Running Deep Learning Applications on Resource Constrained Devices

by

Naveen Vedula

M.Sc., Simon Fraser University, 2016
B.Sc., National Institute of Technology, Warangal, India, 2010

Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of
Doctor of Philosophy

in the
School of Computing Science
Faculty of Applied Sciences

Declaration of Committee

Name:	**Naveen Vedula**
Degree:	**Doctor of Philosophy**
Thesis title:	**Running Deep Learning Applications on Resource Constrained Devices**
Committee:	**Chair:** Keval Vora Assistant Professor, Computing Science
	Arrvindh Shriraman Supervisor Associate Professor, Computing Science
	Zhenman Fang Committee Member Assistant Professor, Engineering Science
	Alaa R. Alameldeen Examiner Associate Professor, Computing Science
	Mieszko Lis External Examiner Associate Professor Electrical and Computer Engineering University of British Columbia

Abstract

The high accuracy of Deep Neural Networks (DNN) come at the expense of high computational cost and memory requirements. During inference, the data is often collected on the edge device which are resource-constrained. The existing solutions for edge deployment include: i) executing entire DNN on the edge (EDGE-ONLY), ii) sending the input from edge to cloud where the DNN is processed (CLOUD-ONLY), and iii) splitting the DNN to execute partially on the edge and partially on the cloud (SPLIT). The choice of deployment between EDGE-ONLY, CLOUD-ONLY and SPLIT is determined by several operating constraints such as device resources and network speed, and application constraints such as latency and accuracy.

The EDGE-ONLY approach requires compact DNN with low compute and memory requirements. Thus, the emerging class of DNNs employ low-rank convolutions (LR-CONVs) which reduce one or more dimensions compared to the spatial convolutions (CONV). Prior research in hardware accelerators has largely focused on CONVs. The LR-CONVs such as depthwise and pointwise convolutions exhibit lower arithmetic intensity and lower data reuse. Thus, LR-CONVs result in low hardware utilization and high latency.

In our first work, we systematically explore the design space of Cross-layer dataflows to exploit data reuse across layers for emerging DNNs in EDGE-ONLY scenarios. We develop novel fine-grain cross-layer dataflows for LR-CONVs that support partial loop dimension completion. Our tool, X-Layer decouples the nested loops in a pipeline and combines them to create a common outer dataflow and several inner dataflows.

The CLOUD-ONLY approach can suffer from high latency due to the high transmission cost of large input data from the edge to the cloud. This could be a problem, especially for latency-critical applications. Thankfully, the SPLIT approach reduces latency compared to the CLOUD-ONLY approach. However, existing solutions only split the DNN in floating-point precision. Executing floating-point precision on the edge device can occupy large memory and reduce the potential options for SPLIT solutions.

In our second work, we expand and explore the search space of SPLIT solutions by jointly applying mixed-precision post-training quantization and DNN graph split. Our work, Auto-Split finds a balance in the trade-off among the model accuracy, edge device capacity, transmission cost, and the overall latency.

Keywords: Cross-layer Dataflow, DNN Accelerator, Edge-Cloud Collaboration, Neural Networks, Mixed Precision, Distributed Inference

Dedication

To Mom, Dad, Niti, and Sanjana.

Acknowledgements

X-Layer is the result of my work at Simon Fraser University in collaboration with my supervisor Dr Arrvindh Shriraman, my labmate Reza Hojabr and his supervisor Ahmad Khonsari. The Auto-Split work has been produced at Huawei Technologies in collaboration with Dr Amin Banitalebi Dehkordi, Dr Fei Xia, Dr Lanjun Wang, Dr Jian Pei and Dr Yong Zhang.

I would like to thank my supervisor, Dr Arrvindh Shriraman for staying with me in tough times and providing his useful insights. This journey would not have been possible without you. I would also like to thank my co-supervisor Dr Zhenman Fang for his valuable feedback on my work from time to time. Thank you Dr Yong Zhang for pushing me to think hard about a problem, Dr Amin Banitalebi Dehkordi for your dedication, work ethic and support, and Dr Lanjun Wang for your critical thinking.

I am very thankful to my collaborators for wearing many hats as friends, mentors, and just being there sometimes. Thank you, Dr Snehasish Kumar, Dr Amirali Sharifian, Dr Reza Hojabr, Dr Apala Guha, Dr Viji Srinivasan, Dr Reynald Hoskinson, Changan Chen, Dr Frederick Tung, Dr Greg Mori, Dr Jian Pei, Dr Andreas Moshovos, and Dr Nick W. Sumner. I would like to thank Dr Alaa Alameldeen, Dr Keval Vora, and Dr Mieszko Lis for serving in different roles during the thesis defence on such short notice. Finally, I would like to thank my lovely wife, Sanjana for her support and patience as I worked my way through this arduous journey.

Table of Contents

List of Tables

List of Figures

Abbreviation

AI Artificial Intelligence

ASR Automatic Speech Recognition

CLOUD-ONLY Cloud Only Approach

CNN Convolutional Neural Network

CONV Spatial Convolution

cross-layer Cross-Layer Dataflow

DNN Deep Neural Network

DP Depthwise Convolution

EDGE-ONLY Edge Only Approach

FC Fully-Connected Layer

HAR Human Activity Recognition

IoT Internet of Things

LR-CONV Low-rank Convolution

LUT Look up Table

MAC Multiply Accumulate Operation

ML Machine Learning

MP Mixed Precision Quantization

NAS Neural Architecture Search

OCR Optical Character Recognition

PE Processing Element

per-layer	Per-Layer Dataflow
PT	Pointwise Convolution
PTQ	Post Training Quantization
QAT	Quantization Aware Training
RNN	Recurrent Neural Network
SPLIT	Split Approach. Partial execution on edge device and partial on cloud device
WCET	Worst Case Execution Time

Chapter 1

Introduction

DEEP LEARNING [6] is a sub branch of Artificial Intelligence (AI) that imitates the workings of the human brain in processing data and creating patterns for use in decision making. Deep-learning architectures such as Deep Neural Network (DNN), deep belief networks, graph neural networks, Recurrent Neural Network (RNN) and Convolutional Neural Network (CNN) have been applied to fields including computer vision, speech recognition, natural language processing, machine translation, bioinformatics, drug design, medical image analysis, material inspection and board game programs, where they have produced results comparable to and in some cases surpassing human expert performance. In 2012, Alexnet [61], a deep learning algorithm became the winner of the ImageNet challenge [60] beating all traditional handcrafted computer vision algorithms. Since then deep learning algorithms have become the de facto for such applications. Deep learning (Figure 1.1) is a subset of Machine Learning (ML) which comes under the large umbrella of AI. ML is the class of algorithms in which the computers have the ability to learn without being explicitly programmed. The AI refers to the theory and development of algorithms that are able to perform tasks normally requiring human intelligence.

Deep Neural Networks use multiple layers to progressively extract higher-level features from the raw input. For example, in image processing, lower layers may identify edges, while higher layers may identify the concepts relevant to a human such as digits or letters or faces. A DNN has a training phase and an inference phase. Typically, during the training phase, the DNN algorithm learns the task at hand by going through many examples from an example dataset. The DNN model learns by minimizing the error between model prediction and ground truth with an algorithm called backpropagation. The Backpropagation updates the model parameters during the training phase. Once the accuracy is sufficiently high, the DNN algorithm stops learning and runs in the inference stage. In the inference stage, the DNN model is frozen i.e, the model parameters are not updated anymore (no backpropagation) and deployed to a target platform.

DNN algorithms are making exponential advances and have been seeping into many applications. A large number of applications such as Human Activity Recognition (HAR), object detection, face recognition, wearable devices, anti-spoofing, Optical Character Recognition (OCR), Automatic Speech Recognition (ASR), authorization entry, and text prediction require AI application processing

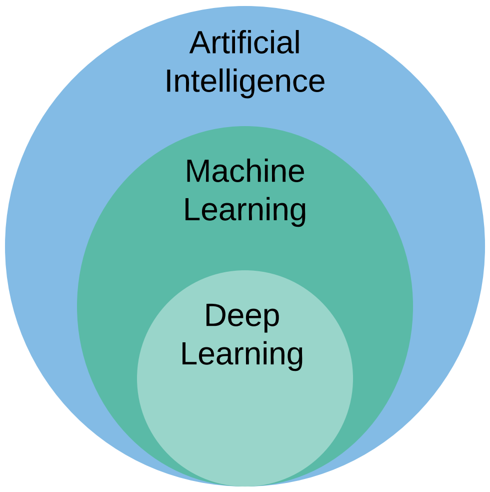


Figure 1.1: The field of artificial intelligence [7]

at the source of the data. This is called EDGE COMPUTING or popularly known as the Internet of Things (IoT). The Edge devices can refer to small devices with limited computing capability such as cameras, phones, microcontrollers and includes any device that collects the data at the source and does the front-end of data processing. The IoT market is estimated to be USD 82.4 Billion in 2020 and is estimated to grow at a compounded annual growth rate of 23% from 2020 to 2028 [8].

We now explain existing solutions for the deployment of AI applications on the edge devices in §1.1. An AI application can choose to i) execute everything on the edge device (EDGE-ONLY), ii) send input data to the cloud for DNN processing on the cloud (CLOUD-ONLY, and iii) Choose to split the AI application to partially execute on the edge and partially on the cloud device (SPLIT). §1.2 discusses the contributions made in this thesis and resulting publications followed by §1.3 which discusses the thesis organization. §1.4 discusses the challenges of deploying Edge Only Approach (EDGE-ONLY) for emerging DNNs based on Low-rank Convolution (LR-CONV). Next in §1.5, we explore the ideas to speed up SPLIT solutions for latency critical applications. We leverage mixed precision post training quantization for the partial DNN executing on the edge device.

1.1 AI application pipeline

The DNN's high accuracy comes at the expense of high computational and memory requirements for both the training and the inference phase of deep learning. The efficiency of DNNs during the inference phase is especially important since training occurs once and inference occurs many times. This section explains various approaches to deploy DNNs for inference on edge devices. We also discuss various constraints that lead to the selection of these approaches.

The DNN deployment on the edge is challenging as the data is collected on the edge device which is generally resource-constrained and may not be able to meet the computation and memory requirements of the DNN execution. Existing solutions can be broadly divided into EDGE-ONLY and DISTRIBUTED. In EDGE-ONLY approach the DNN is executed end to end on the edge device itself, whereas in DISTRIBUTED approach a part of the data is executed on the edge device and the rest is sent to the cloud for further processing. The DISTRIBUTED approach can be further divided into i) CLOUD-ONLY and ii) SPLIT solutions. As the name suggests, in CLOUD-ONLY, the input data is directly sent to the cloud for DNN processing. In the Split solution, the DNN is split into two DNNs. The first half of the DNN executes on the edge DNN and the rest is executed on the cloud. Figure 1.2 shows an example of an AI pipeline of license plate detection. First, the camera takes an image of the license plate, then the image is processed and sent to a DNN (Yolo-v3 head) for object detection. The object detection detects the license plate and crops it from the original image. The cropped license plate is then sent to a large recognition DNN (an LSTM) which converts the license plate image into text. The license plate text is the endpoint of DNN algorithms, but in an end to end application, it requires further processing. For example, the license plate text could be utilized to search into a database and issue a ticket to the driver for speeding. The example shows the deployment of the AI pipeline with EDGE-ONLY, CLOUD-ONLY and SPLIT approaches. The choice of deployment approach depends on the operating and application constraints. For example, the EDGE-ONLY approach requires the AI application to fit on the device memory. The CLOUD-ONLY approach requires the network to be stable, and have relatively high bandwidth compared to the input data being transmitted. The SPLIT approach requires a preprocessing stage to split the AI application into the edge task and the cloud task. It also requires AI applications to contain output activation layers where the transmission cost of activations at the split is significantly lower than the input data. Next, we discuss the operating and application constraints which are a key factor in deciding the choice of deployment.

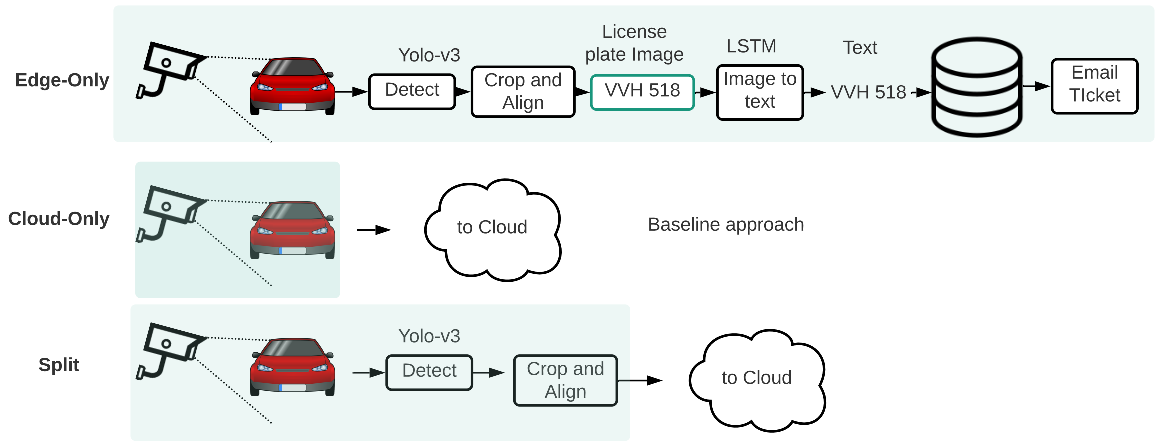


Figure 1.2: An example of issuing ticket with license plate detection and various methods of deployment. The highlighted region shows execution in edge device

Table 1.1: Metrics for Operating constraints and Application requirements

Operating Constraints	Device: memory, compute
	Network: Speed, Stability
Application Requirement	Latency/Throughput, and Cost
	Accuracy

Solution trade-offs based on Operating and Application constraints.

The choice of deployment between EDGE-ONLY, CLOUD-ONLY and SPLIT is determined by operating and application constraints. Table 1.1 shows some choices to consider for operating and application requirements. The operating constraints include factors such as device compute capability and available memory. The application requirements include metrics such as Latency, the DNN model accuracy and device cost. Other important metric is the accuracy of the application. We discuss the application requirements and how it affects the choice of deployment next.

Latency: Real-time inference is critical to many applications. For example, in the case of self-driving cars, the vehicle needs to process the input from camera frames and take a decision within a minimum Worst Case Execution Time (WCET). For example, detecting a pedestrian on the street. Such applications need to use the EDGE-ONLY approach. Since, in a DISTRIBUTED approach, the network conditions can be unreliable such as network speed, stability, and availability of network in remote areas. Some applications may have stable network conditions but require low latency. Such applications would prefer a SPLIT solution, since, transmission cost of large input data to the cloud could be a bottleneck in CLOUD-ONLY approach.

Accuracy: An AI application may demand high accuracy. The DNN's high accuracy comes at the expense of high computational and memory requirements. High application accuracy implies larger models, which in turn requires higher processing power and memory. Thus it may not be possible to deploy an EDGE-ONLY solution in such cases. An extreme example of high accuracy is the text prediction models such as GPT-3 which requires up to 350GB memory and 175 Billion parameters. Such models are far beyond the capacity of edge devices and are run on clusters of GPUs. Such applications would deploy either a CLOUD-ONLY or a SPLIT solution.

The aim of the thesis is to enable efficient execution of DNN inference on edge devices especially for EDGE-ONLY and SPLIT approach.

Challenges of EDGE-ONLY: The EDGE-ONLY devices are limited by compute and memory requirements. In order to fit the DNN on edge device memory, many software optimizations are made to trade-off model accuracy with model size. Techniques such as i) model compression [13,17,24,32, 32,34,56,59,105,108,116,120,124], ii) knowledge distillation, pruning and other hybrid techniques [23,35,47,74,83] and, iii) Neural Architecture Search (NAS) [68,84,98,99,108,109,112,114,126] are used to reduce the model size within the acceptable application constraints. The NAS methods have led to many compact emerging DNNs with high accuracy and, low compute and memory requirements. These DNNs heavily employ LR-CONVs. An LR-CONV is a convolution which has one or more dimensions reduced compared to the standard convolution operation. For example, in a Pointwise Convolution (PT) layer, the filter height and width are reduced to 1×1. The LR-CONVs have low reuse within a layer. This leads to poor hardware utilization which exacerbates the latency problem in spite of having compact DNN with low Multiply Accumulate Operation (MAC) and memory requirement (see chapter 2, Figure 2.4). We explain the challenges of improving latency for EDGE-ONLY approach, especially for executing LR-CONVs in §1.4.

Challenges of SPLIT: When the AI applications are large and do not fit on the edge device, DISTRIBUTED approach has to be used. The SPLIT solutions can be up to $1.6\times$ faster than CLOUD-ONLY solutions and can be a good alternative to CLOUD-ONLY solutions for many AI applications. Existing SPLIT solutions can be categorized into cascaded models [117], multi-exit models [100] and graph based DNN splitting [49, 58, 119]. Only the existing graph based DNN splitting techniques are generic and can be applied to many applications but does not leverage model compression techniques such as quantization to further improve the latency of end to end application. We discuss it in detail in §1.5.

1.2 Dissertation Contribution

To our knowledge, this thesis is the first to explore the design space of cross-layer dataflows for EDGE-ONLY solutions, and explore joint bit-width allocation along with split detection for SPLIT solutions. In this thesis we make two major contributions:

- We propose X-Layer which is the first framework to systematically explore the design space of cross-layer dataflows. It supports i) partial order loops enabled by LR-CONVs, ii) variable pipeline depth, and iii) heterogeneous inner and outer dataflows. We propose novel cross-layer dataflows, $X^pY^pK^pC{-}3$ and $X^pY^pC^p{-}2$. They exploit partial order loops to maximize temporal reuse across layers and minimize the amount of on-chip SRAM. We also develop the X-Layer microarchitecture targeting cross-layer dataflows. The primary novelty is Multi-Input Multi-Output (MIMO) queues that support the multitude of data transformations in heterogeneous dataflows. We navigate exponentially large design space ($\simeq$ millions) of cross-layer dataflows and evaluate six state-of-the-art DNNs (Mobilenet, Xception, Resnet-50, Mobilenet_v2, Mnasnet1_0, EfficientNet-b0). Compared to prior state-of-the-art [10, 36], X-Layer improves performance by $7.8\times$ and $16.6\times$ using only 256KB of on-chip SRAM. [10, 36] requires up to 2.4 MB of SRAM. X-Layer also demonstrates a wider performance band for varied SRAM configurations (32KB – 2MB) whereas the performance drops steeply for prior state-of-the-art as the on-chip SRAM is reduced.

- We propose AUTO-SPLIT, an algorithm to minimize end-to-end latency by jointly searching for a split point (to divide the DNN into edge and cloud devices), as well as searching for optimal bit-widths for edge DNN layers (under pre-specified device memory constraints, network latency, and an accuracy threshold). We demonstrate that by applying quantization on the edge DNN, AUTO-SPLIT reduces the overall latency by 20–80% compared to the state of the art network partitioning algorithms. AUTO-SPLIT also reduces model size requirements on the edge DNN by 43 – 95 % compared to Uniform 4-bit quantized networks. To demonstrate the generality of our method, we study six image classification networks and three object detection networks, at different model sizes/architecture, and input resolutions.

Publications

This dissertation includes the works published in two peer-reviewed conferences. These publications are listed below :-

- PACT 2021 – **X-Layer: Building Composable Pipelined Dataflows for Low-Rank Convolutions**. This work is published at the 30th International Conference on Parallel Architectures and Compilation Techniques with co-authors Reza Hojabr, Ahmad Khonsari, and Arrvindh Shriraman.
  - I was the main contributor of the X-layer framework including the design, development and evaluation of the work. The X-layer scheduler (TB-Scheduler in the repository) extracts the layer information from the PyTorch DNN model and explores all cross-layer dataflows up to a pipeline depth of three. The latency calculation is done by a cycle-accurate DNN accelerator. The optimal loop blocking and loop tiling parameters are searched with the help of NSGA-II, a genetic algorithm.
  - Reza and I came up with the idea of the MIMO queues to realize T-Bricks shown in §4.1. He then developed the RTL for the X-Layer microarchitecture to evaluate FPGA results (§4.3.2).
- KDD 2021 – **Auto-Split: A General Framework of Collaborative Edge-Cloud AI**. This work is published at the Proceedings of the 27th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining with co-authors Amin Banitalebi-Dehkordi, Jian Pei, Fei Xia, Lanjun Wang, and Yong Zhang. Amin and I are the first authors of this paper.
  - I was the main contributor of the AUTO-SPLIT framework, including the idea, algorithm design, framework development, and evaluation of the work. The AUTO-SPLIT framework takes a DNN model in PyTorch and i) collects statistics for each layer to quantize later, ii) applies graph pre-processing and extracts the DNN graph information to collect layer dependencies to find the min-cut in the split, iii) Implements Auto-split algorithm

to find the split and allocate bit-widths to the edge DNN, iv) applies post-training quantization, v) and explores the trade-off in accuracy vs Latency. The Latency estimates are calculated with the help of a mixed-precision edge device cycle-accurate simulator. I also developed the video demo for the face detection application in TensorFlow which i) explores the activation compression techniques, and ii) explores RPC protocols and socket programming for activation transmission from edge device to cloud device.

– Amin did the evaluation and development of the license plate detection application shown in §5.3.5. He also helped with writing the paper and filing for the patent.

.

Chapter 7 explains other contributions made by me which are not included in the thesis. It also includes my third work as the first author – **NACHOS: Software-Driven Hardware-Assisted Memory Disambiguation for Accelerators** which was published in the 2018 IEEE International Symposium on High Performance Computer Architecture, **HPCA 2018** with co-authors Arrvindh Shriraman, Snehasish Kumar, and Nick Sumner.

- I was the main contributor of NACHOS, including the idea, design, development and evaluation of the work. I developed an LLVM Alias analysis framework to replace Load Store queues for memory disambiguation of acceleratable hot paths in SPEC2K, SPEC2K6, and PARSEC workloads. This work is available on `https://github.com/sfu-arch/Nachos`.
- In this work I leveraged Snehasish's framework NEEDLE for extracting acceleratable regions from the workloads.

1.3 Dissertation Organization

Figure 1.3 summarizes the dissertation organization. Chapter 2 provides the background. First we start by explaining the differences between CONV and LR-CONVs, followed by optimizations for DNN model and dataflow for EDGE-ONLY solutions. Then we provide the background for existing SPLIT solutions.

This thesis discusses two of my works X-Layer and AUTO-SPLIT. Chapter 3 and 4 discuss X-Layer which explores the EDGE-ONLY approach. Chapter 5 discusses AUTO-SPLIT which explores the SPLIT approach. In chapter 3, we provide a taxonomy of cross-layer dataflows and explain the cross-layer pipelining for LR-CONV based DNNs. Chapter 4 provides the DNN hardware accelerator microarchitecture changes required to support all cross-layer dataflows. We then present a comprehensive evaluation of cross-layer dataflows and compare them against the state of the art dataflows.

Chapter 5 provides a distributed approach to jointly detect split point to generate an edge DNN and a cloud DNN, and allocate mixed-precision bit-widths for the edge DNN. The edge DNN executes on the edge device and the cloud DNN executes on the cloud device. Chapter 6 provides a

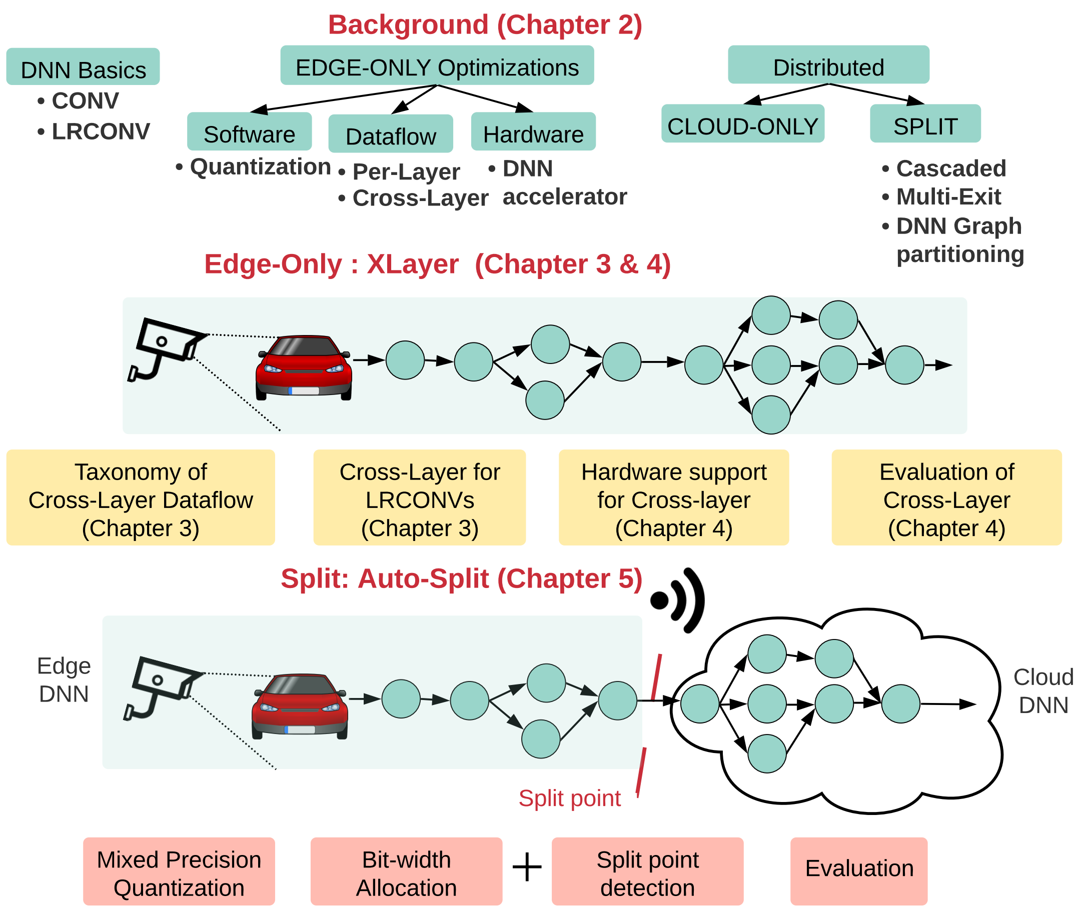
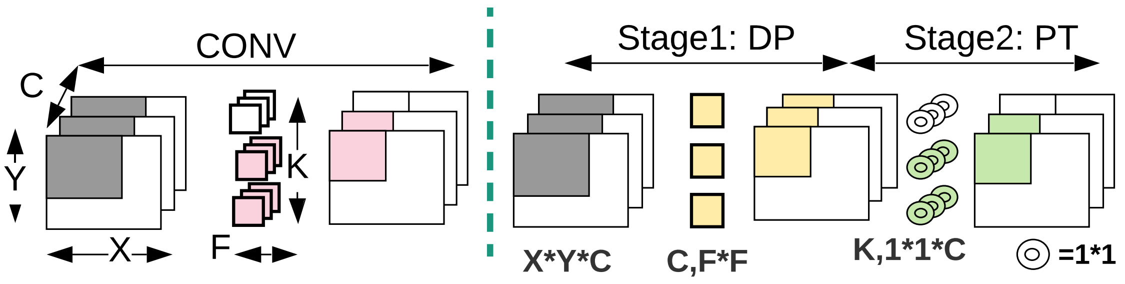


Figure 1.3: Dissertation Organization

summary of the thesis and directions for future work. Chapter 7 explains other contributions made by me which were published in peer-reviewed conferences.

1.4 Improving latency for Edge-Only Approach

The majority of prior DNN accelerators [1, 19] and dataflow toolflows [62, 115] have targeted spatial convolutions (CONV). However, state-of-the-art networks (see Imagenet leaderboard [4]) employ low rank convolutions (LR-CONVs) for their workhorse layers e.g., AmoebaNet [84], RandWire [112], EfficientNet [99], Mnasnet [98], NasNet [127] and PNasNet [68]. LR-CONV networks have only one or two layers of CONV and spend more than 95% of the time in LR-CONVs. Figure 1.4 illustrates a depth-separable layer commonly used in-lieu of CONVs. An LR-CONV is any convolution that has one or more dimensions reduced compared to CONV. For example, a depthwise convolution (DP) has $Filters$=1. Similarly, a pointwise convolutions (PT) has filter height and width $=1\times1$. These LR-CONVs are aggregated to create many composite layers such as depthwise-separable and inverted residual layers in emerging DNNs.



	(**CONV**)	Depthwise (**DP**)	Pointwise (**PT**)
#MACs	$X{*}Y{*}C{*}K{*}F^2$	$X{*}Y{*}C{*}F^2$	$X{*}Y{*}C{*}K$
#Input Bytes	$X{*}Y{*}C$	$X{*}Y{*}C$	$X{*}Y{*}C$
Normalized execution characteristics			
#MACs/#Input Bytes	$K{*}F^2$	$F^2\Downarrow$	$K\Downarrow$
#Filter Bytes	$F^2{*}C{*}K$	$F^2C\Downarrow$	$K{*}C\Downarrow$

Figure 1.4: Spatial vs. Low-rank Convolutions. $F^2 \ll K \ll K{*}F^2$. In Depth-separable layer, majority of MAC is allocated to PT, and DP is memory bound due to F^2. Typically F^2=9 and 1, whereas C,$K \in [1,1000]$.

Given the same input activation of size $X \times Y \times C$, the arithmetic intensity of a CONV layer is $K \times F^2$. A depth separable layer that can replace a CONV layer has an arithmetic intensity of only F^2+K. The DP has "F^2" arithmetic intensity and the PT has "K" arithmetic intensity. In DNNs the order of $F^2 \simeq 9$, whereas the order of C,$K \in [1,1000]$. Thus the depth separable layer, $DP \rightarrow PT$ combined has much less arithmetic intensity compared to a CONV layer. Similarly, the memory required for weights/filters is also much less compared to the CONV layer.

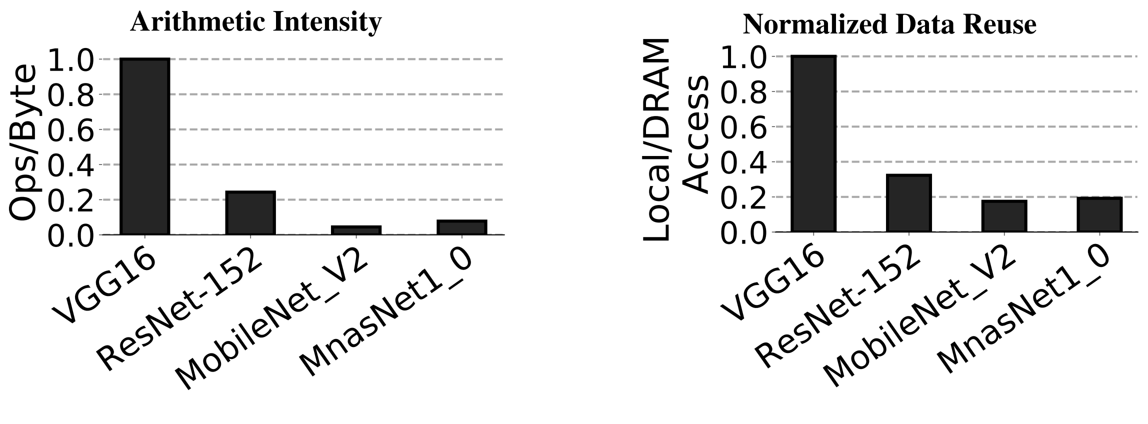


Figure 1.5: LR-CONVs vs CONVs.

Figure 1.5 compares the Arithmetic Intensity for LR-CONV and CONV. We find that DNNs with LR-CONVs (Mnasnet1_0 and Mobilenet_v2) have 10× lower arithmetic intensity (# MACs/byte) and 5-6× lower data-reuse (#SRAM/DRAM).

B	# Batch
X,Y	Activation Width and Height
C	# Activation Channels
K	# Filters
Fx,Fy	Filter Width and Height
$_b$	DRAM→SRAM Blocking parameters. Also determines size of On-chip buffer params
$_t$	Tiling parameters. Controls how on-chip buffer is processed in tiles by the grid-based neural engine.

```
# Outer dataflow: by compiler
def outer_xykc():
  for b in range(0,B,Bb):
   for x in range(0,X,Xb):
    for y in range(0,Y,Yb):
     for k in range(0,K,Kb):
      for c in range(0,C,Cb):
       act=load_act_onchip(Xb,Yb,Cb)
       wgt = load_wgt_onchip(Kb,Cb)
       inner_ck(act,wgt)

# Inner dataflow: in hardware
# (C|K) dataflow
def inner_ckxy(): # loads tile
 for c in range(0,Cb,Ct): #Tile
  for k in range(0,Kb,Kt): #Tile
   for x in range(0,Xb,Xt): #Tile
    for y in range(0,Yb, Yt): #Tile
     act_tile = \
      act[x:x+Xt][y:y+Yt][c:c+Ct]
     wgt_tile = wgt[k,k+Kt][c:c+Ct]
     conv_hw_ck(act_tile,wgt_tile)

# conv2d: by MAC Units
def conv_hw_ck(act_tile,wgt_tile):
 for c in range(0,Cb,Ct): #unroll
  for k in range(0,Kb,Kt): #unroll
   for x in range(0,Xb,Fx): #unroll
    for y in range(0,Yb,Fy): #unroll
     conv(act_tile,wgt_tile)
```

Figure 1.6: Illustration of per-layer schedule and mapping of a CONV layer on a hardware accelerator

Per-Layer dataflow

The Dataflow refers to the scheduling and ordering of the different loops in a DNN. The Baseline DNN accelerators seek to exploit parallelism and reuse in multiple loop dimensions by scheduling each CONV [1, 19] (per-layer dataflow) independently. They focus on improving the per-layer locality and load data in tiles from DRAM [62, 79, 115]. However, in per-layer, the output of each layer is flushed back to the DRAM and re-fetched by the next layer. We show that state-of-the-art per-layer exhibit low-performance since they seek to exploit per-layer locality, which LR-CONVs do not possess.

Figure 1.6 shows an example dataflow schedule of a CONV layer as used in various baseline hardware accelerators. A CONV algorithm has seven loops comprising of b,X,Y,K,C,F_x,F_y. However, when scheduling a layer, the seven loops can be split into three groups of X,Y,K,C dimensions. The F_x,F_y are small dimensions and can be unrolled in either space or time. The outermost loop "outer_xykc()", is scheduled in software by the compiler and controls the on-chip memory allocation for the layer, we call it the **Outer dataflow**. The inner loop "inner_ckxy()" or **inner dataflow** is scheduled in hardware and controls the tiles which are loaded to/from on-chip memory to registers for the execution of convolution. The "conv_hw_ck()" unrolls the loops in hardware and controls the **mapping** of the MAC units.

Accelerator Memory Model We assume a hardware memory model similar to [62, 79, 115]. Figure 1.7 shows the high-level logical view of the accelerator memory model. The DNN hardware accelerator design is composed of several major components i) NN Engine, ii) on-chip buffer, iii) external memory and iv) on/off-chip interconnect. An NN engine is a grid of Processing Element (PE). A PE calculates a MAC operation which is the basic computation unit of convolution operations. Due to resource limitations, the data is first cached in on-chip buffers (SRAM) before being fed to the computation unit. The on-chip interconnect is dedicated to communication between NN engines, between PEs, and between NN engines and on-chip buffers.

Cross-Layer Dataflows for LR-CONVs

Since, LR-CONVs are memory bound and have lower arithmetic intensity, they can benefit from inter-layer reuse. A Cross-layer dataflow schedules loops across concurrent layers simultaneously (per-layer serializes the execution of each layer). In LR-CONVs, the majority of the data re-use is across layers i.e., the producer-consumer reuse of the intermediate output activations of each layer. This provides an opportunity for cross-layers to retain the activations on-chip as they move across the layers.

Unlike CONV pipelines, LR-CONVs provide an opportunity for X-Layer to kickstart neighbouring layers in the pipeline with partially completed activations in (height, width, channel and filter dimensions). This shortens the lifetime of inter-layer activations that need to be held on-chip and allows capturing locality with a smaller amount of SRAM.

Design space of Cross-Layer dataflows for LR-CONV

Figure 1.8 shows an example Cross-layer schedule for an inverted residual layer ($PT \rightarrow DP \rightarrow PT$). To schedule multiple layers at a time, they need to be executed in a pipeline. The outermost loop needs

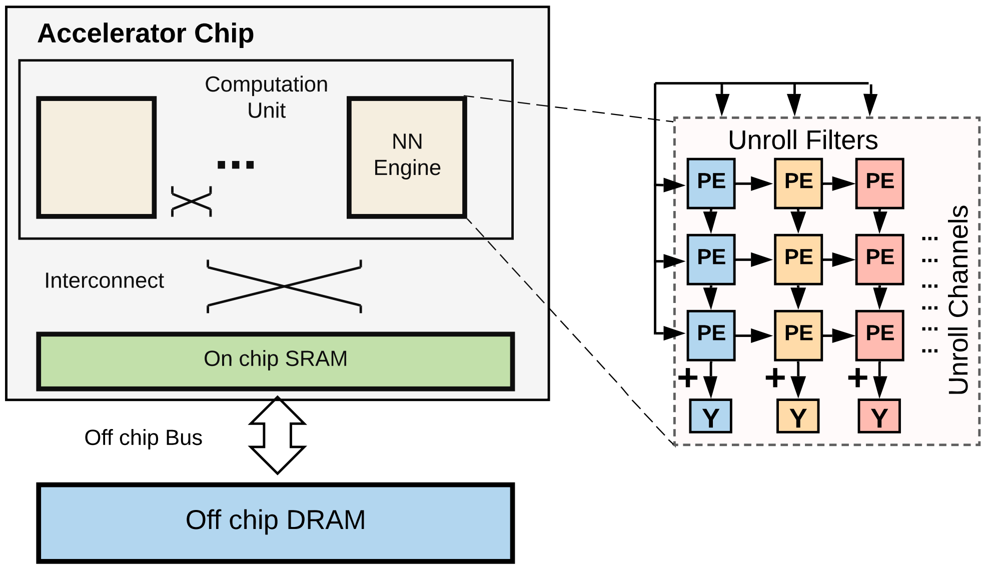


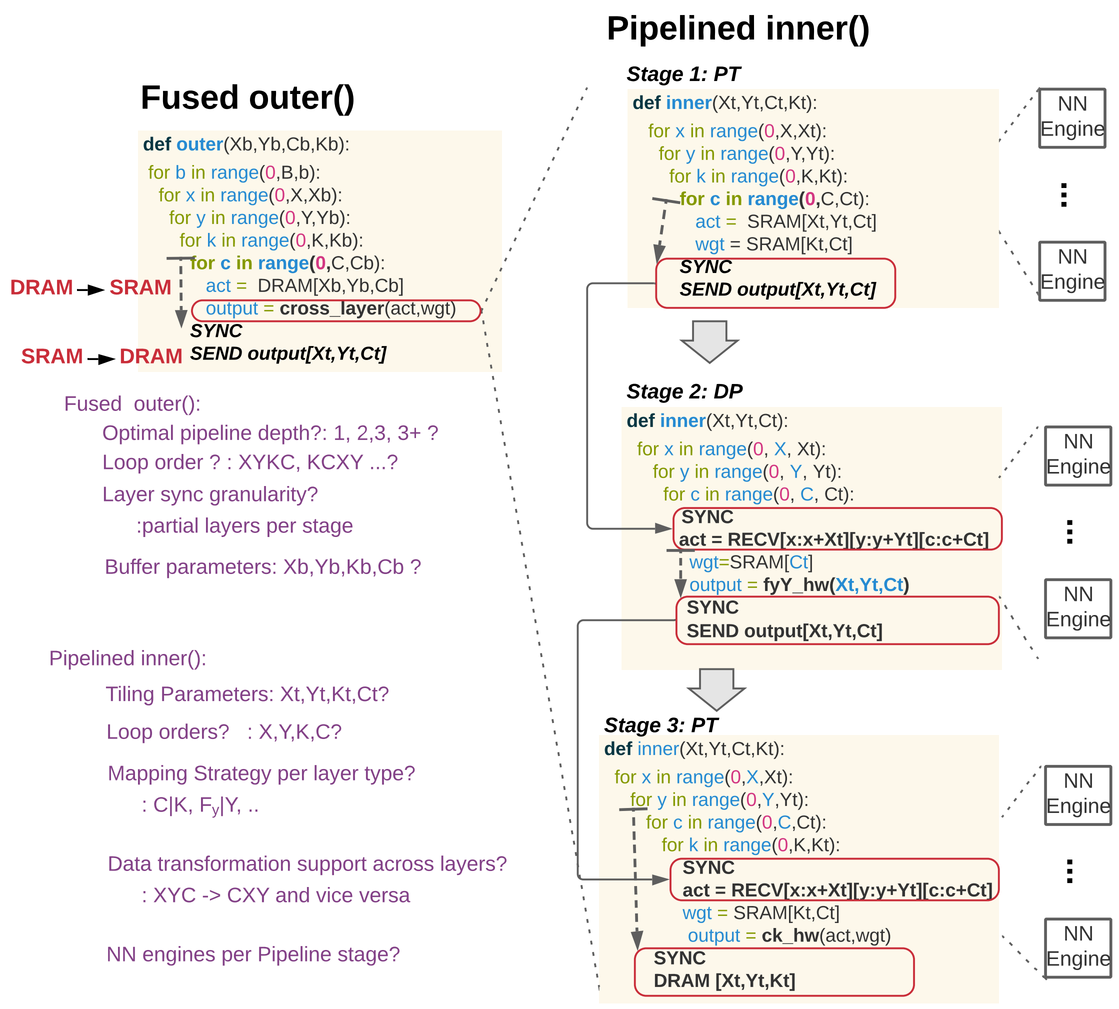
Figure 1.7: Accelerator Overview

to be fused for all pipelined layers which bring the corresponding input activations from DRAM to SRAM and sends the output activations from SRAM to DRAM. Each stage of the layer is executed concurrently with multiple sync points.

This thesis answers the questions such as what is the optimal loop order, pipeline depth, buffer parameters for the fused layer. Within each layer, we seek answers for 1) what are the optimal tiling parameters, 2) optimal loop order, 3) best mapping strategy based on layer type, 4) best mapping strategy and loop orders based on which stage the layer occurs, 5) What are the possible granularities at which different layers can sync based on the layer types and, 6) the optimal number of compute engines (NN) to be allocated per layer.

The outer loop blocking parameters control: i) the minimum on-chip memory required for each pipelined layer stage, ii) the synchronization granularity between layers and how often data is moved between them, and iii) the data reuse across layers and how long data is held on-chip before being consumed by a subsequent stage. The outermost loop has 5 loops, and thus $5!{=}120$ possible loop orders. Within each layer, the buffer parameters vary a lot. For example, $X,Y{\in}[3,224]$, and $K,C{\in}[1,1000]$ for image classification DNN which we study in this thesis.

The outer dataflow controls the pipeline granularity, and the inner loop order along with tiling parameters control the amount of work done in each pipeline stage. The number of loop orders per layer quickly explodes as the pipeline depth increases. Assuming a pipeline depth of “N", the possible loop orders are $(4!)^N$. Within each layer, the tiling parameters can vary between $X,Y{\in}[3,224]$, and $K,C{\in}[1,1000]$. The optimal hardware mapping also varies based on the layer type CONV, DP and



```
def outer(Xb,Yb,Cb,Kb):
 for b in range(0,B,b):
  for x in range(0,X,Xb):
   for y in range(0,Y,Yb):
    for k in range(0,K,Kb):
     for c in range(0,C,Cb):
      act =  DRAM[Xb,Yb,Cb]
      output = cross_layer(act,wgt)
     SYNC
     SEND output[Xt,Yt,Ct]
```

Fused outer():
- Optimal pipeline depth?: 1, 2,3, 3+ ?
- Loop order ? : XYKC, KCXY ...?
- Layer sync granularity?
  - :partial layers per stage
- Buffer parameters: Xb,Yb,Kb,Cb ?

Pipelined inner():
- Tiling Parameters: Xt,Yt,Kt,Ct?
- Loop orders?   : X,Y,K,C?
- Mapping Strategy per layer type?
  - : C|K, F_y|Y, ..
- Data transformation support across layers?
  - : XYC -> CXY and vice versa
- NN engines per Pipeline stage?

```
def inner(Xt,Yt,Ct,Kt):
 for x in range(0,X,Xt):
  for y in range(0,Y,Yt):
   for k in range(0,K,Kt):
    for c in range(0,C,Ct):
     act =  SRAM[Xt,Yt,Ct]
     wgt = SRAM[Kt,Ct]
    SYNC
    SEND output[Xt,Yt,Ct]
```

```
def inner(Xt,Yt,Ct):
 for x in range(0, X, Xt):
  for y in range(0, Y, Yt):
   for c in range(0, C, Ct):
    SYNC
    act = RECV[x:x+Xt][y:y+Yt][c:c+Ct]
    wgt=SRAM[Ct]
    output = fyY_hw(Xt,Yt,Ct)
    SYNC
    SEND output[Xt,Yt,Ct]
```

```
def inner(Xt,Yt,Ct,Kt):
 for x in range(0,X,Xt):
  for y in range(0,Y,Yt):
   for c in range(0,C,Ct):
    for k in range(0,K,Kt):
     SYNC
     act = RECV[x:x+Xt][y:y+Yt][c:c+Ct]
     wgt = SRAM[Kt,Ct]
     output = ck_hw(act,wgt)
   SYNC
   DRAM [Xt,Yt,Kt]
```

Figure 1.8: An example cross layer schedule for $PT \rightarrow DP \rightarrow PT$ layer.

PT which affects hardware utilization. Supporting multiple mapping strategies based on layer type requires dynamic data shape transformation capabilities between layer execution in the hardware.

We systematically explore the search space to find optimal cross-layer dataflow strategies in chapter 3 and provide the hardware microarchitecture which supports shape transformations in chapter 4.

Our Approach: X-Layer

Our first work is X-Layer where we solve the challenges of executing LR-CONVs efficiently for EDGE-ONLY approach using cross-layer dataflows. We introduce multiple novel ideas: i) **Leveraging Cross-layer dataflows to explore pipelining:** It then proceeds to fuse loop dimensions from multiple layers and schedule them across concurrent stages. This explores the tradeoffs in synchronization between pipeline stages and the size of the sliding window mapped to SRAM to capture inter-layer reuse. ii) **Fine-grain partial order loops:** We identify that LR-CONVs have different pipeline dependencies than CONVs and this enables a finer-grain dataflow. With LR-CONVs, the channel and filter loops can be partially complete before a subsequent LR-CONV is kickstarted; CONVs do not provide this opportunity. iii) **Fine-grain vs Coarse-grain Dataflow:** We explore that Coarse-grain is sensitive to on-chip memory and performance drops steeply as on-chip memory reduces below 1.5 MB. On the contrary, X-Layer fine-grain dataflows perform well for a wide range of SRAM ($\geq$ 32KB). iv) **Heterogeneous inner dataflow:** We find that each layer type in the cross-layer pipeline requires different inner dataflows. We develop microarchitecture techniques to support the flexible on-chip data movement required by each layer. There has been some work in pipelining CONVs: batch pipelining [36, 90, 95] and activation pipelining [10, 107]. Their hardware does not support LR-CONVs. Also, their dataflows are coarser and sub-optimal for LR-CONVs. Batch pipelines require 8.3$\times$ more on-chip SRAM than X-Layer. Activation pipelines [10] give up XY locality where X-Layer performs 16$\times$ faster.

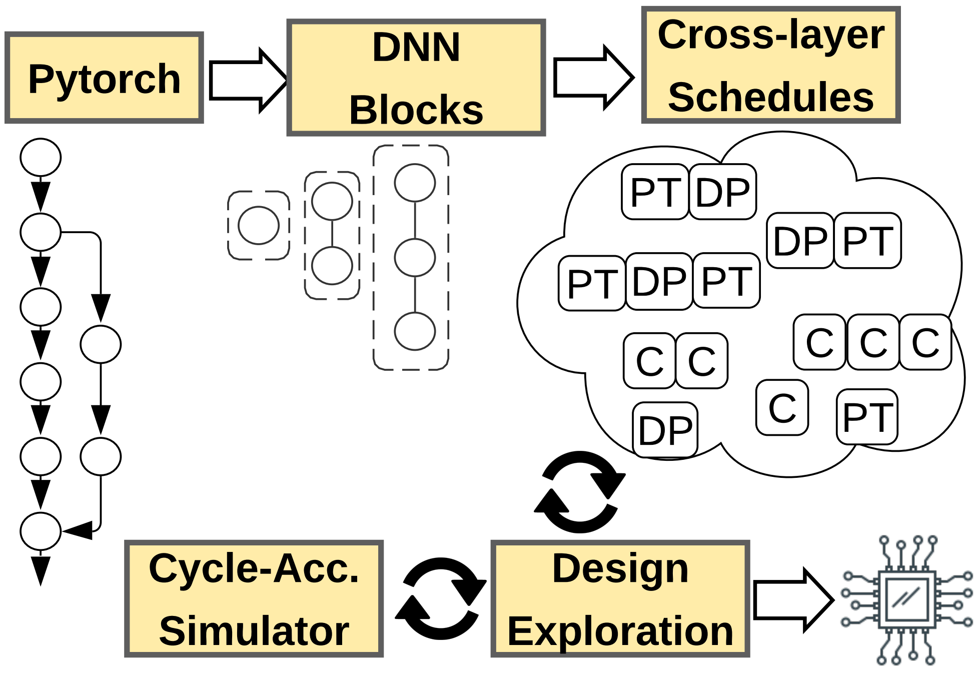


Figure 1.9: X-Layer Overview. DP= Depthwise, PT= Pointwise

Figure 1.9 illustrates X-Layer tool flow. X-Layer first partitions a DNN into composite blocks. The DNN blocks are formed based on the feasibility of pipelining the layers within a block. X-Layer schedules DNN blocks to execute as a concurrent pipeline in a cross-layer schedule. Each layer executes in a decoupled fashion but periodically synchronizes with the next stage in the pipeline to transfer a sliding window of activations. The loop order and blocking control within each layer involves a tradeoff between the size of the sliding window (which impacts on-chip SRAM required), frequency of synchronization (which impacts stage latency), and overall performance. To navigate this design tradeoff and find the Pareto optimal point, we use a multi-objective genetic algorithm [75].

The explorer selects the cross-layer dataflows and allocates on-chip buffers and mac units per layer. Finally, the optimal schedule is mapped onto the underlying hardware. To schedule a cross-layer dataflow, X-Layer splits the convolution loops of multiple layers in the pipeline into a single fused outer dataflow and several inner dataflows which execute in a synchronized manner. The inner dataflows are similar to per-layer dataflows [79]. We lower the scheduled dataflows onto a 2D grid of convolution accelerators. Since multiple layers can be simultaneously active on the hardware, X-Layer also addresses the issue of distributing the compute and SRAM resources in the grid to each layer.

Engineering efforts while developing X-Layer Figure 1.8 shows the large large design space and the questions that need to be answered for an optimal cross-layer dataflow design. The dataflow is closely tied to the hardware and it is impossible to explore the design space of Cross-layer dataflows without basic hardware support such as data transformation capabilities within the hardware, different mapping capabilities in the hardware, data movement between NN engines and, controlling the size of NN engines.

The design space of cross-layer dataflow requires developing an infrastructure to fuse multiple outer layer dataflows together. It should support loop order explorations for both single fused outer dataflow and multiple inner dataflows of the pipelined stages. We need to support these capabilities for primarily three-layer types CONV, DP and PT, where each layer type have different loop dependencies. For instance, in a three-stage CONV pipeline, with the loop order $X \rightarrow Y \rightarrow K \rightarrow C$ for stage 1. Partial loop executions of X,Y,K are allowed but we cannot kickstart stage 2 without finishing all channels (C). Such dependencies vary as the loop order changes.

We first started with a three stage templated cross-layer hardware with a fixed loop order $X^pY^pK^pC \rightarrow X^pY^pC^p \rightarrow CKXY$. The “^{p}" indicate partial loop orders being executed. This limited microarchitecture solution is similar to prior work Fused-Layer [10] and Tangram [36]. The hardware targeted FPGAs and was regenerated for every DNN. This work did not get published, and we had to iterate multiple times until we realized the extent of the large design space.

We realized that the outermost dataflow controls the granularity of the pipeline and has first-order effects on latency. The inner dataflow of each pipeline stage is tied to the mapping of the hardware and it is sufficient to fix them to the optimal mapping of each layer type.

We had to start from scratch multiple times and we ended up with a configurable ASIC microarchitecture with minimal features such as shape transformations which allowed us to explore the design space of cross-layer dataflows.

1.5 Speedup potential for SPLIT Approach

Table 1.2 shows the operating and application constraints when a DISTRIBUTED approach is preferred over an EDGE-ONLY approach. The main scenario is when the AI application does not fit on the

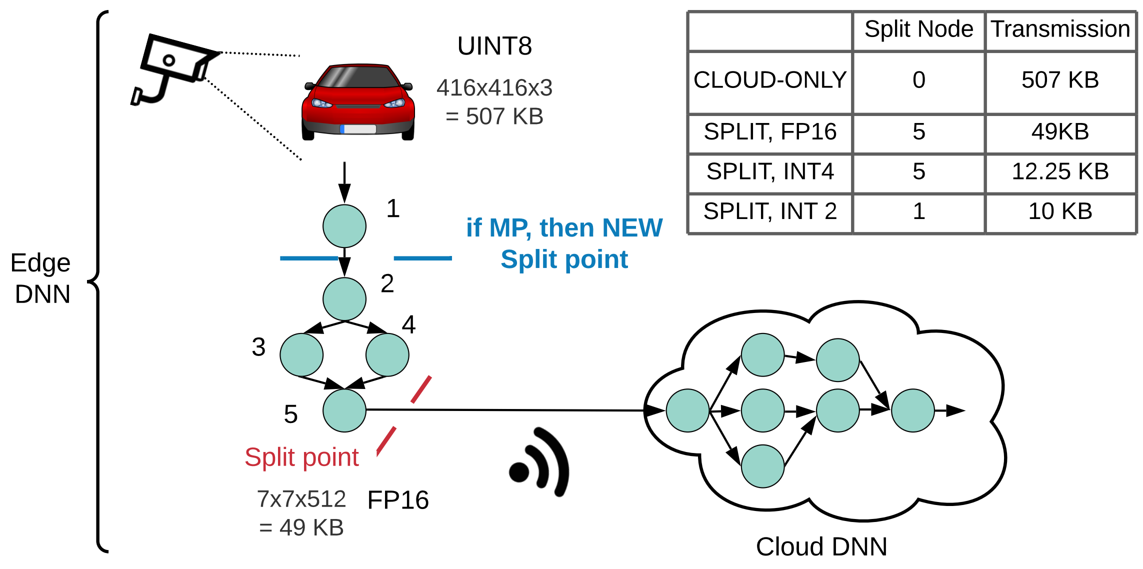
Table 1.2: AI Application scenario for DISTRIBUTED approach

Constraints	Scenario
Operating	DNN does not fit Network connection exists
Application	Latency Critical High Accuracy which require large models

edge device. This is especially true for large scale DNNs that are required for the high accuracy of the application.

Large scale DNN models are typically hosted in cloud servers with unrelenting computational power. At the same time, data is often distributed at the edge of the cloud, that is, the edge of various networks, such as smart-home cameras, authorization entry (e.g. license plate recognition camera), smart-phone and smart-watch AI applications, surveillance cameras, AI medical devices (e.g. hearing aids, and Fitbit), and IoT nodes. However, the gap between huge amounts of data and large deep learning models remains and becomes an arduous challenge for extensive AI applications. Connecting data at the edge with deep learning models at cloud servers is far from straightforward. Through low-power devices at the edge, data is often collected, and machine learning results are often communicated to users or passed to downstream tasks. Large deep learning models cannot be loaded into those low-power devices due to the very limited computation capability. Indeed, deep learning models are becoming more and more powerful and larger and larger. The grand challenge for utilization of the latest extremely large models, such as GPT-3 [16] (350GB memory and 175B parameters in case of GPT-3), is far beyond the capacity of just those low-power devices. For those models, the inference is currently conducted on cloud clusters. It is impractical to run such models only at the edge.

Since the input data is generated at the edge, a network connection is required to transmit i) either the input data (CLOUD-ONLY) or, ii) partially executed DNN output activations (SPLIT). Transmitting high resolution, high volume input data all to cloud servers (the Cloud Only Approach (CLOUD-ONLY) solution) may incur high transmission costs, and may result in high end-to-end latency. Moreover, when original data is transmitted to the cloud, additional privacy risks may be imposed. Recently, the DNN graph splitting methods has been explored [49, 58, 119]. The approach exploits the fact that the data size at some intermediate layer of a deep neural network (DNN for short) is significantly smaller than that of raw input data. This approach partitions a DNN graph into edge DNN and cloud DNN thereby reduces the transmission cost and lowers the end-to-end latency [49, 58, 119]. The edge-cloud collaborative approach is generic, can be applied to a large number of AI applications. Prior work [119] has shown that SPLIT approach can be up to 1.6 $\times$ faster than the CLOUD-ONLY approach. These techniques rely on creating a weighted DNN graph where each edge denotes the transmission cost in case of a split. Then a graph min-cut is calculated to detect the split point with the lowest end to end latency. The existing state of the art SPLIT approaches such as QDMP [119] execute the edge side of the DNN in floating-point precision which is not optimal.



	Split Node	Transmission
CLOUD-ONLY	0	507 KB
SPLIT, FP16	5	49KB
SPLIT, INT4	5	12.25 KB
SPLIT, INT 2	1	10 KB

Figure 1.10: Transmission cost comparison between CLOUD-ONLY and SPLIT. MP: Mixed precision

Joint bit-width allocation and split detection Figure 1.10 shows the intuition behind low latency of SPLIT solutions compared to CLOUD-ONLY approach. In the example, the CLOUD-ONLY approach requires 507 KB of data to be transmitted over the network whereas an existing SPLIT approach executing in 16-bit floating point precision (FP16), requires only 49KB of transmission. By leveraging, mixed precision quantization, one can see that the same split point suggested earlier need to transmit only 12.25 KB. However, when a mixed precision quantization is applied Node 1 was allocated 2-bits whereas Node 5 was allocated 4-bits. Thus, the optimal split point is Node 1 with 10 KB of data transmission. This also demonstrates the need for joint bit-width allocation and graph split point detection.

Mixed Precision Quantization (MP) The mixed precision quantization allocates different bit-widths to weights and activations of different layers such that the total accuracy of the DNN is within the acceptable limit. The intuition is that some layers are more compressible than others and does not affect the final accuracy. The mixed precision quantization requires the allocation of bit widths for weights and activations for every layer of the DNN. Given a DNN with “N" layers, and $\mathbb{B}$=$\{1,2,4,6,8\}$ bit-widths to choose, the number of possible combinations is $\mathbb{B}^{2n}$ which is exponential. Assuming a Resnet-50 example, the allocation of bit widths and execution of a DNN to verify the accuracy can take around $\simeq$8 minutes per configuration on a GPU. Thus, a brute force approach will require $\simeq 5^{2\times 50}\times 8 = 6.3\times 10^{70}$ minutes. The joint bit-width allocation along with split detection increases the search space further.

Next, we argue qualitatively, why the trivial solution of splitting the DNN (QDMP) and then applying MP or vice-versa does not work.

QDMP $\rightarrow$ MP: Finding split points using partitioning algorithms such as [49,58] and then applying mixed-precision quantization on the edge DNN is not sufficient since 1) the split point detected to partition into edge DNN and cloud DNN for reducing end-to-end latency can be different 2) This is because these algorithms do not account for edge device memory to split the DNN. Quantization reduces total memory requirement which in turn allows for deeper layers to be part of the search space for feasible solutions.

MP $\rightarrow$ QDMP: One can also consider applying mixed-precision quantization on the entire DNN first and then use those bit-widths in the edge DNN solution of the partitioning algorithms. (**Mixed precision $\rightarrow$ [49, 58]**). This is also not optimal since: 1) the split point detected did not consider quantization at first and it may be different from the optimal solution. Since the mixed-precision quantization used the entire DNN instead of partial DNN for quantization they will be conservative in assigning low bit-widths, 2) edge DNN quantization in edge-cloud model partitioning approach also requires more compression for activations at the split point, unlike the EDGE-ONLY approach where existing mixed precision techniques treat all layers equally for compression. For example, ZeroQ [17] (page 6) demonstrated mixed-precision quantization for weights and uniform 6-bit quantization for activations to reduce the ResNet-50 model size to 18.27 MB.

We observed that the transmission cost of output activations of the edge DNN can be further reduced by applying mixed-precision quantization along with DNN graph partitioning. However, this raises new challenges: a) existing techniques profile the edge, cloud, and transmission latency of each layer before applying the partitioning algorithm, but a naive mixed-precision quantization method will require this profiling to be done for multiple bit-widths which is computationally expensive, b) bit-widths of DNN layers need to be explored without dropping the accuracy too much.

Problem Formulation The problem requires jointly allocating bit-widths for weights and activations along with detecting the optimal split point “n". Given a DNN with “N" layers, the detected split point should also satisfy the constraints i) the $Edge_{DNN} \in [1,n]$ fits in the edge device memory, ii) the DNN error caused due to quantization is within the user acceptable limit. To summarize, the AUTO-SPLIT problem can be formulated as follows:-

$$\min_{\mathbf{b}^w,\mathbf{b}^a \in \mathbb{B}^n, n} \left(\sum_{i=1}^{n} \mathscr{L}_i^{edge} + \mathscr{L}_n^{tr} + \sum_{i=n}^{N} \mathscr{L}_i^{cloud} \right) \tag{1.1a}$$

$$\mathscr{M}^w + \mathscr{M}^a \leq M, \tag{1.1b}$$

$$\mathscr{F}(\mathbf{b}^w, \mathbf{b}^a) \leq E, \tag{1.1c}$$

where $\mathscr{M}^w$,$\mathscr{M}^a$ is weight and activation memory, M is total edge device memory, $\mathscr{F}$ is the DNN which is a function of weight and activation bit-width sets, and E is the acceptable Error limit set by the user. The $\mathscr{L}$ is the non-linear latency function that depends on the edge and cloud device. $\mathscr{L}_n^{tr}$ denotes the transmission latency of transmitting output activations from $Edge_{DNN} \rightarrow Cloud_{DNN}$.

Our Approach: AUTO-SPLIT

Our second work AUTO-SPLIT improves the end to end latency of AI applications deploying SPLIT solutions on the edge device. This work is explained in detail in chapter 5. In this section, we discuss the problem overview of AUTO-SPLIT. We realize that in SPLIT approach, we can reduce the latency of edge DNNs by leveraging MP. However, this increases the search space of splitting the DNNs further.

Fig. 1.11 shows an overview of our framework. The AUTO-SPLIT applies joint bit-width allocation and split detection to generate a quantized edge DNN which executes on the edge device and a Cloud DNN which executes on the cloud device. The output activations from edge DNN are packed and transmitted to the Cloud device for unpacking and fed as input to the Cloud DNN. Depending on the operating and application constraints, the AUTO-SPLIT can suggest one of the EDGE-ONLY, CLOUD-ONLY and SPLIT approaches. The input to the AUTO-SPLIT can be categorized into i) DNN specific inputs, ii) operating constraints and iii) application constraints. The DNN specific inputs are a pre-trained DNN and sample data for profiling to collect statistics for quantization. The operating constraints are edge device constraints such as device memory and hardware bit-width support, network constraints such as uplink bandwidth based on the network type (e.g., BLE, 3G, 5G, or WiFi). The application constraints include the error limit acceptable by the application user.

The AUTO-SPLIT is part of an edge-cloud collaborative prototype of Huawei Cloud. This patented technology is already validated on selected applications, such as license plate recognition systems with HiLens edge devices [18, 31, 82], is on its way for broader systematic edge-cloud application integration [33], and is being made available for public use as an automated pipeline service for end-to-end edge-cloud collaborative intelligence deployment [9].

Engineering challenges incurred while developing AUTO-SPLIT The Figure 1.11 can be split into input pre-processing, AUTO-SPLIT method and the output post-processing. We list the engineering efforts that were required to develop AUTO-SPLIT framework in each category.

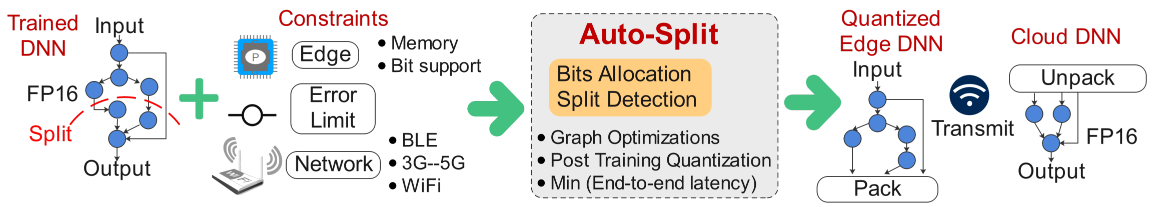


Figure 1.11: AUTO-SPLIT overview: inputs are a trained DNN and the constraints, and outputs are optimal split and bit-widths

DNN preprocessing: The DNN input needs to have DNN layers in a specific format for it to be quantized. The guidelines for preparing a DNN for quantization can be found in [3]. Also pre-processing of DNNs for inference on edge devices requires some layers to be in a particular order. For example, the layers $CONV \rightarrow BatchNorm \rightarrow Relu$, should be in the order presented for DNN processing to apply BatchNorm folding.

Mixed precision edge simulator: At the time of this research, there has been no open source edge device simulator that supports mixed-precision quantization. Existing academia solutions [37, 91] did not have the software stack and existing industry solutions [2, 5] only supported uniform 8-bit quantization. Thus an edge device hardware simulator had to be built from scratch. Prior work [30] used NAS based methods to explore DNN architectures by getting latency feedback from the edge devices. On further exploration, we found that it used an operator Look up Table (LUT) from the Samsung device. We tried using this setup but it did not work, since, the LUT was specific to only one DNN.

DNN layer dependencies: AUTO-SPLIT requires extracting DNN graph information. However, we used TorchScript for the DNN pre-processing stage for inference which i) serializes the DNN graph ii) omits several layers due to optimizations such as BatchNorm folding iii) applies name mangling to new layers. Thus we had to create our own mapping between old and new layers to reconstruct the graph information on the serialized DNN graph. This graph information is necessary to understand the layer dependencies. The layer dependencies are required to find the min-cut for the DNN graph split.

Splitting DNN: The infrastructure for AUTO-SPLIT has been developed in PyTorch. Given the split point, there is no easy way to generate the edge DNN and Cloud DNN. In PyTorch, this requires rewriting the edge DNN and Cloud DNN. For the video demo, we used Tensorflow.

Serializing activations for transmission: Since, there is no native support for sub 8-bit data types, i.e., $\leq 8-bits$. The data transmission from edge device to cloud device requires packing sub 8-bit data into an $signedint8$ data type before serializing the 3D activations. Then it requires to be decoded from serialized activations into a) 3D activations, b) into sub 8-bit data types. This also adds an unnecessary overhead which can be omitted with native sub 8-bit data type support.

Transmission protocol: We explored two transmission protocols i) RPC and ii) socket programming. We first explored RPC protocols used by prior research [49, 119], but it was slow and had a high overhead. Hence, we had to switch to Socket programming for low transmission overhead.

Chapter 2

Background

In this chapter we start with explaining the basics of DNN layers, followed by breakthrough innovations in DNN architecture §2.1. Next we divide the sections into EDGE-ONLY optimizations §2.2 and DISTRIBUTED §2.3 examples. The EDGE-ONLY optimizations lists the optimizations used across the DNN stack for EDGE-ONLY approaches and the DISTRIBUTED examples explains existing distributed approaches.

2.1 Deep Neural Network Basics

A DNN is a class of machine learning techniques inspired by a model of the human brain, the biological neural network. This section explains the basics of DNN and the two stages, namely training and inference. Then we explain the basic operations used in the DNN layers and the difference between the Spatial Convolution (CONV) and Low-rank convolutions (LR-CONV). Finally, we discuss emerging DNNs that heavily employ LR-CONVs to generate compact DNNs with low computation and memory requirements.

A DNN can be represented like a graph. The input to the graph could be an image, text, speech or any data. The output of the graph depends on the DNN task. Figure 2.1 shows an example of digit recognition. The input to the DNN is an image of a handwritten digit and the output is the recognized digit. The DNN can choose from ten classes of digits. Each node of the DNN graph is an operation. For example, a node can represent a convolution operation that consists of parameters such as weights and bias. The edges connecting the nodes pass activations. A DNN can have hundreds of these nodes.

The DNN has two stages, a training stage and an inference stage respectively. During **training stage** the DNN learns to classify an image into the correct digit classes. The parameters are initialized by random values and then the loss is calculated between the output of the DNN and the correct class. This can be done by several loss functions such as a KL-Divergence, Cross-Entropy loss, Mean Square loss, and Cosine loss. The most popular loss function is Cross-Entropy loss which is calculates the entropy or randomness between the predicted output “p" and actual output “q", given by $H(p,q)=-\sum p(x)log(q(x))$. The “p" and “q" are generally a one-hot vector of $1\times C$, where “C" is

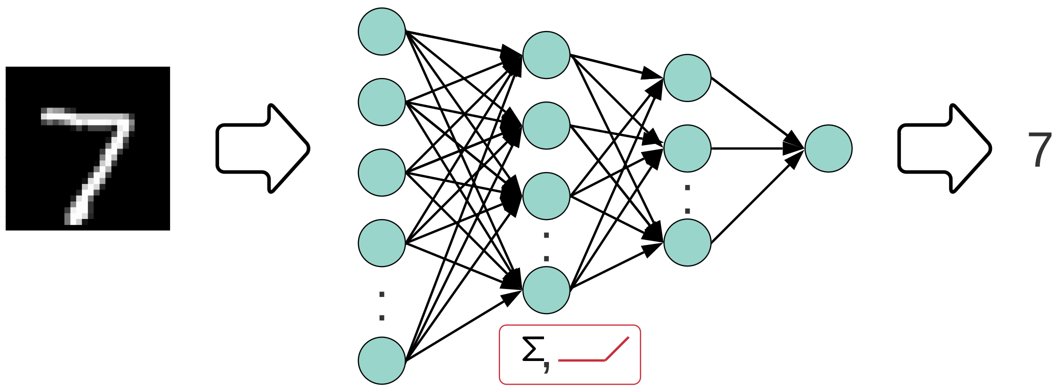


Figure 2.1: DNN example of digit classification from MNIST dataset

equal to the number of classes. During the training phase, the loss is then backpropagated to update all parameters as it goes through the dataset several times.

During **inference stage**, which is the focus of the thesis, the backpropagation algorithms are not used and the parameters of the DNN are constant. The activations need to be calculated during runtime through several operations. These operations/nodes are performed in stages and are also called layers of the DNN. These layers can be broadly categorized into a) convolution operations, b) arithmetic operations, c) activation operations and d) reduction operations. The convolution operations are where a DNN spends ($>95\%$) of execution time. We categorize them into spatial convolution (CONV) and low rank convolutions (LR-CONV). The low-rank convolutions are the type of convolutions where one or more dimension is reduced compared to CONV. For example, depthwise convolution (DP), pointwise convolution (PT), $1\times N$ convolution and $N\times 1$ convolution. The last layer of a DNN for Image classification is generally a fully connected (Fully-Connected Layer (FC)) layer which is an all to all connection between the nodes. The arithmetic operations between tensors include skip connections (addition), scale (multiply operations) and other arithmetic operations which can be applied on matrices. The activation operations are Relu, Tanh and Softmax. The reduction operations are pooling operations such as global, max and average pooling. We now explain how convolution operations work and how the depthwise separable function is functionally similar to a CONV operation.

Spatial Convolutions A two-dimensional discrete convolution can be given as

$$O(i,j)=(I*K)(i,j)=\sum_m\sum_n I(m,n)K(i-m,j-n) \tag{2.1}$$

where I is the input activation, K is the kernel and O is the output activation. Since, convolution is commutative, it can be equivalently written as :-

$$O(i,j) = (K*I)(i,j) = \sum_m \sum_n I(i-m, j-n) K(m,n) \tag{2.2}$$

The later is more straight forward to implement in machine learning libraries, because there is less variation in the range of valid values of m and n. However, many neural networks simplify this further and implement **cross correlation**, since, it does not require flipping of the kernel.

$$O(i,j) = (K*I)(i,j) = \sum_m \sum_n I(i+m, j+n) K(m,n) \tag{2.3}$$

This is what we call as convolution in the thesis. The spatial convolution (CONV) is a three dimensional convolution version of the above as shown in Figure 2.2. The input activation is of size

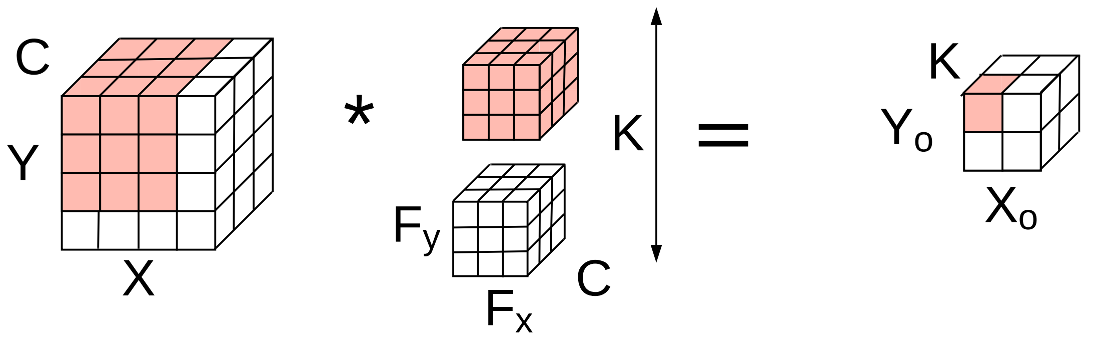


Figure 2.2: Spatial Convolution Example

$X \times Y \times C$. There are K filters each of size $F_x \times F_y \times C$, which produces the output activation $X_o \times Y_o \times K$. The dimensions of X_o and Y_o can be calculated as follows :-

$$\begin{aligned} X_o &= \left\lfloor \frac{X - F_x + 2 \times P_x}{S_x} + 1 \right\rfloor \\ Y_o &= \left\lfloor \frac{Y - F_y + 2 \times P_y}{S_y} + 1 \right\rfloor \end{aligned} \tag{2.4}$$

where S_x, S_y are strides and P_x, P_y are paddings in x,y dimension.

The CONV is a seven loop structure along the dimensions b, X, Y, K, C, F_x, F_y which is largely parallel. Thus, CONV operations have a lot of spatial and temporal parallelism which can be exploited by hardware. The order of loops affects the data reuse within the hardware. It determines which data stays stationary in on-chip memory and which data streams to execute convolutions. This loop order, schedule and data movement in the hardware is called the **dataflow**. A good dataflow can improve the latency up to 18× [20] and is the focus of this thesis. We explain dataflows in detail in §2.2.

Low Rank Convolutions The Low-rank convolutions are partial CONV functions that are created by reducing either one or more dimensions. Low-rank convolutions do the function of CONV but by breaking a three-dimensional CONV operation into two separate two-dimensional convolutions. It can be achieved either by using factorized kernels ($1\times N$ followed by $N\times 1$) or by using depth separable convolutions (*DP* followed by PT). The input and output activation dimensions remain the same but the multiply-accumulate operations (MAC) and memory requirements are considerably reduced.

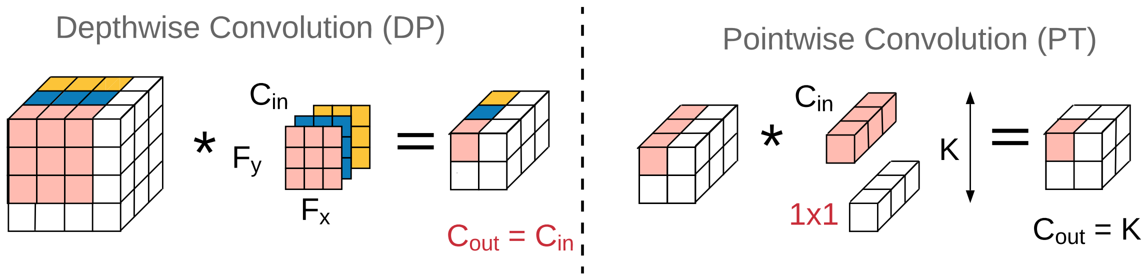


Figure 2.3: Depthwise and Pointwise Convolution Example

Figure 2.3 shows a depthwise convolution (DP) and a pointwise convolution (PT) layer. In DP each input channel convolves with only a single filter, i.e., C dimension is reduced. In PT the filter height and width dimensions are reduced to 1. The total computation cost of depth separable layer is $XYC(F_x{\times}F_y{+}K)$ whereas the computation cost of a CONV layer is $XYCF_xF_yK$. Thus, a depth separable layer requires $(\frac{1}{K}+\frac{1}{(F_x\times F_y)})$ times less MAC operations compared to the CONV layer.

Characteristics of DNN: Figure 2.4 shows a figure from the Google AI Blog [5]. It can be seen that for the same input and output activation, a CONV layer requires 173M MAC operations whereas an equivalent depth separable layer requires only 24M MAC operations. However, the CONV layers have $\simeq 3X$ more effective hardware utilization and thus executes faster on the Edge TPU.

LR-CONV types Figure 2.5 shows different kinds of layers which use convolutions. The residual block found in ResNet architecture uses CONV operations and skip connections. Other complex layers such as Inception, depthwise separable and inverted residual layers employ LR-CONVs. Table 2.1 lists the different DNN that employ LR-CONV layers. Many of these layers are composite layers, made up of multiple sequences of depthwise (DP) and pointwise convolution (PT). The old DNNs such as VGG and AlexNet does not include any of these composite layers.

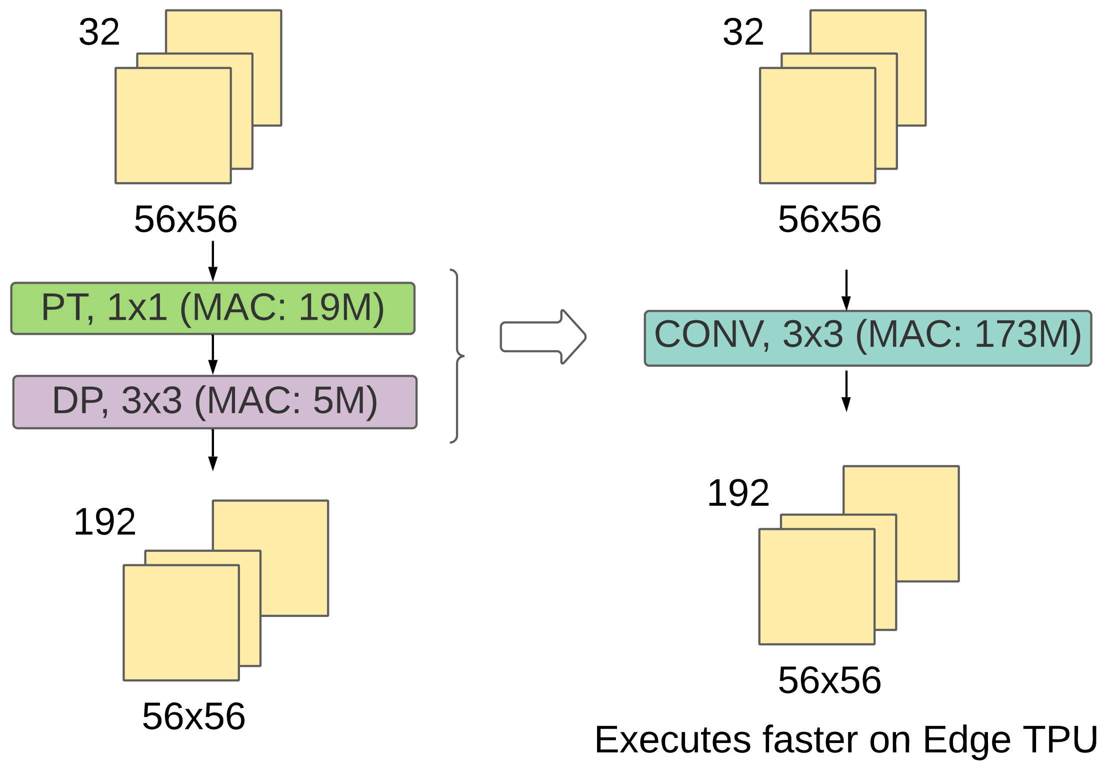


Figure 2.4: MAC requirement for CONV vs depth separable(DP + PT) layers [5]. CONV executes faster on edge TPU due to $\simeq 3X$ more effective hardware utilization.

Table 2.1: Operations used in DNNs. Inv Res.= Inverted Residual Layers, Depth sep.= Depth separable Layers

DNN	Inv. Res.	Depth sep.	Skip	Bottle Neck	1×7 Rank
Efficientnet [99]	✓	✓	✓	✓	-
Xception [27]	-	✓	✓	✓	-
DenseNet [52]	-	-	✓	✓	-
Inception-V3 [97]	-	-	✓	✓	✓
MobileNet-v2 [88]	✓	✓	✓	✓	-
MobileNet [110]	-	✓	✓	✓	-
NasNetMobile [98]	-	✓	✓	✓	✓
MnasNet1_0 [98]	✓	✓	✓	✓	-
ReNets [43]	-	-	✓	✓	-
ShuffleNet_v2 [71]	-	✓	-	✓	-
Squeezenet [53]	-	-	✓	✓	-
VGG [93]			—		
AlexNet [61]			—		

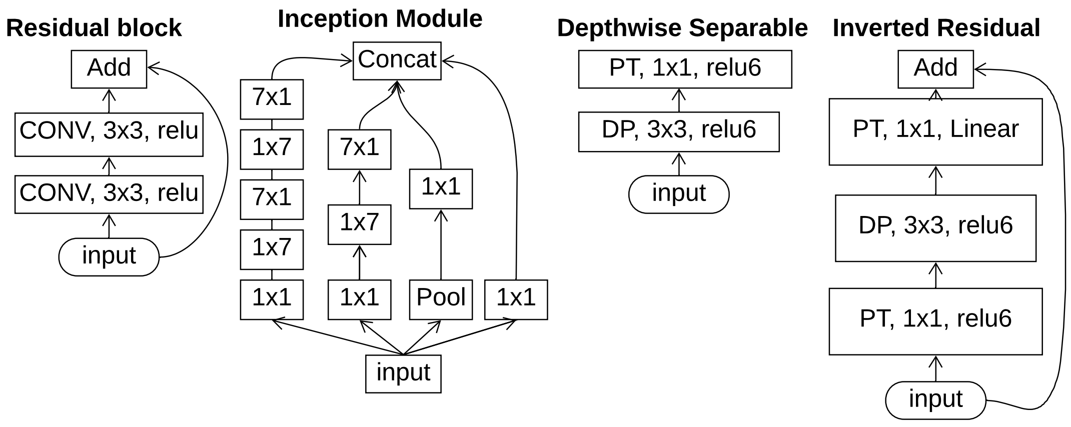


Figure 2.5: Advances in DNN Algorithm designs

Advances in DNN: In the 2012 Imagenet challenge, the first time a neural network beat the state of the art computer vision algorithms. This neural network was Alexnet that lead to a lot of research in computer vision.

It is important to study the characteristics of emerging DNNs, since, it affects the choice of dataflow and hardware accelerator microarchitecture. Therefore, we summarize the breakthrough innovations in DNNs below.

AlexNet, 2012: first successful neural network for image classification tasks consisting of 8 layers (5 CONV and 3 FC layer) with ImageNet Top-1 accuracy of 62.5%. AlexNet consists of 60M parameters. The majority of MAC operations belonged to FC layers but later DNNs after VGG have greater than 90% of MACs in CONV layers. Since later DNNs get rid of expensive FC layers with just one FC layer.

VGG, 2014 was the first deep neural network. It used only $3{\times}3$ kernels instead of $5{\times}5$ kernels used in AlexNet which reduced the number of computations per layer to allow for deeper neural networks. However, DNN is overparameterized. VGG-16 consists of 138M parameters and about 500MB of model size.

Inception-v1,v2,v3, 2014–2015: introduced the inception module (Figure 2.5), batch normalization, bottleneck layers and made use of LR-CONV. The idea of the bottleneck layer is to use the pointwise convolutions (PT) bottleneck layer to reduce the number of activations before the expensive 3x3 CONV. Inception was the first to introduce the idea that CONV layers do not always have to be stacked up sequentially. The inception module has $1{\times}1$, $3{\times}3$, and $5{\times}5$ convolutions all in parallel. Inception module also consists of factorized kernels i.e $1{\times}n$ and $n{\times}1$. Factorized kernels have much less computation cost compared to spatial convolution layers.

ResNets, 2015: introduced Skip connections aka shortcut connections or residual connections which allowed DNNs to go very deep– 100 to 200 layers. ResNets showed that naive stacking of layers to make the network very deep won't always help and can actually make things worse. To address the above issue, they introduce residual learning with skip connections. The idea is that

DNN Stack	**Optimization scope**
DNN Model	Model Compression, NAS, Operator fusion
Dataflow	Loop optimizations – Reorder, Tiling, Unrolling (Spatio-Temporal Trade-off)

Table 2.2: Optimization scope along the DNN stack

by using an additive skip connection as a shortcut, deep layers have direct access to features from previous layers. This allows feature information to more easily be propagated through the network.

MobileNets, 2017: replaced CONV layers with depthwise separable convolutions. Depth-separable convolutions is a single sequence of depthwise convolutions and pointwise convolutions, DP$\rightarrow$PT. It has lower computation requirements compared to CONV § 2.1.

Mobilenet-v2, 2018: introduced inverted residual network and linear bottlenecks with even fewer parameters compared to Mobilenets. Skip connection in original residual block follows a wide$\rightarrow$narrow$\rightarrow$wide approach concerning the number of channels and increased memory. On the other hand, MobileNetV2 follows a narrow$\rightarrow$wide$\rightarrow$narrow approach to reduce memory overhead.

The idea of a linear bottleneck is to use a linear block before the addition in skip connection, i.e., discard the last activation function. To prevent the loss due to narrow$\rightarrow$wide$\rightarrow$narrow approach.

Other emerging DNNs: that have followed, use a combination of LR-CONV, squeeze and excitation networks [50], network architecture search (NAS) [98, 109] based methods which tweak the DNN to run efficiently on a selected target device.

It can be seen that emerging DNNs heavily employ LR-CONVs. In the next section, we show how the DNN inference is optimized across the DNN stack to reduce execution latency and improve hardware utilization.

2.2 Edge Only Optimizations

End to end DNN inference on the edge device preserves privacy and reduces the overhead of communication with the cloud device. However, edge devices have limited on-chip memory and computation capability.

To accommodate large models without sacrificing accuracy, there have been many DNN model optimization techniques that explore the trade-off between model accuracy, model size and model computation requirements. Table 2.2 shows the optimizations applied in DNN model such as Model Compression, NAS and Operator Fusion which we discuss in §2.2.1. We especially focus on quantization which is used in our second work AUTO-SPLIT in chapter 5.

Given a DNN model, it needs to be scheduled and mapped onto a target hardware accelerator. The latency of the DNN inference depends on maximizing the hardware utilization and maximizing available data reuse. We discuss the background of dataflows in §2.2.2 which applies several loop optimizations and hardware mapping techniques to achieve low latency.

2.2.1 Model optimizations

DNN Model optimizations aim to explore DNN architectures along the Pareto-optimal curve of model accuracy, model size and model MAC requirements. The model optimization techniques can be largely divided into NAS, Model Compression, and operator Fusion.

Neural Architecture Search(NAS): There has been a rise in NAS based DNN architectures generate the most compact DNNs with fewer MAC and memory requirements compared to their hand-designed counterparts. NAS techniques employ deep learning algorithms to search for efficient DNN architectures. They solve an objective function that tries to minimize several parameters by taking feedback from the hardware. These parameters include DNN latency, power, on-chip memory and DNN error. Examples of NAS based DNNs are [30, 98, 99, 108, 109, 114]. NAS based DNNs are currently state of the art networks in various DNN tasks and heavily employ LR-CONVs.

Operator Fusion and DNN Scheduling: This is an important stage in pre-processing a DNN before deployment for inference. In this stage, several layers are fused together into a single layer. For example, a $CONV \rightarrow BatchNorm \rightarrow Relu$ can be fused into a single $CONV$ layer with fused Relu operation. The DNN scheduler then serializes the DNN graph and schedules one layer at a time for per-layer dataflows. In the case of a cross-layer dataflow, a sequence of composite layers are scheduled at a time on the hardware.

Model compression: DNNs are over-provisioned, and pruning and quantization are popular methods to reduce the size of the DNNs. Pruning uses several criteria to make the weights and activations below a certain threshold to zero [42]. Quantization restricts the bit-width of weights and activations and represents the values in fixed point [25, 56, 81, 85, 106, 121, 122]. This method significantly reduces the model size up to $8\times$ for a uniform 4-bit DNN but needs several optimizations to recover the accuracy. Model compression is an effective method to reduce the model size, and reduce latency since integer arithmetic on a hardware accelerator is at least $4\times$ faster compared to floating-point. The bit-width assignment for each layer is still an open problem since different layers have different sensitivity to bit-width reduction and can reduce the accuracy of the network. We explore this problem in chapter §5 where we solve the joint problem of bit-width assignment along with DNN graph splitting for distributed inference on edge and cloud devices.

Quantization

A quantization method converts a floating-point precision into fixed-point and uses fewer bits to represent the same number. This representation reduces the precision of the represented numbers. Since the DNNs are over-provisioned and have a lot of redundancy, the accuracy is generally not affected. This is especially true for computer vision-based DNNs. The DNN quantization requires consideration of design choices across many verticals. Table 2.3 summarizes those design verticals and available methods in each of those verticals.

Table 2.3: Different verticals for considering quantization

Quantization Method	When	What	Bits-Allocation	Finetuning
Symmetric	PTQ	Wgt Only	Uniform	Artifically Generated data
Asymmetric	QAT	Act Only	Mixed-Precision	Sample data
-	-	Wgt +Act	-	-

DNN quantization need to consider:- i) what quantization method to use, ii) When to apply DNN quantization, iii) What parameters to apply quantization, iv) how to allocate quantization bits to selected parameters, and whether to use extra data for fine-tuning and recovering accuracy. It can be noticed that the number of possibilities quickly explode. Next, we explain each of the design verticals.

Quantization Method? The quantization methods can be divided into a range based linear quantization or asymmetric quantization and a linear or symmetric quantization [125].

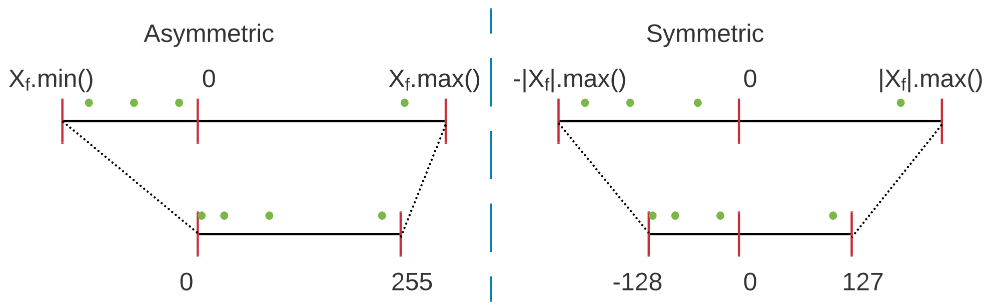


Figure 2.6: Asymmetric vs Symmetric Quantization methods

In **Asymmetric mode**, we map the min/max in the float range to the min/max of the integer range. This is done by using a zero-point in addition to the scale factor. Let's assume the original floating point weight tensor to be X_f, the quantized tensor to be X_q, the number of bits used for quantization be n, say n=8, then the quantization range is given by [0,255] as shown in Figure 2.6. The original floating point range is given by $[X_f.min(),X_f.max()]$. Any scalar value $x_f \in X_f$ can be converted to the quantized scalar value $x_q \in X_q$ as follows :-

$$
\begin{aligned}
x_q &= round\left((x_f - X_f.min()) \times \frac{(2^n - 1)}{(X_f.max() - X_f.min())} \right) && \text{(2.5a)}\\
&= round((x_f - X_f.min()) \times scale) && \text{(2.5b)}\\
&= round(scale \times x_f - scale \times X_f.min()) && \text{(2.5c)}\\
&= round(scale \times x_f - zero\_point) && \text{(2.5d)}\\
& && \text{(2.5e)}
\end{aligned}
$$

The (2.5) shows the asymmetric quantization of x_f to x_q which requires a $scale = \frac{2^n - 1}{X_f.max() - X_f.min()}$ and a $zero\_point = scale \times X_f.min()$. Note that the $zero\_point$ is represented by an integer in the quantization range whereas the scale is represented by a floating point value.

In **Symmetric mode**, instead of mapping the exact min/max of the float range to the quantized range, we choose the maximum absolute value between min/max. In addition, we don't use a zero-point. So, the floating-point range we're effectively quantizing is symmetric with respect to zero, and so is the quantized range.

$$
\begin{aligned}
x_q &= round\left((x_f - |X_f|.max()) \times \frac{(2^n - 1)}{2|X_f|.max()} \right) && \text{(2.6a)}\\
&= round(scale \times x_f - (2^n - 1)/2) && \text{(2.6b)}\\
&\simeq round(scale \times x_f) && \text{(2.6c)}
\end{aligned}
$$

In (2.6) the min and max values are replaced with $|X_f|.max()$. The (2.6)(b) shows the conversion which utilizes full range of quantized values. For example, given $n = 8$, then full range will utilize $[-128, 127]$. The (2.6)(c) shows the restricted range, $[-127, 127]$ where the $scale = (2^n - 1)/|X_f|.max()$.

The symmetric quantization leads to higher speeds but require larger memory and have high quantization error compared to asymmetric quantization. Image classification quantization is more robust to quantization errors compared to object detection algorithms.

When to quantize? The quantization process can be applied after the model is trained, called as Post Training Quantization (PTQ). Another way is to apply quantization while the model is trained with Quantization Aware Training (QAT). QAT requires re-training of the DNN models and can be expensive. However, it leads to high compression. For example, for an Imagenet dataset, the PTQ can take up to $\simeq 8 mins$ on a GPU when the bit widths are already allocated whereas the QAT can take up to $\simeq$8 hrs.

What to quantize? A DNN can have hundreds of layers and for each layer, we can choose to quantize weights only, activations only and both weights and activations. The weights are known offline and can be quantized statically before the DNN inference starts. However, activations are known during runtime and thus, need to be either scaled down or up to the specified bit widths during runtime. The symmetric quantization is hardware friendly and the quantization can be calculated by simply shifting the bits(equivalent to the scale) whereas the range-based quantization requires multiple operations, reducing the benefit of speed.

Bits-Allocation? One can choose to quantize each layer of the DNN with the same bit-width (uniform quantization), but it can suffer from accuracy and have less compression than mixed-precision quantization. This is due to the fact that some layers of the DNN are more sensitive to quantization than others. For example, in practice even for uniform quantization, the first few layers are not quantized. A mixed-precision quantization allocates different bit-width to activations and weights of different layers. This is a hard problem to solve. Suppose a DNN has N layers, and $K \in [1,8]$ allowed bit-widths to select. Then the total number of bit-width selection is $O(K^{2N})$ which is exponential. The “2" term is for weights and activation bit-width selection. The process of PRQ itself takes $\simeq 8 mins$ per configuration on a GPU, which can result in years of exploration time.

Fine tuning? The PTQ method can recover accuracy by fine-tuning with sample data from the same distribution or artificially generating sample data [32]. The most common method is to use the sample data from the same distribution/training data.

There is a wide range of methods besides quantization used for model compression. These methods have been proposed to address the prohibitive memory footprint and inference latency/power of modern NN architectures. They are typically orthogonal to quantization and include techniques such as knowledge distillation [47], model pruning [23, 44], or combination of pruning, distillation and quantization [23, 41, 74, 83].

Quantization Aware Training (QAT) requires training data to fine-tune and recover performance [25, 81]. However, this can be very time-consuming and requires access to the training data, which may not be possible. The alternative used is the uniform post-training quantization (PTQ) technique [13, 14, 56, 59, 66, 76, 120] that assigns equal bit-widths to all layers of the DNN. However, it has a lower compression ratio compared to other methods.

Directly quantizing all DNN layers to low precision can lead to significant accuracy degradation. Thus **Mixed Precision Quantization** [17, 32, 105, 108], uses different bit-precision for different layers. A naive mixed-precision quantization method can be computationally expensive, as the search space for determining the precision of each layer is exponential in the number of layers. To address this [105, 108] introduce network architecture search (NAS) and RL-based search algorithms to explore the configuration space. However, these searching methods are time-consuming, need access to the original training dataset, and require retraining the model which is expensive and sensitive to hyperparameters. Alternatively, recent works [17, 32] introduce low-cost mixed-precision post-training quantization methods, where the bit precision setting is based on the first and second-order

(Hessian) sensitivity of layers. These methods are currently used for the EDGE-ONLY approach and assume that the DNN always fits on the edge device.

In the next section, we discuss Dataflow optimizations which are used to improve hardware utilization and data reuse and reduce latency.

2.2.2 Dataflow

In §1.4 we provided the background for per-layer and cross-layer dataflows. In this section, we first provide the intuition behind low latency of cross-layer compared to per-layer dataflow followed by existing works for per-layer and cross-layer dataflows.

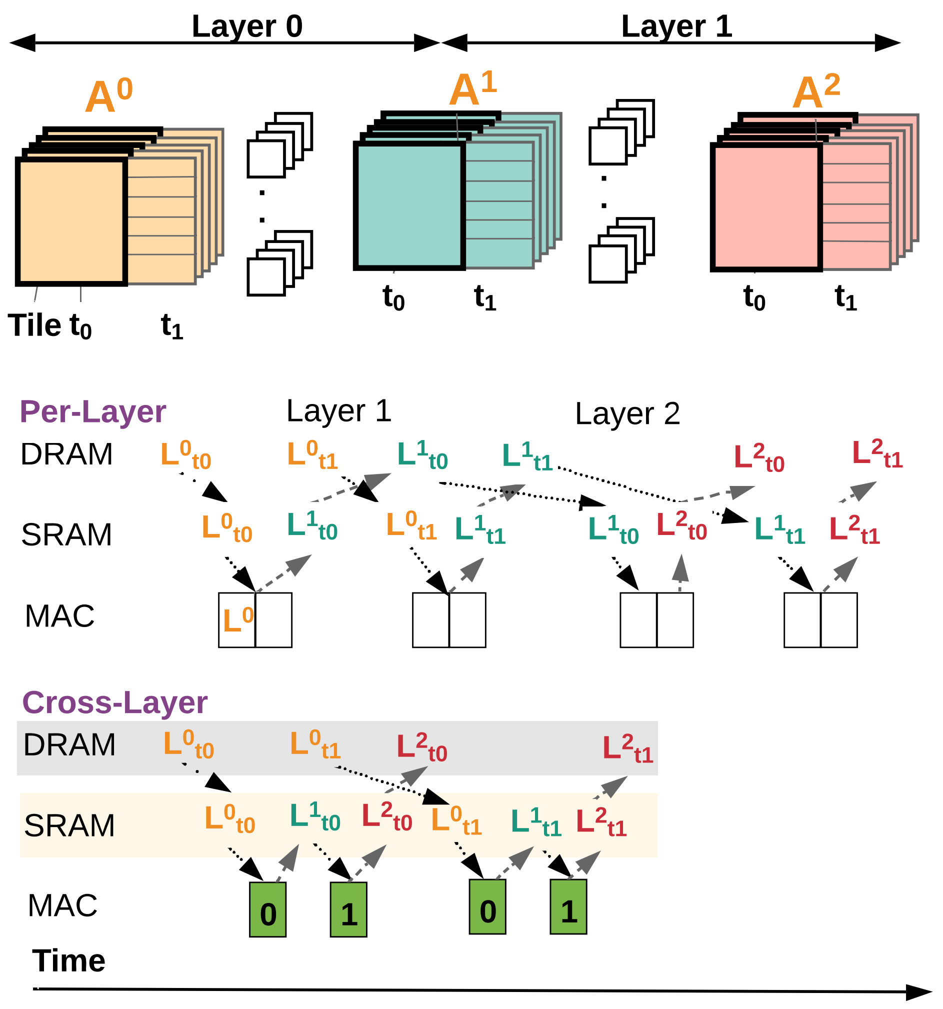


Figure 2.7: Per-Layer and a Cross-Layer dataflow

Per-Layer vs Cross-Layer dataflow Figure 2.7 compares the difference in data movement between per-layer and cross-layer dataflows. The Figure demonstrates two layers, Layer 1 and Layer 2. Layer 1 has input activation (A^0), and output activation (A^1). The second layer takes (A^1) as input and generates (A^2). Each activation has two tiles t_0,t_1. In the case of per-layer, the tiles move several times back and forth between DRAM and SRAM which increases DNN execution latency. The cross-layer dataflow stores activation tiles on-chip (SRAM) and minimizes data movement. Thus, cross-layer dataflow may have lower latency than per-layer dataflow but it comes at the cost of high on-chip memory. The on-chip memory requirement of a cross-layer dataflow can be reduced by reducing the amount of work done in each stage of the cross-layer dataflow pipeline or in other words making the pipeline granularity finer.

Prior work [10, 36] have proposed cross-layer dataflows before but have applied it to CONV based DNNs. Fuse Layer [10] explores fine-grain cross-layer pipelining and only uses 1-D vector products. Using a 1D vector product requires sacrificing spatial parallelism in the XY dimension. It only explores channel dimension and spatial reuse of filters. ALLO dataflow from Tangram [36] explores coarse grain cross-layer pipelining. It uses multiple NN engines to compute alternate pairs of CONV layers and pipeline two layers at a time. The cross-layer dataflow implementation is closely tied to the underlying hardware and thus none of the existing works explores the search space of cross-layer dataflows. In chapter 3 we formally define the taxonomy of cross-layer and provide methods to explore the search space of cross-layer dataflows.

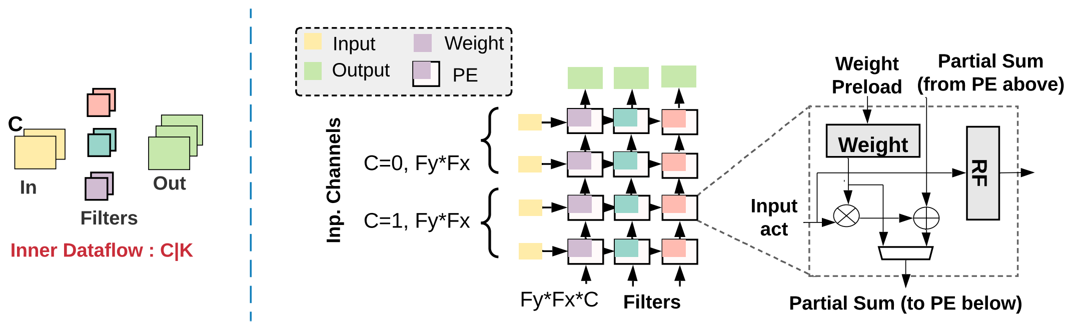


Figure 2.8: Mapping of C|K dataflow

Mapping per-layer dataflow to the hardware Figure 2.8 shows an NN engine which is two dimensional PE array. The DNN accelerator can have several of these NN engines. There has been a plethora of work exploring dataflow for per-layer [19, 20, 62, 115]. These works largely focus on inner dataflow and hardware mapping which is part of the hardware.

Earlier works [19] introduced the concept of dataflow based on which data stays stationary on the hardware. It introduced dataflows such as input stationary, weight stationary, output stationary and No local reuse. These dataflows only mapped one loop dimension to the PE array in space and other loop dimensions are executed temporally. [115] showed that such nomenclature is not unique and can

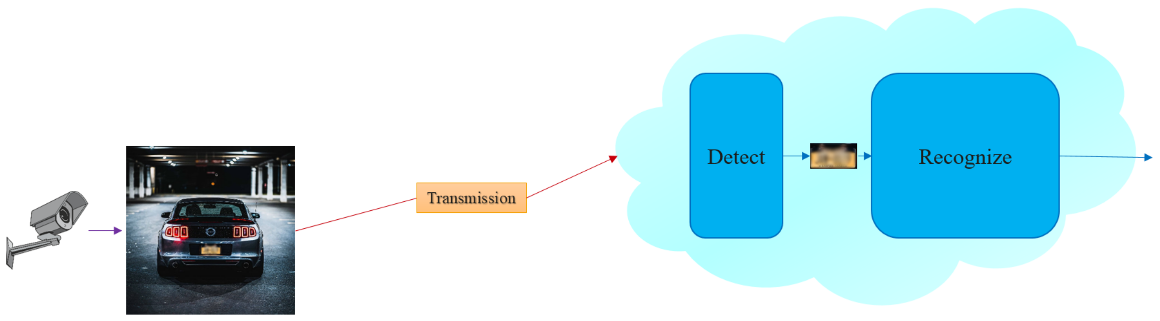


Figure 2.9: **Cloud-Only:** Data is transmitted entirely to the cloud where inference is executed.

incorporate multiple dataflows within each category. To avoid confusion, [115] proposed an intuitive method of naming the mapping strategy. We follow this nomenclature in our thesis. Figure 2.8 demonstrates a C|K mapping strategy. The left loop dimension “C" or channel dimension is mapped to the rows (X-axis) and the right loop dimension “K" or filters is mapped to the columns (Y-axis) of the PE array. The Y-dimension of the NN engine flattens the filters $F_y \times F_x \times C_t$ and executes a new filter in each column. [62, 115] have shown that this is the best mapping strategy for CONV and PT layers. However, it can result in poor utilization for DP layers as shown in chapter 3. In general, LR-CONVs have low MAC operations and data reuse within a layer (per-layer dataflow) and can benefit from inter-layer data reuse with cross-layer dataflows.

2.3 Distributed Examples

The DNN applications as explained previously in chapter 1, may not fit on the edge device even after model compression or may lead to a significant accuracy drop. In such cases, the DNN’s need to be executed with a distributed approach i.e., executing some part of the application on the edge device and the rest on the cloud device. The Edge cloud partitioning techniques can be broadly categorized into a) Inference on the cloud b) Cascaded Edge-Cloud Inference c) Multi exit models and d) DNN graph partitioning techniques.

Cloud-Only A trivial choice is to completely transfer the data to the cloud and let the vast amount of resources on the cloud do the processing (See Fig. 2.9). A major drawback of this approach is the lack of privacy as the original data is being transmitted outside the edge device. It can also lead to high latency, especially for high resolution/size inputs. However, due to its simplicity, this approach is widely used.

Cascaded edge-cloud inference [117] The task can be broken into multiple sub-tasks (Figure 2.10). It might be possible to run some of these sub-models on edge and only transmit their outputs to the cloud so the remaining models run on the cloud. For example, in a face recognition system, the edge model (trained independently) is responsible for face detection only. The detected faces are cropped and transmitted to the cloud where a separate recognition model (trained separately)

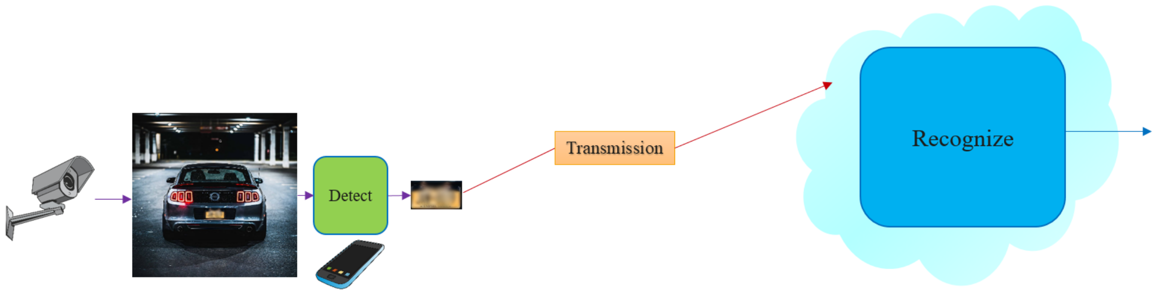


Figure 2.10: **Cascading approach:** The example here is license plate recognition where detection runs on device as it needs less computations. The recognition however requires heavier processing and runs on the cloud.

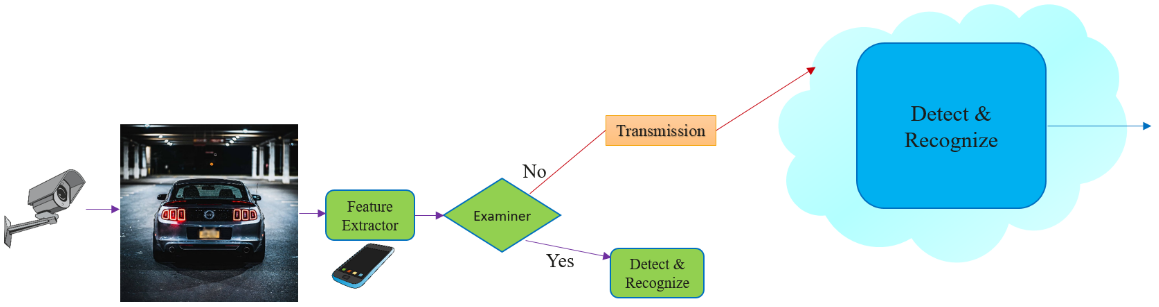


Figure 2.11: **Multi-exit model:** Some features are extracted from the data, and based on some criteria it is decided if the rest of the model runs on edge or needs to be transferred to the cloud. Generally, the edge and cloud side models perform the same task, but they have a different size and therefore different accuracy. The decision criteria tries to evaluate if running the model on edge can result in an acceptable accuracy, otherwise, the features data are transmitted to the cloud.

matches the incoming faces against a face database. Cascaded edge cloud inference works only when tasks can be divided into independent sub-tasks. Therefore, this technique is not generic and cannot be applied to arbitrary AI applications.

Multi-Exit Models Related to our work are progressive-inference/multi-exit models, which are essentially networks with more than one exit (output node). Figure 2.11 summarizes multi-exit models. A lightweight model is stored on the edge device and returns the result as long as the minimum accuracy threshold is met. Otherwise, intermediate features are transmitted to the cloud to execute a larger model. A growing body of work from both the research [51, 65, 67, 100, 118] and industry [55, 101] has proposed transforming a given model into a progressive inference network by introducing intermediate exits throughout its depth. So far the existing works have mainly explored hand-crafted techniques and are applicable to only a limited range of applications, and the latency of the result is non-deterministic (may route to different exits depending on the input data) [100, 123]. Moreover, they require retraining and re-designing DNNs to implement multiple exit points [65, 100, 118].

DNN Graph Partitioning : The most closely related works to AUTO-SPLIT are graph-based DNN splitting techniques [49, 58, 104, 119]. These methods are also the most generic methods for distributed approach. These methods split a DNN graph into edge and cloud parts to process them in edge and cloud devices separately. The aim of these techniques to reduce end to end latency compared to the CLOUD-ONLY approach. They are motivated by the facts that: a) data size of some intermediate DNN layer is significantly smaller than that of raw input data, and b) transmission latency is often a bottleneck in end-to-end latency of an AI application. Therefore, in these methods, the output activations of the edge device are significantly smaller in size, compared to the DNN input data.

The main idea of these techniques is to obtain a Directed Acyclic Graph (DAG) from the DNN where the edges represent the transmission latency estimate in case of a split between two given nodes. The end to end latency is given by

$$\mathscr{L}_{total}=\mathscr{L}_{edge}+\mathscr{L}_{tr}+\mathscr{L}_{cloud} \tag{2.7a}$$

The aim of the split solutions is to minimize $\mathscr{L}_{total}$. The split solution creates two DAGs, an edge DNN which executes on the edge device and its execution latency is given by $\mathscr{L}_{edge}$. Similarly, $\mathscr{L}_{cloud}$ is the execution latency of the cloud DNN on the cloud device. The $\mathscr{L}_{tr}$ gives the transmission latency obtained by transmitting output activations at the split from edge to cloud device. These techniques obtain a min-cut of the DAG which minimizes $\mathscr{L}_{total}$.

Fig. 2.12 shows a comparison between the existing works and AUTO-SPLIT. Earlier techniques such as Neurosurgeon [58] look at primitive DNNs with chains of layers stacked one after another and cannot handle state-of-the-art DNNs which are implemented as complex directed acyclic graphs (DAGs). Other works such as DADS [49] and QDMP [119] can handle DAGs, but only for floating-point DNN models. These works require two copies of the entire DNN, one stored on the edge device and the other on the cloud so that they can dynamically partition the DNN graph depending on the network speed. Both DADS and QDMP assume that the DNN fits on the edge device which may not be true especially with the growing demand of adding more and more AI tasks to the edge device. It is also worth noting that methods such as DADS do not consider inference graph optimizations such as BatchNorm folding and activation fusions. These methods apply the min-cut algorithm on an un-optimized graph, which results in a sub-optimal split [119].

Existing works only work for floating-point precision and require two entire copies of DNN, one stored on the edge device and the other on the cloud, so that they can dynamically partition the DNN graph depending on the network speed. Both DADS and QDMP assume that the DNN fits on the edge device which may not be true especially with the growing demand of adding more and more AI tasks to the edge device (see chapter 1).

Distributed approach with model compression The existing works on edge cloud splitting, do not consider quantization for split layer identification. We wonder what happens if the model

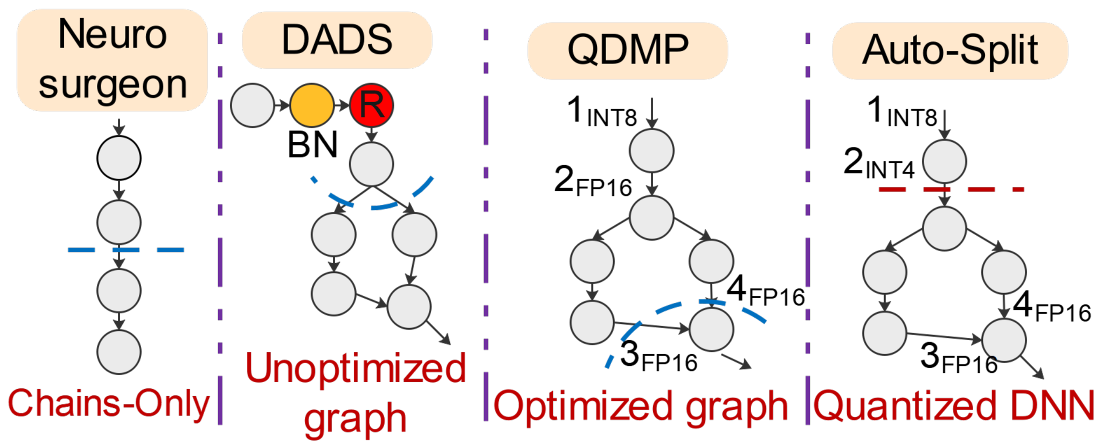


Figure 2.12: Different methods of edge-cloud partitioning. Neurosurgeon [58]: handles chains only. DADS [49]: min-cut on un-optimized DNN. QDMP [119]: min-cut on optimized DNN. All three use Float models. AUTO-SPLIT explores new search space with joint mixed-precision quantization of edge and split point identification. *BN*: Batch norm, *R*: Relu.

compression techniques could be applied to the edge DNNs in Distributed approach. This can lead to a faster end to end AI applications as i) the edge DNN will execute faster due to model compression on the edge device and, ii) mixed-precision quantization also reduces transmission activation latency. Thus we discuss two ideas i) QDMP $\rightarrow$ mixed-precision ii) mixed-precision $\rightarrow$ QDMP

To summarize, in this chapter we first covered the basics of deep neural networks (DNN) and the advances in DNNs which have led to a new class of DNNs based on LR-CONVs. We then covered the optimizations used in the Edge-Only approach followed by examples of a Distributed approach.

In Edge-Only optimizations, we explained the optimizations used across the DNN stack to improve application latency and reduce energy. We showed that LR-CONVs lead to compact DNNs with low MAC operations but still end up with high latency on existing hardware accelerators due to poor hardware utilization. LR-CONVs also have low data reuse Per-layer and can exploit inter-layer data reuse with the help of cross-layer dataflows. We also found that the dataflow implementation is closely tied to the hardware microarchitecture. In the next chapter §3, we define the taxonomy of cross-layer dataflows followed by the microarchitecture tweaks 4 required to support any cross-layer dataflow.

In the Distributed approach, we showed that existing DNN graph partitioning approaches are generic and can be applied to reduce end to end DNN execution latency. However such graph partitioning approaches assume the entire DNN fit on the edge device which may not be the case. These methods work for floating-point models only and may benefit from model compression for edge DNNs. In chapter 5 we propose a new method to jointly compress the edge DNN along with model splitting. It also considers environment constraints such as edge device memory, network speed and acceptable accuracy by the user.

Chapter 3

Composable Pipelined Dataflows for Low-Rank Convolutions

In this chapter we first start with discussing the motivation for exploring cross-layer dataflows for LR-CONVs in §3.1. Next in §3.2, we discuss cross-layer dataflow pipelining with our nomenclature of partial loops with an inverse residual layer example. In the end, we define the taxonomy of cross-layer dataflows in §3.3.

3.1 Motivation

In this section, we first explore the characteristics of LR-CONV based DNNs in §3.1.1. The LR-CONV DNNs have low MAC and memory requirement, can be memory bound, and have inter-layer data reuse. In §3.1.2, we introduce the terminology used by X-Layer and discuss the execution of partial orders for LR-CONVs in §3.1.3.

We then discuss the challenges of creating cross-layer for LR-CONVs. First, we explain inefficient mapping of DP layers (§3.1.4). Second, we discuss the need for heterogeneous inner dataflow (§3.1.5) and, third we explain the need for heterogeneous resource allocation (§3.1.6).

3.1.1 Emerging DNN characteristics

The majority of prior DNN accelerators [1, 19] and dataflow toolflows [62, 115] have targeted spatial convolutions (CONV). However, state-of-the-art networks (see Imagenet leaderboard [4]) employ low rank convolutions (LR-CONVs) for their workhorse layers e.g., AmoebaNet [84], RandWire [112], EfficientNet [99], Mnasnet [98], NasNet [127] and PNasNet [68]. LR-CONV networks have only one or two layers of CONV and spend 95%+ of the time in LR-CONVs.

Low MAC and Memory requirement for LR-CONVs: A single CONV layer requires 3D filters to execute. The filter has height, width and channel dimensions. A low-rank convolution has one or more dimensions reduced for its filters. For example, in a PT the filter size is $1\times1\times Channels$. Similarly, a Depthwise Convolution (DP) has $F\times F\times 1$ filter dimensions. In a DNN, multiple LR-CONV layers can be used to replace a CONV layer. For example, a depth-separable block is commonly used in-lieu of CONVs. This is desired since the two LR-CONV blocks combined have lower MAC and Memory requirements compared to a CONV block. A DP has only “F^2" arithmetic intensity compared to “K" for PT and “$K{*}F^2$" for CONV layers.

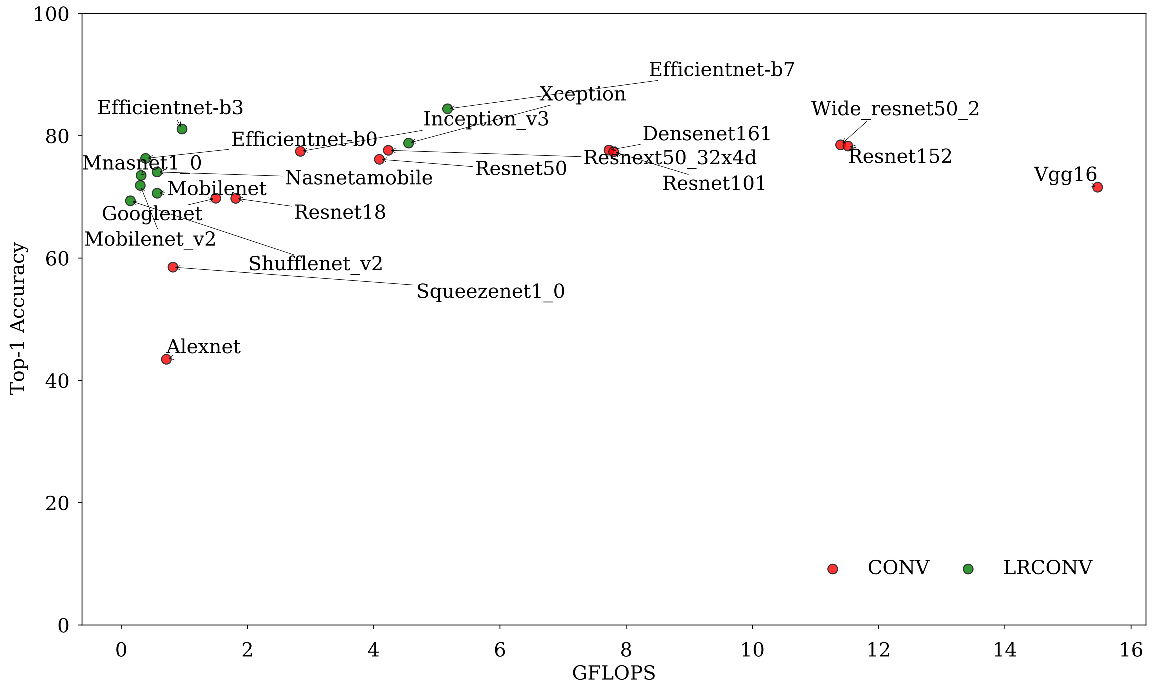


Figure 3.1: MAC requirement for LR-CONV vs CONV dominated DNNs. Top-Left is best.

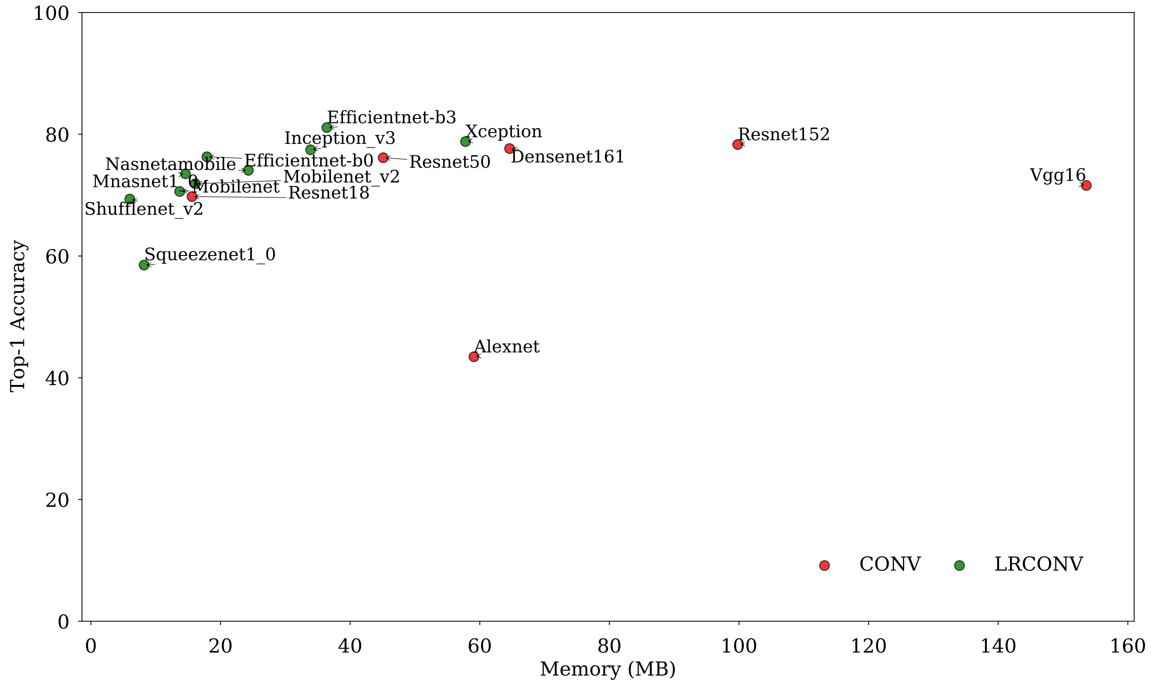


Figure 3.2: Memory requirement for LR-CONV vs CONV dominated DNNs. Top-Left is best

Figure 3.1 shows a scatter plot with various CONV and LR-CONV dominated DNNs. The Y-axis shows the Top-1 Imagenet accuracy and the X-axis shows the MAC(GFLOPS) operations required by each DNN. The LR-CONV DNNs are amongst the ones with the highest accuracy and require the least amount of MAC operations. The low MAC operations per layer allow for the accommodating of deeper layers in the DNN within a given MAC budget, thus increasing the accuracy. For instance, two comparable DNNs Efficientnet-b3 and Resnet-152 have 81.1% and 78.3% Top-1 Imagenet accuracy but the LR-CONV DNN has $4\times$ fewer GFLOPs compared to Resnet-152.

Figure 3.2 compares memory requirement for LR-CONV vs CONV based DNNs. The old DNNs Alexnet and VGG16 require almost 59 MB and 153 MB of memory and have poor Top-1 accuracy of 43.5% and 71.6%. The Resnet-50 which came after VGG16 introduced skip connections and have PT layers in them. PT layers along with skip connections allowed Resnet-50 to go 50 layers deep compared to only 16 layers for VGG16. In spite of having more Layers, Resnet-50 requires $3.4\times$ less memory compared to VGG16. Resnet-50 did not employ DP layers and were still dominated by CONV layers. Emerging DNNs such as Efficientnet-b0 has similar Top-1 accuracy of 76.3% compared to Resnet-50 (71.6%) but requires $2.5\times$ less memory and $10.6\times$ less MAC operations.

Overall LR-CONV dominated DNNs have higher $Accuracy/MAC$ and higher $Accuracy/Memory$ compared to CONV dominated DNNs.

DP Convolutions are Memory Bound

For LR-CONV DNNs, the DP arithmetic intensity is $18\times$– $79\times$ *less compared to PTs. It is* $79\times$ *less for Xception.*

In Figure 3.3, we plot the mac (lines) and memory (bars) requirements of low-rank convolution layers from Mobilenet_v2. Xception employs a sequence of depth separable layers (DS on the x-axis) and Mobilenet_v2 implements a sequence of inverted residual layers (INV on the x-axis). The layers are listed in sequence based on where they appear in the network, early, middle and late. We observe that the “#MACs" is low for DP compared to CONV and PT layers whereas the “Memory" requirement is similar across adjacent layers. For example, compare DP in “Inv3" with adjacent PT layers in Mobilenet_v2 to compare the arithmetic intensity ($\#MAC/Memory$). The memory required for DP is higher than adjacent layers but #MAC is low. This trend can be observed across all Invs (Inv2–Inv17). For Mobilenet_v2, the arithmetic intensity for DPs is $32\times$ less than PTs.

In chapter 1, Fig. 1.4 lists the analytical expression to compare the arithmetic intensity of CONV vs DP and PTs. Out of DP and PT, the majority of MAC operations are distributed to PT layers making DPs memory bound. This leads to under utilization of the mac units [38, 62].

3.1.2 Terminology: Outer and Inner Dataflow

Fig. 3.4 lists the outer and inner dataflow for a PT layer. The **pipeline depth** of a cross-layer dataflow is the number of layers which are executed simultaneously. The CONV, DP and PT layers have up to seven loop dimensions which are split into outer and inner loops. To pipeline several layers, X-Layer

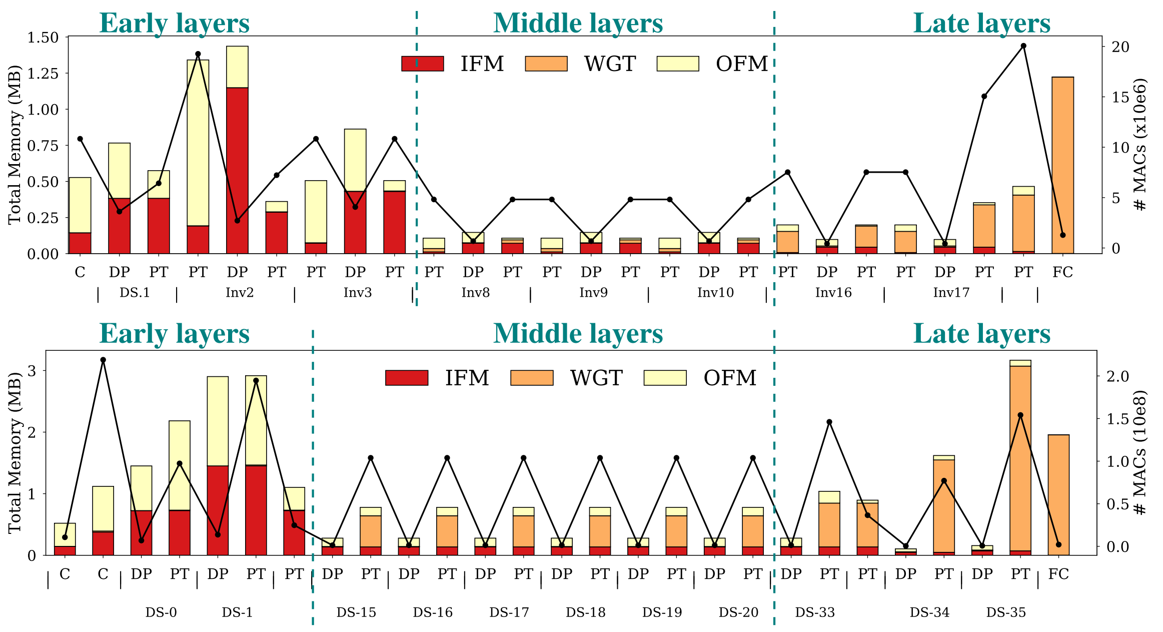


Figure 3.3: Mobilnet_v2(top) and Xception(bottom): Breakdown of #MACs (line) and memory required per layer (bars). Inverted residual (Inv) : PT→DP→PT. Depth-separable (DS): (DP→PT). The numbers suggest layer index, e.g., Inv3 implies third inverted residual layer

fuses the outer loops to create a single outer-dataflow (“Fused Outer(), XYKC") and synchronizes several inner-dataflows (“Inner(), KCXY") to execute simultaneously. Each pipeline stage executes a single layer and the pipeline synchronization is controlled by the loop order of outer loops. The sliding window of SRAM along with the inner loop order controls the amount of work done in each pipeline stage before synchronization.

Timeloop [79] schedules a single outer dataflow and a single inner dataflow for a single layer. On the other hand, X-Layer creates concurrent pipelines for multiple layers where the outer dataflow is fused together to generate a single outer dataflow for all layers. The multiple inner dataflows are synchronized to execute simultaneously. We explain it in detail later §3.3.1.

The outer dataflow loads the data from DRAM in tiles to on-chip SRAM. X-Layer, loop order and blocking parameters (X_b,Y_b,C_b,K_b) of outer dataflow in a stage control: i) the minimum on-chip memory required for each pipelined layer stage, ii) the synchronization granularity between layers and how often data is moved between them, and iii) the data reuse across layers and how long data is held on-chip before being consumed by a subsequent stage. The inner dataflow controls the loading of data from on-chip SRAM to the neural network (NN) engine. Essentially, it is expressing the convolution as a collection of NN operations on fixed tiles (X_t,Y_t,C_t,K_t). The inner dataflow (“Inner(), KCXY") is synonymous to several dataflows discussed in previous literature [19, 20, 62, 115]. However, prior work in cross-layer used the same inner dataflow for both outer and inner [10, 36] dataflow. This is sub-optimal for LR-CONVs as we discuss in § 3.1.5. Finally, the “hardware, $C|K$, Weight Stationary" controls the execution of the Neural Network (NN) engine. While we have shown

B	# Batch
X,Y	Activation Width and Height
C	# Activation Channels
K	# Filters
Fx,Fy	Filter Width and Height
$_b$	DRAM→SRAM Blocking parameters. Also determines size of On-chip buffer params
$_t$	Tiling parameters. Controls how on-chip buffer is processed in tiles by the grid-based neural engine.
$_P$	Partial order completion of loops. See § 3.1.3.

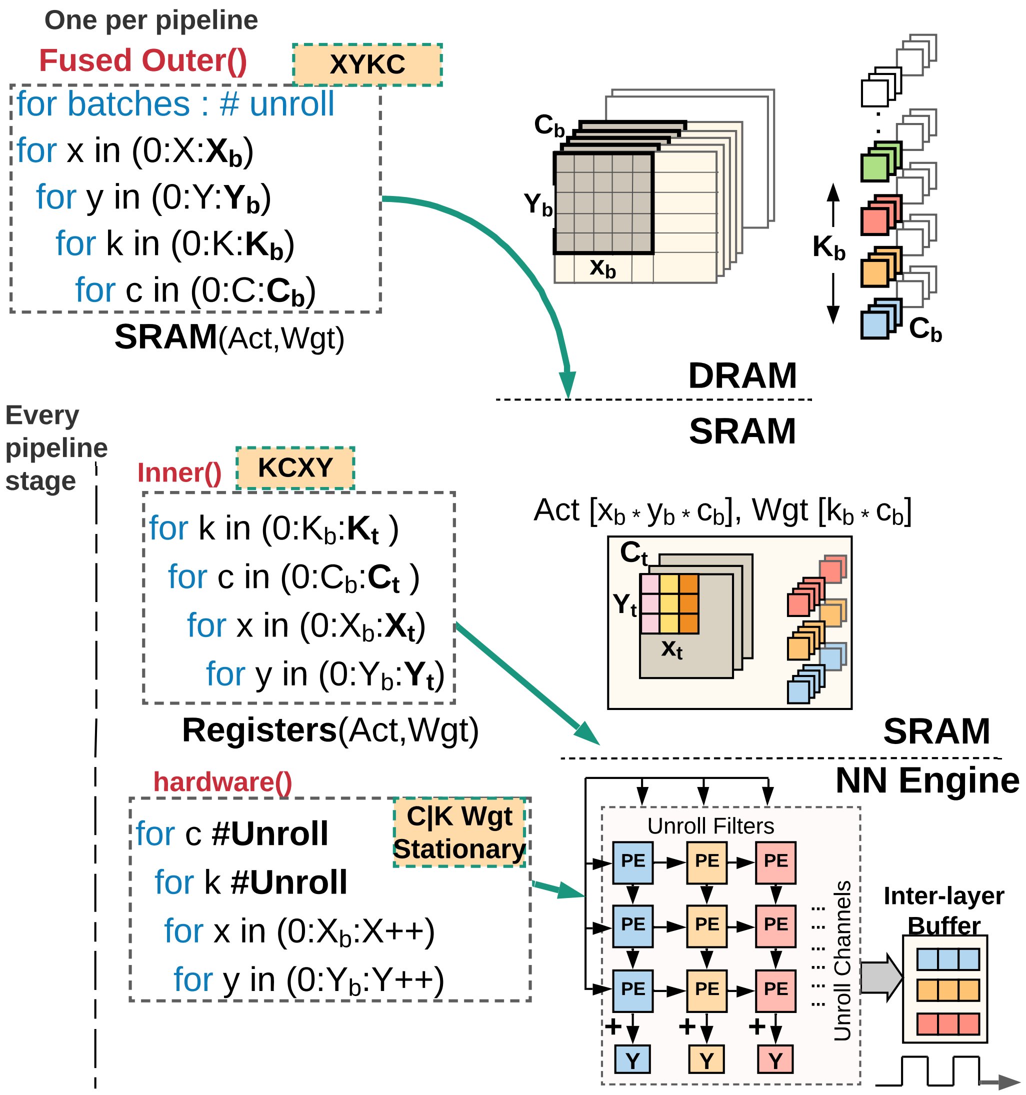


Figure 3.4: Illustration of the PT layer in a cross-layer.

the hardware as loops for clarity, the actual execution is directly controlled by the hardware. Two or more dimensions are unrolled in the hardware depending on the underlying hardware.

3.1.3 Partial-order ("p") pipelines for LR-CONV

Low-rank convolutions provide opportunity to create finer-grain pipelines with higher degree of concurrency, and more inter-layer reuse than spatial convolutions.

This is a key design parameter unexplored by prior work. The superscript "^{p}" indicates that a loop dimension is partially executed before synchronizing with the next stage in a cross-layer. Note that ^{p} and $_b$ serve orthogonal roles in the dataflow. $_b$ determines the tile dimension being loaded from the DRAM. The parameter ^{p} impacts multiple factors: i) it specifies the tile of inter-layer activations retained on-chip and captures the producer-consumer locality. ii) it determines the synchronization granularity between pipelined layers and the computational parallelism within a layer.

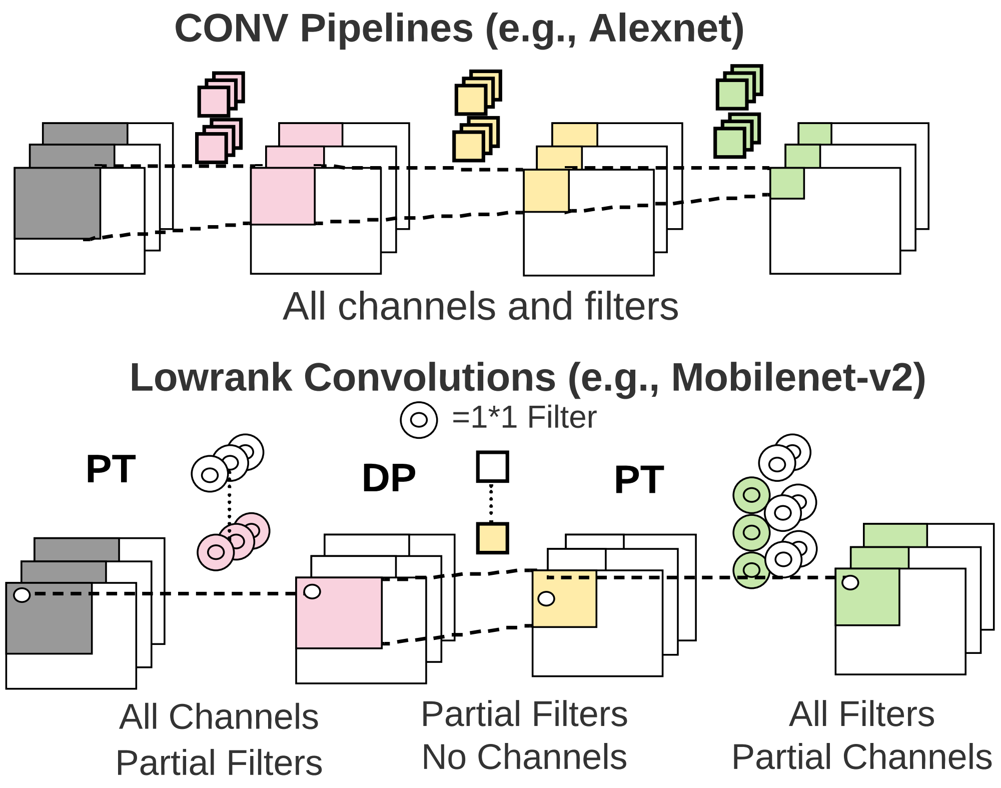


Figure 3.5: Cross-layer for CONVs vs LR-CONVs.

CONVs dependencies restrict parameter ^{p}. Cross-layer dataflows for CONVs [10] do not (and cannot) explore K^p and C^p [36]. Figure 3.5 illustrates a state-of-the-art [10] X^pY^pKC dataflow for $CONV \rightarrow CONV \rightarrow CONV$. Filter and Channels need to be completed before kickstarting the next stage; only X,Y dimensions can be partially executed and even that has been fixed due to hardware constraints (e.g., Alwani et al. [10] only explored X=Y=1).

An LR-CONV based pipeline, allows for a finer-grain pipeline ($X^pY^pK^pC$). It needs to calculate only partial activations and filters in all stages. The figure shows an inverted residual layer, i.e., $PT \rightarrow DP \rightarrow PT$. The first stage (PT), needs to execute only one loop dimension completely, channels.

The second stage DP has a dedicated filter per input channel, all dimensions can be executed partially in Stage 2 of the pipeline. In stage 3, the partial input activation ($X_b \times Y_b \times C_b$) can convolve with all filters with partial channels (K_b,$1 \times 1 \times C_b$).

The "^{p}" factor controls 1) the amount of locality in a dimension, 2) on-chip SRAM and, 3) layer synchronization time. The lack of "^{p}" factor is equivalent to ^{p}=$Complete dimension$. For example, a ^{p}=1 implies no locality, and frequent synchronization, whereas ^{p}=$Complete dimension$ or absence of ^{p} factor indicate maximum per-layer locality with maximum SRAM overhead.

Depth-separable layers (DP$\rightarrow$PT) permit $X^pY^pC^p$ and $C^pX^pY^p$. This is due to the fact that, a DP layer has no filter dimension to unroll. Thus, unlike CONV layer, a partial computation of an input activation $X^pY^pC^p$ leads to an output activation which can be further processed by next layers.

3.1.4 Inefficient Mapping of DP layer

Figure 3.6 shows the "C|K" mapping strategy for a CONV and DP layer. In a "C|K" mapping strategy, the input channels (C) are fed to the rows of the NN engine and the filters (K) are fed to the columns of the NN engine. In a CONV layer, the input activation is reused across all filters. For example, the input channel C_i=0 convolves with C=0 of all filters (K=$\{0,1,2\}$). Thus in the given example, a CONV layer has a 100% MAC utilization.

In case of a DP layer, each channel has a dedicated filter i.e., the input channel C_i=0 convolves with C=0 of a single filter (K=0). Thus, DP layer has minimal input activation reuse. This also leads to poor hardware utilization. We ran Mobilenet [110] using row stationary dataflow ([19]) on [87] and found the mac utilization of $\simeq$7%. Similarly for Mobilenet, [38] reported that though DP layers are few in number it took $\simeq$100% of the runtime. When DP layers were executed on CPU instead, it took 18% of runtime. In another study [5], it was reported that a CONV layer executes faster on edge TPU compared to an equivalent depth separable layer (DP + PT) with same input and output activation dimensions. The CONV layer had 173 million MACs compared to 24 million MACs for depth separable layers (DP: 5 million, PT: 24 million). The reason is due to the severe underutilization of edge TPU for LR-CONVs.

DP is Memory bound: Since input activation reuse is reduced in DP layers, it is also memory bound. For LR-CONV DNNs, the DP arithmetic intensity is 18$\times$– 79$\times$ less compared to PTs. It is 79$\times$ less for Xception.

Since, LR-CONVs have low arithmetic intensity and data reuse within a layer, it can make use of inter-layer data reuse through produces consumer data movement. This makes cross-layer dataflows lucrative. In the next section, we explore the search space of cross-layer dataflows.

3.1.5 Heterogeneous Inner Dataflow

A cross-layer dataflow needs to vary the inner dataflow for each LR-CONV layer in the pipeline. For example, we ran Mobilenet (93% LR-CONV layers) on Eyeriss and resulted in a MAC utilization of

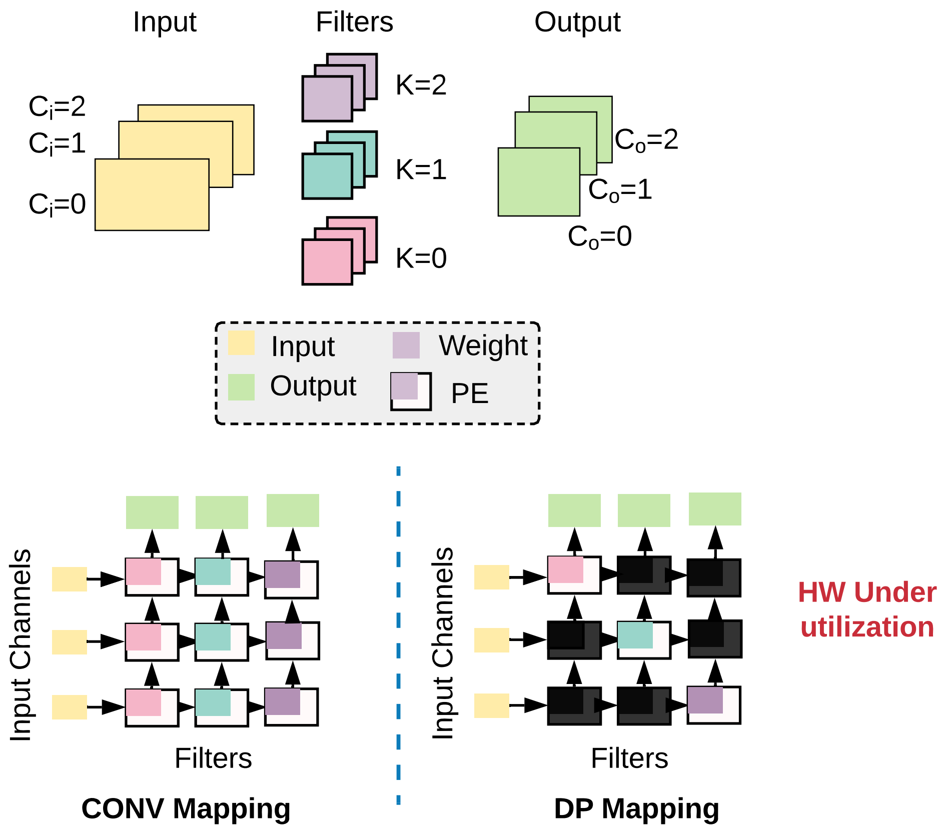


Figure 3.6: CONV vs DP layer C|K mapping strategy

7% for DP layers (also observed by [38, 62]). Even for the same layer type in a cross-layer, different stages of the pipeline require heterogeneous hardware resources.

Fig. 3.7 illustrates the mapping of hardware loops “hardware()" onto a neural network (NN) engine. A $C|K$ mapping maps every channel(C) to row and filters(K) to the columns of the NN engine. A $C|K$ mapping [1, 62, 115] is optimal for CONV/PT layers. However, performs poorly for DP layers. In the case of DP layers with one channel per filter, only the diagonal MACs will be utilized. Unlike, PT or CONV, a DP maps best with $F_y|YC$ [20], i.e., F_y filters as rows and Y,C width and channels as columns of the NN engine. This maximizes input reuse for DP layers.

Thus for optimal performance, a cross-layer pipeline needs to map different inner dataflows to a shared hardware engine. As we discuss in Section 3.2 this leads to both changes in the data storage format as it passes between the layers and requires hardware support for line buffers. Prior cross-layer approaches for CONV [10, 36] fix the same dataflow for outer and inner; they did not investigate LR-CONVs.

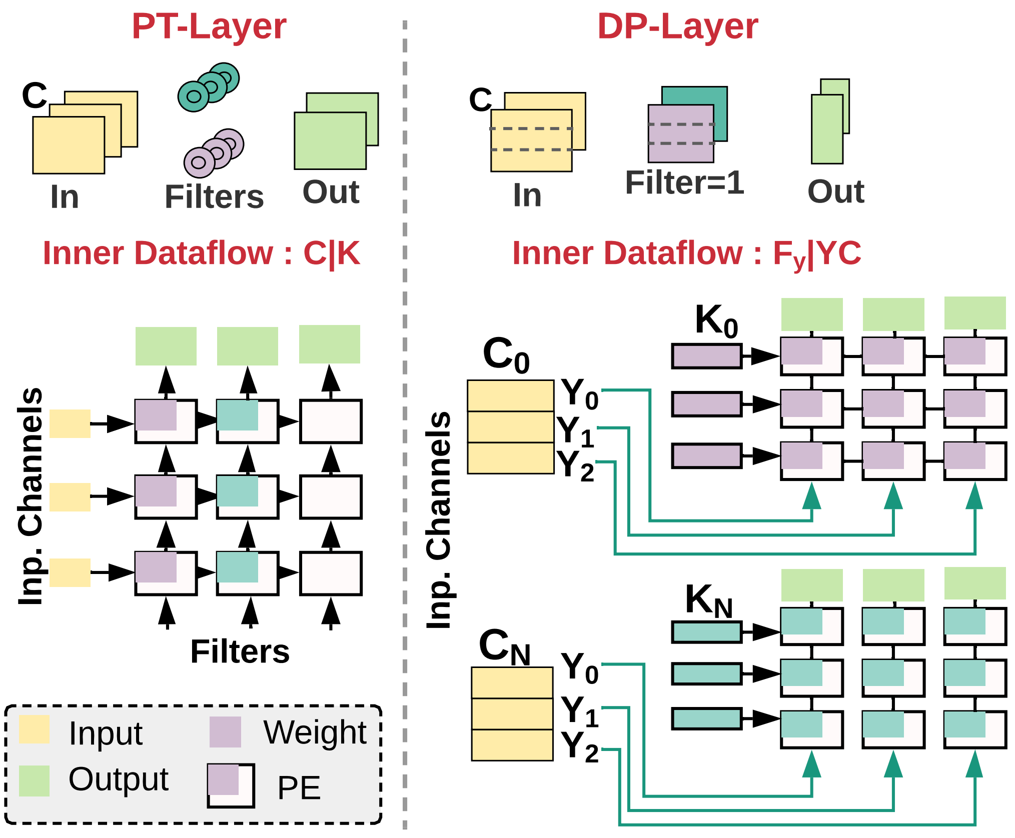


Figure 3.7: Mapping LR-CONVs to hardware. Left shows $C|K$ dataflow for PT layer (F_x=F_y=1). Right shows $F_y|Y$ with channel replication for DP layers.

3.1.6 Heterogeneous Resource Allocation

In Figure 3.3, we plot the mac and memory requirements of low-rank convolution layers from Mobilenet_v2. Xception employs a sequence of depth separable layers (DS on the x-axis) and Mobilenet_v2 implements a sequence of inverted residual layers (INV on the x-axis). The layers are listed in sequence based on where they appear in the network, early, middle and late.

In MobileNet_v2 inverted residual layer (PT $\rightarrow$ DP $\rightarrow$ PT), the first layer (PT) is expansion layer and is used to increase feature dimensions. It always has higher filters and mac operations than the third layer (PT) which is used as a linear bottleneck. In inverted residual networks, the number of channel dimensions across layers is *narrow*$\rightarrow$ *wide*$\rightarrow$ *narrow*. Notice, “Inv2", within an Inverted residual layer, the same layer type PT can vary significantly in MAC and memory between the first PT and the second PT.

In Xception, the depth separable layers have a large number of filters in the late layer and require a large output activation buffer size. The partial product of PT output activation buffer grows along the filter dimension. However, the DP input activation (IFM) buffer size can be small. CONV and LR-CONV vary in their pipeline depth. For instance, due to accumulation and addition of partial products, adjacent CONV layers can only be pipelined to two stages whereas an inverted residual layer (PT$\rightarrow$ DP$\rightarrow$PT) can be pipelined to three stages.

A high-performance cross-layer dataflow will require different resource allocation for inner dataflows of each layer based on the DNN needs. The resources imply different available mac units and inner dataflow tile sizes for different layers.

3.2 Inverted Residual Layer: A Cross-Layer Example

In this section we first illustrate the execution model of a three-layer fine-grain cross-layer dataflow §3.2.1. In §3.2.2, we explain all the data transformations, the key technique necessary to achieve a fine-grain cross-layer dataflow.

3.2.1 Pipeline Overview

Fig. 3.9 illustrates an inverted residual (INV) layer, which is the workhorse of many state-of-the-art DNNs (e.g., EfficientNet, Mnasnet1_0, and Mobilenet_v2). INV blocks are 3-layer pipelines consisting of $PT \rightarrow DP \rightarrow PT$. The dataflow we explore is labelled as $X^pY^pK^pC{-}3$ in our terminology (which we elaborate on in the next section). The X, Y, K, C orders refer to the loop orders of the outer dataflow in the first stage of the pipeline. '=3" indicates the depth of the pipeline and "^{p}" indicate the loops that are executed partially in each stage of the pipeline prior to synchronizing with the next stage. Only the first stage includes an outer dataflow and controls the feed rate through the pipeline. However, every stage can independently set its own inner dataflow. **Stage 1:** The PT layer executes X,Y,K dimensions partially whereas all channels are executed completely. With $X^pY^pK^pC$, stage 1 synchronizes once channel loop are complete. The block of data read from the DRAM to SRAM is controlled using hyper-parameters X_b,Y_b,K_b and C_b. **Stage 2:** The stage 2 shown here implements a fine-grain $X^pY^pC^p$ inner dataflow. There are minimal dependencies to the adjacent stage and DPs can synchronize and stream a tile of $[X_t,Y_t,C_t]$ as soon as it is generated (unlike the PT). It independently parameterize its inner dataflow with hyper-parameters X_t,Y_t,C_t. The hardware mapping can also be different, with $Fy|YC$ vs $C|K$ for stage 1. (see § 3.1 for why this works best for DP). **Stage 3:** The third and final PT layer implements a X^pY^pCK inner dataflow, i.e., all filters and channels are completed before the partial output activations are stored off-chip for the next block. Since all dimensions of C,K are executed their order does not matter. Once they are complete, the layer streams the sliding window back to the DRAM.

3.2.2 Data transformations in pipeline

Fig. 3.10 shows the transformations on the activations as it passes between the layers. The input activation is a 3D array $[C][X][Y]$ stored in DRAM as flattened array in CXY format, where C is the outer most dimension and Y is the inner-most dimension. **Stage 1:** The outer dataflow in PT loads the data on-chip in groups of $[C_b][X_b][Y_b]$. The Transposer ❶ transposes the streaming activations and stores it as tiles of $[X_t][C_t][Y_t]$. The inner dataflow spatially unrolls the channels ❷ and each

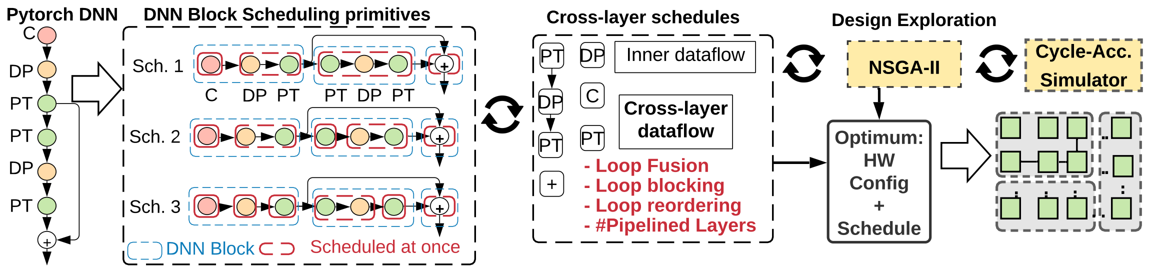
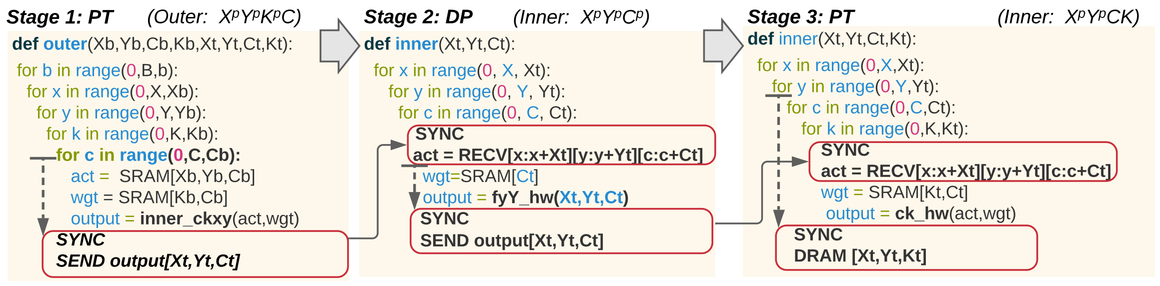


Figure 3.8: X-Layer tool flow. C= CONV layers


Stage 1: PT (Outer: XpYpKpC)
Stage 2: DP (Inner: XpYpCp)
Stage 3: PT (Inner: XpYpCK)

```
def outer(Xb,Yb,Cb,Kb,Xt,Yt,Ct,Kt):
  for b in range(0,B,b):
    for x in range(0,X,Xb):
      for y in range(0,Y,Yb):
        for k in range(0,K,Kb):
          for c in range(0,C,Cb):
            act = SRAM[Xb,Yb,Cb]
            wgt = SRAM[Kb,Cb]
            output = inner_ckxy(act,wgt)
          SYNC
          SEND output[Xt,Yt,Ct]
```

```
def inner(Xt,Yt,Ct):
  for x in range(0, X, Xt):
    for y in range(0, Y, Yt):
      for c in range(0, C, Ct):
        SYNC
        act = RECV[x:x+Xt][y:y+Yt][c:c+Ct]
        wgt=SRAM[Ct]
        output = fyY_hw(Xt,Yt,Ct)
        SYNC
        SEND output[Xt,Yt,Ct]
```

```
def inner(Xt,Yt,Ct,Kt):
  for x in range(0,X,Xt):
    for y in range(0,Y,Yt):
      for c in range(0,C,Ct):
        for k in range(0,K,Kt):
          SYNC
          act = RECV[x:x+Xt][y:y+Yt][c:c+Ct]
          wgt = SRAM[Kt,Ct]
          output = ck_hw(act,wgt)
      SYNC
      DRAM [Xt,Yt,Kt]
```

Figure 3.9: A cross-layer dataflow example for a an Inverted Residual Layer. The cross-layer dataflow is called $X^P Y^P K^P C{-}3$.

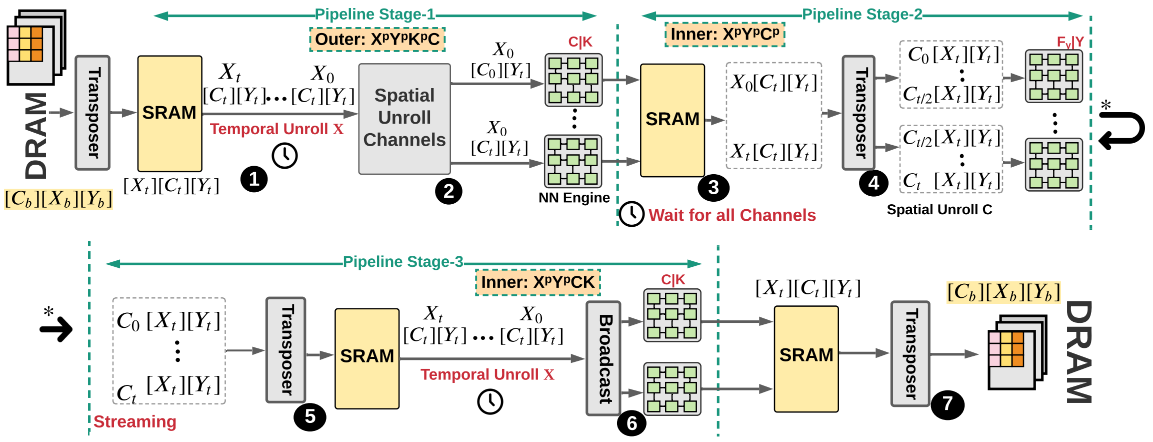


Figure 3.10: The Execution Model of a cross-layer Inverse Residual Layer. Please read in color.

channel is fed to the rows of the $C|K$ mapped NN engine. The X_t is unrolled in time. The output of stage 1 is stored in on-chip SRAM ❸ and Stage 2 cannot start until all channels are executed in Stage 1. **Stage 2:** The Stage 2 streams the activation data from on-chip SRAM and transposes ❹ the data on the fly from $[X_t][C_t][Y_t]$ to $[C_t][X_t][Y_t]$. This transformation is necessary to map onto hardware engines with $F_y|YC$ mapping (see § 3.1 for why this works best for DP). The C_t is spatially unrolled and Y_t is fed to the columns of the NN engines. **Stage 3:** The transposer ❺ then transforms the data back to $[C_t][X_t][Y_t]$ for $C|K$ mapping of the stage 3 PT layer. Unlike the first stage of the pipeline, the same input activation tile is broadcast ❻ to all the NN engines. Finally, the partial output activation in $[X_t][C_t][Y_t]$ format is transposed ❼ to $[C_t][X_t][Y_t]$ format and grouped together to write data to DRAM.

3.3 X-Layer: Design space and Taxonomy

In this section, we elaborate on the design space of cross-layer. In the next section, we elaborate on the microarchitecture and DNN accelerator. Existing compilers for DNNs (e.g., Tensorflow and TVM) fuse operations such as BatchNorm folding and activation functions with convolutions (we do not illustrate such layers in our figure further). We organize this section as follows: i) We discuss how X-Layer schedules various DNN layers onto the underlying hardware. ii) §3.3.1 creates a taxonomy of cross-layer dataflows.

DNN block pipeline: X-Layer partitions a DNN graph into several composite blocks. A block is a pipeline of DNN layers without any conditional branches (Fig. 3.8). Each block is then scheduled using scheduling primitives we create for cross-layer. Layers within a block are pipeline parallelized onto the underlying hardware. The end-to-end DNN is topologically sorted and blocks are serially executed on the underlying hardware. Within each block, the schedule can be chosen between the cross-layer dataflow implementations as shown in Figure 3.8. The number of layers per block can be larger than the pipeline depth feasible in the hardware, in such cases, the scheduler needs to implicitly split up blocks. For instance, in the case of a maximum pipeline depth of two, a PT-DP-PT block can be scheduled as $PT \rightarrow DP - PT$ or $PT - DP \rightarrow PT$. In a nutshell, a block needs to: i) capture enough producer-consumer locality within the pipelined layers to improve performance, ii) capture enough pipeline parallelism for improving end-to-end latency, and iii) include layers without enough macs to keep the hardware utilized.

Hyperparameters: Table 3.1 shows the hyper-parameters for a DNN. It includes the buffer parameters ("$_b$") for the outer-dataflow which control the amount of data movement between the layers and from the layers to the DRAM. Each layer of the pipeline has tiling parameters ("$_t$") which controls the tile size used to execute the inner dataflow on the underlying hardware. It also includes the number of NN engines dedicated to each layer for the computation of each pipeline stage.

The cross-layer DNN block schedule is statically selected during the compile-and-mapping time. The X-Layer exploration tool then profiles the hyper-parameters to select the most optimal hyper-parameters for a design point. The selected hyper-parameters are used to allocate resources to each stage of the pipeline. At runtime, the allocated resource impact the latency of each stage of the pipeline and correlates with the end-to-end pipeline throughput.

X-LAYER Exploration: The main challenge for a cross-layer dataflow is the large search space. Prior work [10, 36] fixed the dataflow and the underlying hardware. As elaborated in §3.1.5 the arithmetic intensity and reuse can vary between two convolution layers in the same block (e.g., different PTs in a DNN block). To explore the large design space of hyper-parameters we first develop a cycle-accurate simulator (details in Section 4.2) to provide an estimation of end-to-end latency and on-chip SRAM requirements. We then rely on a genetic algorithm NSGA-II [75] to find

Table 3.1: Hyper-parameters for Design space exploration.

Args	**Description/ Options**	
Net	DNN Blocks	
Pipeline Depth	1,2,3,3+	
$Hardware\_dict=(Pipeline\_depth)\times Hyperparameters/layer$		
Outer dataflow	X_b,Y_b,C_b,K_b for First layer	
Inner Dataflow	X_t,Y_t,C_t,K_t for all layers	
# mac HW	Number of Mac units for all layers	
Dataflow/layer		
Stage 1:Outer Dataflow	PT/CONV	Fine-grain: $X^pY^pK^pC$,X^pY^pKC, Coarse-grain: K^pCXY,$CKXY$
	DP	$X^pY^pC^p$,$C^pX^pY^p$
Other Stages Inner Dataflow	PT/CONV	Fine-grain: $XYKC$,$XYCK$, Coarse-grain: $KCXY$,$CKXY$
	DP	XYC,CXY
Systolic_engine	$C\|K$, $F_y\|Y$ with channel replication	

Table 3.2: Design Parameter Exploration using NSGA-II.

Objective Function		$\min(Latency)$ $\min(Memory)$
Constraint Function		$\sum_i(mac)=16$ $\sum_i(memory)\leq 256KB$
Parameter Search Space		
First Layer: Outer Dataflow	X_b,Y_b	7 — 224
	Cb,Kb	3 — 900
For each Layer: Inner Dataflow	X_t,Y_t	7 — 224
	K_t	1 — 900
	C_t	3 — 900
	# 7×7 engines	1 — 14

optimal design points. NSGA-II is a multi-objective optimization framework based on evolutionary algorithms. In NSGA-II, we need to define i) objective function, ii) constraints, iii) parameter search space. In our case, the objective function is the minimization of latency and on-chip memory. The constraint function is the total amount of mac units (which is equal to 16, 7×7 NN Engines) and the total on-chip memory. The parameter search space (Table 3.2) provides the limits of each hyper-parameter to explore the design space of cross-layer dataflows.

3.3.1 Taxonomy and search space of cross-layer

In this section, we systematically categorize the universe of cross-layer dataflows. Table3.3 summarizes the qualitative differences between the different cross-layer dataflows. For a CONV pipeline, only X^pY^pKC can pipeline beyond two layers (2+). Any other loop order restricts the pipeline depth to a maximum depth of two. For a pipeline depth of two, the loop order restricts the number of layers which can be partially executed in each stage. This restriction is due to the CONV algorithm. Any loop dimension which comes after a channel (C) need to be executed fully. This implies, the possible loop orders for the first layer is $X^pY^pK^pC$, X^pY^pKC, K^pCXY, and $CKXY$. Most activations' and filters' height and width are square in shape. Thus, the order of X, Y does not matter. The F_x,F_y can be ignored as it does not impact latency.

$CKXY$ runs each loop dimension in a layer to completion and is similar to a single layer that retains all activations on-chip (requires a large amount of SRAM). Both $CKXY$ and K^pCXY are coarse-grain dataflows requiring a large amount of SRAM, since the large C, X and Y are complete dimensions. X^pY^pKC and X^pY^pCK are medium grain dataflows and their performance depends on the ^{p} factor since K and C loop dimensions run to completion like in a single layer. Prior work [10] (Listing 3) chose sub-optimal p=1 and gave up X-Y parallelism and locality leading to moderate performance.

Given four-loop dimensions X,Y,K,C, there can be $(4!)^{depth}$ possible cross-layer pipelined dataflows. However, many of these dataflows overlap in characteristics and we constrain the universe based on the following observations: i) **Order of XY does not matter:** Most DNNs have square activations and kernels. ii) **Loop order does not matter, for complete dimensions within a layer:** The order of complete dimensions does not impact individual layers. For instance, X^pY^pKC and X^pY^pCK will have a similar latency, since in both cases stage throughput is the same.

Stage 1 outer dataflow determines sliding window: In a cross-layer dataflow, the loop order in stage 1 of the pipeline controls the slidinw window processed on-chip, and has first order effect on latency and on-chip SRAM. Hence, we group all possible dataflows with the same stage 1 dataflow together. For example, in Fig. 3.9, the cross-layer dataflow used was $X^pY^pK^pC \rightarrow X^pY^pC^p \rightarrow X^pY^pCK$. There are many possible loop orders for second and third layers. We categorize all those dataflows into $X^pY^pK^pC$ dataflow.

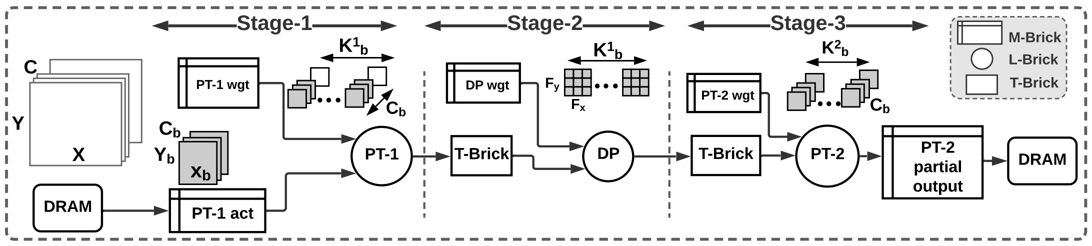
Table 3.3: Design space of cross-layer dataflows.

	Layer depth				
	CONV	**LR-CONV**	**Granularity**	**On-chip SRAM**	**Latency**
$X^pY^pK^pC$	2	2, 3	Fine	Low	Low
X^pY^pKC	2+	2+	Medium	Depends ^{p}	Depends ^{p}
K^pCXY	2	2, 3	Coarse	High	Low
$CKXY$	2	2, 3	Coarse	High	High
$X^pY^pC^p$ (DP-only)	-	2	Fine	Low	Low
$C^pX^pY^p$ (DP-only)	-	2	Fine	Low	Low

3.3.2 Memory Requirement for Cross-Layer dataflows

To understand the granularity of the cross-layer dataflows in Table 3.3, we qualitatively discuss the on-chip memory requirements of four major cross-layer dataflows in Figure 3.11). The K^pCXY-3 uses same outer dataflow as Tangram [36] and X^pY^pKC-3 represents Fused-Layer [10]. The Figure illustrates the memory requirements for pipelining Inverted residual layers. The inverted residual layer is the workhorse layer in Mobilenet_v2 and is a composite layer consisting of three stages of convolution PT→DP→PT. Changes to the loop order and blocking in the outer layer dataflow influences the following: a) the size and shape of on-chip buffers required for holding the activation and weights for all layers in the pipeline (Layer Buffering in Figure 3.11) b) the size of the partial layers to be retained on-chip for subsequent layers. The loop blocking parameters affect the size of activations that need to be convolved before the next layer can start. Thus loop blocking influences the delay a subsequent stage has to wait before kickstarting; this directly impacts the overall throughput. The "Pipeline timing column" column shows the size of activations that need to be computed in each stage of the pipeline which associates with the wait time in each stage of the pipeline. Each row in the table (Figure 3.11) corresponds to the outer-dataflow selected for the first stage in the pipeline, stage1-PT. For a batch size of one, the activations have three dimensions (3D) - X, Y, and C, whereas the filters have four dimensions (4D) which are flattened into 2D by merging $F_x \times F_y \times C$. The color shades is a qualitative indication of the hardware overhead, either on-chip buffering or pipeline initiation latency. Green being most optimized i.e., hardware required is independent of loop dimensions.

Observations: It can be noticed that $CKXY-3$ requires the largest amount of on-chip memory, since, 2 out of 3 activation layers need to be completely stored on-chip (DP,$PT2$). The reason being that when channels (C) are the outermost loops, the CONV/PT algorithm requires entire channel completion before kickstarting the next layer. Similarly, K^pCXY-3 (Tangram) is also a coarse-grain pipeline and requires high on-chip memory for storing activations on-chip. X^pY^pKC-3 (Fused-Layer) is a fine-grain pipeline, but due to the trade-off of being able to pipeline any number of layers, it has to store all filters on-chip for every stage. Thus, it ends up requiring high on-chip memory for filters. X-Layer dataflow, $X^pY^pK^pC$ makes the trade-off of limiting pipeline depth to three layers. Due to this trade-off, it is a finer-grain pipeline compared to other dataflows and requires the least amount of on-chip memory.



On-chip Memory | Pipeline Timing

Outer Dataflow	Tensor	Layer Buffering				Stage 1 Work	Stage 2 Work	Stage 3 Work
		PT-1	DP	PT-2	PT-2 Partial Output			
$X^pY^pK^pC$-3	act	🟩 $Y_b * X_b * C_b$	🟩 $Y_b * X_b * K^1_b$	🟩 $Y_b * X_b * K^1_b$	🟨 $Y_b * X_b * K^2$	🟨 $Y_b * X_b * C$	🟩 $Y_b * X_b * K^1_b$	🟩 $Y_b * X_b * K^1_b$
	wgt	🟩 $K^1_b * C_b$	🟩 $K^1_b * F_x * F_y$	🟩 $K^2_b * K^1_b$	-	🟨 $K^1_b * C$	🟩 $K^1_b * F_x * F_y$	🟨 $K^2 * K^1_b$
X^pY^pKC-3	act	🟩 $Y_b * X_b * C_b$	🟨 $Y_b * X_b * K^1$	🟨 $Y_b * X_b * K^1$	🟨 $Y_b * X_b * K^2$	🟨 $Y_b * X_b * C$	🟨 $Y_b * X_b * K^1$	🟨 $Y_b * X_b * K^1$
	wgt	🟨 $K^1 * C_b$	🟨 $K^1 * F_x * F_y$	🟨 $K^2_b * K^1$	-	🟧 $K^1 * C$	🟨 $K^1 * F_x * F_y$	🟧 $K^2 * K^1$
K^pCXY-3	act	🟧 $Y * X * C_b$	🟧 $Y * X * K$	🟧 $Y * X * K^1$	🟥 $Y * X * K^2$	🟥 $Y * X * C$	🟧 $Y * X * K^1$	🟧 $Y * X * K^1$
	wgt	🟩 $K^1_b * C_b$	🟩 $K^1_b * F_x * F_y$	🟩 $K^2_b * K^1_b$	-	🟨 $K^1_b * C$	🟩 $K^1_b * F_x * F_y$	🟨 $K^2 * K^1_b$
CKXY-3	act	🟧 $Y_b * X_b * C_b$	🟥 $Y * X * K$	🟥 $Y * X * K^1$	🟥 $Y * X * K^2$	🟥 $Y * X * C$	🟥 $Y * X * K^1$	🟥 $Y * X * K^1$
	wgt	🟨 $K^1 * C_b$	🟨 $K^1 * F_x * F_y$	🟨 $K^2_b * K^1$	-	🟧 $K^1 * C$	🟨 $K^1 * F_x * F_y$	🟧 $K^2 * K^1$

🟩 Best - partial 🟨 Good - 1D 🟧 Bad - 2D 🟥 Worst - 3D

Figure 3.11: On chip memory requirement for various Cross-Layer dataflows, for Inverse Residual Layers. Layer Buffering indicates on-chip memory required in each pipeline stage. Pipeline Timing indicate the size of activations required to be computed in each pipeline stage. -1D, -2D, -3D implies the number of complete loop dimensions.

Chapter 4

X-Layer Microarchitecture and Dataflow Exploration

In this chapter we first discuss the cross-layer microarchitecture in §4.1 followed by a comprehensive evaluation of the cross-layer dataflows in §4.2.

4.1 X-Layer Architecture

X-Layer's microarchitecture is a grid-based domain-specific accelerator targeting DNNs. The accelerator is composed of L-Bricks, T-Bricks, and M-Bricks arranged in a configurable grid.

Fig. 4.1 provides an overview of X-Layer microarchitecture. The hardware is a 4$\times$4 grid of L-Bricks connected via a two-dimensional mesh. The T-Bricks are also organized as a separate mesh and super-imposed on the L-Brick mesh to handle inter-layer communication. The periphery of the L-Bricks is connected to M-Bricks which store the activation, weight and partial product tiles. Each **L-Brick** operates as a 7$\times$7 systolic weight stationary NN engine (7: largest filter size).

The hardware microarchitecture is capable of supporting both $C|K$ (for PT and CONV) and $F_y|YC$ (for DP) like Eyeriss-v2 [20]. The M-Brick provides a programmable DMA. They are configured based on the blocking parameters of the outer dataflow. The M-Bricks also control the incoming DRAM burst (512 bit i.e., 64 8-bit activations) across multiple SRAM tiles which are collectively supplied to the tensor required by L-Bricks. For example, for a CONV layer, an M-Brick stores $Y_b{*}C_b$ in each SRAM bank and every SRAM bank stores X dimension.

Fig. 4.1 shows the mapping of an inverted residual layer, PT1(in blue), DP(in green) and PT2(in red) are mapped to the hardware. Filters move top to bottom. Activations move either from left to right or top to bottom. Higher X^p and Y^p improves filter reuse and minimizes block loading overhead. For example, a PT layer with $X^pY^pK^pC$ dataflow and $C|K$ hardware mapping will hold filter stationary in L-Bricks for all $X^p{\times}Y^p$.

T-Bricks are the key architectural motif required to support cross-layer dataflows. They are used for three purposes: i) to forward the data to the next L-Brick through lightweight SRAM queues, 2) to transform the incoming data into a format expected by the dataflow of the consuming stage (❹,

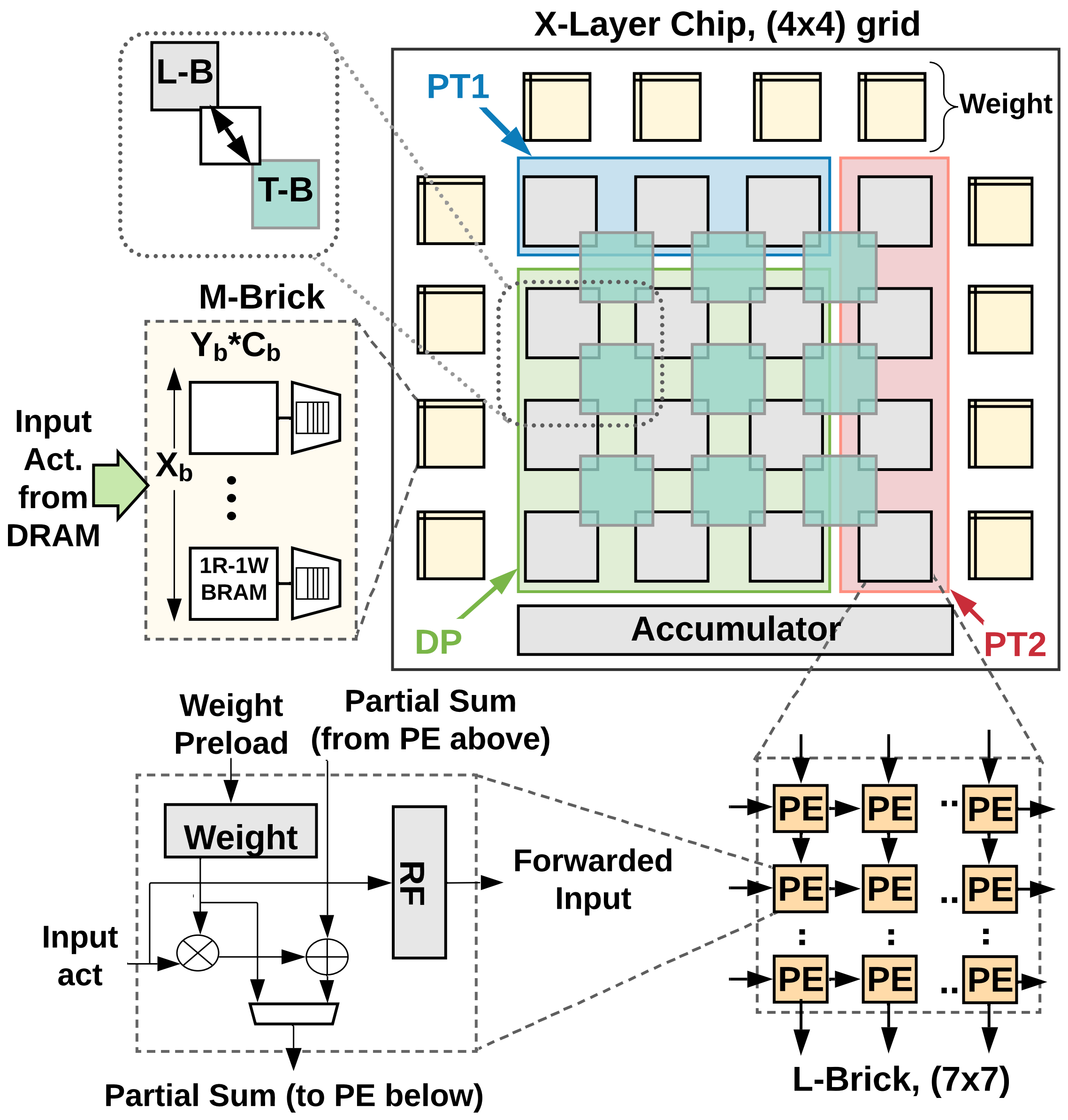


Figure 4.1: The grid-style architecture of X-Layer.

5 in Fig. 3.10), and 3) to apply partial adds (7 in Fig. 3.10) and activation functions such as ReLU operations prior to serialization to the DRAM. The pipeline requires the following transformations i) moving from PT to DP (e.g., inv. residual layers) the hardware has to convert the activation tensor from $XCY \rightarrow CXY$ tensor format. ii) when moving from DP to PT (e.g., depth separable) we perform the reverse conversion from $CXY \rightarrow XCY$.

The T-Bricks between layers are composed of a series of line buffers to transform as well as buffer the multi-dimensional tensor data as it passes through layers. The core hardware in T-Brick is the multiple-input multiple-output queue (MIMO Queue). The MIMO queue is composed of a set of parameterized single-ported line buffers that work in concert to change data format between

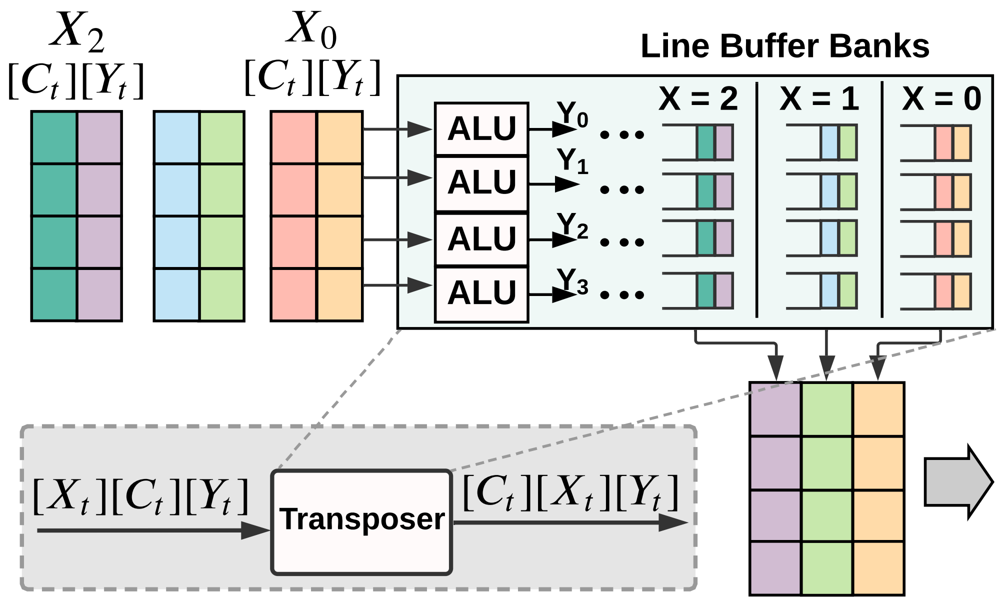


Figure 4.2: Overview of T-Brick (trasposer)

L-Bricks. As shown in Fig. 4.2, T-Brick spatially merges data available at the head of multiple streams into a single vector. Here it performs the reverse conversion from XCY to CXY dataflow storage by combining the head elements of the line buffers. T-Brick is also equipped with ALUs to perform the activation functions such as ReLU before sending the data to the next L-Brick layer. T-Brick implicitly infers the physical wire widths and connections from the tensor type of the stage layers.

4.2 Quantitative Insights and Evaluation

In this section, we first explain the Experimental set-up and then run comprehensive experiments on cross-layer dataflow exploration step by step. Table 4.1 organizes the results and shows the questions we answer with each experiment. Figure 4.8 summarizes the decisions prior state of the art cross-layer dataflows and X-Layer make for Dataflow, partial loop orders, Pipeline depth, Hardware mapping and size of NN engines. We explain these trade-offs in detail in §4.3.

4.2.1 Experimental Setup

For end-to-end DNN evaluation, we implement the X-layer architecture on Amazon F1 FPGAs. The FPGA results enabled us to validate functional simulation of our cycle-accurate model used by X-Layer to explore design space. The bandwidth for data transmission from off-chip to on-chip memory was 16 Mbps. We noticed that the performance of X-Layer found dataflows improves further as the bandwidth reduces. **X-Layer** convolution accelerator is a 4×4 grid of sixteen convolution engines. Each convolution engine (NN engine) organizes multiply-and-accumulate in a 7×7 grid, with 64 bytes of local register file. This mimics an engine which is $\simeq\frac{1}{5}$ size of an Eyeriss engine [20]. We investigate chips with on-chip SRAM between 32KB - 2MB.

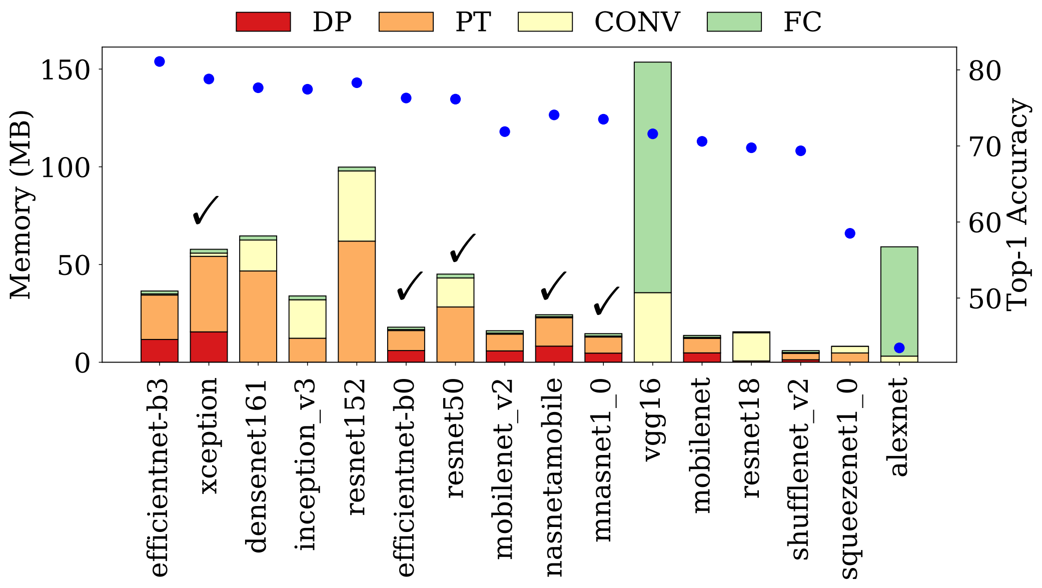


Figure 4.3: Total Memory used for LR-CONVs vs CONVs and FC. Benchmarks are sorted in order of Top-1 accuracy shown as points on Second Y-axis. ✓: Suite we study.

We selected six DNNs that make extensive use of LR-CONVs and are leaders in accuracy(Fig. 4.3). Three are based on inverted residual layers, namely Efficientnet-b0 (56.6% pipelined layers), Mobilenet_v2 (96.2%) and Mnasnet1_0 (94.3%). Two are based on depth-separable layers Mobilenet (93%) and Xception (90.7%). One benchmark is CONV based, Resnet-50 (59%). The percentages indicate the number of layers which execute a cross-layer dataflow out of all layers in a DNN. We schedule three-layer dataflow for Inverted residual layer based DNNs and two-layer dataflow for depth separable and CONV based DNNs. Table 4.1 organizes the results.

Table 4.1: Evaluation Organization

	§ 4.2.2. Fig 4.4	§ 4.2.3.Fig. 4.5	§ 4.2.4.Fig. 4.6
Pipeline	2-stage DP→PT	3-stage PT→DP→PT	1- vs 2- vs 3-stage
Dataflow	Outer: $C^pX^pY^p$-2, $X^pY^pC^p$-2 Inner: $XYCK$,$CKXY$}	Outer: $X^pY^pK^pC-3$, X^pY^pKC-3, K^pCXY-3, $CKXY-3$	Outer: $X^pY^pK^pC-3$, $X^pY^pKC-3,-2,-1$
DNN	Xception	Mobilenet_v2	Mobilenet_v2 & Resnet-50
Study	Which outer and inner is best?	Outer granularity? Partial Layers?	Best Pipeline depth?

4.2.2 Cross-layer for 2-stage depth-separable

Qualitative summary: *Outer dataflow with X-Y outer loops enable finer-grain dataflow that minimizes the temporal distance between adjacent layers. This promotes quicker reuse, improves performance and requires lower SRAM*

Qualitative summary: *Inner dataflow XYKC achieves best tradeoff between latency and on-chip memory. It improves the performance of the critical PT and CONV stages.*
Result 1: *Given large on-chip SRAM (4 MB), the fine-grain $X^pY^pC^p$-2 (outer loops X-Y) is* $1.3\times$ *faster than coarse-grain $C^pX^pY^p$-2 (outer loops C).*
Result 2: *Constrained by smaller on-chip SRAM (256KB), the fine-grain $X^pY^pC^p$-2 is* $4.3\times$ *faster than coarse-grain $C^pX^pY^p$-2.*
Result 3: *For same outer dataflow $X^pY^pC^p$-2, the inner dataflow $XYKC$ requires* $1.55\times$ *less on-chip memory compared to $CKXY$.*

Study: Fine-grain (XY) vs Coarse-grain (C/K) outer dataflow: The major difference between coarse-grain $C^pX^pY^p$-2 and a fine-grain $X^pY^pC^p$-2 is the impact of on-chip SRAM. Having channel (C) vs (XY) as the outermost loop affects the amount of inter-layer activations. The coarse-grain $C^pX^pY^p$-2 requires entire inter-layer activations to be stored on-chip whereas the fine-grain $X^pY^pC^p$-2 requires only partial activations. For edge devices with lower on-chip memory, this has first-order impact on performance of coarse-grain. As, the on-chip memory becomes lesser the performance gap between fine-grain and coarse-grain increases further. For a 256KB of on-chip memory, the fine-grain $X^pY^pC^p$-2 is $4.3\times$ faster than coarse-grain $C^pX^pY^p$-2.

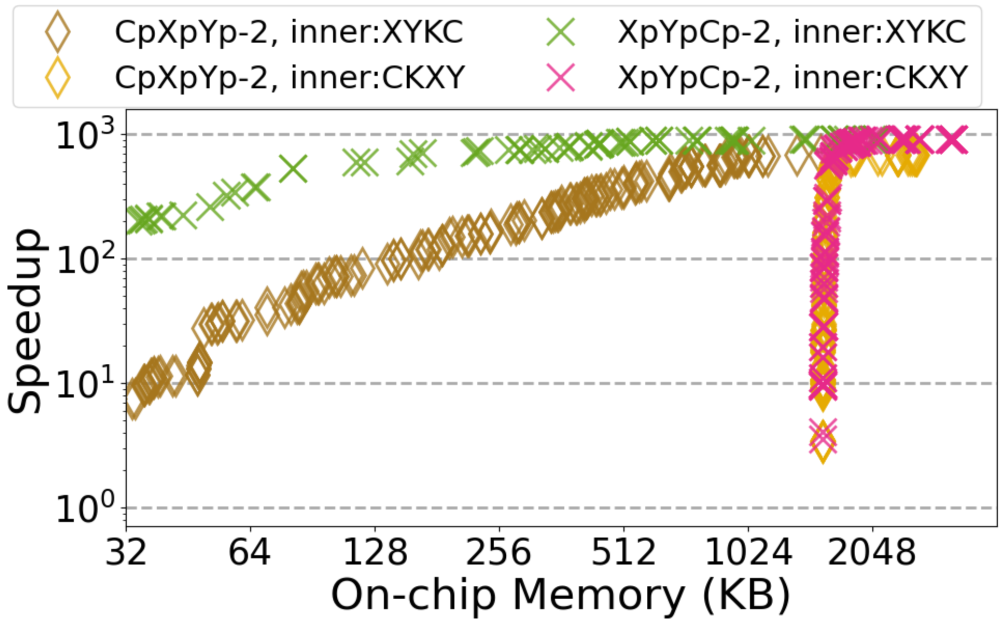


Figure 4.4: Cross-layer for Xception. 2-stage pipelines.

Study: Inner dataflow, XYKC vs CKXY? (Outer dataflow is fixed): We find that even for a fixed outer dataflow $X^pY^pC^p$-2, inner dataflow $XYKC$ requires 2040 KB of on-chip memory compared to 3150KB for $CKXY$, even though they achieve the same latency. In a depth-separable (DP$\rightarrow$PT), the PT is the critical stage. Given sufficient on-chip SRAM, multiple inner dataflows can achieve similar performance on PTs. However, due to the interaction with outer dataflows, different inner dataflows require different amounts of on-chip SRAM.

Study: Prior cross-layer dataflows: Note that prior dataflows [10, 36] cannot execute LR-CONVs as it is. We generously re-implemented their dataflows to support depth-separable layers. Tangram would implement $(C{=}1)XY$-2 outer dataflow which results in a coarse-grain dataflow. It is the worst-performing dataflow for depth-separable layers. Similarly, fused-Layer would implement medium-grain X^pY^pC-2, which executes all channels before starting stage-2 of the pipeline.

4.2.3 Cross-layer for 3-stage inverse-residual

Qualitative summary: Partial loop dimensions provide finer control over computation in each layer of the pipeline. Thus a dataflow with more partial dimensions performs well over a wider range of on-chip SRAM sizes.

Qualitative summary: When XY is selected as the outermost loop it leads to many dataflows with >=2 loop dimensions being partially executed. The following is relative ordering of dataflow with best latency and on-chip SRAM tradeoffs: $X^pY^pK^pC(best)---X^pY^pKC---K^pCXY---CKXY(worst)$.

Qualitative summary: With K/C dataflows, K^pCXY-3, and $CKXY$-3. In the first dataflow only filter dimension is partially executed whereas in the second dataflow no dimension is partially executed, hence, it leads to a coarse-grain dataflow with high on-chip memory requirements.

Result 1: *When on-chip SRAM is limited, fine-grain ($X^pY^pK^pC$-3) achieves 1000× speedup over coarse-grain (K^pCXY-3, and $CKXY$-3).*

Result 2: *While achieving similar performance, the finest-grain $X^pY^pK^pC$-3 requires* $11.7\times$ *and* $2.1\times$ *less on-chip memory compared to coarse-grain $CKXY$-3 and medium-grain X^pY^pKC-3.*

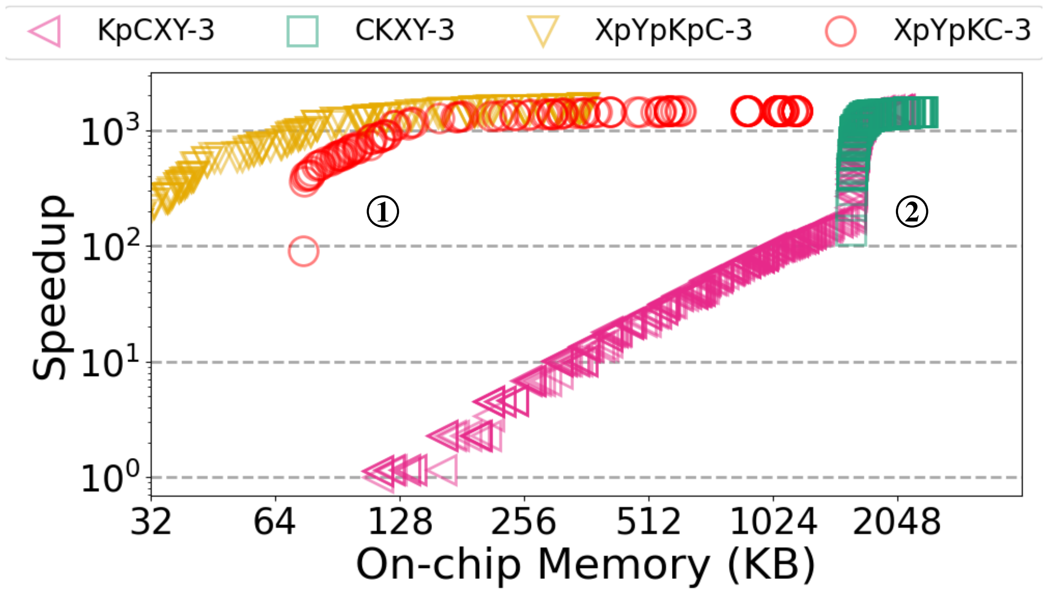
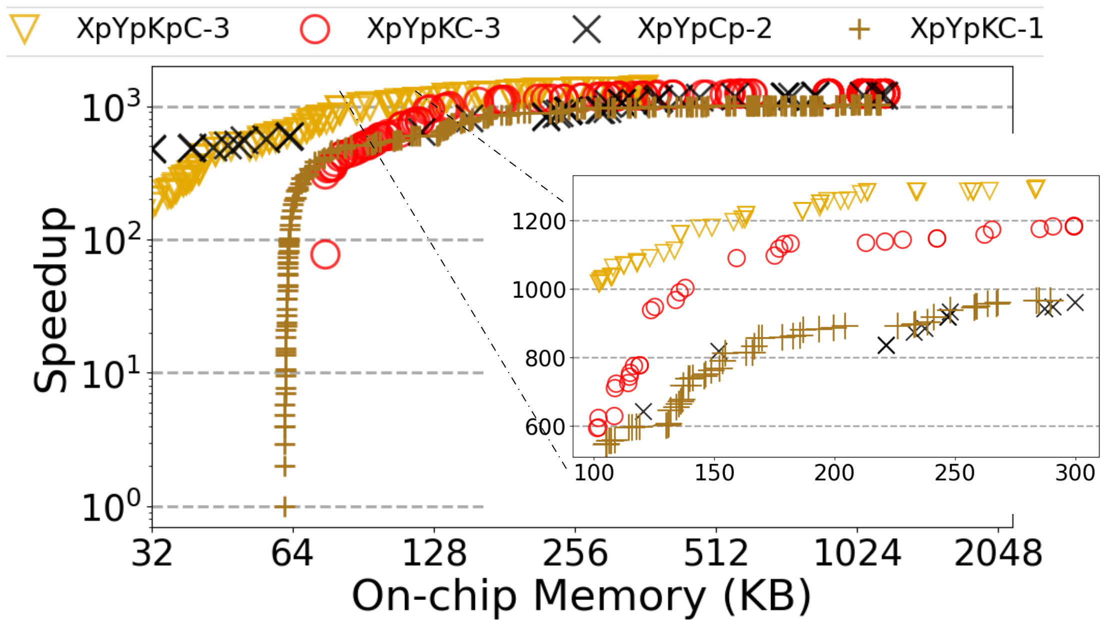


Figure 4.5: Cross-layer for Mobilenet_v2. 3-stage pipelines.



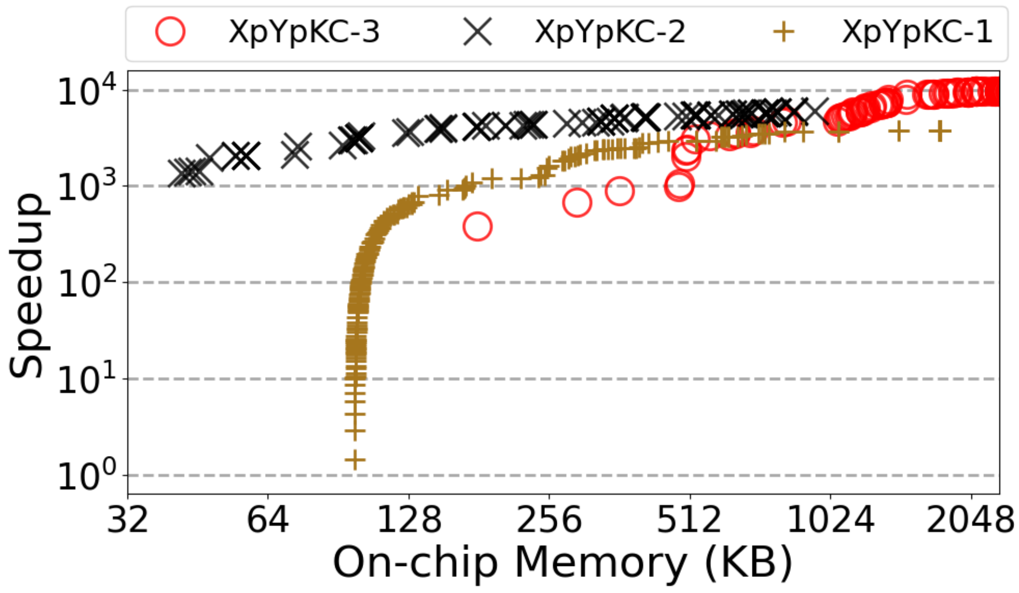


Figure 4.6: Mobilenet_v2(Top) and Resnet-50(Bottom).

Figure 4.5 studies latency vs on-chip SRAM. There are two benefits of fine-grain dataflow: ❶ They exhibit a wider performance band over different SRAM sizes (see $X^pY^pK^pC$-3, X^pY^pKC-3). Coarse-grain is highly sensitive to the amount of memory and experiences a sharp drop in performance when on-chip SRAM is limited. ❷ To achieve the same performance, fine-grain requires less on-chip SRAM. To achieve a performance similar to $CKXY$-3, the fine-grain cross-layer dataflow, $X^pY^pK^pC$-3 requires only 186 KB of on-chip memory compared to 2180 KB of memory for $CKXY$-3. The required on-chip memory is 11.7×, 5.7× and 1.2× less for $X^pY^pK^pC$-3, X^pY^pKC-3, and K^pCXY-3 compared to $CKXY$-3 dataflow. Fine-grain dataflow minimizes on-chip SRAM by

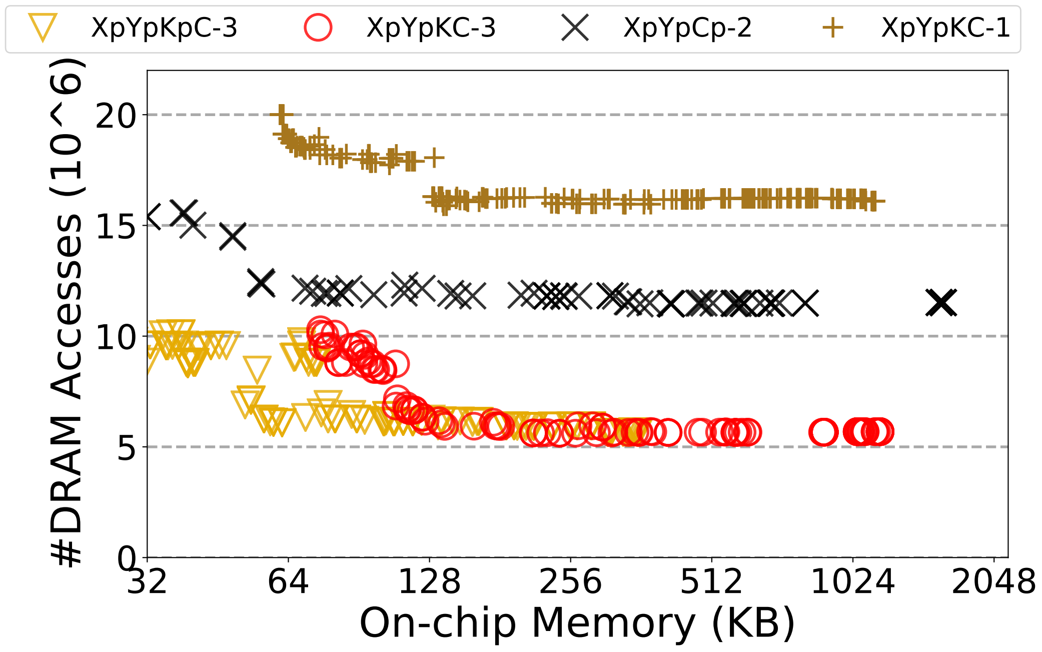


Figure 4.7: Mobilenet_v2, scatter plot comparing Million DRAM accesses (Y-axis) with On-chip Memory (X-axis) for different pipeline depths.

reducing the temporal distance between the producer and consumer layers. Given a sufficient amount of memory on-chip memory, the difference between the different outer dataflow shrinks. $X^pY^pK^pC$-3 (fine) is about 9.7% faster than $CKXY$-3 (coarse).

4.2.4 1, 2, and 3 stage pipelines: LR-CONV and CONV

Experimental Setup: Fig. 4.6 studies two different types of network, LR-CONV-based Mobilenet_v2, and a CONV-based Resnet-50. We vary the pipeline depth from 1 to 3. We study four dataflows, $X^pY^pK^pC$-3, and X^pY^pKC-1/-2/-3. For Mobilenet_v2, -3 pipelines Inverse blocks (PT→DP→PT), -2 pipelines (DP→PT). For Resnet-50, -3 pipelines $PT \rightarrow CONV \rightarrow PT$), and -2 pipelines (PT→CONV).
Qualitative summary : Deeper 3-stage pipelines perform well for LR-CONV. Shallower 2-stage pipelines perform well for CONVs. Finer-grain pipelines perform better for LR-CONVs.
Result 1: *For LR-CONVs, a 3-stage pipeline performs best. Lower on-chip memory highlight speedup better. For example, given a fixed on-chip memory of 128 KB, the three stage pipeline $X^pY^pK^pC$-3 provides a speedup of* $1.48\times$*, and* $1.58\times$ *compared to* X^pY^pKC-3 *and* X^pY^pKC-2
Result 2: *Contrary to LR-CONVs, CONVs perform best with 2-stages. For a fixed on-chip memory of 256 KB,* X^pY^pKC-2 *provides a speedup of* $6.5\times$*, and* $2.7\times$ *compared to* X^pY^pKC-3 *and* X^pY^pKC-1.

In Fig. 4.6 (top), it can be seen that for the same on-chip memory budget, a 3-layer $X^pY^pK^pC$-3 provides up to 48%, and 59% speed up compared to $X^pY^pK^pC$-2 and $X^pY^pK^pC-1$ dataflows. The speed reduces as the on-chip memory increases. For example for a 1024KB of on-chip memory, a $X^pY^pK^pC$-3 is 23.3% and 19.1% faster compared to Per-Layer Dataflow (per-layer) dataflow. In case of CONV dominated DNNs, pipelining 2+ layers leads to higher latency compared to 2−layer deep pipelines. Prior work [10, Table II] also showed that for VGG, a 5-stage pipeline was 6% slower than a single layer pipeline. We verify this for Resnet-50 in Fig. 4.6.

A CONV DNN has high MAC/memory within a layer and can benefit from data reuse within a layer. It may not necessarily benefit by going deeper in Cross-Layer Dataflow (cross-layer) pipeline. The Resnet-50 tops the speed up at 2-layer pipeline depth. For a given on-chip memory of 256KB, X^pY^pKC-2 provides a speedup of 6.5×, and 2.7× compared to X^pY^pKC-3 and X^pY^pKC-1.

#DRAM accesses: The DRAM accesses decreases as the pipeline depth increases. Figure 4.7 compares the #DRAM accesses as the on-chip memory increases across pipeline depths of 1,2 and 3. For a pipeline depth of one, the number of DRAM accesses are $\simeq 3\times$ higher compared to a three stage pipeline. This gap increases to almost $\simeq 4\times$ as the on-chip memory reduces to 64 KB. Given enough on-chip memory ($\simeq$ 256 KB), the number of #DRAM accesses does not increase. [115] also observed similar behaviour.

4.3 X-Layer vs State-of-the-Art

Experimental setup: Table 4.2 shows the set up. We compare X-Layer against state-of-the-art, Tangram [36] and Fused-layer [10]. They do not support LR-CONVs and hence we evaluate improved designs that retain their dataflows. We label them Tangram++ and Fused-layer++. **Tangram++**; The original paper implemented a 2-layer dataflow $K{=}1CXY \rightarrow CKXY$, or $K^1CXY{-}2$ (coarse-grain); see Figure 3.3) on Eyeriss engines that are known to be inefficient for DP [20]. We upgraded Tangram to Eyeriss-v2. **Fused-Layer++:** The original paper implemented X^1Y^1KC dataflow losing input activation reuse and parallelism. We implemented X^7Y^7CK (filter size = 7) which allows at least one convolution at a time in space (XY spatial locality) and minimizes on-chip SRAM overhead.

Table 4.2: X-Layer vs. State-of-the-Art. CC=$CONV{\rightarrow}CONV$, DS= $DP{\rightarrow}PT$, PDP = $PT{\rightarrow}DP{\rightarrow}PT$

	Tangram++	Fused-Layer++			X-Layer		
Layer Type	DS/CC	PDP	DS	CC	PDP	DS	CC
Pipeline Depth	2	3	2	2	3	2	2
NN Engine	16x16	$1{\times}C_t$			7x7		
# Engines	3	$K_t{\times}X_t$			16		
Total MAC	768	784			784		
On-Chip(KB)	As required	As required			256		
Remarks	Eyeriss	Vector, $X{=}Y{=}F_{max}$			-		

Qualitative Summary: *The performance improvement of X-Layer against Tangram++ and Fused-Layer++ is a consequence of systematically improving six different criteria (Fig. 4.8), namely, i) pipeline granularity, ii) partial loop orders, iii) pipeline depth, iv) inner dataflow, v) underlying hardware, vi) and mapping strategy.*

Result 1: *On an average, X-Layer is* $7.83\times$ *and* $16.58\times$ *faster than Tangram++ and Fused-Layer++.*
Result 1: *X-Layer requires* $8.3\times$ *and* $1.9\times$ *lower on-chip memory than Tangram++ and Fused-Layer++.*

Figure 4.9 plots the normalized latency (lower is better). Figure 4.10 compares the on-chip sram (lower is better).

For 3-layer pipelines in efficientnet-b0, Xception and Mobilenet_v2, X-Layer is faster than Tangram++ and Fused-layer++ by 1.4 $\times$ and $4\times$ respectively. For 2-layer pipelines it achieves $50\times$ and $130\times$ respectively.speedup compared to Tangram++ and Fused-Layer++ for depth-separable layers Xception and Mobilenet_v2. For CONV based Resnet-50, X-Layer achieves a speed up of $36\times$ and 18 $\times$ compared to Tangram++ and Fused-Layer++.

Since Tangram++ uses a coarse-grain dataflow it requires up to $8.3\times$ higher on-chip memory compared to X-Layer. It uses between 1.5MB – 2.5 MB of on-chip memory. The medium-grain dataflow based Fused-Layer++ requires up to $1.9\times$ higher on-chip memory compared to X-Layer. It requires on-chip memory between 192KB – 2.5MB.

The improvements seen above is a result of a series of several choices made in dataflow and the hardware as shown in the Fig. 4.8. We discuss the impact of each of these choices in the next section.

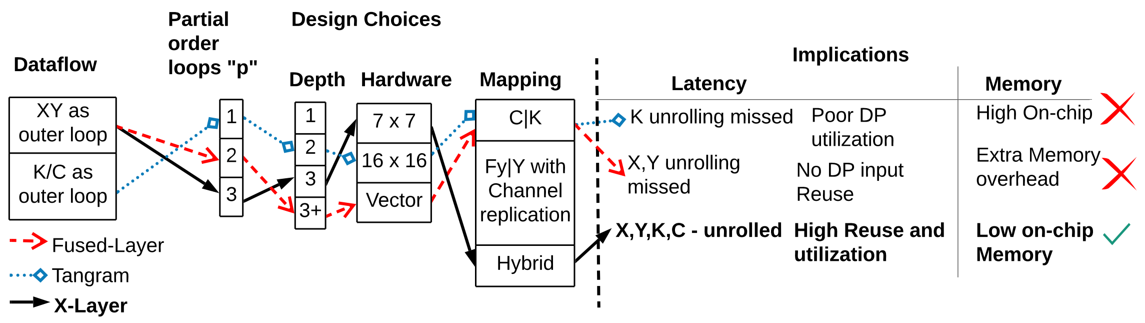


Figure 4.8: Design choices and their implications in prior cross-layer architectures.

4.3.1 Detailed explanation

Performance Breakdown vs. Fused-Layer++

Fig. 4.11 quantitatively analyzes the factors that impact performance. There are four reasons i) **XY Locality and Parallelism (+2—7$\times$):** "Uniform" implies uniform allocation of resources across pipeline stages. Fused-Layer++ (uniform) uses a medium-grain dataflow (X^7Y^7KC-3) with vector hardware (maps C,K). This sacrifices the XY spatial locality and also affects DP layer which has zero filter reuse (no K loop). X-Layer exploits higher dimensions of locality; when we use X^pY^pKC-3 where optimal p varies between 14 — 80 and exploits XY locality. Note that gains depend on % of DP layers. Mobilenet_v2 has 96%+ DPs and speedup is $7\times$. ii) **Non-unfiorm allocation:** When non-uniform resources are allocated per pipeline stage, X^pY^pKC-3 (non-uniform) in the plot, further

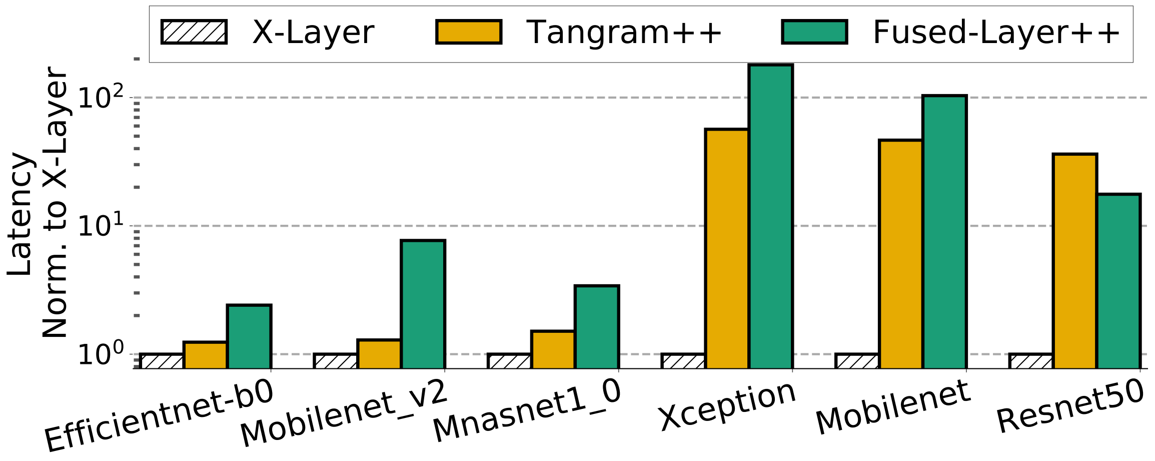


Figure 4.9: End-to-End Latency. X-Layer vs. Tangram [36] and Fused-layer [10]. X-Layer=1. **Lower is better**

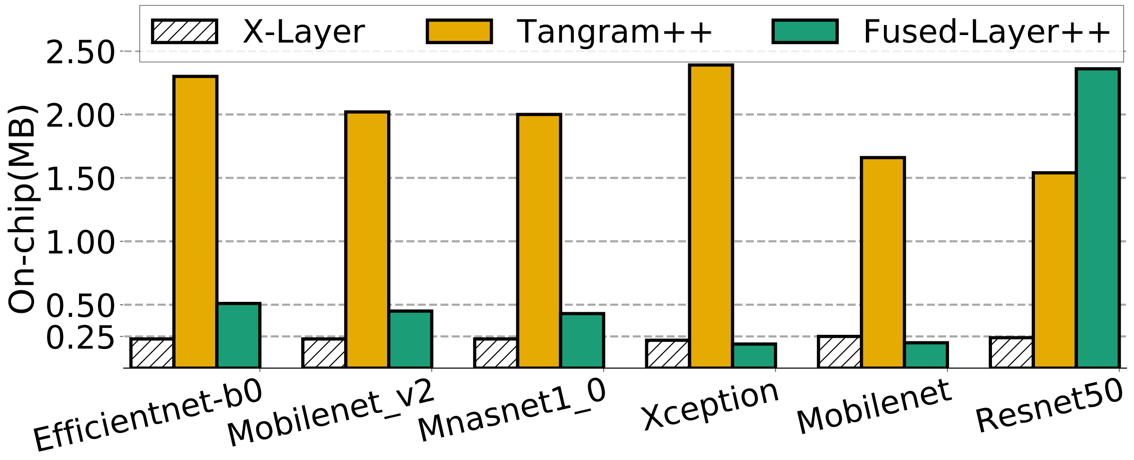


Figure 4.10: On-chip Memory used. X-Layer=1. **Lower is better**

speedup of $\simeq$5% is observed. **Fine-grain dataflow:** Finally, when the fine-grain 3-layer dataflow is used further speedup of 10% is achieved.

Performance Breakdown vs Tangram++

i) **Optimal channel blocking** ($C^pX^pY^p$**-2 +13.5**$\times$) **:** Tangram++ uses 2-layer dataflow of $K{=}1CXY \rightarrow CKXY$ which is sub-optimal for DP layers. $+C^pX^pY^p$-2 introduces partial channel blocking for DPs. ii) **3-stage pipeline** ($+K^pCXY$**-3 1.5**$\times$) **:** Tangram can only pipeline depth of two [36, page 6,Section 3.2]. X-Layer increases the pipeline depth to three layers.iii) **Finer-grain macs (+1.4**$\times$**) :** X-Layer schedules the dataflow onto finer-grain 7$\times$7 NN engine; Tangram relied on 16$\times$16 engines. A finer-grain NN engine enables more optimal allocation of macs to individual layers; a necessity when layers have such disparate requirements (see §3.1.5). Finally, the finest-grain $X^pY^pK^pC$-3 dataflow further improves speedup by 4%.

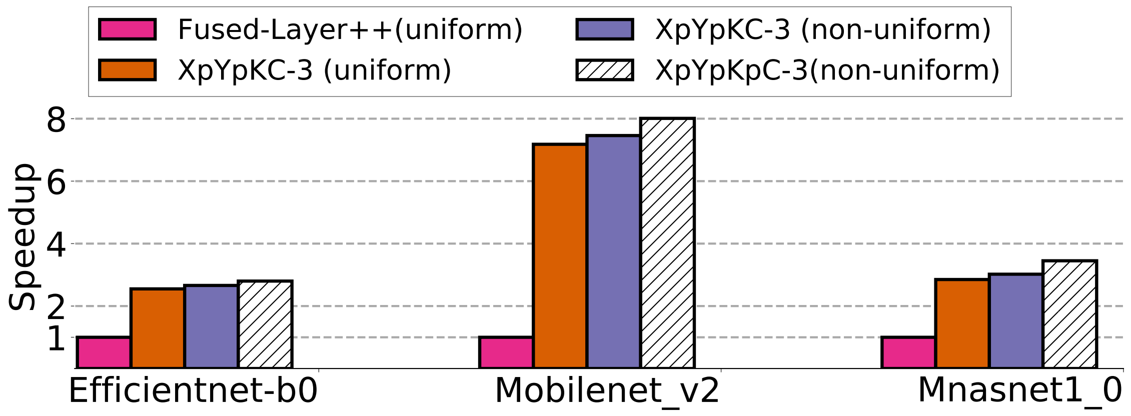


Figure 4.11: Speed up breakdown compared to Fused-Layer++ for inverted residual benchmarks. Higher is better.

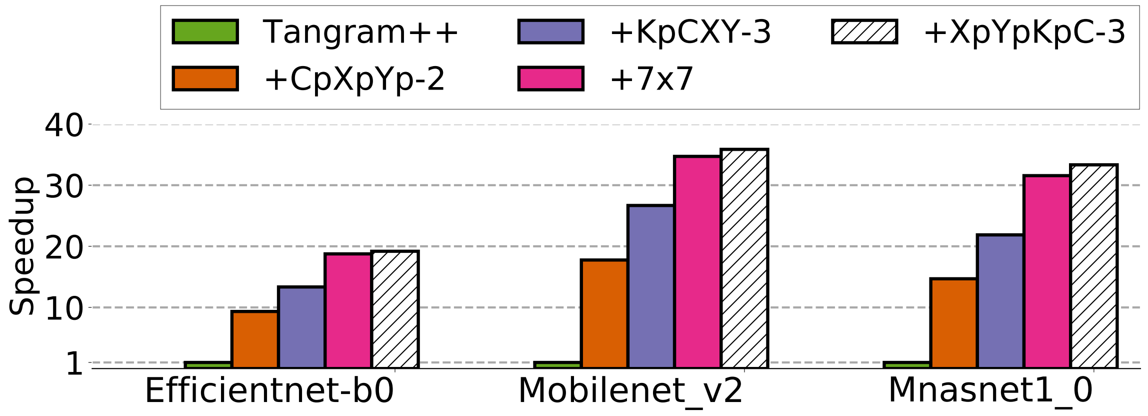
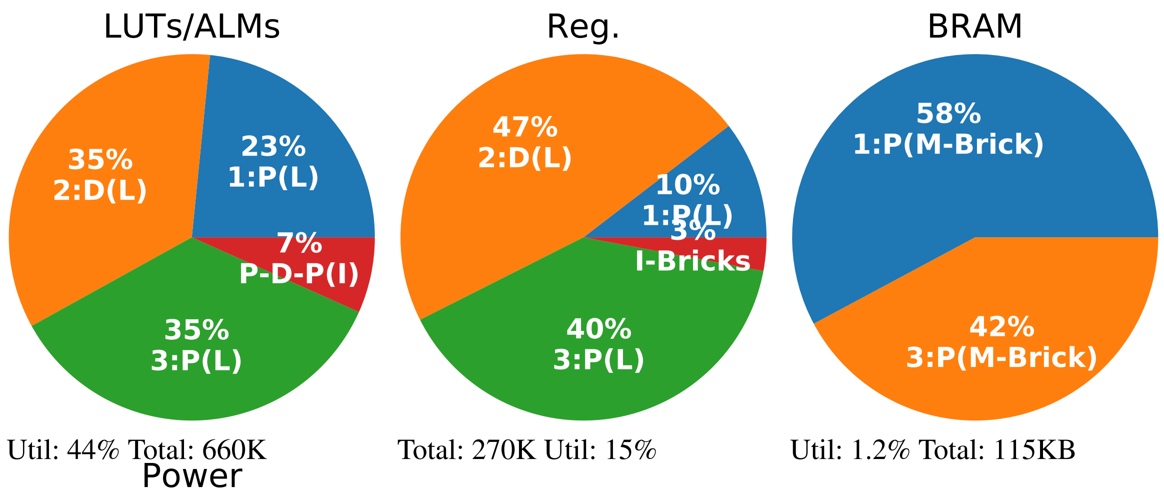


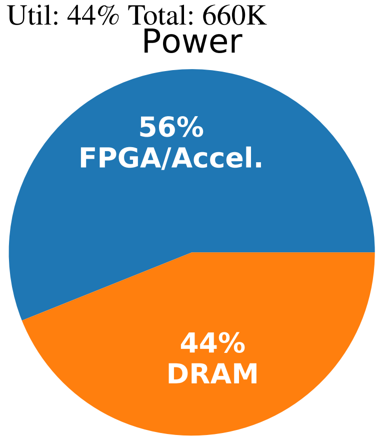
Figure 4.12: Speed up breakdown compared to Tangram++. Higher is better.

4.3.2 X-Layer Hardware Summary

Result: *We run inference at the 0.20ms/inf/W (Xception: 75ms/inf/W).*

In this section, we present the results of FPGA synthesis tools. Here we choose hyper-parameters to enable the design to fit the FPGA and lead to optimal performance for end-to-end DNN. We synthesize our designs on the publicly available Amazon AWS F1 and evaluate performance on FPGA. As shown in Figure 4.13, we breakdown the resource usage into four different components, CLBs/ALMs, Registers, BRAMs, and Power. We also include performance numbers for the same networks running on a Titan GPU. [15]





Power	36.3W
FPS	128 FPS (Xception: 37 FPS)
Batch = 1	0.22 ms/inf/W (Xception: 75ms/inf/W)
Titan XP	299 fps (Xception: 155fps) [15]

Figure 4.13: Hardware Statistics

4.4 Conclusion

In this paper we recognize that current state-of-the-art DNNs provide more opportunity for finer-grain pipelining and profit from deeper pipelines. We developed a new dataflow that only needs to complete partial filters before synchronizing with other layers in the pipeline stage. As far as we are aware, We are the first to create a dataflow exploration tool for pipelined DNNs that can study both coarse-grain and fine-grain pipelines.

Chapter 5

Auto-Split: A General Framework of Collaborative Edge-Cloud AI

This chapter explains our work AUTO-SPLIT which applies joint bit-width allocation along with split point detection in the DNN to reduce end-to-end latency. §5.1 explains the objective function which AUTO-SPLIT solves. §5.2 explains AUTO-SPLIT methodology and solution. In §5.3 we evaluate AUTO-SPLIT with other approaches. In the end, we run several ablation studies in §5.4 and conclude in §5.5.

5.1 Problem Formulation

In this section, we first introduce some basic setup. Then, we formulate a nonlinear integer optimization problem for AUTO-SPLIT. This problem minimizes the overall latency under an edge device memory constraint and a user given error constraint, by jointly optimizing the split point and bid-widths for weights and activation of layers on the edge device.

5.1.1 Basic Setup

Consider a DNN with $N \in \mathbb{Z}$ layers, where $\mathbb{Z}$ defines a non-negative integer set. Let $\mathbf{s}^w \in \mathbb{Z}^N$ and $\mathbf{s}^a \in \mathbb{Z}^N$ be vectors of sizes for weights and activation, respectively. Then, $\mathbf{s}_i^w$ and $\mathbf{s}_i^a$ represent sizes for weights and activation at the i-th layer, respectively. For the given DNN, both $\mathbf{s}^w$ and $\mathbf{s}^a$ are fixed. Next, let $\mathbf{b}^w \in \mathbb{Z}^N$ and $\mathbf{b}^a \in \mathbb{Z}^N$ be bit-widths vectors for weights and activation, respectively. Then, $\mathbf{b}_i^w$ and $\mathbf{b}_i^a$ represent bit-widths for weights and activation at the i-th layer, respectively.

Let $L^{edge}(\cdot)$ and $L^{cloud}(\cdot)$ be latency functions for given edge and cloud devices, respectively. For a given DNN, $\mathbf{s}^w$ and $\mathbf{s}^a$ are fixed. Thus, L^{edge} and L^{cloud} are functions of weights and activation bit-widths. We denote latency of executing the i-th layer of the DNN on edge and cloud by $\mathscr{L}_i^{edge} = L^{edge}(\mathbf{b}_i^w, \mathbf{b}_i^a)$ and $\mathscr{L}_i^{cloud} = L^{cloud}(\mathbf{b}_i^w, \mathbf{b}_i^a)$, respectively. We define a function $L^{tr}(\cdot)$ which measures latency for transmitting data from edge to cloud, and then denote the transmission latency for the i-th layer by $\mathscr{L}_i^{tr} = L^{tr}(\mathbf{s}_i^a \times \mathbf{b}_i^a)$.

To measure quantization errors, we first denote $w_i(\cdot)$ and $a_i(\cdot)$ as weights and activation vectors for a given bit-width at the i-th layer. Without loss of generality , we assume the bit-widths for the original given DNN are 16. Then by using the mean square error function $MSE(\cdot,\cdot)$, we denote the quantization errors at the i-th layer for weights and activation by $\mathscr{D}_i^w = MSE\big(w_i(16), w_i(\mathbf{b}_i^w)\big)$ and $\mathscr{D}_i^a = MSE\big(a_i(16), a_i(\mathbf{b}_i^a)\big)$, respectively. Notice that MSE is a common measure for quantization error and has been widely used in related studies such as [13, 14, 28, 120]. It is also worth noting that while we follow the existing literature on using the MSE metric, other distance metrics such as cross-entropy or KL-Divergence can alternatively be utilized without changing our algorithm.

5.1.2 Problem Formulation

In this subsection, we first describe how to write AUTO-SPLIT's objective function in terms of latency functions and then discuss how we formulate the edge memory constraint and the user given error constraint. We conclude this subsection by providing a nonlinear integer optimization problem formulation for AUTO-SPLIT.

Objective function: Suppose the DNN is split at layer $n \in \{z \in \mathbb{Z} | 0 \le z \le N\}$, then we can define the objective function by summing all the latency parts, i.e.,

$$\mathscr{L}(\mathbf{b}^w, \mathbf{b}^a, n) = \sum_{i=1}^{n} \mathscr{L}_i^{edge} + \mathscr{L}_n^{tr} + \sum_{i=n+1}^{N} \mathscr{L}_i^{cloud}. \tag{5.1}$$

When $n=0$ (resp. $n=N$), all layers of the DNN are executed on the cloud (resp., edge). Since the cloud does not have resource limitations (in comparison to the edge), we assume that the original bit-widths are used to avoid any quantization error when the DNN is executed on the cloud. Thus, $\mathscr{L}_i^{cloud}$ for $i=1,\dots,N$ are constants. Moreover, since $\mathscr{L}_0^{tr}$ represents the time cost for transmitting raw input to the cloud, it is reasonable to assume that $\mathscr{L}_0^{tr}$ is a constant under a given network condition. Therefore, the objective function for CLOUD-ONLY solution $\mathscr{L}(\mathbf{b}^w, \mathbf{b}^a, 0)$ is also a constant.

To minimize $\mathscr{L}(\mathbf{b}^w, \mathbf{b}^a, n)$, we can equivalently minimize

$$\begin{aligned}
&\mathscr{L}(\mathbf{b}^w, \mathbf{b}^a, n) - \mathscr{L}(\mathbf{b}^w, \mathbf{b}^a, 0) \\
&= \left(\sum_{i=1}^{n} \mathscr{L}_i^{edge} + \mathscr{L}_n^{tr} + \sum_{i=n+1}^{N} \mathscr{L}_i^{cloud} \right) - \left(\mathscr{L}_0^{tr} + \sum_{i=1}^{N} \mathscr{L}_i^{cloud} \right) \\
&= \left(\sum_{i=1}^{n} \mathscr{L}_i^{edge} + \mathscr{L}_n^{tr} \right) - \left(\mathscr{L}_0^{tr} + \sum_{i=1}^{n} \mathscr{L}_i^{cloud} \right)
\end{aligned}$$

After removing the constant $\mathscr{L}_0^{tr}$, we write our objective function for the AUTO-SPLIT problem as

$$\sum_{i=1}^{n} \mathscr{L}_i^{edge} + \mathscr{L}_n^{tr} - \sum_{i=1}^{n} \mathscr{L}_i^{cloud}. \tag{5.2}$$

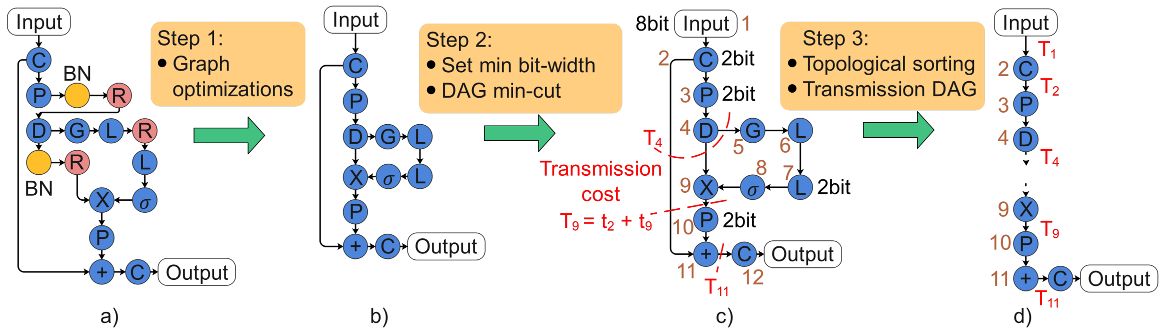


Figure 5.1: An example of graph processing and the steps required to calculate the list of potential split points. 4a) is an example of inverted residual layer with squeeze & excitation from MnasNet. Step1: DAG optimizations for inference such as batchnorm folding and activation fusion. Step2: Set $\mathbf{b}_i^a = b_{min}$ for all activations, create weighted graph, and find the min-cut between sub graphs. Step 3: Apply topological sorting to create transmission DAG where nodes are layers and edges are transmission costs. *C*:Convolution *P*:Pointwise convolution, *D*:Depthwise convolution, *L*:Linear, *G*:Global pool, *BN*:Batch norm, *R*:Relu.

Memory constraint: In hardware, “read-only" memory stores the parameters (weights), and “read-write" memory stores the activations [85]. The weight memory cost on the edge device can be calculated with $\mathscr{M}^w = \sum_{i=1}^n (\mathbf{s}_i^w \times \mathbf{b}_i^w)$. Unlike the weight memory, for activation memory only partial input activation and partial output activation need to be stored in the “read-write" memory at a time. Thus, the memory required by activation is equal to the largest working set size of the activation layers at a time. In case of a simple DNN chain, i.e., layers stacked one by one, the activation working set can be computed as $\mathscr{M}^a = \max_{i=1,\dots,n} (\mathbf{s}_i^a \times \mathbf{b}_i^a)$. However, for complex DAGs, it can be calculated from the DAG. For example, in Fig. 5.1, when the depthwise layer is being processed, both the output activations of layer: 2 (convolution) and layer: 3 (pointwise convolution) need to be kept in memory. Although the output activation of layer: 2 is not required for processing the depthwise layer, it needs to be stored for future layers such as layer 11 (skip connection). Assuming the available memory size of the edge device for executing the DNN is M, then the memory constraint for the AUTO-SPLIT problem can be written as

$$\mathscr{M}^w + \mathscr{M}^a \leq M. \tag{5.3}$$

Error constraint: In order to maintain the accuracy of the DNN, we constrained the total quantization error by a user given error tolerance threshold E. In addition, since we assume that the original bit-widths are used for the layers executing on the cloud, we only need to sum the quantization error on the edge device. Thus, we can write the error constraint for AUTO-SPLIT problem as

$$\sum_{i=1}^{n} (\mathscr{D}_i^w + \mathscr{D}_i^a) \leq E. \tag{5.4}$$

Formulation: To summarize, the AUTO-SPLIT problem can be formulated based on the objective function (5.2) along with the memory (5.3) and error (5.4) constraints

$$\min_{\mathbf{b}^w,\mathbf{b}^a\in\mathbb{B}^n,n} \left(\sum_{i=1}^{n}\mathscr{L}_i^{edge}+\mathscr{L}_n^{tr}-\sum_{i=1}^{n}\mathscr{L}_i^{cloud}\right) \tag{5.5a}$$

$$\mathscr{M}^w+\mathscr{M}^a\leq M, \tag{5.5b}$$

$$\sum_{i=1}^{n}(\mathscr{D}_i^w+\mathscr{D}_i^a)\leq E, \tag{5.5c}$$

where $\mathbb{B}$ is the candidate bit-width set. Before ending this section, we make the following four remarks about problem (5.5).

Remark 1. *For a given edge device, the candidate bit-width set $\mathbb{B}$ is fixed. A typical $\mathbb{B}$ can be $\mathbb{B}=\{2,4,6,8\}$ [37].*

Remark 2. *The latency functions are not given explicitly [30, 68, 114] in practice. Thus, simulators like [79, 86, 111] are commonly used to obtain latency [26, 113, 122].*

Remark 3. *Since the latency functions are not explicitly defined [30, 68, 114] and the error functions are also nonlinear, problem* (5.5) *is a nonlinear integer optimization function and NP-hard to solve. However, problem* (5.5) *does have a feasible solution, i.e., $n=0$, which implies executing all layers of the DNN on cloud.*

Remark 4. *In practice, it is more tractable for users to provide an accuracy drop tolerance threshold A, rather than the error tolerance threshold E. In addition, for a given A, calculating the corresponding E is still intractable. To deal with this issue, we tailor our algorithm in Subsection 5.2.2 to make it user friendly.*

5.2 AUTO-SPLIT Solution

As AUTO-SPLIT problem (5.5) is NP-hard, we propose a multi-step search approach to find a list of potential solutions that satisfy (5.5b) and then select a solution which minimizes the latency and satisfies the error constraint (5.5c).

To find the list of potential solutions, we first collect a set of potential splits $\mathbb{P}$, by analyzing the data size that needs to be transmitted at each layer. Next, for each splitting point $n\in\mathbb{P}$, we solve two sets of optimization problems to generate a list of feasible solutions satisfying constraint (5.5b).

5.2.1 Potential Split Identification

As shown in Fig. 5.1, to identify a list of potential splitting points, we preprocess the DNN graph in three steps. First, we conduct graph optimizations [56] such as batch-norm folding and activation fusion on the original graph (Fig. 5.1a) to obtain an optimized graph (Fig. 5.1b). A potential splitting point n should satisfy the memory limitation with lowest bit-width assignment for DNN, i.e., $b_{min}(\sum_{i=1}^{n}\mathbf{s}_i^w + \max_{i=1,\dots,n} \mathbf{s}_i^a) \leq M$, where b_{min} is the lowest bit-width constrained by the given edge device. Second, we create a weighted DAG as shown in Fig. 5.1c, where nodes are layers and weights of edges are the lowest transmission costs, i.e., $b_{min}\mathbf{s}^a$. Finally, the weighted DAG is sorted in topological order and a new transmission DAG is created as shown in Fig. 5.1d. Assuming the raw data transmission cost is a constant T_0, then a potential split point n should have transmission cost $T_n \leq T_0$ (i.e., $\mathscr{L}_n^{tr} \leq \mathscr{L}_0^{tr}$). Otherwise, transmitting raw data to the cloud and executing DNN on the cloud is a better solution. Thus, the list of potential splitting points is given by

$$\mathbb{P} = \left\{ n \in 0,1,\dots,N \,|\, T_n \leq T_0,\, b_{min}(\sum_{i=1}^{n}\mathbf{s}_i^w + \max_{i=1,\dots,n} \mathbf{s}_i^a) \leq M \right\}. \tag{5.6}$$

5.2.2 Bit-Width Assignment

In this subsection, for each $n \in \mathbb{P}$, we explore all feasible solutions which satisfy the constraints of (5.5). As discussed in Remark 4, explicitly setting E is intractable. Thus, to obtain feasible solutions of (5.5), we first solve

$$\min_{\mathbf{b}^w, \mathbf{b}^a \in \mathbb{B}^n} \sum_{i=1}^{n} (\mathscr{D}_i^w + \mathscr{D}_i^a) \tag{5.7a}$$

$$\mathscr{M}^w + \mathscr{M}^a \leq M, \tag{5.7b}$$

and then select the solutions which are below the accuracy drop threshold A. We observe that for a given splitting point n, the search space of (5.7) is exponential, i.e., $|\mathbb{B}|^{2n}$. To reduce the search space, we decouple problem (5.7) into the following two problems

$$\min_{\mathbf{b}^w \in \mathbb{B}^n} \sum_{i=1}^{n} \mathscr{D}_i^w \quad \mathscr{M}^w \leq M^{wgt}, \tag{5.8}$$

$$\min_{\mathbf{b}^a \in \mathbb{B}^n} \sum_{i=1}^{n} \mathscr{D}_i^a \quad \mathscr{M}^a \leq M^{act}, \tag{5.9}$$

where M^{wgt} and M^{act} are memory budgets for weights and activation, respectively, and $M^{wgt} + M^{act} \leq M$. To solve problems (5.8) and (5.9), we apply the Lagrangian method proposed in [92].

To find feasible pairs of M^{wgt} and M^{act}, we do a two-dimensional grid search on M^{wgt} and M^{act}. The candidates of M^{wgt} and M^{act} are given by uniformly assigning bit-widths $\mathbf{b}^w, \mathbf{b}^a$ in $\mathbb{B}$. Then, the maximum number of feasible pairs is $|\mathbb{B}|^2$. Thus, we significantly reduce the search space from $|\mathbb{B}|^{2n}$ to at most $2|\mathbb{B}|^{n+2}$ by decoupling (5.7) to (5.8) and (5.9).

We summarize the steps above in Algorithm 1. In the proposed algorithm, we obtain a list $\mathbb{S}$ of potential solutions $(\mathbf{b}^w, \mathbf{b}^a, n)$ for problem (5.5). Then, a solution that minimizes the latency and satisfies the accuracy drop constraint can be selected from the list.

Before ending this section, we make a remark about Algorithm 1.

Remark 5. *Due to the nature of the discrete nonconvex and nonlinear optimization problem, it is not possible to precisely characterize the optimal solution of* (5.5). *However, our algorithm guarantees* $\mathscr{L}(\boldsymbol{b}^w,\boldsymbol{b}^a,n) \leq \min\Big(\mathscr{L}(\emptyset,\emptyset,0),\mathscr{L}(\boldsymbol{b}_e^w,\boldsymbol{b}_e^a,N)\Big)$*, where* $(\emptyset,\emptyset,0)$ *is the* CLOUD-ONLY *solution, and* $(\boldsymbol{b}_e^w,\boldsymbol{b}_e^a,N)$ *is the* EDGE-ONLY *solution when it is possible.*

Algorithm 1: Joint DNN Splitting and Bit Assignment

```
   Result: Split & bit-widths for weights and activations
1  $\mathbb{S} \leftarrow [(\emptyset,\emptyset,0)]$.  // Initialize with CLOUD-ONLY solution.
2  $\mathbb{B} \leftarrow$ Bit-widths supported by the edge device
3  for k in $[1,\ldots,|\mathbb{B}|]$ do
4      $M_k^{wgt} = \sum_{i=1}^{n}(\mathbf{s}_i^w \times \mathbb{B}[k])$.
5      $M_k^{act} = \max_{i=1,\ldots,n}(\mathbf{s}_i^a \times \mathbb{B}[k])$.
6  $\mathbb{P} \leftarrow$ Solve (5.6).
7  for n in $\mathbb{P}$ do
8      for $M^{wgt}$ in $[M_1^{wgt},\ldots,M_{|\mathbb{B}|}^{wgt}]$ do
9          $\mathbf{b}^w \leftarrow$ Solve (5.8).
10         for $M^{act}$ in $[M_1^{act},\ldots,M_{|\mathbb{B}|}^{act}]$ do
11             if $M^{wgt}+M^{act}>M$ then
12                 Continue.
13             $\mathbf{b}^a \leftarrow$ Solve (5.9).
14             if $(\mathbf{b}^w,\mathbf{b}^a,n)$ satisfies (5.5b) then
15                 $\mathbb{S}$.append$\big((\mathbf{b}^w,\mathbf{b}^a,n)\big)$.
16 Return $\mathbb{S}$.
```

5.2.3 Post-Solution Engineering Steps

After obtaining the solution of (5.5), the remaining work for deployment is to pack the split-layer activations, transmit them to the cloud, unpack them in the cloud, and feed them to the cloud DNN. There are some engineering details involved in these steps, such as how to pack less than 8-bits data types, transmission protocols, etc. We provide such details in the appendix.

5.3 Experiments

In this section, we present our experimental results. §5.3.1-§5.3.4 cover our simulation-based experiments. We compare AUTO-SPLIT with existing state-of-the-art distributed frameworks. We also perform ablation studies to show step by step, how AUTO-SPLIT arrives at good solutions compared to existing approaches. In addition, in §5.3.5 we demonstrate the usage of AUTO-SPLIT for a real-life example of license plate recognition on a low power edge device.

Table 5.1: Hardware platforms for the simulator experiments

Attribute	Eyeriss [19]	TPU
On-chip memory	192 KB	28 MB
Off-chip memory	4 GB	16 GB
Bandwidth	1 GB/sec	13 GB/sec
Performance	34 GOPs	96 TOPs
Uplink rate	3 Mbps	

5.3.1 Experiments Protocols

Previous studies [30, 68, 114] have shown that tracking multiply-accumulate operations (MACs) or GFLOPs does not directly correspond to measuring latency. We measure edge and cloud device latency on a cycle-accurate simulator based on SCALE-SIM [86] from ARM software (also used by [26, 113, 122]). For edge devices, we simulate Eyeriss [19] and for cloud devices, we simulate Tensor Processing Units (TPU). The hardware configurations for Eyeriss and TPU are taken from SCALE-SIM (see Table 5.1). In our simulations, lower bit precision (sub 8-bit) does not speed up the MAC operation itself, since, the existing hardware has fixed INT-8 MAC units. However, lower bit precision speeds up data movement across off-chip and on-chip memory, which in turn results in an overall speedup. Moreover, we build on quantization techniques [13, 77] using [125]. The appendix includes other engineering details used in our setup. It also includes an ablation study on the network bandwidth.

5.3.2 Accuracy vs Latency Trade-off

As seen in Sections 5.1 and 5.2, AUTO-SPLIT generates a list of candidate feasible solutions that satisfy the memory constraint (5.5b) and error threshold (5.5c) provided by the user. These solutions vary in terms of their error threshold and therefore the downstream task accuracy. In general, good solutions are the ones with low latency and high accuracy. However, the trade-off between the two allows for a flexible selection from the feasible solutions, depending on the needs of a user. The general rule of thumb is to choose a solution with the lowest latency, which has at worst only a negligible drop in the accuracy (i.e., the error threshold constraint is implicitly applied through the task accuracy). However, if the application allows for a larger drop in the accuracy, that may correspond to a solution with even lower latency. It is worth noting that this flexibility is unique to AUTO-SPLIT and the other methods such as Neurosurgeon [58], QDMP [119], U8 (uniform 8-bit quantization), and CLOUD16 (CLOUD-ONLY) only provide one single solution.

Fig. 5.2 shows a scatter plot of feasible solutions for ResNet-50 and Yolov3 DNNs. For ResNet-50, the X-axis shows the Top-1 ImageNet error normalized to CLOUD-ONLY solution and for Yolo-v3, the X-axis shows the mAP drop normalized to CLOUD-ONLY solution. The Y-axis shows the end-to-end latency, normalized to CLOUD-ONLY. The Design points closer to the origin are better.

AUTO-SPLIT can select several solutions (i.e., pink dots in Fig. 5.2-left) based on the user error threshold as a percentage of full precision accuracy drop. For instance, in case of ResNet-50, AUTO-SPLIT can produce Split Approach. Partial execution on edge device and partial on cloud device (SPLIT) solutions with end-to-end latency of 100%, 57%, 43% and 43%, if the user provides an error threshold of 0%, 1%, 5% and 10%. For an error threshold of 0%, AUTO-SPLIT selects CLOUD-ONLY as the solution whereas, in case of an error threshold of 5% and 10%, AUTO-SPLIT selects the same solution.

The mAP drop is higher in the object detection tasks. It can be seen in Fig. 5.2 that uniform quantization of 2-bit (U2), 4-bit(U4), and 6-bit (U6) result in an mAP drop of more than 80%. For a user error threshold of 0%, 10%, 20%, and 50%, AUTO-SPLIT selects a solution with an end-to-end latency of 100%, 37%, 32%, and 24% respectively. For each of the error thresholds, AUTO-SPLIT provides a different split point (i.e., pink dots in Fig. 5.2-right). For Yolo-v3, QDMP and Neurosurgeon result in the same split point which leads to 75% end-to-end latency compared to the CLOUD-ONLY solution.

5.3.3 Overall Benchmark Comparisons

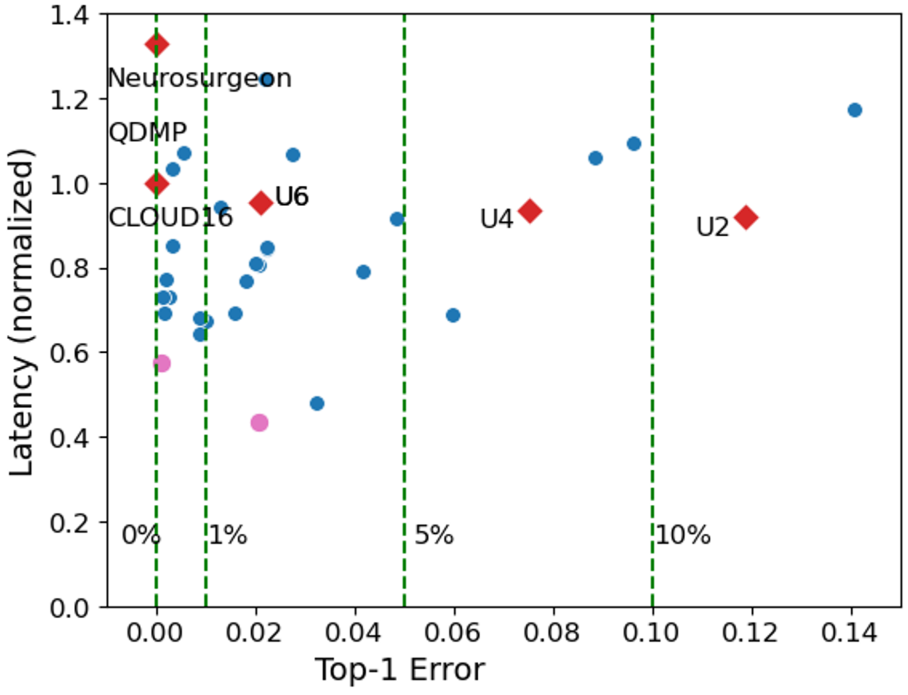
Fig. 5.3 shows the latency comparison among various classification and object detection benchmarks. We compare AUTO-SPLIT, with baselines: Neurosurgeon [58], QDMP [119], U8 (uniform 8-bit quantization), and CLOUD16 (CLOUD-ONLY). It is worth noting that for optimized execution graphs (see §2.3, Figure 2.12)), DADS [49] and QDMP [119] generate a same split solution, and thus we do not report DADS separately here (see Section 5.2 in [119]). The left axis in Fig. 5.3 shows the latency normalized to CLOUD-ONLY solution which is represented by bars in the plot. Lower bars are better. The right axis shows top-1 ImageNet accuracy for classification benchmarks and mAP (IoU=0.50:0.95) from COCO 2017 benchmark for YOLO-based detection models.

Accuracy: Since Neurosurgeon and QDMP run in full precision, their accuracies are the same as the CLOUD-ONLY solution.

For image classification, we selected a user error threshold of 5%. However, AUTO-SPLIT solutions are always within 0-3.5% of top-1 accuracy of the CLOUD-ONLY solution. AUTO-SPLIT selected CLOUD-ONLY as the solution for ResNext50_32x4d, EDGE-ONLY solutions with mixed precision for ResNet-18, Mobilenet_v2, and Mnasnet1_0. For ResNet-50 and GoogleNet AUTO-SPLIT selected SPLIT solution.

For object detection, the user error threshold is set to 10%. Unlike AUTO-SPLIT, uniform 8-bit quantization can lose significant mAP, i.e. between 10–50%, compared to CLOUD-ONLY. This is due to the fact that object detection models are generally more sensitive to quantization, especially if bit-widths are assigned uniformly.

Latency: AUTO-SPLIT solutions can be: a) CLOUD-ONLY, b) EDGE-ONLY, and c) SPLIT. As observed in Fig. 5.3, AUTO-SPLIT results in a CLOUD-ONLY solution for ResNext50_32x4d. EDGE-



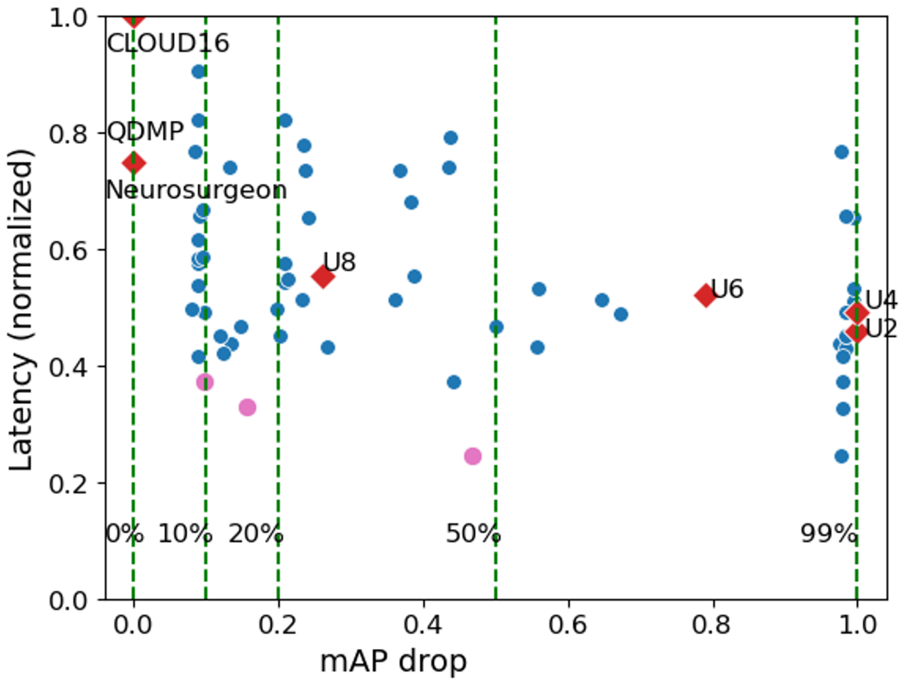


Figure 5.2: Accuracy vs latency trade-off for ResNet-50 (top) and Yolo-v3 (bottom), Towards origin is better. U2, U4, U6 and U8 indicate uniform quantization (EDGE-ONLY). CLOUD16 indicates FP16 configuration (CLOUD-ONLY). The green lines show error threshold which a user can set. AUTO-SPLIT can provide different solutions based on different Error thresholds.

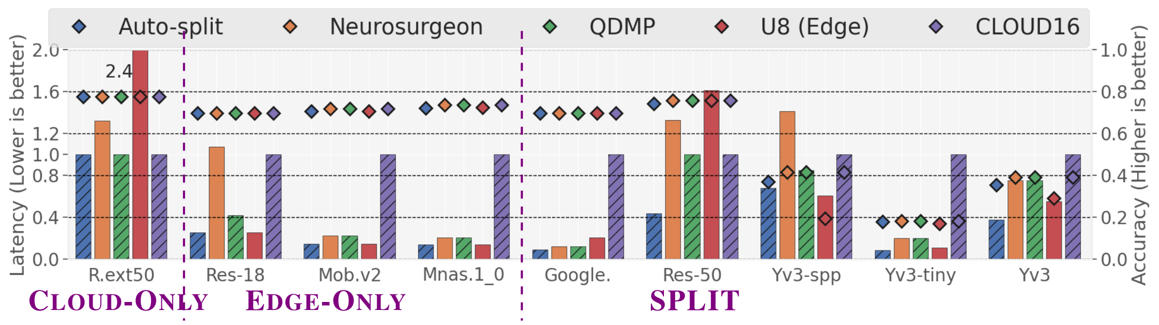


Figure 5.3: Latency (bars) vs Accuracy (points) comparison. Depending on the device constraints, DNN architecture, and network latency, the optimal solution can be achieved from CLOUD-ONLY, EDGE-ONLY, or SPLIT. The user error threshold for image classification workloads is 5%, and for object detection workloads is 10%.

ONLY solutions are reached for ResNet-18, MobileNet-v2, and Mnasnet1_0, and for rest of the benchmarks AUTO-SPLIT results in a SPLIT solution. When AUTO-SPLIT does not suggest a CLOUD-ONLY solution, it reduces latency between 32–92% compared to CLOUD-ONLY solutions.

Neurosurgeon cannot handle DAGs. Therefore, we assume a topological sorted DNN as input for Neurosurgeon, similar to [49, 119]. The optimal split point is missed even for float models due to information loss in topological sorting. As a result, compared to Neurosurgeon, AUTO-SPLIT reduces latency between 24–92%.

DADS/QDMP can find optimal splits for float models. They are faster than Neurosurgeon by 35%. However, they do not explore the new search space opened up after quantizing edge DNNs. Compared to DADS/QDMP, AUTO-SPLIT reduces latency between 20–80%. Note that QDMP needs to save the entire model on the edge device which may not be feasible. We define QDMP$_E$, a QDMP baseline that saves only the edge part of the DNN on the edge device.

To sum up, AUTO-SPLIT can automatically select between CLOUD-ONLY, EDGE-ONLY, and SPLIT solutions. For EDGE-ONLY and SPLIT, our method suggests solutions with mixed precision bit-widths for the edge DNN layers. The results show that AUTO-SPLIT is faster than uniform 8-bit quantized DNN (U8) by 25%, QDMP by 40%, Neurosurgeon by 47%, and CLOUD-ONLY by 70%.

5.3.4 Comparison with QDMP + Quantization

As mentioned, QDMP (and others) operates in floating-point precision. In this subsection, we show that it will not be sufficient, even if the edge part of the QDMP solutions are quantized. To this end, we define QDMP$_E$+U_4, a baseline of adding uniform 4-bit precision to QDMP$_E$. Table 5.2 shows split index and model size for AUTO-SPLIT, QDMP$_E$, and QDMP$_E$+U_4. Note that, ZeroQ reported ResNet-50 with the size of 18.7MB for 6-bit activations and mixed-precision weights. Also, note that 4-bit quantization is the best that any post-training quantization can achieve on a DNN and we discount the accuracy loss on the solution. In spite of discounting accuracy, AUTO-SPLIT reduces the edge DNN size by 14.7× compared to QDMP$_E$ and 3.1× compared to QDMP$_E$+U4.

Table 5.2: Comparing QDMP$_E$, AUTO-SPLIT, and QDMP$_E$+U4

	AUTO-SPLIT		QDMP$_E$		QDMP$_E$+U4
	Split idx	MB	Split idx	MB	MB
Google.	18	0.4	18	3.5	0.44
Resnet-50	12	0.9	53	50	12.5
Yv3-spp	1	1.3	52	99	22.2
Yv3-tiny	7	8.8	7	18.2	4.44
Yv3	33	13.3	58	118	29.6

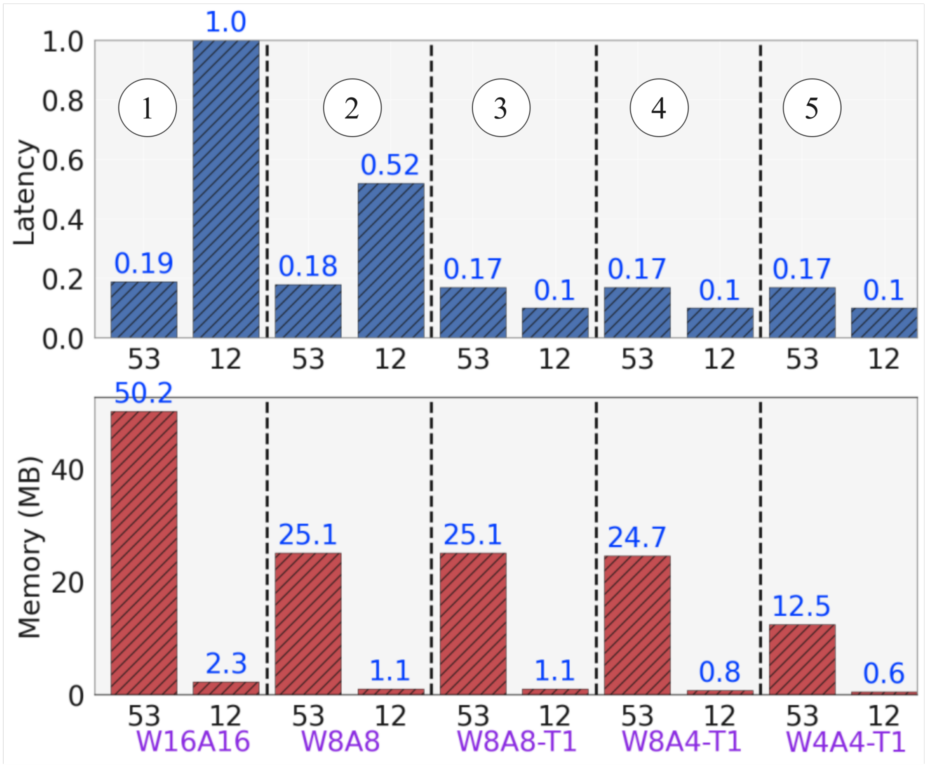


Figure 5.4: ResNet-50 latency & memory for **AUTO-SPLIT (split @12)** and **QDMP (split @53)**. W8A8-T1 implies weight, activation, and transmission bit-widths of 8, 8, and 1, respectively. Transmission cost at split 53 is $\simeq 3\times$ less than split 12.

Next, we demonstrate how QDMP$_E$ can select a different split index compared to AUTO-SPLIT. Fig. 5.4 shows the latency and model size of ResNet-50. ① : W16A16-T16: Weights, activations, and transmission activations are all 16 bits precision. The transmission cost at split index=53 (suggested by QDMP$_E$) is $\simeq 3\times$ less compared to that of split index=12 (suggested by AUTO-SPLIT). Overall latency of split index=53 is 81% less compared to split index=12. A similar trend can be seen when the bit-widths are reduced to 8-bit (②: W8A8-T8). However, in ③: W8A8-T1 we reduce the transmission bit-width to 1-bit and notice that split index=12 is 7% faster compared to split index=53. Since AUTO-SPLIT also considers edge memory constraints in partitioning the DNN, it can be noticed that the model size of the edge DNN in split index=12 is always orders of magnitude less compared to the split index=53. In ④ and ⑤ the model size is further reduced. *Overall* AUTO-SPLIT *solutions are* 7% *faster and* $20\times$ *smaller in edge DNN model size compared to a QDMP + mixed precision algorithm when both models have the same bit-width for edge DNN weights, activations, and transmitted activations.*

5.3.5 Case Study

We demonstrate the efficacy of the proposed method using a real application of license plate recognition used for authorized entry (deployed to customer sites). In the appendix, we also provide a demo of another case study for the task of the person and face detection.

Consider a camera as an edge device mounted at a parking lot that authorizes the entry of certain vehicles based on the license plate registration. Inputs to this system are camera frames and outputs are the recognized license plates (as strings of characters). AUTO-SPLIT is applied offline to determine the split point assuming an 8-bit uniform quantization. AUTO-SPLIT ensures that the edge DNN fits on the device and high accuracy is maintained. The edge DNN is then passed to TensorFlow-Lite for quantization and is stored on the camera. When the camera is online, the output activations of the edge DNN are transmitted to the cloud for further processing. The edge DNN runs parts of a custom YOLOv3 model. The cloud DNN consists of the rest of this YOLO model, as well as an LSTM model for character recognition.

The edge device (Hi3516E V200 SoC with Arm Cortex A7 CPU, 512MB On-chip, and 1GB Off-chip memory) can only run TensorFlow-Lite (cross-compiled for it) with a C++ interface. Thus, we built an application in C++ to execute the edge part, and the rest is executed through a Python interface.

Table 5.3 shows the performance over an internal proprietary license plate dataset. The AUTO-SPLIT solution has similar accuracy to others but has lower latency. Furthermore, as the LSTM in AUTO-SPLIT runs on the cloud, we can use a larger LSTM for recognition. This improves the accuracy further with a negligible latency increase.

Table 5.3: Evaluation of License Plate Recognition solutions

Model	Accuracy	Latency	Edge Size
Float (on edge)	88.2%	Doesn't fit	295 MB
Float (to cloud)	88.2%	970 ms	0 MB
TQ (8 bit)	88.4%	2840 ms	44 MB
AUTO-SPLIT	88.3%	**630 ms**	**15 MB**
AUTO-SPLIT(large LSTM)	**94%**	**650 ms**	**15 MB**

5.4 Ablation Studies

In this section, we perform a series of ablation experiments to deepen our understanding of AUTO-SPLIT. **First**, we explore the effect of compression on transmission bits i) activations for SPLIT and ii) JPEG image compression for CLOUD-ONLY approach. **Second**, we study the effect of network speed on SPLIT solutions. **Third**, we look at the DNN architecture to study which networks are likely to generate SPLIT solutions. For example, a DNN such as Mobilenet_v2 with a smaller volume of activations at DNN graph junctions is a good candidate for SPLIT solutions. By graph junction, we mean, in case of a SPLIT only activations of one layer need to be transmitted. **Fourth**, we study the effect of selecting split points with the same activation volume at different DNN layer indices. **Fifth**, we study the effect of input image resolution on SPLIT solutions. Larger input images imply more options for SPLIT selection. **Sixth**, we study the effect of activation optimization for bit-allocation. We explore whether it is beneficial to compress more weights or more activations.

Compression of SPLIT layer features for edge-cloud splitting

With the availability of mature data compression techniques, it is natural to consider applying data compression before transmission to the cloud, as an attempt to transmit less data, to lower the latency. For CLOUD-ONLY solutions that mean applying image compression (e.g. JPEG), and for AUTO-SPLIT corresponds to applying feature compression. Note that this kind of compression depends highly on the edge device support, both in terms of hardware and software, and thus is not always available. Therefore, we study it as an ablation, rather than the default in the algorithm.

For the CLOUD-ONLY solutions, we studied the use of JPEG compression, as a candidate with relatively low computation overhead and a good compression ratio.

For the task of object detection on Yolov3, at 416×416 resolution, and PIL library for JPEG execution, we observed considerable improvements in the solutions, as shown in Table 5.4. However, strong compression results in severe loss of accuracy in the CLOUD-ONLY case. For feature compression in AUTO-SPLIT, we followed a similar approach and used JPEG compression (We initially tried Huffman coding, but the overhead of compression was high and didn't give good solutions). Results of feature compression showed that AUTO-SPLIT benefits more on compression ratio because activations are sparse (20+%) and are represented by lower bits e.g. 2bits compared to

8bits (0-255) for input images. For activations, we split channels into groups of three and applied JPEG compression.

Table 5.4: Effect of input or feature compression on solutions.

Method	JPEG Quality Factor	Compression Ratio	mAP	Normalized Latency
CLOUD-ONLY	No Compression	1×	0.39	1.0
CLOUD-ONLY	LossLess	2×	0.39	0.56
CLOUD-ONLY	80	5×	0.38	0.23
CLOUD-ONLY	60	8×	0.35	0.15
CLOUD-ONLY	40	10×	0.29	0.13
CLOUD-ONLY	20	17×	0.22	0.09
AUTO-SPLIT	LossLess	15×	0.35	0.08

Note that the overhead of latency for compression/decompression will add another variable to the equation and may result in different splits. For JPEG compression on RaspberryPi3, we measured 28ms overhead for CLOUD-ONLY and 9ms for AUTO-SPLIT. Moreover, compression is better to be lossless or light, otherwise, it will damage the accuracy (even if it's not clearly visible). See Fig7-10 in [73] on object detection or Table1 in [46] on classification. At a similar mAP, AUTO-SPLIT shows 47% speed-up compared to CLOUD-ONLY-QF60. This gap is smaller compared to the non-compressed case of 58% in Fig. 5.3 (Gap depends on architecture/task).

Ablation study on network speed

Table 5.5 shows results on YOLOv3 (416x416 resolution, and latency is normalized in each case to CLOUD-ONLY). It is observed from Table 5.5 that AUTO-SPLIT mAP drops at 20Mbps. The same experiment for YOLOv3SPP (different architecture) resulted in a much better solution. So even at 20Mbps, depending on the architecture, there may be good SPLIT solutions.

Table 5.5: Ablation study on network bandwidth.

Model	Network Bandwidth	Accuracy (mAP) AUTO-SPLIT/ CLOUD-ONLY	Normalized Latency AUTO-SPLIT/ CLOUD-ONLY
Yolov3	1Mbps	0.37/0.39	0.26/1
Yolov3	3Mbps	0.37/0.39	0.37/1
Yolov3	10Mbps	0.34/0.39	0.83/1
Yolov3	20Mbps	0.25/0.39	0.75/1
Yolov3-SPP	20Mbps	0.37/0.41	0.71/1

Splitting object detection models

We studied two styles of object detection models: a) Yolo based: Yolov3-tiny, Yolov3, and Yolov3-spp, and b) FasterRCNN with ResNet-50 backbone. Running AUTO-SPLIT resulted in SPLIT solutions for Yolo models, but suggested the CLOUD-ONLY solution for FasterRCNN. In this subsection, we explain the caveats of selecting the backbone network when searching for a SPLIT solution in object detection models. For a SPLIT solution to be feasible, the transmission cost of activations at the split point should be at least less than the input image, otherwise, a CLOUD-ONLY solution may have lower end-to-end latency.

Most Object detection models have a backbone network that branches off to detection and recognition necks/heads. These detection and recognition heads typically collect intermediate features from the backbone network. For example, Table 5.7 shows the layer indices of the intermediate layers from which the output features are collected in the Feature pyramid network (FPN) for Faster RCNN and YOLO layers for Yolov3. In FasterRCNN, the FPN starts fetching intermediate features from as early as layer index=10, thus in the case of a SPLIT solution, these features are also required to be transmitted to the cloud, unless the entire model is executed in the edge device. As we go deeper in the DNN in search of a SPLIT solution more and more extra intermediate layers need to be transmitted along with the output activation at the split point (see Figure 5.5). Thus, AUTO-SPLIT suggests a CLOUD-ONLY solution for Faster RCNN.

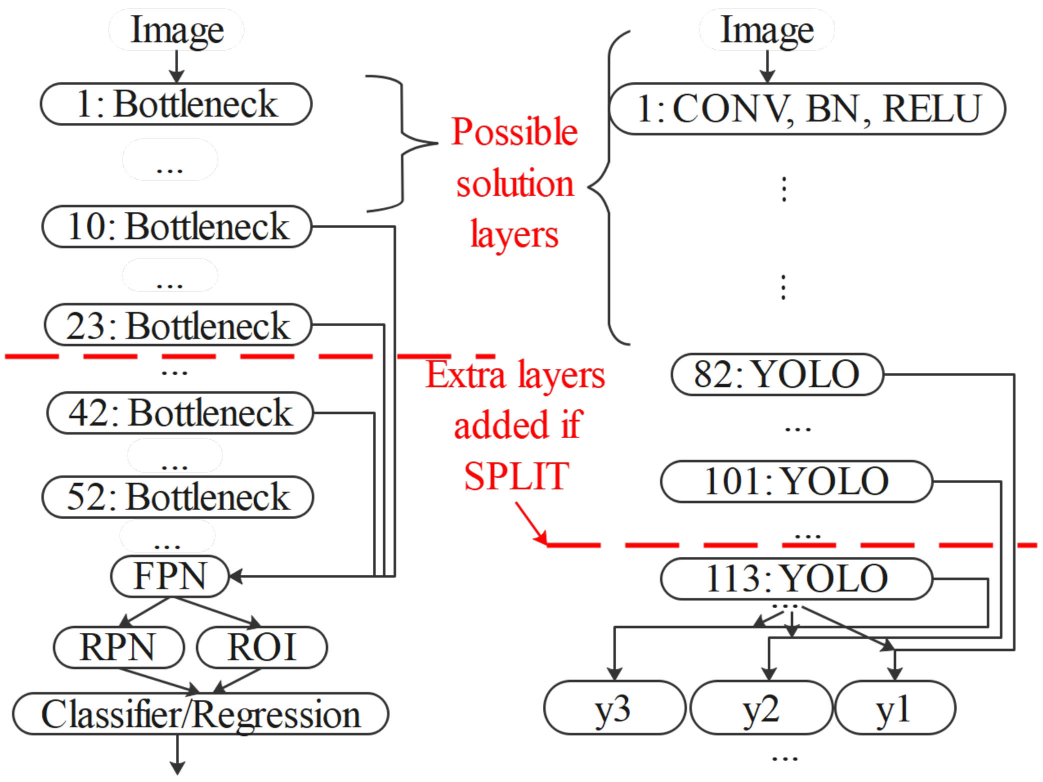


Figure 5.5: Split layers for FasterRCNN (left) and Yolov3 (right)

On the contrary, in YOLO based models there are enough layers to search for a SPLIT solution before the YOLO layers. For example, in YOLOv3-spp, the search space for a SPLIT solution lies between layer index 0 to 82 and AUTO-SPLIT suggested layer index=33 as the split point. Therefore, when co-designing an object detection model for distributed inference, it is crucial to start collecting intermediate features as late as possible in the DNN layers. It will be worth looking at a modified Faster RCNN model retrained with an FPN network collecting features from layer index=[23,42,52] and see if the accuracy does not drop too much. Such a modified model will generate SPLIT solutions.

Selecting split points with equal activation volume

Table 5.6 shows the potential split points towards the end of a pre-trained ResNet-50 network. After the graph optimizations are applied these layers do not have any other dependencies. The output feature map (OFM) volume is the activation volume for potential split points and the volume difference is the difference between OFM volume and input image volume. For a SPLIT to be valid one primary condition is that the volume difference should be negative.

Table 5.6: Potential splits towards the end of ResNet-50

Index	Layer name	Volume	Shape	Vol. Diff
46	layer4.0.conv3	100,352	(2048,7,7)	-5076
49	layer4.1.conv3	100,352	(2048,7,7)	-5076
52	layer4.2.conv3	100,352	(2048,7,7)	-5076
53	fc	1,000	(1,1000)	-149528
-1	i/p image	150,528	(3,224,224)	0

Table 5.7: Layer indices of the collected output activations

Models	Intermediate Layer indices
Yolov3-tiny	[16, 23]
Yolov3	[82, 94, 106]
Yolov3-spp	[89, 101, 113]
Faster RCNN	[10, 23, 42, 52]

The end-to-end latency has three components a) edge latency, b) transmission latency and c) cloud latency. In Table 5.6, layers 46, 49, and 52 have the same shape and thus, the same transmission volume difference. Without quantization, layer 46 will be the SPLIT solution, since the cloud device is faster than the edge device, and in case of the same transmission cost it is preferable to select early layers.

With quantization, however, the transmission cost will vary depending on the bit-widths assigned to each layer. The assignment of bit-widths depends on the sensitivity of each layer to compress without losing accuracy. If layer 49 can be compressed more than layer 46, then layer 49 can be selected as a SPLIT solution. The sensitivity of each layer to bit compression can be measured in different ways. AUTO-SPLIT considers the quantization error of the layers. Other techniques include: Hessian of the feature vector [32] or layer sensitivity by quantizing only one layer at a time [17].

Effect of Image Resolution

Figure 5.6 plots latency and accuracy for different image resolutions for yolov3-spp model. The uniform 8-bit quantization attains a significant loss on accuracy and mAP degrades to 0.19–0.26. On the other hand, AUTO-SPLIT selects SPLIT solutions with mAP between 0.28 – 0.36. As the input resolution increases the mAP gets higher for SPLIT solution as well.

On average, the SPLIT solution is faster than CLOUD-ONLY solution by 50%. Compared to EDGE-ONLY uniform 8-bit quantization, AUTO-SPLIT is faster by 10% for image resolution 416 and 512, however, for 608, it is slower by 10%. The reason for lower latency for AUTO-SPLIT is due to the high accuracy threshold setting.

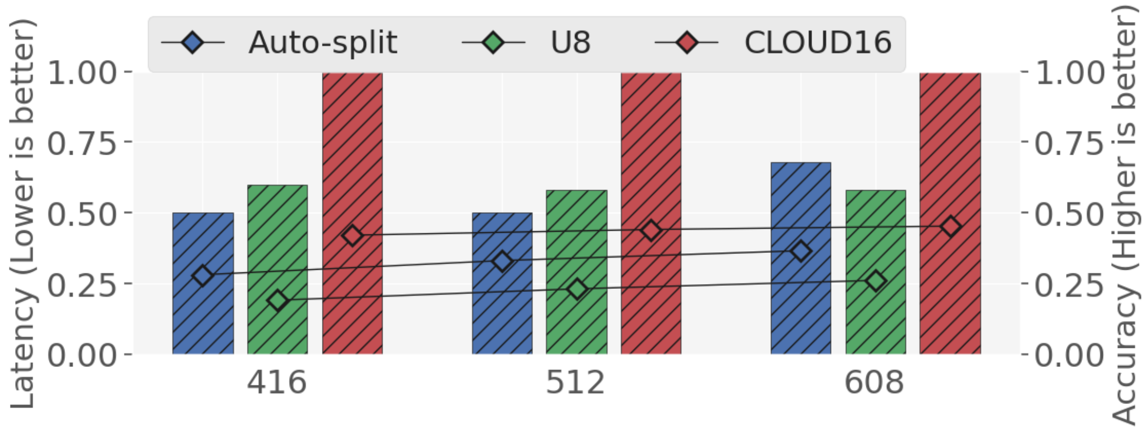


Figure 5.6: Yolo-v3-spp latency (bars) and mAP(line) for different image resolutions.

Table 5.8 compares top two design points for image resolution 608, the *layer_index*=1 results in 0.365 mAP and a normalized latency of 0.68 whereas *layer_index*=27 results in 0.36 mAP and a normalized latency of 0.49. In spite of having a lower latency (by 10%) compared to uniform 8-bit quantization (U8), *layer_index*=27 was dropped by the AUTO-SPLIT algorithm and *layer_index*=1 was selected due to high accuracy threshold for yolo models.

Activation Optimization

Activation optimization, as mentioned above, constraints the total memory usage by activations on the edge device. Figure 5.7 shows the bit allocations for activations and weights for two scenarios with the total memory size constrained to 343.1875 KB (on ResNet-50). In Scenario 1 (Figure 5.7-left), activation bits are traded off for weight bits and in Scenario 2 (Figure 5.7-right), it is vice versa. It

Table 5.8: Comparison of top two design points for image resolution 608×608 in yolo-v3-spp model

Split Index	Trans. Bits	Avg. Bits	Model Size	Latency	mAP
1	2	8	2.8 MB	0.68	0.365
27	6	8	15.6 MB	0.49	0.36

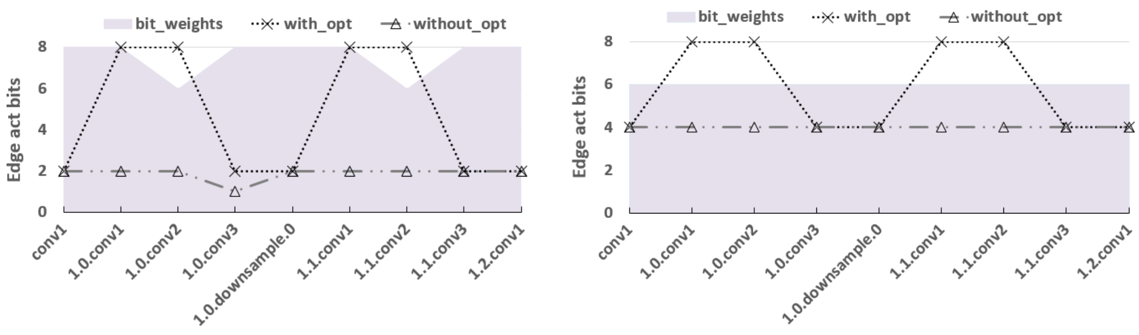


Figure 5.7: A demonstration of bit-width selection for activations w/ and w/o the optimization of (5.9). The shaded region shows the bit-width selection for weights in each scenario. Left: activation bits are traded off for weight bits, and right: vice versa.

Table 5.9: Numerical values for bit assignment of Fig. 5.7. Left: activation bits are traded off for weight bits, and right: vice versa.

Bit regime	Top-1	Model size	act. bits	weight bits
all act. 8	74.57	931 KB	7.71	7.12
with opt.	32.89	343 KB	3.14	7.12
without opt.	18.54	343 KB	1.8	7.12

Bit regime	Top-1	Model size	act. bits	weight bits
all act. 8	75.43	931 KB	7.8	6
with opt.	75.43	343 KB	4.7	6
without opt.	75.4	343 KB	4	6

can be observed that by increasing the activation bits using (5.9), the Top-1 accuracy is improved without affecting the total memory size. Table 5.9 shows the numerical values for bit-widths assigned to weights and activations corresponding to the scenarios of Fig. 5.7. In this table, the “all act 8" case considers all activations as 8 bits with an exception of the split layer. Though “all act 8" has higher accuracy, its model size is 3× larger compared to the other selected bit widths.

5.5 Conclusion

This paper investigates the feasibility of distributing the DNN inference between edge and cloud while simultaneously applying mixed-precision quantization on the edge partition of the DNN. We propose to formulate the problem as an optimization in which the goal is to identify the split and the bit-width assignment for weights and activations, such that the overall latency is reduced without sacrificing accuracy. This approach has some advantages over existing strategies such as being secure, deterministic, and flexible in architecture. The proposed method provides a range of options in the accuracy-latency trade-off which can be selected based on the target application requirements.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

The aim of the thesis is to enable the efficient execution of DNN inference on edge devices. The deployment approach of DNN on edge devices depends on the application scenarios and operating constraints. In general, the EDGE-ONLY and SPLIT approaches result in lower end-to-end latency for many applications and are the preferred approach for deployment.

EDGE-ONLY deployment with X-Layer

In our first work X-Layer, we explored efficient deployment of EDGE-ONLY approach with Cross-layer dataflows. **LR-CONVs for EDGE-ONLY:** For EDGE-ONLY approach, we found that NAS techniques are employed to develop compact DNNs with low MAC and memory requirements. These emerging DNNs heavily employ LR-CONVs. The LR-CONVs have low arithmetic intensity compared to CONVs. The DP layer especially is memory bound and has $18\times - 79\times$ less arithmetic intensity compared to PT layers. This leads to poor hardware utilization on existing edge devices. The [38] reported almost 100% of runtime being used by DP layers for Mobilenet_v2. We also ran Mobilenet on Eyeriss on [87] and found $\simeq$7% utilization.

To solve the above issues, we realized that LR-CONVs can benefit from interlayer reuse, and explored Cross-layer dataflows. We proposed X-Layer, which defines the taxonomy of Cross-layer dataflows. We found that the dataflow is closely tied to the hardware implementation. The Cross-layer dataflows can be divided into outer dataflow (executed in software), and inner dataflow and hardware mapping (both executed in hardware).

Fix hardware mapping: We first fixed the hardware mapping to $C|K$ for PT and CONV layers, and $F_y|YC$ for DP layer which are the optimal hardware mapping for respective layers [20, 62, 115].

Matrix transpose: The exploration of Cross-layer dataflow is not possible without the ability of dynamic data transformation support in the hardware which is enabled by T-Bricks. The T-Bricks are Multi-Input Multi-Output (MIMO) queues that allow dynamic data transpose operation.

Heterogeneous Resource Allocation: We also found that different stages of the Cross-layer dataflow require different resource allocations such as buffer and mac allocations. These resource allocations depend on the DNN algorithm, layer depth in the DNN, and layer type. We chose a 7×7 NN engine as opposed to a 16×16 engine, since, it allows for finer granularity of resource allocation to each layer of the cross-layer dataflow. Also, the largest kernel size we found in our benchmarks was 7×7.

Cross-layer exploration: Given the hardware support for Cross-layer dataflow exploration. We first study **pipeline granularity**. The K,C as the outermost loop leads to coarse-grain dataflows, whereas an X,Y as the outermost loop leads to a fine-grain dataflow. We then explored **partial loop orders,** ^{p}, which controls the amount of work in each stage of the Cross-layer pipeline. We proposed the Cross-layer dataflows with the most amount of partial loop orders $X^pY^pK^pC{-}3$ and $X^pY^pC^p{-}2$. Next we study optimal **pipeline depth**. We found that for CONV layers, a pipeline depth of two is optimal, whereas for LR-CONV layers, pipeline depth of three is optimal. Beyond three, the Cross-layer dataflows have diminishing returns.

In the end, we achieved efficient low latency Cross-layer dataflows whose latency increases linearly as the on-chip memory is reduced. Unlike, the prior state of the art where the latency drops significantly below a threshold on-chip memory. For example, compared to prior state-of-the-art [10, 36], X-Layer improves performance by 7.8× and 16.6× using only 256KB of on-chip SRAM whereas [10, 36] requires up to 2.4 MB of SRAM.

SPLIT deployment with AUTO-SPLIT

The second work explores SPLIT deployment. Our tool AUTO-SPLIT can be used to deploy all three approaches EDGE-ONLY, CLOUD-ONLY and SPLIT and automatically selects the optimal approach. AUTO-SPLIT considers operating constraints such as device memory and network speed, and application constraints such as acceptable error threshold by the user and selects the optimal split point of the DNN along with the allocation of bit-widths for weights and activations to apply post-training quantization of the edge DNN. The edge DNN is the partial split of the DNN which will be executed on the edge device. It allows the user to select the split based on a range of accuracy-latency trade-offs. This approach is most practical when the DNN does not fit on the edge device, and/or the network is slow, typically less than 4G. It is also useful when the input data is large, for example, when the application is using high-resolution images as input.

6.2 Directions for Future Work

This dissertation opens up many new research directions. Here we discuss several high-level future directions where this thesis can be extended to make significant contributions in improving the latency of end to end application for EDGE-ONLY and SPLIT approaches. We also highlight the necessary work which is required towards the productization of this thesis.

Cross-layer for complex NAS based DNN: Recent advancements in Automated Machine Learning (AutoML) aims to remove humans out of the loop in developing DNNs. This includes Neural Architecture Search (NAS) [22, 68, 69, 84, 98, 99, 127] that focus on automation of designing neural architectures. NAS based DNNs (see Figure 6.1) have become state of the art in various DNN tasks. They generate compact DNNs with irregular connections to different layers compared to conventional networks with a regular graph topology. They also heavily use LR-CONVs. These networks outperform manually designed architectures in terms of accuracy and use fewer multiply-accumulate (mac) computations and memory. In fact, the primary goal of NAS based DNNs is to reduce resources while achieving superior performance.

In the Figure, sep is a depth-separable layer i.e., DP $\rightarrow$ PT. X-Layer employs cross-layer dataflow to pipeline multiple layers simultaneously. We need to explore the search space of different order of schedules. The Figure, shows one example of the schedule to execute all layers of a DNN end to end. This schedule may not be optimal in minimizing on-chip memory and latency. How can we efficiently and automatically extract cross-layer pipelines from the DNN which improves hardware utilization. The scheduler should also aim to reduce on-chip memory usage and minimize data movement between on-chip and off-chip memory.

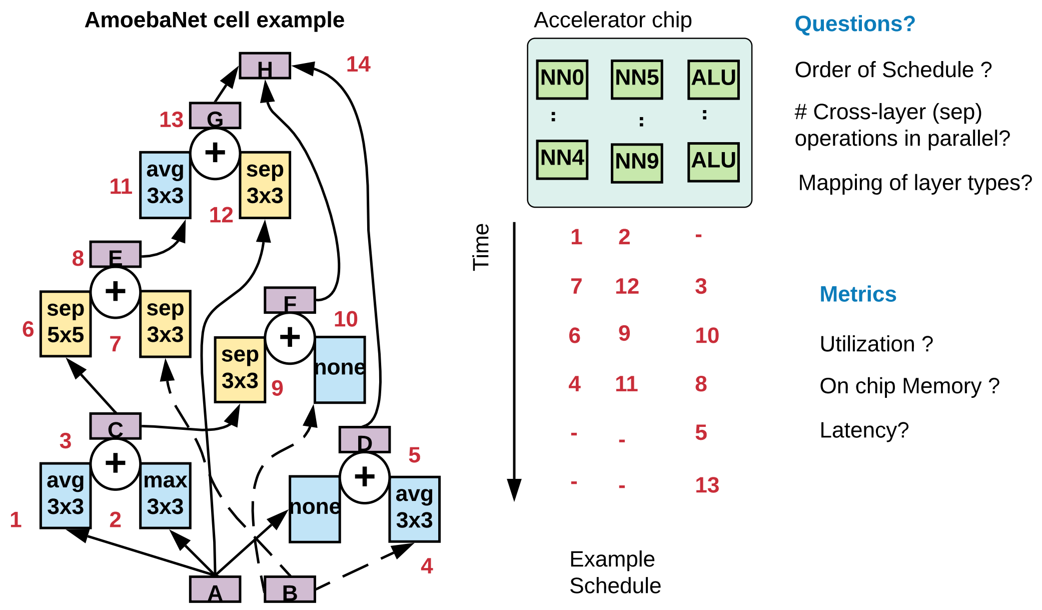


Figure 6.1: Scheduling DNN layers for NAS based DNN. sep= Depth-separable layer

Searching new DNN architectures with Cross-Layer dataflow: AI software community has already developed pipelines to explore compact DNNs using NAS [108, 114] methods. In these methods, the AI takes latency and energy feedback from the hardware to explore compact DNNs. This hardware is generally a CPU, since, the edge devices generally require copying the data over a

UART, SPI or other communication protocols which makes it difficult for NAS based methods to explore new architectures. Assuming, such pipelines have been built, one can then explore efficient DNNs assuming cross-layer dataflows.

Mixed precision latency feedback for AUTO-SPLIT: The mixed precision hardware has been proposed in academia [37, 91] but the software stack to read latency feedback is missing. The industry [2, 5] on the other hand, only executes DNN with uniform 8-bit. Thus, AUTO-SPLIT relies on edge device simulators to explore the latency vs accuracy trade-offs for SPLIT approach. New efficient algorithms can be developed, if we could read the latency from real hardware.

Dynamic graph partition for DNN splits suggested by AUTO-SPLIT: At present, AUTO-SPLIT needs to manually partition the DNN once, the split point has been suggested. The reason is that there is no support in PyTorch to dynamically partition the graph given the node names. Once, dynamic graph partitioning is supported one can leverage AUTO-SPLIT to support many applications where AUTO-SPLIT will automatically select between EDGE-ONLY, CLOUD-ONLY and SPLIT approaches on the fly based on operating and application constraints. Tensorflow 2.0 allows to partition the DNN using the node names. Update, [70] is an open source tool released at the time of AUTO-SPLIT submission built using tensorflow 2.0 which allows for on the fly graph partition. We chose PyTorch since many state of the art post training quantization algorithms were open sourced in PyTorch. We used distiller [3].

X-Layer dataflows for edge DNN generated by AUTO-SPLIT: The SPLIT solutions generated by AUTO-SPLIT require edge DNN to executed on the edge devices. The edge DNN can leverage cross-layer dataflows for scheduling on to the edge device to reduce the latency further. This will require the edge device to support cross-layer schedules for mixed precision hardware. However, this will require exploration of cross-layer dataflows in the presence of sub 8-bit activations and weights.

6.3 Reflections

In this section, I present what I have learnt over the course of my PhD program and present my views on how to approach dataflows and splitting techniques.

Lessons learnt from X-Layer

- ***1. There is no best dataflow for every DNN and for every hardware***. Every dataflow requires a minimum number of hardware resources and coarse-grain dataflows may not even work for many resource-constrained devices. DNNs come in all shapes and sizes and depending on the class of DNN, the workhorse layer characteristics can vary.
- ***2. A dataflow is closely tied to the hardware***. A dataflow designed for particular hardware may not even work for another hardware. This is especially true in the case of cross-layer

dataflows where multiple layers require i) fused outer dataflows, ii) execute multiple layers in a pipelined manner, iii) need to sync at different granularity based on the choice of outer dataflow.

Every dataflow paper needs to target a class of DNNs and have a target class of hardware. This realization is important for dataflow research. It is also hard to convince other researchers to agree on the rules of the playing field.

Types of DNNs: The DNNs can be categorized based on the AI fields, AI tasks, and deployment targets. Examples of the AI field include computer vision, reinforcement learning, and natural language processing. Examples of AI tasks in computer vision include image classification, object detection, semantic segmentation and Image generation. Sometimes the DNNs are also designed for a particular deployment target such as an ARM Cortex-M, a GPU in a mobile phone, a desktop, and a TPU in the cloud.

The DNN algorithms come in all shapes and sizes. For example, an LSTM or a transformer model used in **natural language processing** use many CONV, Tanh, Sigmoid, Multiply and Add functions. These DNNs can require many gigabytes of memory and computations. The DNN graph is also non-linear and requires efficient node scheduling and parallel batch processing. An **object detection algorithm**, have a base DNN similar to Image classification networks but the detection part such as Single Shot Detector, Faster-RCNN, and Yolo require parallel thread processing and utilize a lot of CPU computations which is currently not accelerated by the standard DNN hardware accelerators. The DNNs designed for **mobile targets** utilize a lot of composite layers based on LR-CONV layers which have poor utilization for existing DNN accelerators.

Bias of the dataflow papers Prior work [10, 19, 21, 36, 45, 57, 62–64, 79, 80, 87, 90] mostly looked at dataflow as one size fits all approach and heavily focused on CONV layers. It is also challenging to compare different dataflows designed for different target platforms and based on different target hardware. We summarize all the design choices that are made by a dataflow paper in Figure 4.8 in §4.3. For example, it is challenging to compare the dataflows of Tangram [36] with Fused-Layer [10]. Tangram is a coarse-grain dataflow that has a lower limit on the minimum number of resources required to work. Fused-Layer is a fine-grain dataflow that runs on vector hardware. The Fused-Layer hardware cannot execute Tangram cross-layer dataflow, and vice-versa. We list their detailed configuration for readers to understand the differences.

Tangram configuration: It is designed for Cloud, it uses 256 tiles of 16$\times$16 Eyeriss style ASIC NN engines. Overall it has the computation resource of 16384 PEs and 8 MB of on-chip memory. It uses a coarse-grain dataflow with a pipeline depth of two and uses $K^1CXY \rightarrow CKXY$. Tangram demonstrated AlexNet, VGG, GoogleNet, ResNet, MLP and LSTM.

Fused-Layer: It is an HLS based tool. It generates new hardware for every DNN, i.e., different mac, on-chip memory, and tile sizes. It targets FPGA and uses vector hardware. Fused-Layer implements a fine-grain dataflow X^1Y^1KC which can pipeline an arbitrary number of layers in theory.

It demonstrated two pipelines of depth 2 (Alexnet) and 5 (VGG). Alexnet and VGG does not contain LR-CONVs.

Both Tangram and Fused-Layer did not implement LR-CONVs. They cannot run DP layers. They will have to re-implement their cross-layer dataflows to support LR-CONVs.

Lessons learnt from AUTO-SPLIT

AUTO-SPLIT idea is a little ahead of its time. The framework required to support AUTO-SPLIT in real products does not exist yet. For example, it is challenging to find a hardware accelerator that can support mixed precision on the edge device. I am not aware of any method to split the DNNs automatically in PyTorch by providing a node name. The sub 8-bit data type support does not exist in the edge devices yet, thus, it creates an overhead of encoding and decoding the data into 8-bit data type during transmission of activations.

The AUTO-SPLIT framework was developed with Pytorch due to the large support of open source libraries for mixed-precision quantization. In practice, TensorFlow seems like a better option for graph splitting methods. It is a double-edged sword. On one side, it provides new opportunities to develop the necessary infrastructure and realize novel edge cloud collaborative products. On the other side, publishing one idea requires heavy engineering effort.

Chapter 7

Other Works of the Author

In addition to the works presented in the thesis, **I was also the main contributor on the research work NACHOS, which could not be included in the thesis**. Apart from NACHOS, I have also contributed to several other projects. A Brief overview of these works is described next.

NACHOS: Software-Driven Hardware-Assisted Memory Disambiguation for Accelerators

*Published in: 2018 IEEE International Symposium on High Performance Computer Architecture (HPCA). The authors were **Naveen Vedula**, Arrvindh Shriraman, Snehasish Kumar and Nick Sumner.*

Hardware accelerators have relied on the compiler to extract instruction parallelism but may waste significant energy in enforcing memory ordering and discovering memory parallelism. Accelerators tend to either serialize memory operations [103] or reuse power hungry load-store queues (LSQs) [39, 78]. Recent Works [40,48] use the compiler for scheduling but continue to rely on LSQs for memory disambiguation.

NACHOS is a hardware-assisted software-driven approach to memory disambiguation for accelerators. In NACHOS, the compiler classifies pairs of memory operations as NO alias (i.e., independent memory operations), MUST alias (i.e., ordering required), or MAY alias (i.e., compiler uncertain). We developed a compiler-only approach called NACHOS-SW that serializes memory operations both when the compiler is certain (MUST alias) and uncertain (MAY alias). Our study analyzes multiple stages of alias analysis on 135 acceleration regions extracted from SPEC2K, SPEC2k6, and PARSEC. NACHOS-SW is energy efficient, but serialization limits performance; 18%–100% slowdown compared to an optimized LSQ. We then proposed NACHOS a low-overhead, scalable, hardware comparator assist that dynamically verifies MAY alias and executes independent memory operations in parallel. NACHOS is a pay-as-you-go approach where the compiler filters out memory operations to save dynamic energy, and the hardware dynamically checks to find MLP. NACHOS achieves performance comparable to an optimized LSQ; in fact, it improved performance in 6 benchmarks(6%—70%) by reducing load-to-use latency for cache hits. NACHOS imposes no energy overhead in 15 out of 27 benchmarks i.e., compiler accurately determines all memory dependencies;

the average energy overhead is $\simeq$6% of total (accelerator and L1 cache); in comparison, an optimized LSQ consumes 27% of total energy. NACHOS is released as free and open-source software. **Github:** `https://github.com/sfu-arch/nachos`

Deepframe: A Profile-guided, Speculative Compiler for Spatial Hardware Accelerators

*Published in 28th International Conference on Parallel Architectures and Compilation Techniques (PACT 2019). The authors were Apala Guha, **Naveen Vedula**, Arrvindh Shriraman*

Tracing code paths to form extended basic blocks is useful in many areas, compiler optimizations [89], improving instruction cache behavior [12] and custom-hardware offloading [29]. Prior work has been plagued by small traces, limited either by the overheads of dynamic profiling, information available statically [11], or side-exit branches [72]. In this work, we rethink how to determine what code path sequences should we fuse to construct long traces for offloading to spatial accelerators while minimizing the occurrence of side-exits from limiting dynamic coverage.

We introduce a novel technique that recasts a program's dynamic execution history as a natural-language-processing problem, CBOW (Continuous Bag of Words). We then use a deep learning network to learn the relationships between different sequences of paths. During the compilation phase, the compiler uses a sequence miner to decide what paths are likely to occur. The learning network's classification is used to predict a Deepframe online, an extended basic block of multiple path sequences (each path itself is composed of multiple basic blocks). We offload the frequent frames to a spatial accelerator and demonstrate the following: i) Deepframe can construct up to 5$\times$ (max: 25$\times$) longer offload regions compared to prior approaches. ii) Surprisingly far–flung ILP and MLP can be mined from the frames statically (5.5$\times$ increase in ILP and 10.5$\times$ increase in MLP). iii) The frames offloaded to the spatial accelerator have minimal side exits (mis-speculation) and achieve sufficient dynamic coverage to improve overall application performance (up to 9$\times$ improvement). We will be releasing open-source an end-to-end compiler prototype based on LLVM.

Constraint-Aware Deep Neural Network Compression

*Published in Proceedings of the European Conference on Computer Vision (ECCV), 2018. The authors were Changan Chen, Frederick Tung, **Naveen Vedula**, Greg Mori.*

Deep neural network compression has the potential to bring modern resource-hungry deep networks to resource-limited devices. However, in many of the most compelling deployment scenarios of compressed deep networks, the operational constraints matter: for example, a pedestrian detection network on a self-driving car may have to satisfy a latency constraint for safe operation. We propose the first principled treatment of deep network compression under operational constraints. We formulate the compression learning problem from the perspective of constrained Bayesian optimization, and introduce a cooling (annealing) strategy to guide the network compression towards the target constraints. Experiments on ImageNet demonstrate the value of modelling constraints directly in network compression.

Fusion Design Tradeoffs in Coherent Cache Hierarchies for Accelerators

*Published in Proceedings of the 42nd Annual International Symposium on Computer Architecture, ISCA 2015. The authors were Snehasish Kumar, Arrvindh Shriraman, and **Naveen Vedula**.* Chip designers have shown increasing interest in integrating specialized fixed-function coprocessors into multicore designs to improve energy efficiency. Recent work in academia [39, 102] and industry [54] has sought to enable more fine-grain offloading at the granularity of functions and loops. The sequential program now needs to migrate across the chip utilizing the appropriate accelerator for each program region. As the execution migrates, it has become increasingly challenging to retain the temporal and spatial locality of the original program as well as manage the data sharing. We show that with the increasing energy cost of wires and caches relative to compute operations, it is imperative to optimize data movement to retain the energy benefits of accelerators. We develop FUSION, a lightweight coherent cache hierarchy for accelerators and study the tradeoffs compared to a scratchpad based architecture. We find that coherency, both between the accelerators and with the CPU, can help minimize data movement and save energy. FUSION leverages temporal coherence [94] to optimize data movement within the accelerator tile. The accelerator tile includes small per-accelerator L0 caches to minimize hit energy and a per-tile shared cache to improve localized-sharing between accelerators and minimize data exchanges with the host LLC. We find that overall FUSION improves performance by 4.3$\times$ compared to an oracle DMA that pushes data into the scratchpad. In workloads with inter-accelerator sharing, we save up to 10$\times$ the dynamic energy of the cache hierarchy by minimizing the host-accelerator data ping-ponging.

DASX: Hardware accelerator for software data structures

*Published in Proceedings of the 29th ACM on International Conference on Supercomputing (ICS 2015). The authors were Snehasish Kumar, **Naveen Vedula**, Arrvindh Shriraman, and Vijayalakshmi Srinivasan*. Recent research [39, 96, 103] has proposed compute accelerators to address the energy efficiency challenge. While these compute accelerators specialize and improve the compute efficiency, they have tended to rely on address-based load/store memory interfaces that closely resemble a traditional processor core. The address-based load/store interface is particularly challenging in data-centric applications that tend to access different software data structures. While accelerators optimize the compute section, the address-based interface leads to wasteful instructions and low memory level parallelism (MLP). We study the benefits of raising the abstraction of the memory interface to data structures. We propose DASX (Data Structure Accelerator), a specialized state machine for data fetch that enables compute accelerators to efficiently access data structure elements in iterative program regions. DASX enables the compute accelerators to employ data structure based memory operations and relieves the compute unit from having to generate addresses for each individual object. DASX exploits knowledge of the program's iteration to i) run ahead of the compute units and gather data objects for the compute unit (i.e., compute unit memory operations do not encounter cache misses) and ii) throttle the fetch rate, adaptively tile the dataset based on the locality characteristics and guarantee cache residency. We demonstrate accelerators for three types of data structures, Vector, Key-Value (Hash) maps, and BTrees. We demonstrate the benefits of DASX on data-centric applications which have varied compute kernels but access few regular data structures. DASX achieves higher energy efficiency by eliminating data structure instructions and enabling energy efficient compute accelerators to efficiently access the data elements. We demonstrate that DASX can achieve 4.4$\times$ the performance of a multicore system by discovering more parallelism from the data structure.

Bibliography

[1] Nvdla: nvidia deep learning accelerator. NVIDIA Deep Learning Accelerator. http://nvdla.org/.

[2] Nvidia jetson. https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/.

[3] Preparing a model for quantization. Neural Network Distiller. https://intellabs.github.io/distiller/prepare_model_quant.html.

[4] Sota imagenet. github. https://paperswithcode.com/sota/image-classification-on-imagenet.

[5] Efficientnet-edgetpu: Creating accelerator-optimized neural networks with automl. https://ai.googleblog.com/2019/08/efficientnet-edgetpu-creating.html, 2019.

[6] Computer vision- wikipedia. https://en.wikipedia.org/wiki/Deep_learning, 2021.

[7] Deep learning vs machine learning - what's the difference. https://levity.ai/blog/difference-machine-learning-deep-learning, 2021.

[8] Quince industrial internet of things market insights. https://www.globenewswire.com/news-release/2020/08/29/2085754/0/en/Industrial-Internet-of-Things-IoT-Market-To-Grow-At-A-CAGR-of-21-3-During-2020-To-2028-Quince-Market-Insights.html#:~:text=29%2C%202020%20(GLOBE%20NEWSWIRE),period%20from%202020%20to%202028., 2021.

[9] ModelArts: Deploying a Model as an Edge Service. 2021.

[10] Manoj Alwani, Han Chen, Michael Ferdman, and Peter Milder. Fused-layer CNN accelerators. In Proc. of the 49th MICRO, pages 1–12, 2016.

[11] Matthew Arnold, Stephen Fink, Vivek Sarkar, and Peter F. Sweeney. A comparative study of static and profile-based heuristics for inlining. In Proc. of the ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization, 2000.

[12] Thomas Ball and James R Larus. Branch prediction for free. In PROC of the 1993 PLDI, 1993.

[13] R. Banner, Y. Nahshan, E. Hoffer, and D. Soudry. Aciq: Analytical clipping for integer quantization of neural networks. ArXiv, abs/1810.05723, 2018.

[14] R. Banner, Y. Nahshan, and D. Soudry. Post training 4-bit quantization of convolutional networks for rapid-deployment. In NeurIPS, 2019.

[15] Simone Bianco, Rémi Cadène, Luigi Celona, and Paolo Napoletano. Benchmark analysis of representative deep neural network architectures. IEEE Access, 6:64270–64277, 2018.

[16] T. B. Brown et al. Language models are few-shot learners. arXiv preprint arXiv:2005.14165, 2020.

[17] Y. Cai, Zh. Yao, Zh. Dong, et al. Zeroq: A novel zero shot quantization framework. 2020 IEEE/CVF CVPR, pages 13166–13175, 2020.

[18] Software-Defined Camera. 2021.

[19] Yu-Hsin Chen, Tushar Krishna, Joel S Emer, and Vivienne Sze. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. IEEE Journal of Solid-State Circuits, 52(1):127–138, 2016.

[20] Yu-Hsin Chen, Tien-Ju Yang, Joel Emer, and Vivienne Sze. Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices. arXiv preprint arXiv:1807.07928, 2018.

[21] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, and Olivier Temam. Dadiannao: A machine-learning supercomputer. 2014 47th Annual IEEE/ACM International Symposium on Microarchitecture, pages 609–622, 2014.

[22] Hsin-Pai Cheng, Tunhou Zhang, Yukun Yang, Feng Yan, Shiyu Li, Harris Teague, Hai Li, and Yiran Chen. Swiftnet: Using graph propagation as meta-knowledge to search highly representative neural architectures. ArXiv, abs/1906.08305, 2019.

[23] Y. Cheng, D. Wang, P. Zhou, and T. Zhang. A survey of model compression and acceleration for deep neural networks. arXiv:1710.09282, 2017.

[24] J. Choi, Z. Wang, S. Venkataramani, et al. Pact: Parameterized clipping activation for quantized neural networks. ArXiv, abs/1805.06085, 2018.

[25] Jungwook Choi, Zhuo Wang, Swagath Venkataramani, Pierce I-Jen Chuang, Vijayalakshmi Srinivasan, and Kailash Gopalakrishnan. Pact: Parameterized clipping activation for quantized neural networks. arXiv preprint arXiv:1805.06085, 2018.

[26] Yujeong Choi and Minsoo Rhu. Prema: A predictive multi-task scheduling algorithm for preemptible neural processing units. In HPCA, pages 220–233. IEEE, 2020.

[27] François Chollet. Xception: Deep learning with depthwise separable convolutions. In Proceedings of the IEEE conference on computer vision and pattern recognition, pages 1251–1258, 2017.

[28] Y. Choukroun, E. Kravchik, and P. Kisilev. Low-bit quantization of neural networks for efficient inference. 2019 IEEE/CVF ICCVW, pages 3009–3018, 2019.

[29] Nathan Clark, Manjunath Kudlur, Hyunchul Park, Scott Mahlke, and Krisztian Flautner. Application-Specific Processing on a General-Purpose Core via Transparent Instruction Set Customization. In PROC of the 37th MICRO, 2004.

[30] Xiaoliang Dai, Peizhao Zhang, Bichen Wu, Hongxu Yin, Fei Sun, Yanghan Wang, Marat Dukhan, Yunqing Hu, Yiming Wu, Yangqing Jia, Peter Vajda, Matthew Uyttendaele, and Niraj K. Jha. Chamnet: Towards efficient network design through platform-aware model adaptation. 2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), pages 11390–11399, 2019.

[31] ModelArts Pro Model Deployment. 2021.

[32] Zh. Dong, Zh. Yao, A. Gholami, et al. Hawq: Hessian aware quantization of neural networks with mixed-precision. ICCV, pages 293–302, 2019.

[33] MindX Edge. 2021.

[34] S. Esser, J. McKinstry, et al. Learned step size quantization. arXiv preprint arXiv:1902.08153, 2019.

[35] D. Gao, X. He, Z. Zhou, et al. Rethinking pruning for accelerating deep inference at the edge. In 26th ACM SIGKDD, pages 155–164, 2020.

[36] Mingyu Gao, Xuan Yang, Jing Pu, Mark Horowitz, and Christos Kozyrakis. Tangram: Optimized coarse-grained dataflow for scalable nn accelerators. In Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, pages 807–820. ACM, 2019.

[37] Angelo Garofalo, Manuele Rusci, Francesco Conti, Davide Rossi, and Luca Benini. Pulp-nn: accelerating quantized neural networks on parallel ultra-low-power risc-v processors. Philosophical Transactions of the Royal Society A, 378(2164):20190155, 2020.

[38] H. Genc, Ameer Haj-Ali, Vighnesh Iyer, Alon Amid, Howard Mao, J. Wright, C. Schmidt, J. Zhao, Albert Ou, Max Banister, Y. Shao, B. Nikolić, I. Stoica, and K. Asanovic. Gemmini: An agile systolic array generator enabling systematic evaluations of deep-learning architectures. ArXiv, abs/1911.09925, 2019.

[39] Venkatraman Govindaraju, Chen-Han Ho, and Karthikeyan Sankaralingam. Dynamically specialized datapaths for energy efficient computing. In HPCA, 2011.

[40] Tae Jun Ham, Juan L Aragn, and Margaret Martonosi. DeSC: decoupled supply-compute communication management for heterogeneous architectures. In MICRO, 2015.

[41] S. Han, H. Mao, and W. Dally. Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding. CoRR, abs/1510.00149, 2016.

[42] Song Han, Huizi Mao, and William J. Dally. Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding. CoRR, abs/1510.00149, 2016.

[43] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In Proceedings of the IEEE conference on computer vision and pattern recognition, pages 770–778, 2016.

[44] Y. He, X. Zhang, and J. Sun. Channel pruning for accelerating very deep neural networks. ICCV, pages 1398–1406, 2017.

[45] Kartik Hegde, Jiyong Yu, R. Agrawal, Mengjia Yan, Michael Pellauer, and Christopher W. Fletcher. Ucnn: Exploiting computational reuse in deep neural networks via weight repetition. 2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA), pages 674–687, 2018.

[46] D. Hendrycks and Th. Dietterich. Benchmarking neural network robustness to common corruptions and perturbations. arXiv preprint arXiv:1903.12261, 2019.

[47] G. E. Hinton, O. Vinyals, and J. Dean. Distilling the knowledge in a neural network. ArXiv, abs/1503.02531, 2015.

[48] Chen-Han Ho, Sung Jin Kim, and Karthikeyan Sankaralingam. Efficient execution of memory access phases using dataflow specialization. In ISCA, 2015.

[49] Ch. Hu, W. Bao, D. Wang, and F. Liu. Dynamic adaptive dnn surgery for inference acceleration on the edge. IEEE INFOCOM, pages 1423–1431, 2019.

[50] Jie Hu, Li Shen, and Gang Sun. Squeeze-and-excitation networks. 2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition, pages 7132–7141, 2017.

[51] Gao Huang, Danlu Chen, T. Li, et al. Multi-scale dense networks for resource efficient image classification. In ICLR, 2018.

[52] Forrest Iandola, Matt Moskewicz, Sergey Karayev, Ross Girshick, Trevor Darrell, and Kurt Keutzer. Densenet: Implementing efficient convnet descriptor pyramids. arXiv preprint arXiv:1404.1869, 2014.

[53] Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and< 0.5 mb model size. arXiv preprint arXiv:1602.07360, 2016.

[54] Intel. Xeon chip with integrated fpga. 2014.

[55] Intel. Intel nervana. 2020. nervana's early exit inference. 2020.

[56] B. Jacob, S. Kligys, Bo Chen, et al. Quantization and training of neural networks for efficient integer-arithmetic-only inference. CVPR, 2018.

[57] Norman P. Jouppi, Cliff Young, Nishant Patil, David A. Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh K Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, M. T. Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, Richard C. 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 Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan 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. 2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA), pages 1–12, 2017.

[58] Y. Kang, J. Hauswald, C. Gao, et al. Neurosurgeon: Collaborative intelligence between the cloud and mobile edge. ASPLOS, 2017.

[59] R. Krishnamoorthi. Quantizing deep convolutional networks for efficient inference: A whitepaper. ArXiv, abs/1806.08342, 2018.

[60] A. Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. Communications of the ACM, 60:84 – 90, 2012.

[61] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In Advances in neural information processing systems, pages 1097–1105, 2012.

[62] Hyoukjun Kwon, Prasanth Chatarasi, Michael Pellauer, Angshuman Parashar, Vivek Sarkar, and Tushar Krishna. Understanding reuse, performance, and hardware cost of dnn dataflow: A data-centric approach. In Proc. of the 52nd MICRO, pages 754–768, 2019.

[63] Hyoukjun Kwon, Ananda Samajdar, and Tushar Krishna. Maeri: Enabling flexible dataflow mapping over dnn accelerators via reconfigurable interconnects. In ASPLOS '18, 2018.

[64] Kiseok Kwon, Alon Amid, Amir Gholami, Bichen Wu, Krste Asanovic, and Kurt Keutzer. Invited: Co-design of deep neural nets and neural net accelerators for embedded vision applications. 2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC), pages 1–6, 2018.

[65] Stefanos Laskaridis, Stylianos I. Venieris, et al. Spinn: synergistic progressive inference of neural networks over device and cloud. MobiCom, 2020.

[66] J. H. Lee, S. Ha, S. Choi, et al. Quantization for rapid deployment of deep neural networks. ArXiv, abs/1810.05488, 2018.

[67] En Li, Liekang Zeng, , et al. Edge ai: On-demand accelerating deep neural network inference via edge computing. TWC, 2020.

[68] Chenxi Liu, Barret Zoph, Jonathon Shlens, Wei Hua, Li-Jia Li, Li Fei-Fei, Alan L. Yuille, Jonathan Huang, and Kevin Murphy. Progressive neural architecture search. In ECCV, 2018.

[69] Hanxiao Liu, Karen Simonyan, and Yiming Yang. Darts: Differentiable architecture search. ArXiv, abs/1806.09055, 2019.

[70] Luke Lockhart, P. Harvey, Pierre Imai, P. Willis, and B. Varghese. Scission: Performance-driven and context-aware cloud-edge distribution of deep neural networks. 2020 IEEE/ACM 13th International Conference on Utility and Cloud Computing (UCC), pages 257–268, 2020.

[71] Ningning Ma, Xiangyu Zhang, Hai-Tao Zheng, and Jian Sun. Shufflenet v2: Practical guidelines for efficient cnn architecture design. In Proceedings of the European Conference on Computer Vision (ECCV), pages 116–131, 2018.

[72] Wen mei W. Hwu, Scott A. Mahlke, William Y. Chen, Pohua P. Chang, Nancy J. Warter, Roger A. Bringmann, Roland G. Ouellette, Richard E. Hank, Tokuzo Kiyohara, Grant E. Haab, John G. Holm, and Daniel M. Lavery. The superblock: An effective technique for vliw and superscalar compilation. THE JOURNAL OF SUPERCOMPUTING, 7:229–248, 1993.

[73] C. Michaelis, B. Mitzkus, R. Geirhos, et al. Benchmarking robustness in object detection: Autonomous driving when winter is coming. arXiv preprint arXiv:1907.07484, 2019.

[74] A. Mishra and D. Marr. Apprentice: Using knowledge distillation techniques to improve low-precision network accuracy. arXiv:1711.05852, 2018.

[75] P. Murugan, S. Kannan, and S. Baskar. Nsga-ii algorithm for multi-objective generation expansion planning problem. Electric Power Systems Research, 79:622–628, 2009.

[76] M. Nagel, M. v. Baalen, T. Blankevoort, and M. Welling. Data-free quantization through weight equalization and bias correction. ICCV, pages 1325–1334, 2019.

[77] Y. Nahshan, B. Chmiel, Ch. Baskin, et al. Loss aware post-training quantization. ArXiv, abs/1911.07190, 2019.

[78] Tony Nowatzki, Vinay Gangadhar, and Karthikeyan Sankaralingam. Exploring the potential of heterogeneous von neumann/dataflow execution models. ISCA, 2015.

[79] Angshuman Parashar, Priyanka Raina, Yakun Sophia Shao, Yu-Hsin Chen, Victor A. Ying, Anurag Mukkara, Rangharajan Venkatesan, Brucek Khailany, Stephen W. Keckler, and Joel S. Emer. Timeloop: A systematic approach to dnn accelerator evaluation. 2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), pages 304–315, 2019.

[80] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel S Emer, Stephen W Keckler, and William J Dally. SCNN: An Accelerator for Compressed-sparse Convolutional Neural Networks. In Proc. of the 44th ISCA, pages 27–40, 2017.

[81] Eunhyeok Park, Dongyoung Kim, Sungjoo Yoo, and Peter Vajda. Precision highway for ultra low-precision quantization. arXiv preprint arXiv:1812.09818, 2018.

[82] HiLens Platform and Edge Device. 2021.

[83] A. Polino, R. Pascanu, and Dan Alistarh. Model compression via distillation and quantization. ArXiv, abs/1802.05668, 2018.

[84] Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V. Le. Regularized evolution for image classifier architecture search. In AAAI, 2018.

[85] M. Rusci, A. Capotondi, and L. Benini. Memory-driven mixed low precision quantization for enabling deep network inference on microcontrollers. ArXiv, abs/1905.13082, 2020.

[86] A. Samajdar, Y. Zhu, P. Whatmough, et al. Scale-sim: Systolic cnn accelerator. ArXiv, abs/1811.02883, 2018.

[87] Ananda Samajdar, Yuhao Zhu, Paul N. Whatmough, Matthew Mattina, and Tushar Krishna. Scale-sim: Systolic CNN accelerator. CoRR, abs/1811.02883, 2018.

[88] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pages 4510–4520, 2018.

[89] Steven S Lumetta Sanjay J Patel. rePLay: A Hardware Framework for Dynamic Program Optimization. IEEE Transactions on Computers archive. Volume 50, 1999.

[90] Yakun Sophia Shao, Jason Clemons, Rangharajan Venkatesan, Brian Zimmer, Matthew Fojtik, Nan Jiang, Ben Keller, Alicia Klinefelter, Nathaniel Pinckney, Priyanka Raina, et al. Simba: Scaling deep-learning inference with multi-chip-module-based architecture. In Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, pages 14–27. ACM, 2019.

[91] Hardik Sharma, Jongse Park, Naveen Suda, Liangzhen Lai, Benson Chau, Joon Kyung Kim, Vikas Chandra, and Hadi Esmaeilzadeh. Bit fusion: Bit-level dynamically composable architecture for accelerating deep neural network. 2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA), pages 764–775, 2017.

[92] Y. Shoham and A. Gersho. Efficient bit allocation for an arbitrary set of quantizers. IEEE Trans. Acoustics, Speech, and Signal Processing, 36, 1988.

[93] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. arXiv preprint arXiv:1409.1556, 2014.

[94] Inderpreet Singh, Arrvindh Shriraman, Wilson WL Fung, Mike O'Connor, and Tor M Aamodt. Cache coherence for gpu architectures. In HPCA, pages 578–590, 2013.

[95] 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. IEEE, 2017.

[96] Shreesha Srinath, Berkin Ilbeyi, Mingxing Tan, Gai Liu, Zhiru Zhang, and Christopher Batten. Architectural Specialization for Inter-Iteration Loop Dependence Patterns. In MICRO, 2014.

[97] Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, and Alexander A Alemi. Inception-v4, inception-resnet and the impact of residual connections on learning. In Thirty-First AAAI Conference on Artificial Intelligence, 2017.

[98] Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, Mark Sandler, Andrew Howard, and Quoc V Le. Mnasnet: Platform-aware neural architecture search for mobile. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pages 2820–2828, 2019.

[99] Mingxing Tan and Quoc V Le. Efficientnet: Rethinking model scaling for convolutional neural networks. arXiv preprint arXiv:1905.11946, 2019.

[100] S. Teerapittayanon, B. McDanel, and H.-T. Kung. Branchynet: Fast inference via early exiting from deep neural networks. In ICPR, pages 2464–2469. IEEE, 2016.

[101] Tenstorrent. Tenstorrent's grayskull ai chip. 2020.

[102] Ganesh Venkatesh, J. Sampson, N. Goulding, Sravanthi Kota Venkata, M. Taylor, and S. Swanson. Qscores: Trading dark silicon for scalable energy efficiency with quasi-specific cores. 2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), pages 163–174, 2011.

[103] Ganesh Venkatesh, Jack Sampson, Nathan Goulding, Saturnino Garcia, Vladyslav Bryksin, Jose Lugo-Martinez, Steven Swanson, and Michael Bedford Taylor. Conservation cores: reducing the energy of mature computations. In ASPLOS, 2010.

[104] J. Wang, J. Zhang, W. Bao, et al. Not just privacy: Improving performance of private deep learning in mobile cloud. In 24th ACM SIGKDD, pages 2407–2416, 2018.

[105] K. Wang, Zh. Liu, Y. Lin, J. Lin, and Song H. Haq: Hardware-aware automated quantization with mixed precision. In CVPR, pages 8612–8620, 2019.

[106] Kuan Wang, Zhijian Liu, Yujun Lin, Ji Lin, and Song Han. Haq: Hardware-aware automated quantization with mixed precision. In Proceedings of the IEEE conference on computer vision and pattern recognition, pages 8612–8620, 2019.

[107] Paul Whatmough, Chuteng Zhou, Patrick Hansen, Shreyas Kolala Venkataramanaiah, Jae-Sun Seo, and Matthew Mattina. Fixynn: Efficient hardware for mobile computer vision via transfer learning. SysML, 2019.

[108] B. Wu, Y. Wang, P. Zhang, et al. Mixed precision quantization of convnets via differentiable neural architecture search. ArXiv, abs/1812.00090, 2018.

[109] Bichen Wu, Xiaoliang Dai, Peizhao Zhang, Yanghan Wang, Fei Sun, Yiming Wu, Yuandong Tian, Peter Vajda, Yangqing Jia, and Kurt Keutzer. Fbnet: Hardware-aware efficient convnet design via differentiable neural architecture search. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pages 10734–10742, 2019.

[110] 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. IEEE, 2019.

[111] Y. N. Wu, J. S. Emer, and V. Sze. Accelergy: An architecture-level energy estimation methodology for accelerator designs. In ICCAD, 2019.

[112] Saining Xie, A. Kirillov, Ross B. Girshick, and Kaiming He. Exploring randomly wired neural networks for image recognition. 2019 IEEE/CVF International Conference on Computer Vision (ICCV), pages 1284–1293, 2019.

[113] Haichuan Yang et al. Energy-constrained compression for deep neural networks via weighted sparse projection and layer input masking. arXiv preprint arXiv:1806.04321, 2018.

[114] Tien-Ju Yang, Andrew Howard, Bo Chen, Xiao Zhang, Alec Go, Mark Sandler, Vivienne Sze, and Hartwig Adam. Netadapt: Platform-aware neural network adaptation for mobile applications. In Proceedings of the European Conference on Computer Vision (ECCV), pages 285–300, 2018.

[115] Xuan Yang, Mingyu Gao, Qiaoyi Liu, Jeff Setter, Jing Pu, Ankita Nayak, Steven Bell, Kaidi Cao, Heonjae Ha, Priyanka Raina, Christos Kozyrakis, and Mark Horowitz. Interstellar: Using halide's scheduling language to analyze dnn accelerators. In Proc, of the 25th ASPLOS, 2020.

[116] Dongqing Zhang, Jiaolong Yang, Dongqiangzi Ye, and Gang Hua. Lq-nets: Learned quantization for highly accurate and compact deep neural networks. In Proceedings of the European conference on computer vision (ECCV), pages 365–382, 2018.

[117] Kaipeng Zhang, Zhanpeng Zhang, Zhifeng Li, and Yu Qiao. Joint face detection and alignment using multitask cascaded convolutional networks. IEEE Signal Processing Letters, 23(10):1499–1503, 2016.

[118] Linfeng Zhang, Zhanhong Tan, et al. Scan: A scalable neural networks framework towards compact and efficient models. In NeurIPS, 2019.

[119] Shigeng Zhang, Yinggang Li, et al. Towards real-time cooperative deep inference over the cloud and edge end devices. IMWUT, 2020.

[120] R. Zhao, Y. Hu, J. Dotzel, C. D. Sa, and Z. Zhang. Improving neural network quantization without retraining using outlier channel splitting. In ICML 2019, 2019.

[121] W. Zhe, J. Lin, et al. Optimizing the bit allocation for compression of weights and activations of deep neural networks. In ICIP, pages 3826–3830. IEEE, 2019.

[122] W. Zhe, J. Lin, M. M. Sabry Aly, S. Young, V. Chandrasekhar, and B. Girod. Towards effective 2-bit quantization: Pareto-optimal bit allocation for deep cnns compression. 2020.

[123] H.Y. Zhou, B.B. Gao, and J. Wu. Adaptive feeding: Achieving fast and accurate detections by adaptively combining object detectors. In ICCV, 2017.

[124] Sh. Zhou, Z. Ni, et al. Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients. ArXiv, abs/1606.06160, 2016.

[125] N. Zmora, G. Jacob, et al. Neural network distiller: A python package for dnn compression research. October 2019.

[126] Barret Zoph and Quoc V. Le. Neural architecture search with reinforcement learning. ArXiv, abs/1611.01578, 2017.

[127] Barret Zoph, V. Vasudevan, Jonathon Shlens, and Quoc V. Le. Learning transferable architectures for scalable image recognition. 2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition, pages 8697–8710, 2018.

Appendix A

AUTO-SPLIT Engineering details

Guidelines proposed for users of our system: a) For models with float size of $<$ 50MB, EDGE-ONLY is likely the optimal solution (on typical edge chips). b) Compared to the input image, the activation volumes are generally large for initial layers, but small for deep layers. When a large number of initial layers receive high activation volumes i.e., their volume is greater than input image volume, then CLOUD-ONLY is likely the optimal solution. c) For deep but thin networks or when the input image has high resolution (say >= $416\times416\times3$), the SPLIT solution is likely optimal.

Activation transmission protocol: In practice, we found that python's xmlRPC protocol was orders of magnitude slower compared to using socket programming. The reason is xmlRPC cannot transfer binary data over a network, so activations are encoded and decoded into ASCII characters. This adds extra overhead. Thus, we used socket programming (in C++) for data transmission. Table A.1 shows RPC vs socket transmission for a Yolov3 based face detection model. On a single server (31 Gbps), Auto-split takes 1.13s on RPC and 0.27 ms with socket programming. Note that in both xmlRPC or socket programming, quantized activations (say 4-bits) are still stored as "int8" data type (by padding with zeros). Thus, it requires some pre-processing before transmission for existing edge/cloud devices. Table A.2 shows the API of the transmission protocol.

Table A.1: Comparison of RPC vs Socket programming

Benchmark	Img/Act shape	KB	RPC/Socket
Cloud-Only	432,768,3	972	3566
Auto-Split	36,64,256	288	3981

Table A.2: API for activation transmission

Data Type	Parameters
Bytes (int8)	Transmitted Activation
Float32	Scale
Float32	Zero-point
List(int32)	Input image shape
Int8	#Bits used for activations

Table A.3: Packing & unpacking overhead of 4-bit activations

Benchmark	**Act shape**	**KB**	**Height-Width**	**Channel**
Auto-Split	36,64,256	288	1.45 (s)	0.01 (s)

Handling sub 8-bit activations for transmission: AUTO-SPLIT may provide solutions with activation layers of lower than 8-bits, e.g. 4-bits. To minimize the transmission cost one needs to: 1) either implement “int4" data type (or lower) for both edge and cloud devices, or 2) pack two 4-bit (or lower) activations into “int8" data type on the edge device, transmit over the network, and unpack into “float" data type on the cloud device. For existing devices that do not support $<8-bit$ data type, we also implemented an API to pack/unpack to 8-bit data types. We tried to pack/unpack activations along i) “Height-Width" and ii) “Channel" dimensions. If the edge device supports python, then it is more efficient to use NumPy libraries for packing multiple channels of activation ($<8-bit$) to a single 8-bit channel. Table A.3 shows details of Height-Width (HW) vs Channel (C) packing & unpacking overhead of 4-bit activations before transmission (Intel(R) Xeon(R) CPU E5-2690 v4 @ 2.60GHz). For edge devices with a C++ interface, SIMD units should be utilized to speed up the packing of activations.