers, i.e., $T_{\text{SA}}^{\max} = T_{\text{SA}}^{(1)} = \dots = T_{\text{SA}}^{(N)}$. In this case, all PEs should have the same number of DSPs (denoted as $R_{\text{SA}}^{\text{dsp}}$), since to achieve identical processing time for layers of the same configuration, the parallelism parameters for all layers should be the same, which leads to the same DSP usage across layers. The resource efficiency for this design is $1/(T_{\text{SA}}^{\max} \times R_{\text{SA}}^{\text{dsp}})$. On the other hand, a CLP design has identical processing time for all layers as well, as it processes any layer at the same speed, and all layers have the same amount of workload. Suppose this CLP design uses $R_{\text{CLP}}^{\text{dsp}}$ number of DSPs overall, and each layer has processing time as T_{CLP} , we have its resource efficiency as $1/(N \times T_{\text{CLP}} \times R_{\text{CLP}}^{\text{dsp}})$.

Assuming one layer now has a much less workload than the others, we argue that the SA design still has the same resource efficiency, while the resource efficiency of the CLP design improves. It is because even one layer has much less workload, $T_{\text{SA}}^{\max}$ does not change, and the level of parallelism is constrained by Eq. (5.6), the $R_{\text{SA}}^{\text{dsp}}$ stays the same as well. For the CLP design, however, the total processing time is reduced, and therefore, its resource efficiency improves. Although it is an oversimplified scenario, we have observed similar circumstances in the networks we empirically study.

5.4.2 Optimisation Algorithm

An optimisation algorithm is required to find an optimal design configuration regarding the cost function. Each point in the design space is a configuration, which consists of the computation sequence ($\mathbf{1}_{\text{cm}}, \mathbf{1}_{\text{fm}}$) and the parallelism parameters ($\mathbf{P}_{\text{C}}, \mathbf{P}_{\text{F}}$) for each PE (Table 5.1). Despite the constraints on parallelism (Section 5.4.1), the number of combinations is still enormous, which imposes a serious challenge to the optimisation algorithm.

Design Space Pruning

The design space is *pruned* by eliminating the choices on computation sequences that are heuristically sub-optimal, i.e., have identical computation sequences in adjacent PEs. There are two cases:

1. if the graph of PE is 2-colourable, we run a 2-colouring algorithm to get all the 2-coloured candidates, using channel-major or filter-major as colours;
2. otherwise, the graph should be 3-colourable (Lemma 1), and we apply 3-colouring with an extra colour representing channel-/filter-major.

The 2-colouring scenario has linear-time complexity, which can be solved by Depth-First-Search (DFS) straightforwardly, and each solution is a valid PE graph with interleaved computation sequences. The 3-colouring problem is NP-Complete [79], for which we apply the well-studied heuristic DSatur algorithm [36]. For each 3-coloured graph we get, we duplicate it by deciding the value for the extra colour, either channel-major or filter-major. Pruning for a 2-colourable graph reduces the design space by at most $\mathcal{O}(2^{N-1})$ for a chain graph, which appears frequently according to common DNN graph topologies after applying graph-level optimisations (Fig. 5.1b).

Global Optimisation

Our optimisation objective is to improve resource efficiency of a design, or equivalently, we want to minimise a *cost*, which we can define as the reciprocal of resource efficiency:

$$\text{Cost} = T_{\text{max}} \times R_{\text{dsp}} \quad (5.11)$$

Algorithm 4: Design space exploration algorithm.**Input:** PE graph G after graph-level optimisation**Output:** Optimised design configuration D^*

```

1 Design space pruning. Initialise candidates  $\mathcal{G} \leftarrow \emptyset$ 
2 if  $G$  is 2-colourable then
3   |  $\mathcal{G} \leftarrow \text{TwoColouring}(G)$ 
4 else /*  $G$  is 3-colourable */ */
5   |  $\mathcal{G} \leftarrow \text{DSatur}(G)$ 

6 Simulated annealing. Initialise state  $S$ 
7 for  $i \leftarrow 0$  to  $N$  do
8   | Temperature  $T \leftarrow 1 - (i + 1)/N$ 
9   | Sample a neighbour  $S'$  by tweaking a parallelism parameter or switching to
   |   an element  $\in \mathcal{G}$ 
10  | if  $E(S) > E(S')$  or  $\exp(-[E(S') - E(S)]/T) > \mathcal{U}[0, 1]$  then
11  |   |  $S \leftarrow S'$ 

12 return  $S$  as the optimised design configuration  $D^*$ 

```

Table 5.4: Xilinx UltraScale+ VU9P resource capacity.

LUTs	FFs	DSP	BRAM	URAM
1,182.2 K	2,364.5 K	6,840	4,320	960

Design space exploration is a global optimisation problem: we are trying to find a global minima of the cost function over the whole set of design configurations constrained by certain conditions (Section 5.4.1). It is extremely difficult to apply deterministic algorithms, as the problem falls into the category of NP-hard, Mixed-Integer Non-Linear Programming (MINLP) due to the solutions $(P_C, P_F, \mathbf{1}_{fm}, \mathbf{1}_{cm})$ are integers, and the cost function and constraints have non-linear components $T_{\max}$ and R_{dsp} .

Leveraging simulated annealing [150], a heuristic, probabilistic global optimisation technique, we can approximately solve our design space optimisation problem. The definition of its key components in our context is as follows:

- a *state* is a vector of the design parameters;
- *neighbours* of a state are those generated by tweaking either one parallelism parameter (P_C, P_F) , obeying the corresponding constraints, or changing to a different colouring found from the pruning step, with the undecided sequence randomly initialised by uniform distribution if the graph is 3-colourable;
- the *energy* of a state equals to the cost function defined in Eq. (5.11), and if the resource constraints are broken, the energy is modified to value that is

Table 5.5: Comparing with results from related works on MobileNet v1 and v2. Numbers in bold are considered the best among their corresponding groups. (a) and (b) versions of [151] refer to the single instance and the two instances designs. (S) and (L) in [306] target small (ZU2) and large (ZU9) FPGAs. DSPs refer to the number of DSP Slices with 27x18 multipliers for Xilinx boards and the number of 27x27 multipliers for Arria 10 FPGAs. BRAMs are of 36Kb per block on Xilinx FPGAs (BRAM36K) and of 20Kb per block on Arria 10 FPGAs (M20K). Memory bandwidth (Mem. BW) is calculated by multiplying the data transfer amount per frame (GB/frame) with frame rate (FPS). Deacon designs only requires to transfer the initial input and final output, while other designs should transfer the input, output, and weights of each layer. We also normalise (listed as norm.) the FPS/DSP by scaling performance to match 200 MHz clock frequency.

	MobileNet v1					MobileNet v2						
	[328]	[264]	Ours	[328]	Arria 10 AS066	[16]	[151] (a)	[151] (b)	[306] (S)	[306] (L)	Ours	
FPGA	K325T	ZU9EG	VU9P	K325T	Arria 10 AS066	Arria 10 AS066	Arria 10 AS066	AX115	ZU2EG	ZU9EG	VU9P	
Clock (MHz)	200	150	200	200	150	150	250	200	430	333	200	
Precision	8	8/4	16	8	8	8	16	16	8	8	16	
Architecture	OVL-CLP	CLP	SA	OVL-CLP	CLP	CLP	CLP	2×CLPs	CLP	CLP	SA	
DSPs (used/total)	704/840	1452/2520	1863/6840	704/840	1278/1687	610/1518	610/1518	1220/1518	212/240	2070/2520	2715/6840	
BRAMs (used/total)	387/890	1729/1824	3197/4320	387/890	1844/2133	451/2713	451/2713	783/2713	290/300	1542/1824	3476/4320	
Mem. BW (Gb/s)	14.59	7.023/3.512	0.03231	7.604	6.215	30.73	30.73	49.03	4.793	18.91	0.09074	
Power (W)	16.85	—	14.57	16.47	31.59	29.40	29.40	34.00	—	—	14.86	
OPS (G)	1.14	0.72	1.14	0.58	0.58	0.58	0.58	0.58	0.58	0.58	0.58	
GFLOPs	301.64	91.72	448.64	189.41	154.81	382.66	382.66	610.62	119.39	470.93	895.32	
GFLOPs/DSP	0.21	0.03	0.24	0.14	0.06	0.63	0.63	0.50	0.28	0.11	0.17	
GFLOPs/W	17.90	—	30.79	11.50	4.90	13.02	13.02	17.96	—	—	51.90	
FPS	264.60	127.39	395.12	325.70	266.20	658.00	658.00	1050.00	205.30	809.80	970.86	
FPS/DSP	0.19	0.04	0.21	0.23	0.10	1.08	1.08	0.86	0.48	0.20	0.37	
(norm.)	0.19	0.06	0.21	0.23	0.14	0.86	0.86	0.86	0.23	0.12	0.37	
FPS/W	15.70	—	27.12	19.78	8.43	22.38	22.38	30.88	—	—	65.33	

larger than all possible cost values.

The sketch of the full design space exploration algorithm is summarised in Algorithm 4.

5.5 Experiments

The designs that Deacon generates target MaxCompiler (version 2021.1) [202], a tool that describes in and synthesises from hardware specifications within the SA paradigm. The target platform is MAX5, a system that is shipped together a Xilinx UltraScale+ VU9P FPGA with an Intel Xeon E5-2643 CPU. Table 5.4 summarises the resource capacity of VU9P.

There are four DNN models we evaluate: MobileNet v1 [118], v2 [251], SqueezeNet (1.1) [125], and ResNet-18 [110]. We use a 16-bit fixed-point data format for the designs accelerating compact models like MobileNets and SqueezeNet. While for ResNet-18, due to its rather larger number of parameters, we need to quantize the model to 8-bit integer [131]. Deacon is not optimised for data types and quantisation schemes, and the bit-width chosen are the largest available that will not deplete hardware resources. The accuracy of the input models will be preserved during Deacon processing.

Input data are streamed into Deacon designs: an input image will be consumed once the design is ready, i.e., the previous image is processed by the first layer and passed to the subsequent layers. There is no batch processing, and Deacon designs are under fully utilisation if there is a continuous stream of input data.

The performance metrics we mainly look at are resource efficiency, measured by FPS per DSP block, and power efficiency, which is calculated by FPS per Watt. For a fair comparison, the resource efficiency is normalised by 0.5 if the bit-width is 8, since a single DSP block (DSP48E2) can be shared by at most two 8-bit operations on VU9P. Due to the reported numbers from different work can use different clock frequency, we will use results normalised to 200 MHz for comparison, assuming the resource efficiency from past papers can scale linearly with the clock speed.

5.5.1 Compare with Existing Work

Our objective is to exceed all existing approaches on power efficiency, since there is no off-chip memory transactions except the initial input and the final output; and

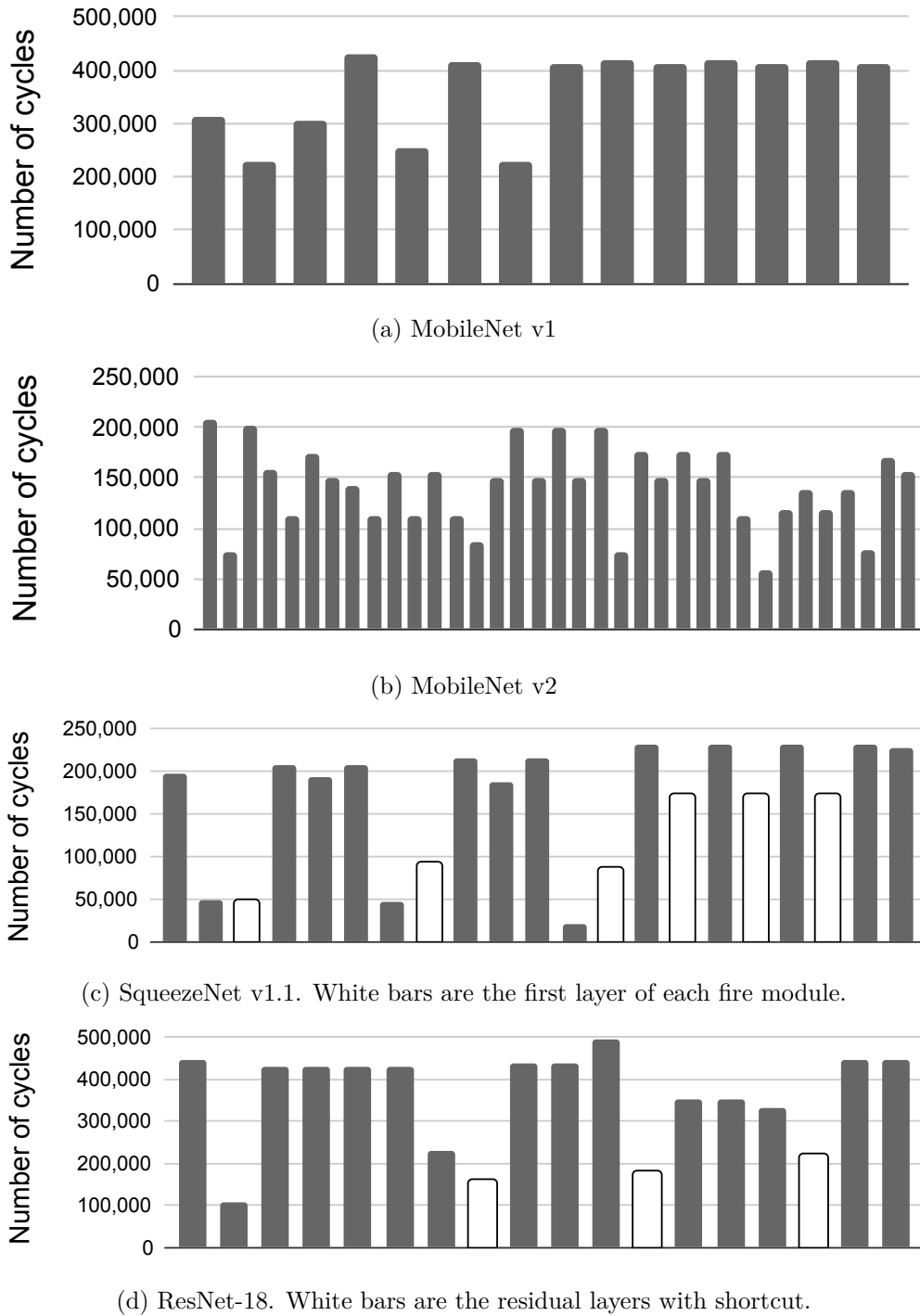


Figure 5.11: Pipeline balancing results of the best Deacon designs. From left to right are the latency of pipeline stages from the top to the bottom.

try to have resource efficiency on par, and even surpass existing work.

MobileNet v1 and v2

Table 5.5 compares MobileNet v1 and v2 with existing work. For MobileNet v1, the two other approaches based on a single Convolution Layer Processor (CLP)

perform worse than Deacon, regarding resource efficiency (FPS/DSP) and power efficiency (FPS/W). Specifically, Deacon is 1.13 times more resource efficient than Light-OPU [328], and 3.63 times better than RR-MobileNet [264]. The depthwise separable convolution is optimised by the corresponding technique in Section 5.3, which explains the higher resource efficiency.

Deacon produces MobileNet v2 accelerators more resource and power efficient than the rest of the designs for comparison, with a geomean of 1.22 and 3.54 times improvement. Deacon detects and transforms the depthwise separable convolution within each inverted bottleneck block (the middle and the last layer) as well, which guarantees resource efficiency improvement despite the imbalanced pipeline stages as seen in Fig. 5.11a and Fig. 5.11b.

Deacon is better than most existing MobileNet v2 designs regarding DSP resource efficiency (GFLOPs/DSP), except those two provided by [151] and the small design from [306]. The underlying reason is that, due to the existence of residual connections, there are more constraints on the parallelism parameters for MobileNet v2 than v1, which increases the difficulty of balancing pipeline stages in Deacon designs, and therefore, some DSP resources are idle.

Moreover, Deacon designs require much less off-chip memory bandwidth, specifically, the geomean of memory bandwidth reduction rate from using Deacon over prior papers are 220.1 and 150.4 respectively for MobileNet v1/v2.

SqueezeNet

Regarding resource efficiency, Deacon is not significantly better than the others when the target model is SqueezeNet (Table 5.6). The major reason is that, based on the SqueezeNet definition and the discussion in Section 5.3.2, the P_F of a split-concatenation layer is the sum of the P_F of its two inner convolution layers. This constraint cannot be eased to make sure there is no blocking in the SA, while it can produce convolution layers that run much faster than expected, e.g., the first layer of a fire module being marked as white in Fig. 5.11c. Still, although being less efficient than unzipFPGA [284], Deacon can generate a design that has performance on par with Shen et al. [255], and 1.2 times more efficient than Light-OPU [328]. The geomean of memory bandwidth reduction is 17.25 comparing the prior examples and Deacon.

Table 5.6: SqueezeNet v1.1 performance comparison. The definition of rows DSPs, BRAMs, and Mem. BW are the same as in Table 5.5.

	[328]	[255]	[284]	Ours
FPGA	K325T	V690T	ZU7EV	VU9P
Clock (MHz)	200	170	200	200
Precision	8	16	16	16
Architecture	OVL-CLP	CLPs	CLP	SA
DSPs	704/840	2880/3600	1728/1728	2248/6840
BRAMs	387/890	317.5/1470	135/312	1877/4320
Mem. BW	16.52	92.10	62.20	2.642
Power (W)	18.57	—	—	13.04
OPS (G)	0.70	0.70	0.70	0.70
GFLOPs	293.92	909.70	553.20	601.17
GFLOPs/DSP	0.19	0.23	0.32	0.27
GFLOPs/W	15.83	—	—	46.10
FPS	420.90	1173.0	792.20	860.90
FPS/DSP	0.30	0.33	0.46	0.38
(norm.)	0.30	0.39	0.46	0.38
FPS/W	22.67	—	—	66.02

Table 5.7: ResNet-18 performance comparison. We list three results from Deacon here, (a), (b), and (c), in which (a) is the baseline without parallelism, (b) is constrained by around 2k DSP blocks, and (c) is capped by 5k. DSPs, BRAMs, and Mem. BW are the same as in Table 5.5.

	[284]	[41]	Ours (a)	Ours (b)	Ours (c)
FPGA	Z7045	Z7045	VU9P	VU9P	VU9P
Clock (MHz)	150	250	200	200	200
Precision	16	16	8	8	8
Arch.	CLP	ISA	SA	SA	SA
DSPs	900	256	292	2011	5216
BRAMs	547	547	2591	2643	3458
Mem. BW	10.43	4.468	0.004	0.070	0.209
Power (W)	—	—	12.13	14.41	17.30
OPS (G)	3.59	3.59	3.59	3.59	3.59
GFLOPs	179.07	76.73	29.57	550.54	1,637.75
FPS	49.90	21.38	8.07	150.19	446.78
FPS/DSP	0.058	0.084	0.014	0.037	0.043
(norm.)	0.077	0.074	0.014	0.037	0.043
FPS/W	—	—	0.66	10.42	25.83

ResNet-18

Compared with existing ResNet-18 results, what Deacon produces is not as appealing as others. Even though we can reach much higher throughput, the resource

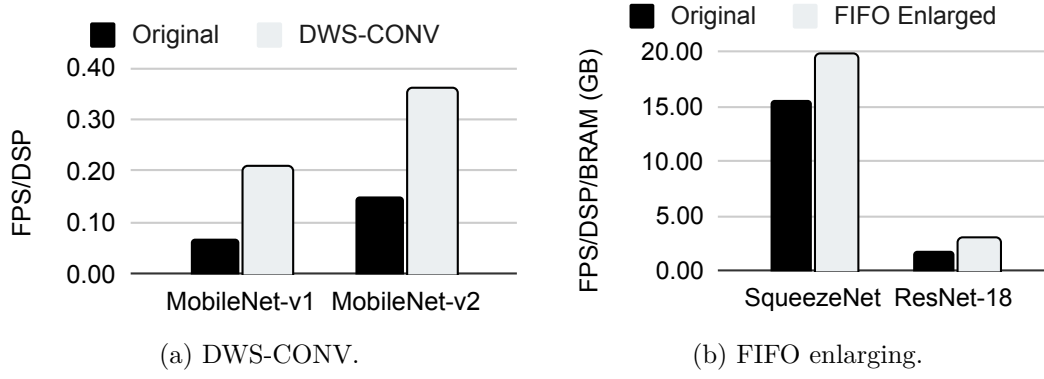


Figure 5.12: Empirical study on Deacon optimisation techniques.

efficiency is about half of the other work. This is primarily due to a reason similar to SqueezeNet: the shortcut layers are constrained to have the same parallelism as the main branches they are attached to (Fig. 5.11d), which makes these layers run much faster than the blocking pipeline stage, and therefore, the resource efficiency is limited. As shown in Table 5.7, increasing the resource usage will improve throughput, however, the resource efficiency is still capped by the faster-than-required shortcut layers. Other approaches are not based on SA, so their resource efficiency can become higher. Nevertheless, due to their high memory bandwidth requirement, scaling their throughput up to the same level as Deacon designs, e.g., 446.78 FPS got from (c), the most resource-efficient design from [284] requires 93.38 Gb/s off-chip memory bandwidth, which can hardly achieve with the state-of-the-art Dynamic Random-Access Memory³.

5.5.2 Empirical Study on Optimisation Techniques

This section demonstrates at what scale the optimisation techniques used by Deacon can help improve the performance, compared with a corresponding SA design baseline.

Table 5.5 already shows how the DWS-CONV optimisation helps improve resource efficiency. In Fig. 5.12a, we further evaluate the resource efficiency gain by comparing the designs generated by Deacon for MobileNet v1 and v2 under the same resource constraints, using either the original DWS design, or the optimised DWS-CONV. There can be about 2 times improvement over the original baseline.

Section 5.2.4 mentions the necessity to enlarge the FIFO depth after pooling layers, otherwise, the performance of the pipelined designs can be significantly reduced.

³DDR4, the commonly used memory, can only support 2400 Mb/s data bandwidth.

However, enlarging FIFO depth involves more BRAM resources. Fig. 5.12b evaluates the memory efficiency, defined by the resource (DSP) efficiency over the total on-chip memory space used. Results on SqueezeNet and ResNet-18, the two models with pooling layers, show that despite the overhead in BRAM usage, the memory efficiency is still improved.

5.6 Discussion

Despite Deacon demonstrating its power efficiency advantage through empirical evaluation across numerous benchmark examples, certain limitations persist. Primarily, Deacon presupposes that the target FPGA boards possess sufficient size to accommodate the entirety of the DNN model. This assumption may not hold true in instances involving the deployment of an extraordinarily large network or the utilization of a compact FPGA board on embedded devices. Various strategies can mitigate these challenges, such as implementing lower-precision versions of the target network or harnessing the multi-board streaming technique as exemplified by *fpgaConvNet* [283]. However, an exhaustive examination of the repercussions of these potential solutions falls outside the scope of this thesis.

This thesis incorporates up-to-date research, including some of the most recent advancements in the field at the time of its composition, such as the work by Venieris et al. [284]. However, the rapidly evolving nature of this field inevitably leads to some oversights, which is another acknowledged limitation of this study. Furthermore, while we strive to select and analyze the most relevant accelerator designs for the convolutional neural network (CNN) models examined in this thesis, these models might not reflect the most contemporary CNN design spaces. The introduction of novel models since the invention of ResNet is continuous; regrettably, due to the boundaries of this study, we are unable to include all these developments.

5.7 Summary

We present Deacon, a novel framework that generates efficient SA that processes a single DNN model. Deacon addresses the challenges on DNN SA by the following advanced pipelining techniques: exploiting the computation sequences to avoid pipeline stalls and reduce buffer usage, transforming the original DNN graphs for better resource efficiency and data locality when heterogeneous convolutions and residual connections present, and exploring the enormous design space by a graph

colouring-based pruning method and simulated annealing. While SA is theoretically bounded by CLP in resource efficiency, Deacon can still perform on par with, or even better than prior approaches on compact DNNs, including MobileNet v1, v2, and SqueezeNet. For larger DNNs like ResNet-18, Deacon can scale well over the baseline, even though the designs produced can be less resource efficient than those from related work.

Chapter 6

Conclusion

6.1 Summary of Thesis Achievements

This thesis presents tools for efficient Deep Learning within three different domains. Please refer to Fig. 1.2 for an overview of the thesis structure.

First, in Chapter 3, we describe two tools for efficient Deep Learning training and inference. One is Aegis, which is an improvement over the existing mixed precision training algorithm that particularly aims to address the challenges from gradient underflow. Aegis demonstrates the potential of its gradient scaling algorithm by the empirical evaluation results on various computer vision tasks, which show that Aegis can make mixed precision training stabler and produce models with better accuracy. The other one is SPGC, which is a structured pruning method that prunes a trained network into ones that use more efficient group convolution without losing much accuracy. SPGC works around the combinatorial complexity challenges from this problem and provides a heuristic algorithm based on a block matrix formulation. SPGC also enables neural architecture search based on the number of groups per group convolution. By combining Aegis and SPGC, we can train a Deep Learning model more efficiently, and result in an efficient inference model through pruning. Both tools are published in mainstream Machine Learning conferences and workshops [341, 340], and their codebases are released as open-source projects ¹.

Next, in Chapter 4, we present three frameworks, Polygeist, Polymer, and POLSCA, all of which aim to address the challenges from constructing a fully fledged polyhedral compiler. Since most of the Deep Learning workloads can be described by

¹Aegis: <https://github.com/kumasento/gradient-scaling>, SPGC: <https://github.com/kumasento/gconv-prune>.

polyhedral representations, these efforts can be translated to better Deep Learning runtime performance. Specifically, Polygeist connects C/C++ to MLIR and raises certain parts of a program into polyhedral representation, Polymer leverages external polyhedral schedulers to apply comprehensive polyhedral transformations on MLIR, and POLSCA aims to fill the gap of a missing polyhedral high-level synthesis tool following the compiler transformation based paradigm as Polygeist and Polymer. Evaluated on Polybench/C, we see that Polygeist can compile example programs to executable binaries that have similar performance as those from Clang; Polymer offers about $2.34\times$ and $1.41\times$ times speed-ups over Pluto and Polly when compiled in sequential execution, and $7.54\times$ and $3.26\times$ times speed-ups when in parallel; and POLSCA results in hardware designs that are on average 1.5 times faster than those synthesised by existing high-level synthesis tools without polyhedral transformations. Not only are these projects published in mainstream compiler and hardware conferences [214, 342], but also gained attention from the industry and even become a LLVM incubation project².

Finally, in Chapter 5, we present Deacon, a framework that synthesises a whole Deep Neural Network onto a single FPGA with a streaming architecture. Deacon mainly addresses the pipelining challenges when dealing with different network topologies, especially those with various convolution layers and branches. The advance pipelining techniques can novelly solve the problems appeared when aiming to synthesise a whole residual network and other efficient convolutional neural networks on an FPGA. Compared with prior designs for MobileNets and SqueezeNet, Deacon shows resource/power consumption efficiency improvement of $1.2\times/3.5\times$, and $1.0\times/2.8\times$ averaged by geomean. Deacon can optimise large DNNs such as ResNet-18, with $2.7\times/35.6\times$ efficiency improvement over the baseline. Deacon is originated from our previous publication [339], and is released as an open-source project³.

Reflecting upon the central research inquiry of this dissertation: *What approaches can be adopted to develop tools that enhance the efficiency of Deep Learning?* Throughout this investigation, we have traversed numerous areas including efficient Deep Learning training and inference, compilation, and hardware synthesis. While substantial strides have been made in each realm, we have not yet, and it may even prove infeasible, succeeded in developing a comprehensive framework that integrates these elements seamlessly. However, the idea of an integrated framework, although appealing, may not necessarily be the optimal solution. Each chapter of this dissertation has specified a variety of constraints and assumptions about the specific cases where the proposed tools are most effective. An amalgamation of these tools

²<https://github.com/llvm/Polygeist>

³<https://github.com/kumasento/deacon>

could introduce further limitations, thereby considerably restricting their applicable scenarios. In lieu of attempting to directly answer the initial research question, the proposition that emerges from this thesis is a shift in perspective. We propose that it would be more efficient to design tools that are customised to address the unique requirements at each stage of the Deep Learning workflow.

6.2 Future Work

This thesis presents various tools for efficient Deep Learning, the contents from each chapter are not connected, and one major future work of our thesis is to demonstrate the potential from these tools by combining them together. Specifically for each tool:

- Regarding future work of Aegis, we want to evaluate gradient scaling on other tasks and models, especially those for Natural Language Processing, as well as extending it to even lower-precision representations such as 8-bit floating-point, fixed-point, etc.
- For SPGC, future work includes exploring different importance criteria to improve the quality of explored models, tuning the pruning hyperparameters to achieve higher model accuracy after the fine-tuning phase, and investigating the contribution of the regularisation effect on the better accuracy from CondenseNet. It is also possible to utilise NAS to improve our group configuration optimisation solution, since each group configuration basically determines an architecture.
- For Polygeist, as it is part of the LLVM incubation projects, we will make sure it can work smoothly with other components in the LLVM tool-chain and expand its domain to other high-performance computing application suites besides Polybench/C.
- For Polymer, since we only support Pluto at the moment, it will further extend its impact if other polyhedral schedulers are supported, especially those based on isl [286].
- For POLSCA, we will further explore the design space to optimise designs for resource efficiency. It is also interesting to search for the best set of polyhedral scheduler hyper-parameters with POLSCA. We will also adapt POLSCA for other polyhedral schedulers, e.g., isl, to provide alternative solutions to Pluto.

- For Deacon, we will explore other novel topologies and their impact on SA, as well as larger models by scaling their size down using reduced bit-width. In conjunction with Neural Architecture Search (NAS) [67], we can also adapt Deacon to find DNN models that are more suitable for streaming architectures, i.e., the pipeline stages can be better balanced, while maintaining the overall accuracy. We can also leverage NAS to find models that use less parameters and/or fewer bits in data representations when the FPGA boards have limited amounts of on-chip memory.

6.3 Final Thoughts

As Deep Learning techniques become more popular, there is growing demand for better tools that can make Deep Learning practice more efficient. The topics covered by this thesis are what industry cares about, e.g., PyTorch continuously works on improving the support for better mixed precision training⁴ and their approach is aligned with what we propose in Aegis; Polygeist is part of the LLVM incubation community, and MLIR in general is accepted by many Deep Learning frameworks including TensorFlow and PyTorch; and as the FPGAs have more resource capacity, Deacon can synthesise larger networks on to a single FPGA, so that people can benefit from more power efficient acceleration. We believe these approaches will lay the foundation for next-generation Deep Learning tools not just with superior performance and energy efficiency, but also with enhanced ease of use to significantly improve productivity of application builders.

In a long-term, although this thesis provides tools that are built upon the assumption of a heterogeneous system involving CPU/GPU/FPGA, we believe these tools will benefit future generations of systems. For instance, Aegis will be useful for other low-precision data types that future devices may support, since it provides a generic framework to solve the underflow problems under backpropagation. Another example is Polygeist, which is a tool based on polyhedral optimisation that is agnostic to a particular (preferably parallelisable) device.

The tools we present in this thesis are designed in a composable manner. They serve at particular stages in the end-to-end workflow from high-level DNN descriptions to device-specific implementation. They can be aggregated together and customised to create domain-specific tools for different purposes; for example, we can leverage POLSCA (Section 4.4) to directly produce Deacon designs (Chapter 5) from high-level programming languages. Additionally, we can interface these tools to other

⁴https://pytorch.org/docs/stable/notes/amp_examples.html

frameworks. For instance, we can support other polyhedral scheduling algorithms as in [347, 295, 51] in Polygeist or POLSCA, and we can extend Deacon to interface with HeteroCL [162], without changing the core design and algorithm within the tools. Therefore, we believe the tools in this thesis are likely to remain influential in the foreseeable future.

Bibliography

- [1] ChainerCV: A Library for Deep Learning in Computer Vision. <https://github.com/chainer/chainercv>, November 2021.
- [2] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. *arXiv preprint arXiv:1603.04467*, 2016.
- [3] Santosh G. Abraham and B. R. Rau. Efficient design space exploration in PICO. In *Proceedings of the 2000 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, CASES '00, pages 71–79, New York, NY, USA, November 2000. Association for Computing Machinery. ISBN 978-1-58113-338-7. doi: 10.1145/354880.354891.
- [4] Intel Altera. FPGA Architecture, July 2006.
- [5] Corinne Ancourt and François Irigoin. Scanning polyhedra with do loops. In *Proceedings of the third ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 39–50, 1991.
- [6] Irving H. Anellis. Peirce’s Truth-functional Analysis and the Origin of the Truth Table. *History and Philosophy of Logic*, 33(1):87–97, February 2012. ISSN 0144-5340. doi: 10.1080/01445340.2011.621702.
- [7] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una May O’Reilly, and Saman Amarasinghe. OpenTuner:

- An extensible framework for program autotuning. In *PACT*, pages 303–315, 2014. ISBN 978-1-4503-2809-8. doi: 10.1145/2628071.2628092.
- [8] Tom M Apostol. One-Variable Calculus with Introduction to Linear Algebra. *Calculus*, 1:686, 1967. ISSN 00225193. doi: 10.1016/s0022-5193(69)80067-6.
- [9] Matthew Aubury, Ian Page, Geo Randall, Jonathan Saul, and Robin Watts. Handel-C Language Reference Guide, August 1996.
- [10] Utku Aydonat, Shane O’Connell, Davor Capalija, Andrew C Ling, and Gordon R Chiu. An OpenCL(TM) Deep Learning Accelerator on Arria 10. In *FPGA*, 2017. ISBN 978-1-4503-4354-1. doi: 10.1145/3020078.3021738.
- [11] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avizienis, John Wawrzynek, and Krste Asanovic. Chisel: Constructing Hardware in a Scala Embedded Language. In *DAC*, 2012. ISBN 978-1-4503-1199-1. doi: 10.1243/pime_proc_1916_090_011_02.
- [12] Vijay Badrinarayanan, Alex Kendall, and Roberto Cipolla. SegNet: A Deep Convolutional Encoder-Decoder Architecture for Image Segmentation. *PAMI*, 39(12):2481–2495, 2017. ISSN 01628828. doi: 10.1109/TPAMI.2016.2644615.
- [13] Riyadh Baghdadi, Ulysse Beaugnon, Albert Cohen, Tobias Grosser, Michael Kruse, Chandan Reddy, Sven Verdoolaege, Adam Betts, Alastair F. Donaldson, Jeroen Ketema, Javed Absar, Svenvan V. Haastregt, A. Kravets, Anton Lokhmotov, Robert David, and Elnar Hajiyev. PENCIL: A Platform-Neutral Compute Intermediate Language for Accelerator Programming. In *PACT*, pages 138–149. IEEE, 2015. doi: 10.1109/PACT.2015.17.
- [14] 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*, pages 193–205, 2019. ISBN 978-1-72811-436-1. doi: 10.1109/CGO.2019.8661197.
- [15] R Bagnara, PM Hill, and E Zaffanella. The ppl: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. *Dipartimento di Matematica, Università di Parma, Italy, Tech. Rep*, 2006.
- [16] Lin Bai, Yiming Zhao, and Xinming Huang. A CNN Accelerator on FPGA Using Depthwise Separable Convolution. *arXiv:1809.01536 [cs, eess]*, September 2018.

- [17] Ron Banner, Itay Hubara, Elad Hoffer, and Daniel Soudry. Scalable methods for 8-bit training of neural networks. In *NeurIPS*, 2018.
- [18] Cédric Bastoul. Efficient code generation for automatic parallelization and optimization. In *ISPDC*, volume 2, pages 23–30. Citeseer, 2003.
- [19] Cédric Bastoul. Code generation in the polyhedral model is easier than you think. In *PACT*, pages 7–16, 2004.
- [20] Cédric Bastoul. OpenScop: A Specification and a Library for Data Exchange in Polyhedral Compilation Tools. Technical report, 2012.
- [21] AG Baydin, Barak A Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic Differentiation in Machine Learning: A Survey. 2017.
- [22] Mohamed Walid Benabderrahmane, Louis Noël Pouchet, Albert Cohen, and Cédric Bastoul. The polyhedral model is more widely applicable than you think. In *CC*, pages 283–303, 2010. ISBN 3-642-11969-7. doi: 10.1007/978-3-642-11970-5_16.
- [23] James Bergstra and Yoshua Bengio. Random Search for Hyper-Parameter Optimization. *Journal of Machine Learning Research*, 13(10):281–305, 2012. ISSN 1533-7928.
- [24] James Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for Hyper-Parameter Optimization. In *Advances in Neural Information Processing Systems*, volume 24. Curran Associates, Inc., 2011.
- [25] Jesse Bettencourt, Matthew J Johnson, and David Duvenaud. Taylor-Mode Automatic Differentiation for Higher-Order Derivatives in JAX. In *NeurIPS Workshop*, 2019.
- [26] Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. Lava: Hardware design in Haskell. *ACM SIGPLAN Notices*, 34(1):174–184, September 1998. ISSN 0362-1340. doi: 10.1145/291251.289440.
- [27] Pierre Blanchard, Nicholas J. Higham, Florent Lopez, Theo Mary, and Srikara Pranesh. Mixed Precision Block Fused Multiply-Add : Error Analysis and Application to GPU Tensor Cores. *MIMS EPrint: 2019.18*, 2019.
- [28] Uday Bondhugula. High Performance Code Generation in MLIR: An Early Case Study with GEMM. *arXiv:2003.00532 [cs]*, March 2020.

- [29] Uday Bondhugula, J. Ramanujam, and P. Sadayappan. Automatic mapping of nested loops to FPGAS. In *PPoPP*, pages 101–111, 2007. ISBN 1-59593-602-5. doi: 10.1145/1229428.1229446.
- [30] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *PLDI*, 2008.
- [31] Uday Bondhugula, Vinayaka Bandishti, and Irshad Pananilath. Diamond tiling: Tiling techniques to maximize parallelism for stencil computations. *IEEE Transactions on Parallel and Distributed Systems*, 28(5):1285–1298, 2017. ISSN 10459219. doi: 10.1109/TPDS.2016.2615094.
- [32] Uday Kumar Reddy Bondhugula. Effective Automatic parallelization and locality optimization using the Polyhedral model. *ProQuest Dissertations and Theses*, page 190, 2008.
- [33] Léon Bottou. On-line Learning and Stochastic Approximations. In David Saad, editor, *On-Line Learning in Neural Networks*, pages 9–42. Cambridge University Press, first edition, January 1999. ISBN 978-0-521-65263-6 978-0-521-11791-3 978-0-511-56992-0. doi: 10.1017/CBO9780511569920.003.
- [34] Léon Bottou. Stochastic Learning. In Olivier Bousquet, Ulrike von Luxburg, and Gunnar Rätsch, editors, *Advanced Lectures on Machine Learning*, volume 3176, pages 146–168. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004. ISBN 978-3-540-23122-6 978-3-540-28650-9. doi: 10.1007/978-3-540-28650-9_7.
- [35] Leon Bottou. Large-Scale Machine Learning with Stochastic Gradient Descent. *COMPSTAT*, 2010.
- [36] Daniel Brélaz. New methods to color the vertices of a graph. *Communications of the ACM*, 22(4):251–256, April 1979. ISSN 0001-0782. doi: 10.1145/359094.359101.
- [37] R. L. Brooks. On colouring the nodes of a network. *Mathematical Proceedings of the Cambridge Philosophical Society*, 37(2):194–197, April 1941. ISSN 1469-8064, 0305-0041. doi: 10.1017/S030500410002168X.
- [38] Stephen Brown. FPGA Architectural Research:. page 7, 1996.
- [39] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger,

- Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language Models are Few-Shot Learners. In *NeurIPS*, 2020.
- [40] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H. Anderson, Stephen Brown, and Tomasz Czajkowski. LegUp: High-level synthesis for FPGA-based processor/accelerator systems. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '11, pages 33–36, New York, NY, USA, February 2011. Association for Computing Machinery. ISBN 978-1-4503-0554-9. doi: 10.1145/1950413.1950423.
- [41] Andre Xian Ming Chang, Aliasger Zaidy, Vinayak Gokhale, and Eugenio Culurciello. Compiling Deep Learning Models for Custom Hardware Accelerators. *arXiv:1708.00117 [cs]*, December 2017.
- [42] Chun Chen. Polyhedra scanning revisited. *ACM SIGPLAN Notices*, 47(6): 499–508, 2012. ISSN 15232867. doi: 10.1145/2345156.2254123.
- [43] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Haichen Shen, Eddie Yan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: End-to-End Optimization Stack for Deep Learning. *arXiv preprint arXiv:1802.04799*, 2018.
- [44] Yu-Hsin Chen, Tushar Krishna, Joel Emer, and Vivienne Sze. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE Journal of Solid-State Circuits*, 52(1):262–263, 2016. ISSN 01936530. doi: 10.1109/ISSCC.2016.7418007.
- [45] Yukang Chen, Tong Yang, Xiangyu Zhang, Gaofeng Meng, Chunhong Pan, and Jian Sun. DetNAS: Neural Architecture Search on Object Detection. pages 1–11, 2019.
- [46] Jianyi Cheng, Lana Josipović, George A. Constantinides, Paolo Ienne, and John Wickerson. Combining dynamic & static scheduling in high-level synthesis. In *FPGA*, pages 288–298, 2020. ISBN 978-1-4503-7099-8. doi: 10.1145/3373087.3375297.
- [47] Jungwook Choi, Pierce I-Jen Chuang, Zhuo Wang, Swagath Venkataramani, Vijayalakshmi Srinivasan, and Kailash Gopalakrishnan. Bridging the Accuracy Gap for 2-bit Quantized Neural Networks (QNN). pages 1–10, 2018.

- [48] François Chollet. Xception: Deep Learning with Depthwise Separable Convolutions. *CoRR*, 2016. doi: 10.1109/CVPR.2017.195.
- [49] CIRCT. "CIRCT" / Circuit IR Compilers and Tools. <https://github.com/llvm/circt>.
- [50] J. Cong and Sung Kyu Lim. Physical planning with retiming. In *IEEE/ACM International Conference on Computer Aided Design. ICCAD - 2000. IEEE/ACM Digest of Technical Papers (Cat. No.00CH37140)*, pages 2–7, San Jose, CA, USA, 2000. IEEE. ISBN 978-0-7803-6445-5. doi: 10.1109/ICCAD.2000.896441.
- [51] Jason Cong and Jie Wang. PolySA : Polyhedral-Based Systolic Array Auto-Compilation. In *ICCAD*, 2018. ISBN 978-1-4503-5950-4.
- [52] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. BinaryConnect: Training Deep Neural Networks with binary weights during propagations. In *NeurIPS*, pages 1–9, 2015. doi: arXiv:1412.7024.
- [53] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Training deep neural networks with low precision multiplications. In *ICLR Workshop*, 2015. doi: arXiv:1412.7024.
- [54] Matthieu Courbariaux, Itay Hubara, Ran El-Yaniv, and Yoshua Bengio. Binarized Neural Networks: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1. *CoRR*, page 9, 2016.
- [55] Philippe Coussy and Adam Morawiec, editors. *High-Level Synthesis: From Algorithm to Digital Circuit*. Springer, New York, 2008. ISBN 978-1-4020-8587-1.
- [56] Alain Darte and Yves Robert. Constructive methods for scheduling uniform loop nests. *IEEE transactions on Parallel and Distributed Systems*, 5(8): 814–822, 1994.
- [57] Alain Darte and Yves Robert. Affine-by-statement scheduling of uniform and affine loop nests over parametric domains. *Journal of Parallel and Distributed Computing*, 29(1):43–59, 1995.
- [58] Dipankar Das, Naveen Mellempudi, Dheevatsa Mudigere, Dhiraj Kalamkar, Sasikanth Avancha, Kunal Banerjee, Srinivas Sridharan, Karthik Vaidyanathan, Bharat Kaul, Evangelos Georganas, Alexander Heinecke, Pradeep Dubey, Jesus Corbal, Nikita Shustrov, Roma Dubtsov, Evarist

- Fomenko, and Vadim Pirogov. Mixed Precision Training of Convolutional Neural Networks using Integer Operations. In *ICLR*, 2018.
- [59] Eddie C Davis, Michelle Mills Strout, and Catherine Olschanowsky. Transforming loop chains via macro dataflow graphs. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, pages 265–277, 2018.
- [60] James Demmel and Yozo Hida. Accurate floating point summation. Technical report, 2002.
- [61] Johannes Doerfert, Kevin Streit, Sebastian Hack, and Zino Benaissa. Polly’s Polyhedral Scheduling in the Presence of Reductions, 2015.
- [62] Xin Dong, Shangyu Chen, and Sinno Jialin Pan. Learning to Prune Deep Neural Networks via Layer-wise Optimal Brain Surgeon. In *NeurIPS*, 2017.
- [63] John Duchi, Elad Hazan, and Yoram Singer. Adaptive Subgradient Methods for Online Learning and Stochastic Optimization. *Journal of Machine Learning Research*, 12, 2011.
- [64] Christine Eisenbeis and Jean-Claude Sogno. A general algorithm for data dependence analysis. In *Proceedings of the 6th International Conference on Supercomputing, ICS ’92*, pages 292–302, New York, NY, USA, August 1992. Association for Computing Machinery. ISBN 978-0-89791-485-7. doi: 10.1145/143369.143422.
- [65] Conal Elliott. The simple essence of automatic differentiation. *Proceedings of the ACM on Programming Languages*, 2(ICFP):1–29, 2018. ISSN 2475-1421. doi: 10.1145/3236765.
- [66] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Neural Architecture Search: A Survey. pages 1–17, 2018.
- [67] Hongxiang Fan, Martin Ferianc, Shuanglong Liu, Zhiqiang Que, Xinyu Niu, and Wayne Luk. Optimizing FPGA-Based CNN Accelerator Using Differentiable Neural Architecture Search. In *2020 IEEE 38th International Conference on Computer Design (ICCD)*, pages 465–468, Hartford, CT, USA, October 2020. IEEE. ISBN 978-1-72819-710-4. doi: 10.1109/ICCD50377.2020.00085.
- [68] Julius Farkas. Theorie der einfachen ungleichungen. *Journal für die reine und angewandte Mathematik (Crelles Journal)*, 1902(124):1–27, 1902.

- [69] Paul Feautrier. Array Expansion. In *ACM International Conference on Supercomputing 25th Anniversary Volume*, pages 99–111, 1988.
- [70] Paul Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20(1):23–53, 1991. ISSN 08857458. doi: 10.1007/BF01407931.
- [71] Paul Feautrier. Some efficient solutions to the affine scheduling problem. I. One-dimensional time. *International Journal of Parallel Programming*, 21(5): 313–347, 1992. ISSN 08857458. doi: 10.1007/BF01407835.
- [72] Paul Feautrier. Some efficient solutions to the affine scheduling problem. Part II. Multidimensional time. *International Journal of Parallel Programming*, 21(6):389–420, 1992. ISSN 08857458. doi: 10.1007/BF01379404.
- [73] Paul Feautrier. Automatic parallelization in the polytope model. *The Data Parallel Programming Model*, pages 79–103, 1996.
- [74] Michael Figurnov, Aijan Ibraimova, Dmitry Vetrov, and Pushmeet Kohli. PerforatedCNNs: Acceleration through Elimination of Redundant Convolutions. In *NeurIPS*, 2015. ISBN 0-8153-4076-1. doi: 10.1249/FIT.0000000000000198.
- [75] Cheng-Yang Fu, Wei Liu, Ananth Ranga, Ambrish Tyagi, and Alexander C. Berg. DSSD : Deconvolutional Single Shot Detector. 2017.
- [76] Yao Fu, Ephrem Wu, Ashish Sirasao, Sedny Attia, Kamran Khan, and Ralph Wittig. Deep Learning with INT8 Optimization on Xilinx Devices. page 11, 2017.
- [77] Christophe Garcia. Simplifying ConvNets for Fast Learning Simplifying ConvNets for Fast Learning. In *ICANN*, 2012. ISBN 978-3-642-33266-1. doi: 10.1007/978-3-642-33266-1.
- [78] Roman Gareev, Tobias Grosser, and Michael Kruse. High-Performance Generalized Tensor Operations: A Compiler-Oriented Approach. *ACM Transactions on Architecture and Code Optimization*, 15(3):34:1–34:27, September 2018. ISSN 1544-3566. doi: 10.1145/3235029.
- [79] M. R. Garey, D. S. Johnson, and L. Stockmeyer. Some simplified NP-complete graph problems. *Theoretical Computer Science*, 1(3):237–267, February 1976. ISSN 0304-3975. doi: 10.1016/0304-3975(76)90059-1.
- [80] R. S. Garfinkel and K. C. Gilbert. The Bottleneck Traveling Salesman Problem: Algorithms and Probabilistic Analysis. *Journal of the ACM*, 25(3): 435–448, 1978. ISSN 00045411. doi: 10.1145/322077.322086.

- [81] Sylvain Girbal, Nicolas Vasilache, Cédric Bastoul, Albert Cohen, David Par-ello, Marc Sigler, and Olivier Temam. Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies. *International Journal of Parallel Programming*, 34(3):261–317, 2006. ISSN 08857458. doi: 10.1007/s10766-006-0012-3.
- [82] Ross Girshick. Fast R-CNN. In *Proceedings of the IEEE International Conference on Computer Vision*, volume 11-18-Dece, pages 1440–1448, 2016. ISBN 978-1-4673-8391-2. doi: 10.1109/ICCV.2015.169.
- [83] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *AISTATS*, 2010. ISBN 978-1-937284-27-5. doi: 10.1.1.207.2059.
- [84] David Goldberg. *What Every Computer Scientist Should Know about Floating-Point Arithmetic*, volume 23. 1991. ISBN 00000000000000. doi: 10.1145/103162.103163.
- [85] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. 2016.
- [86] Google. TensorFlow MLIR. <https://www.tensorflow.org/mlir>, 2020.
- [87] Georgios Goumas, Maria Athanasaki, and Nectarios Koziris. An Efficient Code Generation Technique for Tiled Iteration Spaces. *IEEE Transactions on Parallel and Distributed Systems*, 14(10):1021–1034, 2003. ISSN 10459219. doi: 10.1109/TPDS.2003.1239870.
- [88] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, Kaiming He, Yangqing Jia Kaiming, and He Facebook. Accurate, large minibatch SGD: Training ImageNet in 1 Hour. 2017. ISSN 10495258. doi: 10.1124/dmd.107.016501.CYP3A4-Mediated.
- [89] Martin Griebel and Christian Lengauer. The loop parallelizer loopo. In *Proc. Sixth Workshop on Compilers for Parallel Computers*, volume 21, pages 311–320. Citeseer, 1996.
- [90] Tobias Grosser, Hongbin Zheng, Raghesh Aloor, Andreas Simbürger, Armin Größlinger, and Louis-Noël Pouchet. Polly - Polyhedral optimization in LLVM. In *IMPACT*, page None, 2011.
- [91] Tobias Grosser, Groesslinger Armin, and Christian Lengauer. Polly - Performing Polyhedral Optimizations on a Low-level Intermediate Representa-

- tion. *Parallel Processing Letters*, 22(04):1250010, December 2012. ISSN 0129-6264. doi: 10.1142/S0129626412500107.
- [92] Tobias Grosser, Albert Cohen, Paul HJ Kelly, J Ramanujam, Ponuswamy Sadayappan, and Sven Verdoolaege. Split tiling for gpus: automatic parallelization using trapezoidal tiles. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, pages 24–31, 2013.
- [93] Tobias Grosser, Albert Cohen, Justin Holewinski, P. Sadayappan, and Sven Verdoolaege. Hybrid Hexagonal/Classical Tiling for GPUs. *CGO '14 Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 66–75, 2014. doi: 10.1145/2544137.2544160.
- [94] Tobias Grosser, Sven Verdoolaege, Albert Cohen, and P Sadayappan. The relation between diamond tiling and hexagonal tiling. *Parallel Processing Letters*, 24(03):1441002, 2014.
- [95] Tobias Grosser, J. Ramanujam, Louis-Noel Pouchet, P. Sadayappan, and Sebastian Pop. Optimistic Delinearization of Parametrically Sized Arrays. In *Proceedings of the 29th ACM on International Conference on Supercomputing, ICS '15*, pages 351–360, New York, NY, USA, June 2015. Association for Computing Machinery. ISBN 978-1-4503-3559-1. doi: 10.1145/2751205.2751248.
- [96] Tobias Grosser, Sven Verdoolaege, and Albert Cohen. Polyhedral AST Generation Is More Than Scanning Polyhedra. *ACM Transactions on Programming Languages and Systems*, 37(4):1–50, 2015. ISSN 01640925. doi: 10.1145/2743016.
- [97] Tobias Grosser, Johannes Doerfert, and Zino Benaissa. Analyzing and Optimizing your Loops with Polly, 2016.
- [98] Thorsten Grötzer, Stan Liao, Grant Martin, and Stuart Swan. *System Design with SystemC*. 2002.
- [99] Yijin Guan, Hao Liang, Ningyi Xu, Wenqiang Wang, Shaoshuai Shi, Xi Chen, Guangyu Sun, Wei Zhang, and Jason Cong. FP-DNN: An Automated Framework for Mapping Deep Neural Networks onto FPGAs with RTL-HLS Hybrid Templates. In *FCCM*, 2017. ISBN 978-1-5386-4037-1. doi: 10.1109/FCCM.2017.25.
- [100] Yijin Guan, Zhihang Yuan, Guangyu Sun, Jason Cong, and Los Angeles. FPGA-based Accelerator for Long Short-Term Memory Recurrent Neural Networks. In *ASP-DAC*, 2017.

- [101] Qingbei Guo, Xiao-Jun Wu, Josef Kittler, and Zhiquan Feng. Self-grouping convolutional neural networks. *Neural Networks*, 2020. ISSN 0030-4018. doi: 10.1016/j.knosys.2020.105893.
- [102] Shaori Guo and Wayne Luk. Compiling Ruby into FPGAs. *FPL*, page 10, 1995.
- [103] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. Deep Learning with Limited Numerical Precision. In *ICML*, pages 1737–1746, 2015. ISBN 978-1-5108-1058-7. doi: 10.1109/72.80206.
- [104] Tobias Gysi, Christoph Mueller, Oleksandr Zinenko, Stephan Andreas Herhut, Eddie Davis, Tobias Wicky, Oliver Fuhrer, Torsten Hoefer, and Tobias Grosser. Domain-Specific Multi-Level Rewriting for HPC: A Case Study with MLIR. *ACM Transactions on Architecture and Code Optimization*, 4: 51:1–51:23, 2021.
- [105] Song Han, Huizi Mao, and William J. Dally. Deep Compression - Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding. In *ICLR*, 2016.
- [106] Boris Hanin. Which Neural Net Architectures Give Rise To Exploding and Vanishing Gradients? In *NeurIPS*, 2018.
- [107] Babak Hassibi and David G. Stork. Second order derivatives for network pruning: Optimal brain surgeon. *NeurIPS*, pages 164–171, 1992.
- [108] Babak Hassibi, David G. Stork, and Gregory J. Wolff. Optimal brain surgeon and general network pruning. In *IEEE International Conference on Neural Networks*, 1993. ISBN 0-7803-0999-5. doi: 10.1109/ICNN.1993.298572.
- [109] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *ICCV*, pages 1026–1034, 2015. ISBN 978-1-4673-8391-2. doi: 10.1109/ICCV.2015.123.
- [110] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. In *CVPR*, 2016.
- [111] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Identity Mappings in Deep Residual Networks. In *ECCV*, 2016.
- [112] Nicholas J. Higham. The accuracy of floating point summation. *SIAM J. Scientific Computing*, 14(4):783–799, 1993.

- [113] Nicholas J Higham, Srikara Pranesh, and Mawussi Zounon. Squeezing a Matrix into Half Precision, with an Application to Solving Linear Systems. *SIAM J. Sci. Comput.*, 41(5):585–602, 2019.
- [114] Geoffrey E. Hinton. Neural Networks for Machine Learning, Lecture 6. http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf.
- [115] Robert. B. Hitchcock, Gordon L. Smith, and David D. Cheng. Timing Analysis of Computer Hardware. *IBM Journal of Research and Development*, 26(1): 100–105, January 1982. ISSN 0018-8646, 0018-8646. doi: 10.1147/rd.261.0100.
- [116] Edin Hodzic and Weijia Shang. On time optimal supernode shape. *IEEE Transactions on Parallel and Distributed Systems*, 13(12):1220–1233, 2002.
- [117] Elad Hoffer, Ron Banner, Itay Golan, and Daniel Soudry. Norm matters: Efficient and accurate normalization schemes in deep networks. In *NeurIPS*, 2018.
- [118] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. *CoRR*, 2017. doi: arXiv:1704.04861.
- [119] Hengyuan Hu, Rui Peng, Yu-Wing Tai, and Chi-Keung Tang. Network Trimming: A Data-Driven Neuron Pruning Approach towards Efficient Deep Architectures. 2016.
- [120] Jie Hu, Li Shen, Samuel Albanie, Gang Sun, and Enhua Wu. Squeeze-and-Excitation Networks. *TPAMI*, May 2019.
- [121] Gao Huang, Zhuang Liu, Laurens van der Maaten, and Kilian Q. Weinberger. Densely Connected Convolutional Networks. *CoRR*, abs/1608.0, 2016. ISSN 0002-9645. doi: 10.1109/CVPR.2017.243.
- [122] Gao Huang, Shichen Liu, Laurens van der Maaten, and Kilian Q. Weinberger. CondenseNet: An Efficient DenseNet using Learned Group Convolutions. *CoRR*, abs/1711.0, 2017.
- [123] Jonathan Huang, Vivek Rathod, Chen Sun, Menglong Zhu, Anoop Korattikara, Alireza Fathi, Ian Fischer, Zbigniew Wojna, Yang Song, Sergio Guadarrama, and Kevin Murphy. Speed/accuracy trade-offs for modern convolutional object detectors. 2016.

- [124] Kyuyeon Hwang and Wonyong Sung. Fixed-point feedforward deep neural network design using weights +1, 0, and -1. In *2014 IEEE Workshop on Signal Processing Systems (SiPS)*, pages 1–6, Belfast, United Kingdom, October 2014. IEEE. ISBN 978-1-4799-6588-5. doi: 10.1109/SiPS.2014.6986082.
- [125] Forrest N. Iandola, Matthew W. Moskewicz, Khalid Ashraf, Song Han, William J. Dally, and Kurt Keutzer. SqueezeNet: AlexNet-Level Accuracy with 50x Fewer Parameters and 1MB Model Size. *arXiv*, pages 1–5, 2016. doi: 10.1007/978-3-319-24553-9.
- [126] Intel. Implementing FPGA Design with the OpenCL Standard. Technical report, 2013.
- [127] Intel. Intel High Level Synthesis Compiler Pro Edition: User Guide, October 2021.
- [128] Yani Ioannou, Duncan Robertson, Roberto Cipolla, and Antonio Criminisi. Deep Roots: Improving CNN Efficiency with Hierarchical Filter Groups. In *CVPR*, 2017. doi: 10.1109/CVPR.2017.633.
- [129] Sergey Ioffe and Christian Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. In *ICML*, pages 448–456, 2015. ISBN 978-0-87421-656-1. doi: 10.1007/s13398-014-0173-7.2.
- [130] F. Irigoin and R Triolet. Supernode Partitioning. In *POPL*, pages 319–329, 1988.
- [131] 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 *CVPR*, 2018. doi: 10.1109/CVPR.2018.00286.
- [132] Rajeev Jayaraman. (When) Will FPGAs Kill ASICs?, 2001.
- [133] Yangqing Jia. *Learning Semantic Image Representations at a Large Scale*. PhD thesis, 2014.
- [134] Tian Jin, Gheorghe-Teodor Bercea, Tung D. Le, Tong Chen, Gong Su, Haruki Imai, Yasushi Negishi, Anh Leu, Kevin O’Brien, Kiyokuni Kawachiya, and Alexandre E. Eichenberger. Compiling ONNX Neural Network Models Using MLIR. *arXiv:2008.08272 [cs]*, September 2020.
- [135] Lana Josipović, Radhika Ghosal, and Paolo Ienne. Dynamically scheduled high-level synthesis. In *FPGA*, pages 127–136, 2018. ISBN 978-1-4503-5614-5. doi: 10.1145/3174243.3174264.

- [136] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-Luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon Mackean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagara-jan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omer-nick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *ISCA*, 2017.
- [137] Pierre Jouvelot and Babak Dehbonei. A unified semantic approach for the vectorization and parallelization of generalized reductions. In *Proceedings of the 3rd International Conference on Supercomputing*, ICS '89, pages 186–194, New York, NY, USA, June 1989. Association for Computing Machinery. ISBN 978-0-89791-309-6. doi: 10.1145/318789.318810.
- [138] Andrew B. Kahng, Jens Lienig, Igor L. Markov, and Jin Hu. *VLSI Physical Design: From Graph Partitioning to Timing Closure*. Springer Publishing Company, Incorporated, 1st edition, 2011. ISBN 978-90-481-9590-9.
- [139] Dhiraj Kalamkar, Dheevatsa Mudigere, Naveen Mellempudi, Dipankar Das, Kunal Banerjee, Sasikanth Avancha, Dharma Teja Vooturi, Nataraj Jammala-madaka, Jianyu Huang, Hector Yuen, Jiyan Yang, Jongsoo Park, Alexander Heinecke, Evangelos Georganas, Sudarshan Srinivasan, Abhishek Kundu, Misha Smelyanskiy, Bharat Kaul, and Pradeep Dubey. A Study of BFLOAT16 for Deep Learning Training. 2019.
- [140] Nal Kalchbrenner, Edward Grefenstette, and Phil Blunsom. A convolutional neural network for modelling sentences. In *ACL*, pages 655–665, 2014. ISBN 978-1-937284-72-5. doi: 10.3115/v1/p14-1062.
- [141] Richard M Karp, Raymond E Miller, and Shmuel Winograd. The organiza-tion of computations for uniform recurrence equations. *Journal of the ACM (JACM)*, 14(3):563–590, 1967.

- [142] Ryan Kastner, Janarбек Matai, and Stephen Neuendorffer. *Parallel Programming for FPGAs*. 2018.
- [143] Victor J. Katz. *The Mathematics of Egypt, Mesopotamia, China, India, and Islam*. Sun, 08/05/2007 - 12:00. ISBN 978-0-691-11485-9.
- [144] W. Kelly and W. Pugh. A unifying framework for iteration reordering transformations. In *Proceedings 1st International Conference on Algorithms and Architectures for Parallel Processing*, volume 1, pages 153–162 vol.1, 1995. doi: 10.1109/ICAPP.1995.472180.
- [145] W. Kelly, W. Pugh, and E. Rosser. Code generation for multiple mappings. *Proceedings Frontiers '95. The Fifth Symposium on the Frontiers of Massively Parallel Computation*, pages 332–341, 1995. doi: 10.1109/FMPC.1995.380437.
- [146] Wayne Kelly. *Optimization within a Unified Transformation Framework*. PhD thesis, 1996.
- [147] Wayne Kelly and William Pugh. A framework for unifying reordering transformations. Technical report, 1998.
- [148] Yoon Kim. Convolutional neural networks for sentence classification. In *EMNLP 2014 - 2014 Conference on Empirical Methods in Natural Language Processing, Proceedings of the Conference*, pages 1746–1751, 2014. ISBN 978-1-937284-96-1. doi: 10.3115/v1/d14-1181.
- [149] Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. In *ICLR*, 2015. ISBN 978-1-4503-0072-8. doi: <http://doi.acm.org.ezproxy.lib.ucf.edu/10.1145/1830483.1830503>.
- [150] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by Simulated Annealing. *Science*, 220(4598), May 1983.
- [151] Justin Knapheide, Benno Stabernack, and Maximilian Kuhnke. A High Throughput MobileNetV2 FPGA Implementation Based on a Flexible Architecture for Depthwise Separable Convolution. In *2020 30th International Conference on Field-Programmable Logic and Applications (FPL)*, pages 277–283, August 2020. doi: 10.1109/FPL50879.2020.00053.
- [152] David Koeplinger, Matthew Feldman, Raghu Prabhakar, Yaqi Zhang, Stefan Hadjis, Ruben Fiszal, Tian Zhao, Luigi Nardi, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. Spatial: A language and compiler for application accelerators. In *PLDI*, pages 296–311, 2018. ISBN 978-1-4503-5698-5. doi: 10.1145/3192366.3192379.

- [153] Konrad Komisarczyk, Lorenzo Chelini, Kanishkan Vadivel, Roel Jordans, and Henk Corporaal. Pet-to-mlir: A polyhedral front-end for mlir. In *2020 23rd Euromicro Conference on Digital System Design (DSD)*, pages 551–556. IEEE, 2020.
- [154] Alexandros Kouris, Stylianos I. Venieris, and Christos-Savvas Bouganis. CascadeCNN: Pushing the Performance Limits of Quantisation in Convolutional Neural Networks. In *FPL*, 2018.
- [155] Alex Krizhevsky. Learning Multiple Layers of Features from Tiny Images. Technical report, 2009.
- [156] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In *NeurIPS*, 2012. ISBN 978-1-62748-003-1.
- [157] Michael Kruse and Tobias Grosser. DeLICM: Scalar dependence removal at zero memory cost. In *CGO*, pages 241–253, 2018. ISBN 978-1-4503-5617-6. doi: 10.1145/3168815.
- [158] Hana Kubátová. Finite State Machine Implementation in FPGAs. In Marian Andrzej Adamski, Andrei Karatkevich, and Marek Wegrzyn, editors, *Design of Embedded Control Systems*, pages 175–184. Springer US, Boston, MA, 2005. ISBN 978-0-387-28327-2. doi: 10.1007/0-387-28327-7_15.
- [159] Oleksii Kuchaiev, Boris Ginsburg, Igor Gitman, Vitaly Lavrukhin, Jason Li, Huyen Nguyen, Carl Case, and Paulius Micikevicius. Mixed-Precision Training for NLP and Speech Recognition with OpenSeq2Seq. In *Workshop for Natural Language Processing Open Source Software (NLP-OSS), Co-Located with ACL2018*, 2018.
- [160] Aditya Kumar and Sebastian Pop. Scop detection: a fast algorithm for industrial compilers. In *IMPACT 2016 Sixth International Workshop on Polyhedral Compilation Techniques*. Citeseer, 2016.
- [161] H. T. Kung. Why Systolic Architectures? *Computer*, 15(1):37–46, 1982. ISSN 00189162. doi: 10.1109/MC.1982.1653825.
- [162] Yi-Hsiang Lai, Yuze Chi, Yuwei Hu, Jie Wang, Cody Hao Yu, Yuan Zhou, Jason Cong, and Zhiru Zhang. Heterocl: A multi-paradigm programming infrastructure for software-defined reconfigurable computing. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 242–251, 2019.

- [163] Leslie Lamport. The hyperplane method for an array computer. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 24 LNCS:113–131, 1975. ISSN 16113349. doi: 10.1007/3-540-07135-0_114.
- [164] Rasmus Wriedt Larsen and Troels Henriksen. Strategies for regular segmented reductions on GPU. In *Proceedings of the 6th ACM SIGPLAN International Workshop on Functional High-Performance Computing*, FHPC 2017, pages 42–52, New York, NY, USA, September 2017. Association for Computing Machinery. ISBN 978-1-4503-5181-2. doi: 10.1145/3122948.3122952.
- [165] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Life-long Program Analysis & Transformation. In *CGO*, 2014.
- [166] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. MLIR: A Compiler Infrastructure for the End of Moore’s Law. 2020.
- [167] 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, February 2021. doi: 10.1109/CGO51591.2021.9370308.
- [168] Andrew Lavin and Scott Gray. Fast Algorithms for Convolutional Neural Networks. In *CVPR*, 2016. ISBN 978-1-4673-8851-1. doi: 10.1109/CVPR.2016.435.
- [169] Hervé Le Verge, Christophe Mauras, and Patrice Quinton. The alpha language and its use for the design of systolic arrays. *Journal of VLSI signal processing systems for signal, image and video technology*, 3(3):173–182, 1991.
- [170] Vadim Lebedev and Victor Lempitsky. Fast ConvNets Using Group-Wise Brain Damage. In *CVPR*, pages 2554–2564, 2016. ISBN 978-1-4673-8851-1. doi: 10.1109/CVPR.2016.280.
- [171] Yann LeCun, John S Denker, and Sara a Solla. Optimal Brain Damage. In *NeurIPS*, pages 598–605, 1990. ISBN 1-55860-100-7. doi: 10.1.1.32.7223.
- [172] Charles E. Leiserson and James B. Saxe. Retiming Synchronous Circuitry. *Algorithmica*, pages 5–35, 1991.

- [173] Charles E. Leiserson, Flavio M. Rose, and James B. Saxe. Optimizing Synchronous Circuitry by Retiming (Preliminary Version). In Randal Bryant, editor, *Third Caltech Conference on Very Large Scale Integration*, pages 87–116, Berlin, Heidelberg, 1983. Springer. ISBN 978-3-642-95432-0. doi: 10.1007/978-3-642-95432-0_7.
- [174] Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. Pruning Filters for Efficient Convnets. In *ICLR*, 2017.
- [175] Amy W. Lim and Monica S. Lam. Maximizing parallelism and minimizing synchronization with affine partitions. *Parallel Computing*, 24(3-4):445–475, 1998. ISSN 01678191. doi: 10.1016/S0167-8191(98)00021-0.
- [176] Amy W. Lim, Shih-Wei Liao, and Monica S. Lam. Blocking and array contraction across arbitrarily nested loops using affine partitioning. *ACM SIGPLAN Notices*, 36(7):103–112, 2001. ISSN 03621340. doi: 10.1145/568014.379586.
- [177] Chenxi Liu, Barret Zoph, Maxim Neumann, Jonathon Shlens, Wei Hua, Li Jia Li, Li Fei-Fei, Alan Yuille, Jonathan Huang, and Kevin Murphy. Progressive Neural Architecture Search. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 11205 LNCS:19–35, 2018. ISSN 16113349. doi: 10.1007/978-3-030-01246-5_2.
- [178] Hanxiao Liu, Karen Simonyan, Oriol Vinyals, Chrisantha Fernando, and Koray Kavukcuoglu. Hierarchical Representations for Efficient Architecture Search. In *ICLR*, 2018.
- [179] Hanxiao Liu, Karen Simonyan, and Yiming Yang. DARTS: Differentiable Architecture Search. In *ICLR*, 2019.
- [180] Junyi Liu, John Wickerson, and George A. Constantinides. Loop Splitting for Efficient Pipelining in High-Level Synthesis. In *FCCM*, pages 72–79. IEEE, 2016. ISBN 978-1-5090-2356-1. doi: 10.1109/FCCM.2016.27.
- [181] Junyi Liu, John Wickerson, Samuel Bayliss, and George A. Constantinides. Polyhedral-based dynamic loop pipelining for high-level synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(9):1802–1815, 2018. ISSN 02780070. doi: 10.1109/TCAD.2017.2783363.
- [182] Shuanglong Liu and Wayne Luk. Towards an Efficient Accelerator for DNN-Based Remote Sensing Image Segmentation on FPGAs. In *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, pages 187–193, September 2019. doi: 10.1109/FPL.2019.00037.

- [183] Shuanglong Liu, Hongxiang Fan, Martin Ferianc, Xinyu Niu, Huifeng Shi, and Wayne Luk. Toward Full-Stack Acceleration of Deep Convolutional Neural Networks on FPGAs. *IEEE Transactions on Neural Networks and Learning Systems*, pages 1–14, 2021. ISSN 2162-237X, 2162-2388. doi: 10.1109/TNNLS.2021.3055240.
- [184] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C Berg. SSD: Single Shot MultiBox Detector. In *ECCV*, 2016.
- [185] Zhuang Liu, Jianguo Li, Zhiqiang Shen, Gao Huang, Shoumeng Yan, and Changshui Zhang. Learning Efficient Convolutional Networks through Network Slimming. In *ICCV*, pages 2755–2763, 2017. ISBN 978-1-5386-1032-9. doi: 10.1109/ICCV.2017.298.
- [186] Zhuang Liu, Mingjie Sun, Tinghui Zhou, Gao Huang, and Trevor Darrell. Rethinking the Value of Network Pruning. In *ICLR*, 2019. ISBN 0006-3606. doi: 10.1111/j.1744-7429.2006.00114.x.
- [187] Guoping Long, Jun Yang, Kai Zhu, and Wei Lin. FusionStitching: Deep Fusion and Code Generation for Tensorflow Computations on GPUs. 2018.
- [188] Ilya Loshchilov and Frank Hutter. SGDR: Stochastic Gradient Descent with Warm Restarts. In *ICLR*, 2017.
- [189] Liqiang Lu and Yun Liang. SpWA: An Efficient Sparse Winograd Convolutional Neural Networks Accelerator on FPGAs. In *DAC*, pages 1–6. IEEE, 2018. ISBN 978-1-4503-5700-5. doi: 10.1109/dac.2018.8465842.
- [190] Jian Hao Luo, Jianxin Wu, and Weiyao Lin. ThiNet: A Filter Level Pruning Method for Deep Neural Network Compression. In *ICCV*, 2017. ISBN 978-1-5386-1032-9. doi: 10.1109/ICCV.2017.541.
- [191] Yufei Ma, Naveen Suda, Yu Cao, Jae-sun Seo, and Sarma Vrudhula. Scalable and Modularized RTL Compilation of Convolutional Neural Networks onto FPGA. In *FPL*. IEEE, August 2016. ISBN 978-2-8399-1844-2. doi: 10.1109/FPL.2016.7577356.
- [192] Yufei Ma, Yu Cao, Sarma Vrudhula, and Jae-sun Seo. Optimizing Loop Operation and Dataflow in FPGA Acceleration of Deep Convolutional Neural Networks. In *FPGA*, 2017. ISBN 978-1-4503-4354-1. doi: 10.1145/3020078.3021736.

- [193] Yufei Ma, Yu Cao, Sarma Vrudhula, and Jae-sun Seo. An Automatic RTL Compiler for High-Throughput FPGA Implementation of Diverse Deep Convolutional Neural Networks. In *FPL*, 2017. ISBN 978-90-90-30428-1. doi: 10.23919/FPL.2017.8056824.
- [194] N. Maheshwari and S. Sapatnekar. Efficient retiming of large circuits. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 6(1):74–83, March 1998. ISSN 1557-9999. doi: 10.1109/92.661250.
- [195] Kingshuk Majumder and Uday Bondhugula. HIR: An MLIR-based Intermediate Representation for Hardware Accelerator Description. *arXiv:2103.00194 [cs]*, February 2021.
- [196] Michael A. Malcolm. On Accurate Floating-Point Summation. *Communications of the ACM*, 14(11):731–736, 1971. ISSN 15577317. doi: 10.1145/362854.362889.
- [197] Alwani Manoj, Chen Han, and Ferdman Michael. Fused-Layer CNN Accelerators. In *MICRO*, 2016. ISBN 978-1-5090-3508-3.
- [198] 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. In *NeurIPS*, 2017. ISBN 978-1-5386-0733-6. doi: 10.1109/CVPRW.2017.241.
- [199] Stefano Markidis, Steven Wei Der Chien, Erwin Laure, Ivy Bo Peng, and Jeffrey S. Vetter. NVIDIA tensor core programmability, performance & precision. In *IPDPSW*, 2018. ISBN 978-1-5386-5555-9. doi: 10.1109/IPDPSW.2018.00091.
- [200] Marc Masana, Joost Van De Weijer, Luis Herranz, Andrew D. Bagdanov, and Jose M. Alvarez. Domain-Adaptive Deep Network Compression. In *ICCV*, pages 4299–4307, 2017. ISBN 978-1-5386-1032-9. doi: 10.1109/ICCV.2017.460.
- [201] C. Maura, P. Quinton, S. Rajopadhye, and Y. Saouter. Scheduling affine parameterized recurrences by means of. In *Proceedings of the International Conference on Application Specific Array Processors*, pages 100–110, 1990. doi: 10.1109/ASAP.1990.145447.
- [202] Maxeler. Maxcompiler White Paper. Technical report, 2011.
- [203] Alexander McCaskey and Thien Nguyen. A MLIR Dialect for Quantum Assembly Languages. *arXiv:2101.11365 [quant-ph]*, January 2021.

- [204] M.C. McFarland, A.C. Parker, and R. Camposano. The high-level synthesis of digital systems. *Proceedings of the IEEE*, 78(2):301–318, February 1990. ISSN 1558-2256. doi: 10.1109/5.52214.
- [205] T.M. McWilliams. Verification of Timing Constraints on Large Digital Systems. In *17th Design Automation Conference*, pages 139–147, June 1980. doi: 10.1109/DAC.1980.1585240.
- [206] Benoit Meister, Nicolas Vasilache, David Wohlford, Muthu Manikandan Baskaran, Allen Leung, and Richard Lethin. R-Stream Compiler. In David Padua, editor, *Encyclopedia of Parallel Computing*, pages 1756–1765. Springer US, Boston, MA, 2011. ISBN 978-0-387-09766-4. doi: 10.1007/978-0-387-09766-4_515.
- [207] Benoît Meister, Eric Papenhausen Akai Kaeru, and Benoît Pradelle Silexica. Polyhedral Tensor Schedulers. In *2019 International Conference on High Performance Computing Simulation (HPCS)*, pages 504–512, July 2019. doi: 10.1109/HPCS48598.2019.9188233.
- [208] Naveen Mellempudi, Sudarshan Srinivasan, Dipankar Das, and Bharat Kaul. Mixed Precision Training With 8-bit Floating Point. 2019.
- [209] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, and Hao Wu. Mixed Precision Training. In *ICLR*, 2018.
- [210] MLIR. MLIR Affine. <https://mlir.llvm.org/docs/Dialects/Affine/>.
- [211] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharmashan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, February 2015. ISSN 1476-4687. doi: 10.1038/nature14236.
- [212] Pavlo Molchanov, Stephen Tyree, Tero Karras, Timo Aila, and Jan Kautz. Pruning Convolutional Neural Networks for Resource Efficient Transfer Learning. In *ICLR*, 2017. ISBN 978-1-61197-068-5. doi: 10.1051/0004-6361/201527329.
- [213] Antoine Morvan, Steven Derrien, and Patrice Quinton. Polyhedral bubble insertion: A method to improve nested loop pipelining for high-level synthesis.

- IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 32(3):339–352, 2013. ISSN 02780070. doi: 10.1109/TCAD.2012.2228270.
- [214] William S Moses, Lorenzo Chelini, Ruizhe Zhao, and Oleksandr Zinenko. Polygeist : Raising C to Polyhedral MLIR. In *PACT*, 2021.
- [215] Jean-michel Muller, Nicolas Brisebarre, Florent De Dinechin, Claude-pierre Jeannerod, Guillaume Melquiond, Nathalie Revol, and Serge Torres. *Handbook of Floating-Point Arithmetic*. 2010. ISBN 978-0-8176-4704-9.
- [216] Kevin P. Murphy. *Machine Learning: A Probabilistic Perspective*. Adaptive Computation and Machine Learning Series. MIT Press, Cambridge, MA, 2012. ISBN 978-0-262-01802-9.
- [217] Syed Waqar Nabi and Wim Vanderbauwhede. FPGA design space exploration for scientific HPC applications using a fast and accurate cost model based on roofline analysis. *Journal of Parallel and Distributed Computing*, 133:407–419, November 2019. ISSN 0743-7315. doi: 10.1016/j.jpdc.2017.05.014.
- [218] Luigi Nardi, David Koeplinger, and Kunle Olukotun. Practical Design Space Exploration. 2018.
- [219] Rachit Nigam, Samuel Thomas, Zhijing Li, and Adrian Sampson. A Compiler Infrastructure for Accelerator Generators. *arXiv:2102.09713 [cs]*, February 2021.
- [220] Wei Niu, Jiexiong Guan, Yanzhi Wang, Gagan Agrawal, and Bin Ren. DNN-Fusion: Accelerating deep neural networks execution with advanced operator fusion. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, pages 883–898, New York, NY, USA, June 2021. Association for Computing Machinery. ISBN 978-1-4503-8391-2. doi: 10.1145/3453483.3454083.
- [221] NVIDIA. NVIDIA Tesla V100 GPU Architecture. Technical report, 2017.
- [222] NVIDIA. NVIDIA A100 Tensor Core GPU Architecture. Technical report, 2020.
- [223] Wei Pan, Hao Dong, and Yike Guo. DropNeuron: Simplifying the Structure of Deep Neural Networks. In *NeurIPS*, 2016.
- [224] 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 *ISCA*, 2017. ISBN 978-1-4503-4892-8. doi: 10.1145/3079856.3080254.
- [225] Adam Paszke, Gregory Chanan, Zeming Lin, Sam Gross, Edward Yang, Luca Antiga, and Zachary Devito. Automatic differentiation in PyTorch. In *NeurIPS Workshop*, 2017. ISBN 978-85-7811-079-6. doi: 10.1017/CBO9781107707221.009.
- [226] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *NeurIPS*, 2019.
- [227] Bo Peng, Wenming Tan, Zheyang Li, Shun Zhang, Di Xie, and Shiliang Pu. Extreme Network Compression via Filter Group Approximation. In *ECCV*, 2018. doi: 10.1007/978-3-030-01237-3_19.
- [228] Jacques Pienaar. MLIR in TensorFlow Ecosystem, 2020.
- [229] Arjun Pitchanathan, Christian Ulmann, Michel Weber, Torsten Hoefer, and Tobias Grosser. Fpl: Fast presburger arithmetic through transprecision. *Proceedings of the ACM on Programming Languages*, 5(OOPSLA), 2021.
- [230] Louis-Noël Pouchet, Peng Zhang, P. Sadayappan, and Jason Cong. Polyhedral-based data reuse optimization for configurable computing. In *FPGA*, pages 29–38, 2013. ISBN 978-1-4503-1887-7. doi: 10.1145/2435264.2435273.
- [231] Benot Pradelle, Benot Meister, Muthu Baskaran, Jonathan Springer, and Richard Lethin. Polyhedral Optimization of TensorFlow Computation Graphs. 2017. doi: 10.13140/RG.2.2.10754.58565.
- [232] William Pugh. The Omega test: A fast and practical integer programming algorithm for dependence analysis. In *Supercomputing*, pages 4–13, 1991. ISBN 0-89791-459-7. doi: 10.1145/125826.125848.
- [233] Zhiqiang Que, Hiroki Nakahara, Hongxiang Fan, He Li, Jiuxi Meng, Kuen Hung Tsoi, Xinyu Niu, Eriko Nurvitadhi, and Wayne Luk. Remarn: A reconfigurable multi-threaded multi-core accelerator for recurrent neural networks. *TRETS*, may 2022.

- [234] Fabien Quilleré, Sanjay Rajopadhye, and Doran Wilde. Generation of efficient nested loops from polyhedra. *International Journal of Parallel Programming*, 28(5):469–498, 2000. ISSN 08857458. doi: 10.1023/A:1007554627716.
- [235] Patrice Quinton. Automatic synthesis of systolic arrays from uniform recurrent equations. *ACM SIGARCH Computer architecture news*, 12(3):208–214, 1984.
- [236] Jonathan Ragan-Kelley, Andrew Adams, Sylvain Paris, Frédo Durand, Connelly Barnes, and Saman Amarasinghe. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. In *PLDI*, 2013. ISBN 978-1-4503-2014-6. doi: 10.1145/2491956.2462176.
- [237] Atul Rahman, Sangyun Oh, Jongeun Lee, and Kiyoun Choi. Design Space Exploration of FPGA Accelerators for Convolutional Neural Networks. In *DATE*, 2017. ISBN 978-3-9815370-8-6. doi: 10.23919/DATE.2017.7927162.
- [238] Sanjay V Rajopadhye. Synthesizing systolic arrays with control signals from recurrence equations. *Distributed Computing*, 3(2):88–105, 1989.
- [239] Sanjay V Rajopadhye and Richard M Fujimoto. Synthesizing systolic arrays from recurrence equations. *Parallel computing*, 14(2):163–189, 1990.
- [240] Jagannathan Ramanujam and Ponnuswamy Sadayappan. Tiling multidimensional iteration spaces for multicomputers. *Journal of Parallel and Distributed Computing*, 16(2):108–120, 1992.
- [241] Brandon Reagen, Robert Adolf, Yakun Sophia Shao, Gu-Yeon Wei, and David Brooks. MachSuite: Benchmarks for accelerator design and customized architectures. In *2014 IEEE International Symposium on Workload Characterization (IISWC)*, pages 110–119, October 2014. doi: 10.1109/IISWC.2014.6983050.
- [242] Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V Le. Regularized Evolution for Image Classifier Architecture Search. pages 52–54, 2018.
- [243] Chandan Reddy, Michael Kruse, and Albert Cohen. Reduction Drawing: Language Constructs and Polyhedral Compilation for Reductions on GPU. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*, pages 87–97, Haifa Israel, September 2016. ACM. ISBN 978-1-4503-4121-9. doi: 10.1145/2967938.2967950.
- [244] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You Only Look Once: Unified, Real-Time Object Detection. June 2015.

- [245] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks. In *NeurIPS*, pages 91–99, 2015.
- [246] T. G. Robertazzi and S. C. Schwartz. Best “ordering” for floating-point addition. *ACM Transactions on Mathematical Software*, 14(1):101–110, 1988. ISSN 00983500. doi: 10.1145/42288.42343.
- [247] Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Global value numbers and redundant computations. *Conference Record of the Annual ACM Symposium on Principles of Programming Languages*, Part F1301:12–27, 1988. ISSN 07308566. doi: 10.1145/73560.73562.
- [248] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *IJCV*, 2015. ISSN 15731405. doi: 10.1007/s11263-015-0816-y.
- [249] Stuart J. Russell, Peter Norvig, and Ernest Davis. *Artificial Intelligence: A Modern Approach*. Prentice Hall Series in Artificial Intelligence. Prentice Hall, Upper Saddle River, 3rd ed edition, 2010. ISBN 978-0-13-604259-4.
- [250] Charbel Sakr, Naigang Wang, Chia-Yu Chen, Jungwook Choi, Ankur Agrawal, Naresh Shanbhag, and Kailash Gopalakrishnan. Accumulation Bit-Width Scaling For Ultra-Low Precision Training Of Deep Networks. In *ICLR*, 2019.
- [251] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Inverted Residuals and Linear Bottlenecks: Mobile Networks for Classification, Detection and Segmentation. In *CVPR*, 2018.
- [252] Herman H. Schmit, Srihari Cadambi, Matthew Moe, and Seth C. Goldstein. Pipeline Reconfigurable FPGAs. *Journal of VLSI Signal Processing Systems*, 24:129–146, 2000.
- [253] Fabian Schuiki, Andreas Kurth, Tobias Grosser, and Luca Benini. LLHD: A multi-level intermediate representation for hardware description languages. In *PLDI*, pages 258–271, 2020. doi: 10.1145/3385412.3386024.
- [254] Minghua Shen, Guojie Luo, and Nong Xiao. Exploring GPU-Accelerated Routing for FPGAs. *IEEE Transactions on Parallel and Distributed Systems*, 30(6):1331–1345, June 2019. ISSN 1558-2183. doi: 10.1109/TPDS.2018.2885745.

- [255] Yongming Shen, Michael Ferdman, and Peter Milder. Maximizing CNN Accelerator Efficiency Through Resource Partitioning. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pages 535–547, Toronto ON Canada, June 2017. ACM. ISBN 978-1-4503-4892-8. doi: 10.1145/3079856.3080221.
- [256] Masayuki Shimoda, Youki Sada, and Hiroki Nakahara. FPGA-Based Inter-layer Pipelined Accelerators for Filter-Wise Weight-Balanced Sparse Fully Convolutional Networks with Overlapped Tiling. *Journal of Signal Processing Systems*, 93(5):499–512, May 2021. ISSN 1939-8018, 1939-8115. doi: 10.1007/s11265-021-01642-6.
- [257] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, January 2016. ISSN 1476-4687. doi: 10.1038/nature16961.
- [258] Karen Simonyan and Andrew Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition. In *ICLR*, 2015.
- [259] Deshanand P Singh and Stephen D Brown. Integrated Retiming and Placement for Field Programmable Gate Arrays. page 10, 2002.
- [260] Byoungro So, Mary W. Hall, and Pedro C. Diniz. A compiler approach to fast hardware design space exploration in FPGA-based systems. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, pages 165–176, New York, NY, USA, May 2002. Association for Computing Machinery. ISBN 978-1-58113-463-6. doi: 10.1145/512529.512550.
- [261] Linghao Song, Xuehai Qian, Hai Li, and Yiran Chen. PipeLayer: A Pipelined ReRAM-Based Accelerator for Deep Learning. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 541–552, Austin, TX, February 2017. IEEE. ISBN 978-1-5090-4985-1. doi: 10.1109/HPCA.2017.55.
- [262] Ryan Stansifer. Presburger’s article on integer airthmetic: Remarks and translation. Technical Report TR84-639, Cornell University, Computer Science Department, September 1984. URL <http://techreports.library.cornell.edu:8081/Dienst/UI/1.0/Display/cul.cs/TR84-639>.

- [263] Emma Strubell, Ananya Ganesh, and Andrew McCallum. Energy and Policy Considerations for Deep Learning in NLP. (1), 2019. doi: 10.18653/v1/p19-1355.
- [264] Jiang Su, Julian Faraone, Junyi Liu, Yiren Zhao, David B. Thomas, Philip H. W. Leong, and Peter Y. K. Cheung. Redundancy-Reduced MobileNet Acceleration on Reconfigurable Logic for ImageNet Classification. In *ARC*, 2018.
- [265] Naveen Suda, Vikas Chandra, Ganesh Dasika, Abinash Mohanty, Yufei Ma, Sarma Vrudhula, Jae-sun Seo, and Yu Cao. Throughput-Optimized OpenCL-based FPGA Accelerator for Large-Scale Convolutional Neural Networks. In *FPGA*, pages 16–25, New York, New York, USA, 2016. ACM Press. ISBN 978-1-4503-3856-1. doi: 10.1145/2847263.2847276.
- [266] Xiao Sun, Wang Naigang, Chen Chia-yu, Ni Jia-min, Agrawal Ankur, Cui Xiaodong, Venkataramani Swagath, Maghraoui Kaoutar El, Srinivasan Vijayalakshmi, and Gopalakrishnan Kailash. Ultra-Low Precision 4-bit Training of Deep Neural Networks. In *NeurIPS*, 2020.
- [267] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. 2004. ISBN 978-0-262-03924-6. doi: 10.1021/bk-1997-0674.ch022.
- [268] Mingxing Tan and Quoc V. Le. EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks. In *ICML*, 2019.
- [269] Mingxing Tan and Quoc V. Le. EfficientNetV2: Smaller Models and Faster Training. In *ICML*, June 2021.
- [270] Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, Mark Sandler, Andrew Howard, and Quoc V. Le. MnasNet: Platform-Aware Neural Architecture Search for Mobile. 2018.
- [271] The NVIDIA Apex team. NVIDIA Apex. NVIDIA Corporation, October 2021.
- [272] The PyTorch team. Automatic Mixed Precision package - torch.cuda.amp. <https://pytorch.org/docs/stable/amp.html#gradient-scaling>, October 2021.
- [273] T.J. Todman, G.A. Constantinides, S.J.E. Wilton, O. Mencer, W. Luk, and P.Y.K. Cheung. Reconfigurable computing: Architectures and design methods. *IEE Proceedings - Computers and Digital Techniques*, 152(2):193, 2005. ISSN 13502387. doi: 10.1049/ip-cdt:20045086.

- [274] Seiya Tokui, Ryosuke Okuta, Takuya Akiba, Yusuke Niitani, Toru Ogawa, Shunta Saito, Shuji Suzuki, Kota Uenishi, Brian Vogel, and Hiroyuki Yamazaki Vincent. Chainer: A deep learning framework for accelerating the research cycle. In *KDD*, 2019. ISBN 978-1-4503-6201-6. doi: 10.1145/3292500.3330756.
- [275] The torch-mlir developers. The Torch-MLIR Project. <https://github.com/llvm/torch-mlir>. Accessed: 2022-10-03.
- [276] Konrad Trifunovic, Albert Cohen, David Edelsohn, Feng Li, Tobias Grosser, Harsha Jagasia, Razya Ladelsky, Sebastian Pop, Jan Sjödin, and Ramakrishna Upadrasta. Graphite two years after: First lessons learned from real-world polyhedral compilation. In *GCC Research Opportunities Workshop (GROW'10)*, 2010.
- [277] Yaman Umuroglu, Nicholas J Fraser, Giulio Gambardella, and Michaela Blott. FINN: A Framework for Fast, Scalable Binarized Neural Network Inference. In *FPGA*, 2017. ISBN 978-1-4503-2138-9. doi: 10.1145/1235.
- [278] Nanda K. Unnikrishnan and Keshab K. Parhi. LayerPipe: Accelerating Deep Neural Network Training by Intra-Layer and Inter-Layer Gradient Pipelining and Multiprocessor Scheduling. *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pages 1–8, November 2021. doi: 10.1109/ICCAD51958.2021.9643567.
- [279] Frank Vahid. *Digital Design with RTL Design, VHDL, and Verilog*. John Wiley & Sons, March 2010. ISBN 978-0-470-53108-2.
- [280] 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. *CoRR*, 2018.
- [281] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary Devito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. The Next 700 Accelerated Layers: From Mathematical Expressions of Network Computation Graphs to Accelerated GPU Kernels, Automatically. *ACM Transactions on Architecture and Code Optimization*, 16(4): 1–26, January 2020. ISSN 1544-3566, 1544-3973. doi: 10.1145/3355606.
- [282] Nicolas Vasilache, Oleksandr Zinenko, Aart JC Bik, Mahesh Ravishankar, Thomas Raoux, Alexander Belyaev, Matthias Springer, Tobias Gysi, Diego Caballero, Stephan Herhut, et al. Composable and modular code generation in

- mlir: A structured and retargetable approach to tensor compiler construction. *arXiv preprint arXiv:2202.03293*, 2022.
- [283] Stylianos I Venieris and Christos Savvas Bouganis. fpgaConvNet: A Framework for Mapping Convolutional Neural Networks on FPGAs. In *FCCM*, 2016. ISBN 978-1-5090-2356-1. doi: 10.1109/FCCM.2016.22.
- [284] Stylianos I. Venieris, Javier Fernandez-Marques, and Nicholas D. Lane. unzipFPGA: Enhancing FPGA-based CNN Engines with On-the-Fly Weights Generation. *arXiv:2103.05600 [cs]*, April 2021.
- [285] Anand Venkat, Manu Shantharam, Mary Hall, and Michelle Mills Strout. Non-affine extensions to polyhedral code generation. In *GCO*, page 185–194, 2014.
- [286] Sven Verdoolaege. Isl: An integer set library for the polyhedral model. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 6327 LNCS:299–302, 2010. ISSN 03029743. doi: 10.1007/978-3-642-15582-6_49.
- [287] Sven Verdoolaege. Presburger Formulas and Polyhedral Compilation, 2016.
- [288] Sven Verdoolaege. Live-Range Reordering. Technical report, 2016.
- [289] Sven Verdoolaege and Tobias Grosser. Polyhedral Extraction Tool. In *IMPACT*, page 8, 2012.
- [290] Sven Verdoolaege and Alexandre Isoard. Consecutivity in the isl polyhedral scheduler. *Report CW*, 709, 2017.
- [291] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. Polyhedral parallel code generation for CUDA. *ACM Transactions on Architecture and Code Optimization*, 9(4): 1–23, 2013. ISSN 15443566. doi: 10.1145/2400682.2400713.
- [292] Sven Verdoolaege, Serge Guelton, Tobias Grosser, and Albert Cohen. Schedule Trees. In *IMPACT*, 2014. doi: 10.13140/RG.2.1.4475.6001.
- [293] Nils Voss, Pablo Quintana, Oskar Mencer, Wayne Luk, and Georgi Gaydadjiev. Memory Mapping for Multi-die FPGAs. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 78–86, San Diego, CA, USA, April 2019. IEEE. ISBN 978-1-72811-131-5. doi: 10.1109/FCCM.2019.00021.

- [294] Erwei Wang, James J. Davis, Peter Y. K. Cheung, and George A. Constantinides. LUTNet: Learning FPGA Configurations for Highly Efficient Neural Network Inference. *IEEE Transactions on Computers*, 69(12):1795–1808, December 2020. ISSN 1557-9956. doi: 10.1109/TC.2020.2978817.
- [295] Jie Wang, Licheng Guo, and Jason Cong. AutoSA: A Polyhedral Compiler for High-Performance Systolic Arrays on FPGA. In *FPGA*, 2021. ISBN 978-1-4503-8218-2.
- [296] Naigang Wang, Jungwook Choi, Daniel Brand, Chia-Yu Chen, and Kailash Gopalakrishnan. Training Deep Neural Networks with 8-bit Floating Point Numbers. In *NeurIPS*, 2018.
- [297] Xijun Wang, Meina Kan, Shiguang Shan, and Xilin Chen. Fully Learnable Group Convolution for Acceleration of Deep Neural Networks. In *CVPR*, pages 9049–9058, 2019.
- [298] Yuxin Wang, Peng Li, Peng Zhang, Chen Zhang, and Jason Cong. Memory partitioning for multidimensional arrays in high-level synthesis. In *Proceedings of the 50th Annual Design Automation Conference*, pages 1–8, 2013.
- [299] Yuxin Wang, Peng Li, and Jason Cong. Theory and Algorithm for Generalized Memory Partitioning in High-Level Synthesis. In *FPGA*, pages 199–208, 2014. ISBN 978-1-4503-2671-1. doi: 10.1145/2554688.2554780.
- [300] Markus Weinhardt and Wayne Luk. Pipeline vectorization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20(2):234–248, February 2001.
- [301] Wei Wen, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. Learning Structured Sparsity in Deep Neural Networks. In *NeurIPS*, 2016. ISBN 978-1-4799-8930-0. doi: 10.1109/HPCA.2015.7056066.
- [302] Doran K Wilde. A library for doing polyhedral operations. *Parallel Algorithms and Application*, 15(3-4):137–166, 2000.
- [303] Ronald J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8(3-4):229–256, 1992. ISSN 0885-6125. doi: 10.1007/bf00992696.
- [304] Samuel Webb Williams, Andrew Waterman, and David Patterson. Roofline: An Insightful Visual Performance Model for Floating-Point Programs and Multicore Architectures. *Commun. ACM*, 2009. ISSN 00010782. doi: 10.1145/1498765.1498785.

- [305] Shmuel Winograd. *Arithmetic Complexity of Computations*, volume 33. Siam, 1980.
- [306] Di Wu, Yu Zhang, Xijie Jia, Lu Tian, Tianping Li, Lingzhi Sui, Dongliang Xie, and Yi Shan. A High-Performance CNN Processor Based on FPGA for MobileNets. In *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, pages 136–143, September 2019. doi: 10.1109/FPL.2019.00030.
- [307] Shuang Wu, Guoqi Li, Feng Chen, and Luping Shi. Training and Inference with Integers in Deep Neural Networks. In *ICLR*, 2018.
- [308] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, Łukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, Keith Stevens, George Kurian, Nishant Patil, Wei Wang, Cliff Young, Jason Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, Greg Corrado, Macduff Hughes, and Jeffrey Dean. Google’s Neural Machine Translation System: Bridging the Gap between Human and Machine Translation. *arXiv:1609.08144 [cs]*, October 2016.
- [309] Qingcheng Xiao, Yun Liang, Liqiang Lu, Shengen Yan, and Yu-Wing Tai. Exploring Heterogeneous Algorithms for Accelerating Deep Convolutional Neural Networks on FPGAs. In *DAC*, pages 1–6, 2017. ISBN 978-1-4503-4927-7. doi: 10.1145/3061639.3062244.
- [310] Guotian Xie, Jingdong Wang, Ting Zhang, Jianhuang Lai, Richang Hong, and Guo-Jun Qi. IGCVS2\$: Interleaved Structured Sparse Convolutional Neural Networks. In *CVPR*, 2018. doi: 10.1109/BIBM.2007.29.
- [311] Saining Xie, Ross Girshick, Piotr Dollár, Zhuowen Tu, and Kaiming He. Aggregated Residual Transformations for Deep Neural Networks. In *CVPR*, 2017. ISBN 978-1-5386-0457-1. doi: 10.1109/CVPR.2017.634.
- [312] Xilinx. MLIR-based AIEngine toolchain. <https://github.com/Xilinx/mlir-aie>. Accessed: 2022-10-03.
- [313] Xilinx. Large FPGA Methodology Guide, October 2012.
- [314] Xilinx. UltraRAM: Breakthrough Embedded Memory Integration on Ultra-Scale+ Devices, June 2016.
- [315] Xilinx. 7 Series FPGAs Configurable Logic Block User Guide (UG474), 2016.

- [316] Xilinx. Zynq-7000 All Programmable SoC Data Sheet: Overview, 2017.
- [317] Xilinx. 7 Series DSP48E1 Slice User Guide (UG479), 2018.
- [318] Xilinx. Vivado Design Suite User Guide: Synthesis, 2020.
- [319] Xilinx. Vitis High-Level Synthesis User Guide, 2021.
- [320] Xilinx. Vivado Design Suite User Guide: High-Level Synthesis, May 2021.
- [321] Xilinx. UltraScale Architecture and Product Data Sheet: Overview (DS890), March 2021.
- [322] Xilinx. Vivado Design Suite User Guide: Implementation. page 208, 2021.
- [323] Yu Xing, Shuang Liang, Lingzhi Sui, Xijie Jia, Jiantao Qiu, Xin Liu, Yushun Wang, Yu Wang, and Yi Shan. DNNVM : End-to-End Compiler Leveraging Heterogeneous Optimizations on FPGA-based CNN Accelerators. 2019. doi: arXiv:1902.07463v1.
- [324] Jingling Xue. Communication-minimal tiling of uniform dependence loops. *Journal of Parallel and Distributed Computing*, 42(1):42–59, 1997.
- [325] Guandao Yang, Tianyi Zhang, Polina Kirichenko, Junwen Bai, Andrew Gordon Wilson, and Christopher De Sa. SWALP : Stochastic Weight Averaging in Low-Precision Training. In *ICML*, 2019.
- [326] Wei Yang. Pytorch-classification, November 2021.
- [327] Hanchen Ye, Cong Hao, Jianyi Cheng, Hyunmin Jeong, Jack Huang, Stephen Neuendorffer, and Deming Chen. ScaleHLS: A New Scalable High-Level Synthesis Framework on Multi-Level Intermediate Representation. *arXiv:2107.11673 [cs]*, November 2021.
- [328] Yunxuan Yu, Tiandong Zhao, Kun Wang, and Lei He. Light-OPU: An FPGA-based Overlay Processor for Lightweight Convolutional Neural Networks. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 122–132, Seaside CA USA, February 2020. ACM. ISBN 978-1-4503-7099-8. doi: 10.1145/3373087.3375311.
- [329] Ming Yuan and Yi Lin. Model selection and estimation in regression with grouped variables. *Journal of the Royal Statistical Society. Series B: Statistical Methodology*, 68(1):49–67, 2006. ISSN 13697412. doi: 10.1111/j.1467-9868.2005.00532.x.

- [330] Tomofumi Yuki and Louis-Noël Pouchet. PolyBench 4.0. Technical report, 2015.
- [331] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks. In *FPGA*, pages 161–170, 2015. ISBN 978-1-4503-3315-3. doi: 10.1145/2684746.2689060.
- [332] Chen Zhang, Zhenman Fang, Peipei Zhou, Peichen Pan, and Jason Cong. Caffeine: Towards Uniformed Representation and Acceleration for Deep Convolutional Neural Networks. In *ICCAD*, 2016. ISBN 978-1-4503-4466-1. doi: 10.1145/2966986.2967011.
- [333] Ting Zhang, Guo-Jun Qi, Bin Xiao, and Jingdong Wang. Interleaved Group Convolutions for Deep Neural Networks. In *ICCV*, 2017. ISBN 1-74052-098-X. doi: 10.1109/ICCV.2017.469.
- [334] Xiangyu Zhang, Xinyu Zhou, Mengxiao Lin, and Jian Sun. ShuffleNet: An Extremely Efficient Convolutional Neural Network for Mobile Devices. *CoRR*, 2017.
- [335] Xiaofan Zhang, Junsong Wang, Chao Zhu, Yonghua Lin, Jinjun Xiong, Wenmei Hwu, and Deming Chen. DNNBuilder: An Automated Tool for Building High-Performance DNN Hardware Accelerators for FPGAs. In *ICCAD*, pages 1–8, November 2018. doi: 10.1145/3240765.3240801.
- [336] Xiaofan Zhang, Hanchen Ye, Junsong Wang, Yonghua Lin, Jinjun Xiong, Wenmei Hwu, and Deming Chen. DNNE Explorer: A Framework for Modeling and Exploring a Novel Paradigm of FPGA-based DNN Accelerator. *arXiv:2008.12745 [cs]*, March 2021.
- [337] Xishan Zhang, Shaoli Liu, Rui Zhang, Chang Liu, DI Huang, Shiyi Zhou, Jiaming Guo, Qi Guo, Zidong Du, Tian Zhi, and Yunji Chen. Fixed-Point Back-Propagation Training. In *CVPR*, pages 2327–2335, 2020. doi: 10.1109/CVPR42600.2020.00240.
- [338] Zhiru Zhang, Yiping Fan, Wei Jiang, Guoling Han, Changqi Yang, and Jason Cong. AutoPilot: A Platform-Based ESL Synthesis System. In Philippe Coussy and Adam Morawiec, editors, *High-Level Synthesis: From Algorithm to Digital Circuit*, pages 99–112. Springer Netherlands, Dordrecht, 2008. ISBN 978-1-4020-8588-8. doi: 10.1007/978-1-4020-8588-8_6.

- [339] Ritchie Zhao, Yuwei Hu, Jordan Dotzel, Christopher De Sa, and Zhiru Zhang. Building Efficient Deep Neural Networks with Unitary Group Convolutions. 2018.
- [340] Ruizhe Zhao and Wayne Luk. Efficient Structured Pruning and Architecture Searching for Group Convolution. In *ICCV Workshop*, 2019.
- [341] Ruizhe Zhao, Brian Vogel, Tanvir Ahmed, and Wayne Luk. Reducing Underflow in Mixed Precision Training by Gradient Scaling. In *IJCAI*, 2020. doi: 10.24963/ijcai.2020/404.
- [342] Ruizhe Zhao, Jianyi Cheng, Wayne Luk, and George A Constantinides. Polysca: Polyhedral high-level synthesis with compiler transformations. In *FPL*, 2022.
- [343] Shixuan Zheng, Xianjue Zhang, Daoli Ou, Shibin Tang, Leibo Liu, Shaojun Wei, and Shouyi Yin. Efficient Scheduling of Irregular Network Structures on CNN Accelerators. *TCAD*, 39(11):3408–3419, November 2020. ISSN 1937-4151. doi: 10.1109/TCAD.2020.3012215.
- [344] Guanwen Zhong, Vanchinathan Venkataramani, Yun Liang, Tulika Mitra, and Smail Niar. Design space exploration of multiple loops on FPGAs using high level synthesis. In *2014 IEEE 32nd International Conference on Computer Design (ICCD)*, pages 456–463, October 2014. doi: 10.1109/ICCD.2014.6974719.
- [345] Guanwen Zhong, Alok Prakash, Siqi Wang, Yun Liang, Tulika Mitra, and Smail Niar. Design Space exploration of FPGA-based accelerators with multi-level parallelism. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, pages 1141–1146, Lausanne, Switzerland, March 2017. IEEE. ISBN 978-3-9815370-8-6. doi: 10.23919/DATE.2017.7927161.
- [346] Michael Zhu and Suyog Gupta. To prune, or not to prune: Exploring the efficacy of pruning for model compression. In *ICLR Workshop*, 2018.
- [347] Oleksandr Zinenko, Sven Verdoolaege, Chandan Reddy, Jun Shirako, Tobias Grosser, Vivek Sarkar, and Albert Cohen. Unified Polyhedral Modeling of Temporal and Spatial Locality. page 41, 2017. ISSN 0249-6399.
- [348] Oleksandr Zinenko, Sven Verdoolaege, Chandan Reddy, Jun Shirako, Tobias Grosser, Vivek Sarkar, and Albert Cohen. Modeling the conflicting demands of parallelism and temporal/spatial locality in affine scheduling. In *CC*, pages 3–13, 2018. ISBN 978-1-4503-5644-2. doi: 10.1145/3178372.3179507.
- [349] Barret Zoph and Quoc V. Le. Neural Architecture Search with Reinforcement Learning. 2016. ISSN 1938-7228.

- [350] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V Le. Learning Transferable Architectures for Scalable Image Recognition. 2017.
- [351] Wei Zuo, Peng Li, Deming Chen, Louis Noël Pouchet, Shunan Zhong, and Jason Cong. Improving polyhedral code generation for high-level synthesis. In *CODES+ISSS*, 2013. ISBN 978-1-4799-1417-3. doi: 10.1109/CODES-ISSS.2013.6659002.
- [352] Wei Zuo, Yun Liang, Peng Li, Kyle Rupnow, Deming Chen, and Jason Cong. Improving high level synthesis optimization opportunity through polyhedral transformations. In *FPGA*, 2013. ISBN 978-1-4503-1887-7. doi: 10.1145/2435264.2435271.