# HERO: A Hybrid General Matrix Multiplication and Direct Convolution Accelerator

A Thesis Presented
by

**Aly Sultan**

to

**The Department of Electrical and Computer Engineering**

in partial fulfillment of the requirements
for the degree of

**Master of Science**

in

**Electrical and Computer Engineering**

**Northeastern University**
**Boston, Massachusetts**

MM YYYY

NORTHEASTERN UNIVERSITY

**Graduate School of Engineering**

**Thesis Title:** HERO: A Hybrid General Matrix Multiplication and Direct Convolution Accelerator

**Author:** Aly Sultan.

**Department:** Electrical and Computer Engineering.

Approved for Thesis Requirements of the Master of Science Degree

| | |
|---|---|
| Thesis Advisor: Professor Gunar Schirner | Date |
| Thesis Reader: AA BB | Date |
| Thesis Reader: CC DD | Date |
| Department Chair: ZZ EE | Date |

Director of the Graduate School:

| | |
|---|---|
| Dean: EE FF | Date |

*To my family.*

# Contents

**List of Figures** vi

**List of Tables** ix

**List of Acronyms** x

**Acknowledgments** xi

**Abstract of the Thesis** xii

**1 Introduction** 1
- 1.1 AI and Edge Computing . . . 1
- 1.2 Convolution accelerators . . . 2
- 1.3 Problem Definition . . . 3
- 1.4 Solution Overview . . . 4
- 1.5 Thesis Structure . . . 7

**2 Background** 8
- 2.1 Convolutional Neural Networks And The Convolution Operation . . . 9
- 2.2 Reinterpreting Convolutions As GEMM . . . 12
- 2.3 Implementing Convolutions In Hardware . . . 16
  - 2.3.1 The Dataflow Taxonomy . . . 16
  - 2.3.2 The Hardware Implementation taxonomy . . . 18
- 2.4 Related work . . . 19
  - 2.4.1 Eyeriss V1 and V2 . . . 20
  - 2.4.2 Tensor Processing Unit . . . 21
  - 2.4.3 Maeri . . . 22

**3 Data-Aware Accelerator Design** 23
- 3.1 Pruning the dataflow design space with CIGAR . . . 24
  - 3.1.1 CIGAR: The ConvolutIon statIstics GAtherer . . . 24
  - 3.1.2 Applying CIGAR to prune the dataflow design space . . . 28
- 3.2 Enabling GEMM mode through layer equivalence . . . 34
  - 3.2.1 Functional equivalence between GEMM and (1, 1) Convolutions . . . 35

3.2.2 Layer Equivalence . . . 37
3.2.3 Layer Dimensionality . . . 39
3.2.4 Accelerator Spatial Axis mapping . . . 39
3.2.5 Overhead of lowering and lifting . . . 40
3.3 Exploring The Hardware Implementation Design Space . . . 41
3.3.1 Temporal Reuse Analysis . . . 41
3.3.2 Spatial Reuse Analysis . . . 49
3.3.3 Simplifying the memory hierarchy . . . 52
3.4 HERO: A Hybrid GEMM and Direct Conv. Accelerator . . . 55

**4 Architecture Dimensioning** **58**
4.1 Exploring what remains of the dataflow design space with TEMPO . . . 58
4.2 TEMPO Algorithm . . . 60
4.3 TEMPO analytical model . . . 61
4.3.1 Utilization . . . 62
4.3.2 Latency . . . 64
4.3.3 Memory access counts . . . 65
4.4 TEMPO results . . . 65
4.5 Memory Hierarchy Sizing . . . 66

**5 On-Chip Data Orchestration** **71**
5.1 Structure of a SAM . . . 72
5.2 Address generator controller . . . 73
5.3 Descriptor based programs . . . 73
5.3.1 Descriptor Types . . . 74
5.3.2 Creating timed memory operations with descriptor programs . . . 75

**6 Network Compilation** **78**
6.1 Phase one: Layer Transformation . . . 78
6.1.1 Layer Decomposition . . . 78
6.2 Phase Two: Descriptor Program Generation . . . 81
6.2.1 1x1 convolution programs . . . 82
6.2.2 3x3 convolution programs . . . 85

**7 HERO Architecture Simulation** **87**
7.1 Simulation Environment . . . 87
7.1.1 Python Frontend . . . 87
7.1.2 SystemC backend . . . 88
7.2 Experimental Results . . . 90
7.2.1 Utilization . . . 91
7.2.2 Latency and speedup over CPU Baseline . . . 93
7.2.3 DRAM Bandwidth . . . 95
7.2.4 Energy . . . 97
7.2.5 Area . . . 98
7.2.6 Per network results . . . 98

**8 Conclusion** **105**

**Bibliography** **107**

# List of Figures


1.1 Visual illustration of the solution overview . . . . . 4

2.1 Example of the different layers in a convolution neural networks . . . . . 9
2.2 Convolution Operation Illustrated . . . . . 11
2.3 Im2Col (Expensive lowering/ lifting) Illustrated . . . . . 13
2.4 Balanced Lowering/Lifting Illustrated . . . . . 15
2.5 Illustration of different dataflow implementations (adapted from[24]) arising from (a) Unrolling F and C loops (b) unrolling Ky and Y loops (c) unrolling C loops . . 18
2.6 Hardware Implementation Taxonomy adapted from [13] . . . . . 19
2.7 Illustration of Eyeriss V1 and Eyeriss v2's architecture from [7] . . . . . 20
2.8 Block diagram of the TPU architecture from [11] . . . . . 21
2.9 An overview of Maeri's architecture [14] . . . . . 22

3.1 ConvoluTional neural network Statistics GAtheRer (CIGAR) attachment of forward hooks to all model convolution layers . . . . . 25
3.2 CIGAR extraction of convolution layers . . . . . 25
3.3 Illustration of CIGAR's library diversity based on (a) Model sizes and number of MACS (b) Model layer types . . . . . 27
3.4 Exploration of data element reuse behavior in convolution layers of models from the TIMM library, a) shows overall reuse behavior as a boxplot b) shows reuse behavior trends within models with multiple convolution layers . . . . . 28
3.5 Percentage of total layers in the TIMM library's networks that have a stride size (k, k) 30
3.6 (a) Percentage of total layers in the TIMM library's networks that have a kernel size (k, k) (b) The percentage of networks in TIMM that have at least one kernel of size (k, k) . . . . . 31
3.7 (a) Kernel size vs number of layer MACs (b) Kernel size vs number of layer MACs adjusted by kernel frequency . . . . . 32
3.8 An illustration of weight tiling by loop unroll factors . . . . . 33
3.9 General Matrix Multiplication (GEMM) and (1, 1) Convolution Equivalence . . . . 36
3.10 Illustration of approach Conv2Gemm2Conv approach . . . . . 38
3.11 Initial hardware template incorporating buffers IFmap and OFmap temporal reuse . 45
3.12 IFmap Reuse Behavior w.r.t individual feature map channels . . . . . 46
3.13 Hardware template incorporating a reuse chain for reuse within an IFmap channel 49

3.14 Illustration of different partial sum reduction styles assuming kernel size is (3, 3) (a) Tree Reduction (b) Systolic array reduction . . . . . 51
3.15 Reinterpretation of IFmap memory hierarchy outputs as a stream function . . . . . 52
3.16 Using a systolic reduce and forward to calculate OFmaps . . . . . 54
3.17 Hardware Implementation Taxonomy adapted from [13] . . . . . 56
3.18 Hardware Implementation Taxonomy adapted from [13] . . . . . 57

4.1 GEMM and (1, 1) Convolution Equivalence . . . . . 59
4.2 GEMM and (1, 1) asd Equivalence . . . . . 62
4.3 Utilization results for different optimal configurations found using TEMPO . . . . 66
4.4 Boxplot of storage required for different data elements assuming no lowering . . . 67
4.5 Illustration of different tiling schedules (a) ASAP scheduling (F, C) (b) ALAP scheduling (C, F) . . . . . 69
4.6 Illustration of storage tradeoff between OFmaps and IFmaps depending on tile scheduling (loop ordering) and accelerator configuration parameters (loop unroll factors) (a) IFmap storage (b) OFmap storage (c) architectural illustration . . . . . 70

5.1 (a) SAM structure (b) Address generator structure . . . . . 72
5.2 Address generator Finite State Machine . . . . . 73
5.3 Illustration of different descriptor based programs with single address generators (a) Loop program (b) Delayed loop program . . . . . 76
5.4 Illustration of different descriptor based programs with dual address generators (a) Memory to memory transfer (b) variable sized FIFO . . . . . 77

6.1 Illustration of layer decomposition's effect on Ifmap tensors . . . . . 79
6.2 Illustration of channel based layer decomposition's effect on weight tiling . . . . . 80
6.3 Illustration of filter based layer decomposition's effect on weight tiling . . . . . 80
6.4 Illustration of interconnects for control and data in on-chip featuremap memories . 81
6.5 Illustration of weight tiling under (1, 1) convolution . . . . . 82
6.6 Illustration of channel banking in Ifmap L3 on-chip SRAMs . . . . . 83
6.7 Illustration of (1, 1) convolution scheduling . . . . . 84
6.8 Illustration of (3, 3) convolution scheduling . . . . . 86

7.1 Illustration of the simulation environment's python frontend . . . . . 88
7.2 Illustration of the simulation environment's SystemC backend . . . . . 89
7.3 Percent of computation represented by layers studied in the TIMM library . . . . . 90
7.4 Average layer utilization distributions by a) Network, b)Layer . . . . . 92
7.5 Shaded contour plot of average layer utilization vs number of MACs . . . . . 93
7.6 Scatter plot of utilization vs layer a) IFmap size and b) OFmap size. The dotted blue line represents the on-chip memory allocated for both types of feature maps. . . . . 94
7.7 Illustration of the causes of low utilization due to A) Layer type and B) Layer dimensions . . . . . 94

7.8 A) Barplot of the ratios between layer channel/ filters vs effective available channel/ filter concurrency for layers with speedup $< 1$ and utilization $> 95\%$ b) Barplot of the MAC factor increase due to lowering/lifting for layers with speedup $< 1$ and utilization $> 95\%$ . . . . . 95
7.9 a) Histogram of average network speedup over CPU baseline b) Histogram of layer speedup over CPU baseline c) Histogram of average network speedup with offloading of poorly mapped layers with low utilization and low speedup c) Histogram of FPS upper bounds for simulated networks . . . . . 96
7.10 Hardware Implementation Taxonomy adapted from [13] . . . . . 97
7.11 Median network energy breakdown by operation type, note that the vertical axis uses a log scale . . . . . 98
7.12 a) Inferences/J when factoring in DRAM access cost b) Inferences/J excluding DRAM access cost . . . . . 99
7.13 Area breakdown of HERO in the configuration specified in Table 7.1 . . . . . 100
7.14 Resnet50 performance results on HERO with the configuration specified in Table 7.1 103
7.15 Hardware Implementation Taxonomy adapted from [13] . . . . . 104

# List of Tables

2.1 Breakdown of the dimensionalities and complexity (in FLOPs and DRAM Reads) of the different available lowering and lifting strategies adapted from [2] . . . . . 12

4.1 Table of median storage requirements for data elements in convolution layers of networks in the TIMM Library . . . . . 67

7.1 HERO configuration used for analysis . . . . . 91
7.2 Median bandwidth requirements for simulated networks . . . . . 96
7.3 Energy model from [9] . . . . . 96
7.4 Energy model from [9] . . . . . 98
7.5 Area model from [9] assuming a 14nm technology node . . . . . 99
7.6 Resent50 layer breakdown . . . . . 101
7.7 Mobilenetv3 layer breakdown . . . . . 102

# List of Acronyms

**ASIC** Application Specific Integrated Circuit.

**FPGA** Field Programmable Gate Array.

**GEMM** General Matrix Multiplication.

**CNN** Convolutional Neural Network.

**DNN** Deep Neural Network.

**ACC** Accelerator.

**Conv** Convolution.

**MAC** Multiply and Accumulate.

**PE** Processing Engine.

**HERO** Hybrid GEMM and Direct Convolution Accelerator

**FC** Fully Connected layer

**EMPIRE** Network Compiler For Hero

**SAM** Self Addresable Memory

**TIMM** PyTorch Image Models

**TEMPO** accelerator TEMPlate Optimizer.

**CIGAR** ConvoluTional neural network Statistics GAtheRer.

**SAM** Self Addressable SRAMs.

# Acknowledgments

To my family, for their endless love and support.

# Abstract of the Thesis


HERO: A Hybrid General Matrix Multiplication and Direct Convolution Accelerator

by

Aly Sultan

Master of Science in Electrical and Computer Engineering

Northeastern University, MM YYYY

Professor Gunar Schirner, Adviser

Deep Neural Networks (DNNs) currently dominate in regression and classification problems in image recognition, sequence to sequence learning [22], and speech recognition [3]. Given the substantial variety of neural networks, there is a need for a general architecture that can support a wide range of networks while maintaining efficiency when running common configurations of the convolution operation. This thesis introduces the Hybrid GEMM and Direct Convolution Accelerator (HERO), a neural network accelerator that preserves computation generality by supporting matrix multiplication and maintaining computational efficiency when running common configurations of the convolution operation. Additionally, this thesis also introduces a toolchain that aids in the design, dimensioning, and programming of HERO. HERO's design is data-aware, optimized for the common case of convolutions by supporting common configurations directly without the need for conversion techniques like Im2Col [2]. To identify the common case of convolutions, this thesis introduces CIGAR, a tool that can gather configuration statistics for convolution and linear layers in a library of DNNs written in PyTorch. Using the HERO accelerator TEMPlate Optimizer (TEMPO), this thesis finds utilization optimal configurations for HERO. Additionally, a novel descriptor-driven on-chip memory primitive called Self-Addressable Memory (SAM) is proposed to orchestrate data movements. Additionally, this thesis introduces a HERO network compiler called EMPIRE that converts arbitrary convolution and linear layers described in pytorch to SAM descriptors. Finally, this thesis presents a cycle-accurate simulation platform driven by a SystemC simulation backend and a Python evaluation frontend to estimate performance and energy efficiency of arbitrary HERO configurations on different DNN networks described in Pytorch. An analysis of a utilization optimal HERO configuration when running 695 networks in the TIMM library was performed, the configuration achieved a median FPS of 91 FPS across all 695 networks with a median speedup of

4.87X over CPU baseline. The estimated bandwidth required for the configuration of HERO studied was 19.65GiB/s, which is within the PC4-21300 DDR4 specification. With that configuration of DRAM, the median inferences/J was 57. The total on-chip area for the configuration was estimated at 0.34 $mm^2$.

# Chapter 1

# Introduction

## 1.1 AI and Edge Computing

Deep Neural Networks (DNNs) currently represent the state of the art in complex regression and classification problems in image recognition, sequence to sequence learning [22], and speech recognition [3]. As such, they are being deployed on both cloud platforms and edge devices at scale. Among the various types of DNNs, Convolutional Neural Networks (CNNs) are widely used and demonstrate great accuracy in image/video recognition. The main computation layer of CNNs that consumes most of the runtime of a network is the convolutional layer. This layer operates on multi-dimensional tensors as part of a CNN's feature extraction process. Convolution layers exhibit a significant amount of parallel behavior and data reuse, making them suitable for parallel processing on GPUs and specialized hardware accelerators. The proliferation of CNNs in various environments, particularly on edge devices, has led to the increased demand for specialized hardware accelerators for CNNs with a particular emphasis on accelerating convolution layers. This is due to the tight latency, throughput, and energy constraints that these environments impose. The design of such accelerators is a complex task, requiring a balance between computational efficiency, parallelism, and energy efficiency. A number of different architectures have been proposed in the literature to address this problem, including systolic arrays, dataflow architectures, and hybrid architectures that combine both systolic arrays and dataflow. These architectures have their own set of advantages and trade-offs, and a careful analysis of the design space is necessary to decide on the optimal architecture for a specific use case. Additionally, optimizing the memory hierarchy and communication patterns between the accelerator and other system components is also crucial for achieving high performance. The use of specialized memory hierarchies, such as on-chip buffers,

can be effective in reducing data movement and improving energy efficiency.

## 1.2 Convolution accelerators

Prior work on CNN accelerators can be broadly classified based on three factors: the implementation technology, the level of network specificity, and the mathematical interpretation of the convolution operation. Some prior work has implemented CNN accelerators as hardened ASICs, which offer high performance and energy efficiency, but may lack flexibility and the ability to adapt to new CNN architectures without redesign. Other prior work has implemented CNN accelerators in FPGA fabrics, which offer high flexibility and adaptability, but may have lower performance and energy efficiency compared to hardened ASIC-based approaches. It is important to note that the choice of implementation technology should not be based solely on performance and energy efficiency, but should also consider factors such as ease of design, programmability, and the target application scenario, whether it be for edge or cloud deployment.

Regardless of the target execution platform chosen for a novel CNN accelerator architecture, there exists a need to create a general enough architecture that can support a wide variety of networks and network layer types. CNN accelerator generality can be decomposed into 1) Convolution generality, which can be defined as the range of support for convolution layers, and 2) Network generality, which can be defined as the types of convolution network layers supported. The ability to support a wide range of convolutional layers and network types can greatly increase the flexibility and usability of the accelerator. However, achieving this level of generality can also come at the cost of decreased performance and efficiency for the most common case of convolution layers and network types. Therefore, it is important to strike a balance between generality and efficiency when designing CNN accelerator architectures. Additionally, as CNNs are constantly evolving with new layers and network architectures being developed, it is important for accelerator architectures to have the ability to adapt and support new developments in the field.

FPGAs inherently have an advantage with respect to both types of generality, given their reconfigurable nature. FPGA-centric approaches incorporate the architecture of a target CNN network into their architecture compilation process [26]. This allows FPGA-based architectures to tackle network generality by adding layer-specific accelerators (provided they are available) at compile time and tailoring convolution accelerator primitives to the target network in order to provide the appropriate amount of convolution generality and performance. The disadvantage of FPGA-based architectures is the need to recompile the architecture prior to deployment of a new CNN,

potentially with no support for a new CNN without recompilation, and inferior performance and energy efficiency compared to a hardened architecture. To the best of this author's knowledge, no Field Programmable Gate Array (FPGA)-based Accelerator (ACC) compilation processes incorporate more than one Convolutional Neural Network (CNN) network architectures into their architecture optimization process.

ASIC-based architectures tackle network generality by introducing a wide variety of hardened layer accelerator primitives on-chip [11]. Additionally, they tackle convolution generality by either 1) reinterpreting convolution layers as GEMM operations or 2) creating a general enough convolution accelerator capable of supporting a wide range of convolution layer dimensions directly [6]. However, given the pace of development in DNNs, new layers like self-attention layers [19] can arise and become integral to improving DNN model performance [5]. These new layers may not be fully compatible with the chosen accelerator primitives in the ASIC-based design approach, resulting in only partial acceleration. Additionally, ASIC-based designs must balance their dedication of on-chip resources to convolution vs. other resource-intensive layers, which may cause convolution performance to suffer. Approaches that reinterpret convolutions to increase convolution generality like GEMM tend to dramatically increase data volume, which may strain on-chip memory resources as well as decrease energy efficiency [26]. Finally, supporting a wide range of convolutions can come at the cost of reduced performance/energy efficiency for the statistical common case of convolution layer dimensions in a wide range of CNNs.

## 1.3 Problem Definition

The literature highlights the need for an ASIC-based convolution accelerator that offers improved performance and energy efficiency for the common case of convolution layers across a broad range of CNNs, while maintaining operational generality to support a wide variety of layer types found in modern DNNs without sacrificing performance. To optimize for the common case of convolutions, a statistical analysis of a wide range of networks is necessary to identify the common characteristics of convolution layers across different networks, providing crucial insights on how to configure the accelerator's memory hierarchy and on-chip network structure to optimize performance and energy efficiency. To cater to the wide variety of DNNs, the architecture must be configurable at compile-time to adapt to changing trends in network architectures. Additionally, flexible on-chip data movement is necessary to map diverse layers to the architecture. However, since a static communication and memory hierarchy structure will not enable that flexibility, the

architecture should incorporate programmable on-chip data movement primitives that can be programmed using a network compiler, generating data movement instructions based on the layer configuration within the network. To evaluate the performance and energy metrics of the accelerator, a cycle-accurate simulator should be developed, which will enable the assessment of different configurations of the architecture on a wide variety of networks and evaluate the performance and energy efficiency of different configurations.

## 1.4 Solution Overview

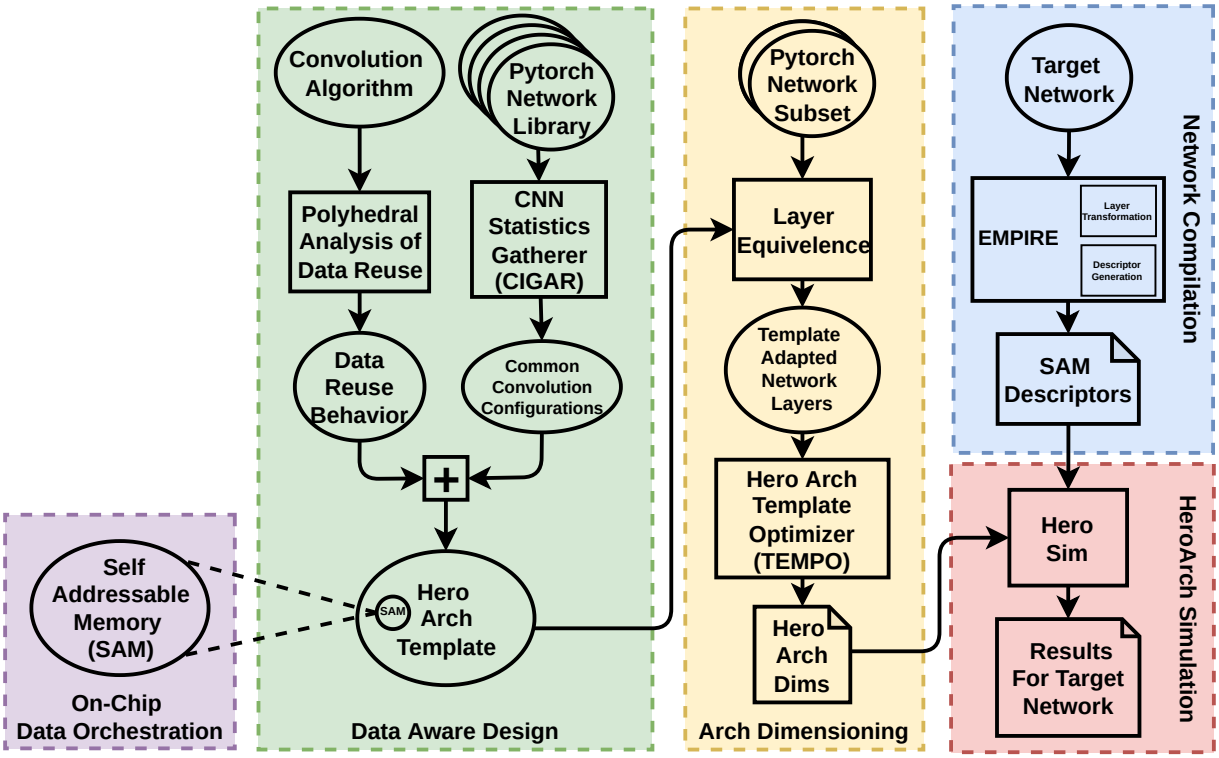



Figure 1.1: Visual illustration of the solution overview

The full solution overview is presented in Figure 1.1. This thesis presents (HERO), a Hybrid GEMM and Direct Convolution Accelerator. HERO supports general matrix multiplication to maintain generality across different network layers. General matrix multiplication is the backbone of many computationally intensive layers in modern DNNs, for example, self attention layers in

transformer networks [19] as well as fully connected layers in many CNNs like Resnet [10]. Extending support to GEMM should not detract from the primary goal of accelerating convolutions since they represent a larger portion of most network's runtime. HERO is derived from a data aware design process. It is optimized for the common case of convolutions in the literature by supporting these convolution configurations directly without the need to convert them into GEMM using data transformation techniques like Im2Col.

To find the common case of convolutions, this thesis introduces CIGAR. CIGAR can gather configuration statistics for convolution and linear layers in a libray of DNNs networks written in PyTorch [17]. The heart of CIGAR is it's model dim collector that can collect convolution and linear layer configuration for any network written in pytorch. CIGAR can analyze a network or a library of networks and reveal the most common convolution layer configurations (e.g their kernel and stride sizes) across the entire library. The networks analyzed by CIGAR are all provided by the PyTorch Image Models (TIMM) python library of networks [20], which has over 695 networks written in pytorch available for analysis. Convolution layer configurations that are not supported directly by the architecture are converted into equivalent GEMM operations using data transformation techniques like Im2Col [2]. This thesis introduces the Hybrid GEMM and Direct Convolution Accelerator (HERO), a neural network accelerator that balances generality and efficiency in its design. HERO supports general matrix multiplication, the backbone of many computationally intensive layers in modern DNNs, without detracting from its primary goal of accelerating convolutions.

HERO's design is data-aware, optimized for the common case of convolutions by supporting common configurations directly without the need for conversion techniques like Im2Col. To identify the common case of convolutions, this thesis introduces CIGAR, a tool that can gather configuration statistics for convolution and linear layers in a library of DNNs written in [17]. CIGAR can analyze a network or a library of networks and reveal the most common convolution layer configurations across the entire library. The networks analyzed by CIGAR are provided by the [20] python library of networks, which has over 695 networks written in PyTorch available for analysis. Any layer configurations not supported directly by HERO are converted into equivalent GEMM operations using data transformation techniques like Im2Col [2].

HERO is more than just a single architecture; it represents a configurable template with flexible allocation of on-chip compute resources called processing engines (PEs) used in processing convolution layer channels and filters. This configurability allows for compile-time optimization of HERO by changing the number of PEs allocated to processing different channels and filters in a convolution layer. To determine the optimal configuration, this thesis introduces TEMPO, a tool

that uses analytical models for estimating different architecture metrics like latency, utilization, and memory access counts when running different DNNs. TEMPO takes advantage of CIGAR's model dimension collection to extract configurations of convolution layers and applies the aforementioned analytical models to them. TEMPO provides the initial architecture dimensions for the HERO template in order to get the first estimates for other performance metrics. TEMPO only defines layer dimensions for channel and filter concurrency. A separate analysis of memory usage for different convolution elements is provided in this thesis. The networks used by TEMPO is the entire TIMM library to find the most general architecture dimensions for HERO.

This thesis introduces the Self-Addressable Memory (SAM), an on-chip programmable memory primitive, to manage data movement within the Hybrid GEMM and Direct Convolution Accelerator (HERO). SAMs are programmed using descriptors that define time-dependent address streams, which, in combination with sufficiently flexible on-chip communication, provide the necessary flexibility to map arbitrary network layers onto HERO. SAMs enable for more efficient data movement on-chip, with the aim of maximizing data reuse. To determine the reuse behavior in the convolution operation, this thesis applies the polyhedral model to different data elements in convolution layers, using techniques outlined in [16]. Additionally, this thesis introduces a HERO network compiler called EMPIRE that aids in the mapping of arbitrary PyTorch models to SAM descriptors, which drive on-chip data movement.

To evaluate the performance and energy efficiency of HERO, a cycle accurate simulation platform is developed. This simulation platform is driven by a SystemC simulation backend and a Python evaluation frontend. The simulation platform allows for the evaluation of different configurations of HERO on a wide variety of networks. An analysis of an optimal HERO configuration when running 695 networks from the TIMM library was performed. The HERO configuration analyzed was found to perform well on a wide variety of network configurations, achieving a median FPS of 91 FPS with a median speedup of 4.87X over CPU baseline. The estimated bandwidth required for the configuration of HERO studied was 19.65GiB/s which is within the PC4-21300 DDR4 specification. With that configuration of DRAM, the median inferences/J is 57. The total on-chip area is estimated at 0.34 $mm^2$.

Overall, HERO provides a high-performance, energy-efficient solution for accelerating convolution layers in a wide range of DNNs. Its general architecture and support for GEMM, in addition to its direct support for the common case of convolution layers, make it a versatile accelerator that can adapt to changing trends in DNNs. The introduction of CIGAR and the HERO layer compiler, in addition to the use of SAM, further improves the design process and ease of use

of HERO. The cycle accurate simulation platform developed allows for the accurate evaluation of HERO's performance

## 1.5 Thesis Structure

In this thesis, Chapter 2 discusses background and related work in the literature. Chapter 3 introduces the CIGAR and the data-aware design approach from which HERO is derived. Chapter 4 introduces TEMPO, from which several candidate configurations of HERO are presented. Chapter 5 introduces the SAM primitive. Chapter 6 discusses HERO's network compilation process and how SAM descriptors are generated from arbitrary models in Pytorch using EMPIRE. Finally, Chapter 7.1 discusses the HERO simulation platform as well as results from running a HERO configuration optimized by TEMPO on all 695 networks in the TIMM library.

# Chapter 2

# Background

This chapter introduces and explores the following concepts to provide the necessary context for the remaining chapters in this thesis. Firstly, convolutional neural networks will be introduced, along with a mathematical representation of the convolution operation. After that, a brief discussion on reinterpreting convolutions as GEMM operations will be presented. Reinterpreting convolutions as GEMM is integral to HERO's support of arbitrary convolution layer configurations.

GEMM conversion is not a requirement for to support all convolution layers. HERO is optimized for the common case of convolution and hence has to support a subset of the convolution operations that exist in the literature directly. This means that HERO needs to implement some configurations of the convolution operation as dataflow operations in hardware. Prior to exploring the dataflow design space of convolution accelerators, we must first define it.

In this chapter, a taxonomy of convolution accelerator dataflows is introduced from [15]. An exploration of this design space, when combined with network configuration data provided by CIGAR, allows us to define HERO's dataflow for the common case of convolutions in the literature.

After that, the HW implementation taxonomy from [13] is introduced. The HW implementation taxonomy defines the space of possible HW implementation options available when implementing a convolution accelerator in hardware. The available hardware implementation options depend on the communication and reuse behavior of different data elements defined by an accelerator's chosen dataflow. HERO's hardware implementation is hence driven by an analysis of its chosen dataflow's data element communication and reuse behavior. This chapter concludes with a discussion of related work in the literature.

## 2.1 Convolutional Neural Networks And The Convolution Operation

Neural networks are a class of machine learning models that are used for various tasks such as image recognition and object detection. They can be trained to model complex mathematical functions by adjusting their internal parameters. These models are composed of multiple layers, each of which performs a specific computation on the input data in order to produce the desired output.

Convolutional Neural Networks (CNNs) are a subtype of neural networks that place a particular emphasis on the use of the convolution layer in computing the network's output. Convolution layers are responsible for detecting patterns and features in the input data by sliding a small filter over the image and computing the dot product between the filter and the image at every position.

In addition to convolution layers, CNNs commonly make use of other layer types such as fully connected layers, activation functions and batch normalization layers. However, it is worth noting that the majority of a network's inference runtime is spent computing the convolution layers [8]. An illustration of the different layers that can be present in a CNN is presented in figure Figure 2.1

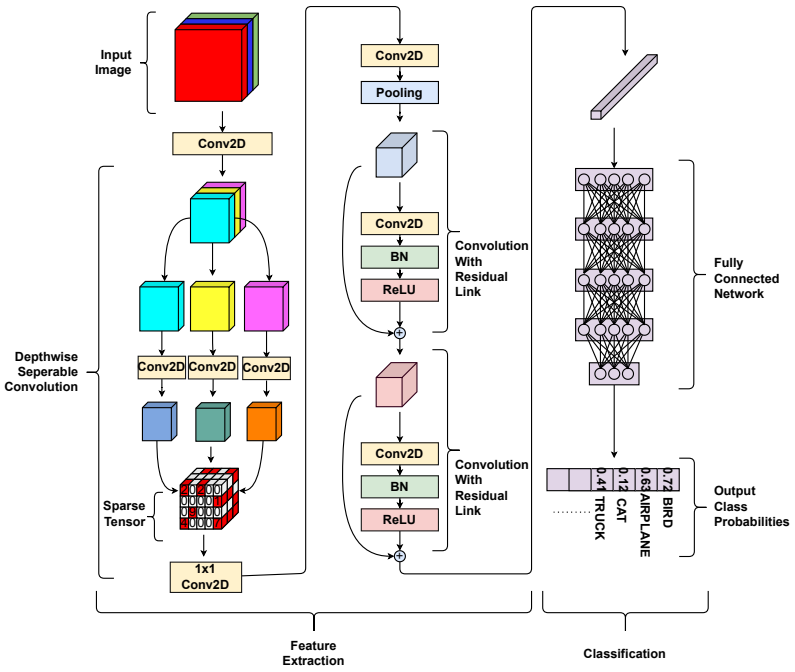



Figure 2.1: Example of the different layers in a convolution neural networks

Assuming the input feature map (IFmap), output feature map (OFmap) and weight dimensionalities of a layer are defined as in equation (Equation 2.1). A mathematical representation of the convolution operation is given in equation (Equation 2.2). This equation represents a stencil-based operation where a sliding window, called a kernel, is moved across an input feature and at each position, a multiply-and-accumulate operation is performed to compute an output element of the output feature map. This stencil operation is repeated for each kernel present in the layer. The number of kernels in a layer is referred to as the number of filters. This multiply-and-accumulate operation occurs across all three dimensions of the IFmap tensor.

In addition to the mathematical description in equation (Equation 2.2), a visual representation of the convolution operation is provided in Figure 2.2. This figure shows an IFmap tensor with 3 channels (red, green, blue) being convolved with 4 separate filters, each with kernels of size 2x2. The contents of the kernels are called the weights of the layer. Each filter's kernels operate on separate IFmap channels. Each kernel is convolved with each IFmap channel using a sliding window operation. The outputs across all IFmap channels are aggregated into one OFmap channel. The total number of OFmap channels equals the total number of filters in the convolution layer.

$$\begin{aligned} IFmap &\in R^{C\times n\times n} \\ OFmap &\in R^{F\times m\times m} \\ Weight &\in R^{F\times C\times K\times K} \end{aligned} \tag{2.1}$$

$$OFmap[f][y][x] = \sum_{c=0}^{C-1}\sum_{k_x=0}^{K-1}\sum_{k_y=0}^{K-1} Weight[f][c][k_y][k_x] * IFmap[c][y+ky][x+kx] \tag{2.2}$$

Listing 2.1: Convolution implemented as nested loops

```
1 for(int f = 0; f < F; f++) // Filter loop
2     for(int c = 0; c < C; c++) // Channel loop
3         for(int y = 0; y < Y; y++) // Output feature map row
4             for(int x = 0; x < X; x++)  // Output feature map col
5                 for(int ky = 0; ky < KY; ky++)  // Kernel row
6                     for(int kx = 0; kx < KX; kx++)  // Kernel col
7                         O[f][y][x] += I[c][y+ky][x+kx]*W[f][c][ky][kx];
```

To compute the convolution operation in software we can implement the expression in Equation 2.2 as a series of nested for loops as in Listing 2.1. This represents a direct approach to computing convolution layers. An alternative approach would be to converted convolution layers to general matrix multiplication operations through several input/ output data transformations. The

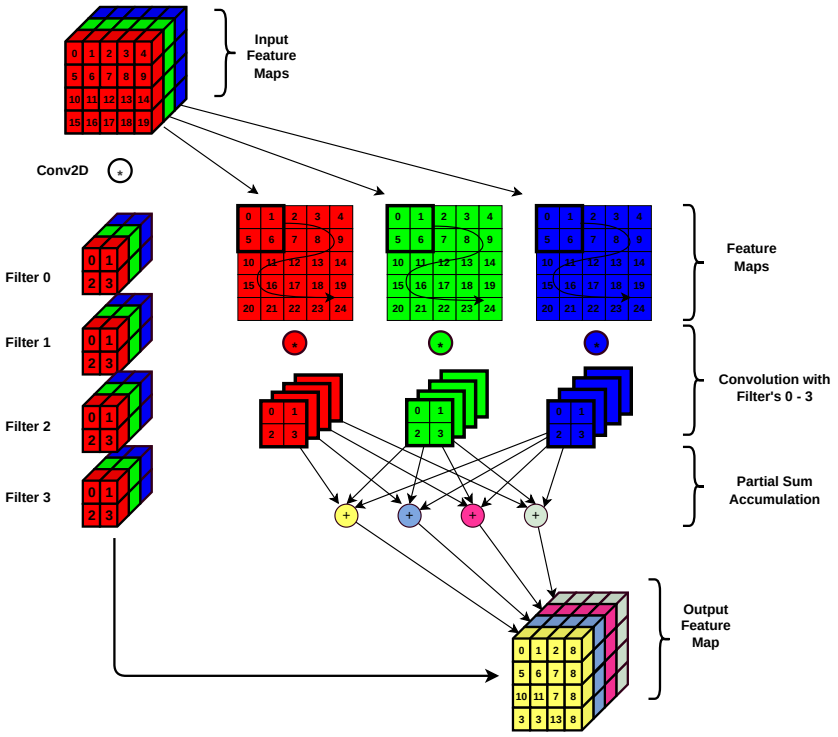



Figure 2.2: Convolution Operation Illustrated

advantage of reinterpreting convolutions as matrix multiplication arises from the use highly optimized libraries for computing GEMM like [4] on CPUs and [1] on CUDA compliant GPUs. The next section will provide a brief overview of some of the data transformation operations required to convert convolution into GEMM. HERO makes extensive use of these transformations to expand it's support for a wide variety of convolution operations outside of the common case of convolutions in the literature.

## 2.2 Reinterpreting Convolutions As GEMM

In the literature [2] there are several techniques used to to transform convolution operations into GEMM using data transformations applied to the IFmaps and OFmaps of a convolution layer. Table 2.2 shows the different costs, in floating point operations (FLOPs) and memory overhead, of the different transformation techniques discussed in [2].

| | | Expensive Lowering/ Lifting | Balanced Lowering and Lifting | Lowering/ Expensive Lifting |
|---|---|---|---|---|
| Lowering | Lowered IFmap Size | ($k^2$C, $m^2$) | (kC, mn) | (C, $n^2$) |
| | Lowered Kernel Size | (F, $k^2$C) | (Fk, kC) | (F$k^2$, C) |
| Matrix Multiply | Input Size | (F, $k^2$C)x($k^2$C, $m^2$) | (Fk, kC)x(kC, mn) | (F$k^2$, C)x(C, $n^2$) |
| | # FLOPS | 2F$k^2$C$m^2$ | 2F$k^2$Cmn | 2F$k^2$C$n^2$ |
| | Output Size | (F, $m^2$) | (Fk, mn) | (F$k^2$, $n^2$) |
| Lifting | # FLOPS | 0 | $m^2$kF | $m^2k^2$F |
| | # Ram Read | F$m^2$ | Fkmn | F$k^2n^2$ |

Table 2.1: Breakdown of the dimensionalities and complexity (in FLOPs and DRAM Reads) of the different available lowering and lifting strategies adapted from [2]

The first of the transformation techniques used in computing convolution operations as general matrix multiplication discussed in [2] is the Im2Col transformation or as [2] refers to it "Expensive lowering/lifting". The lowering/ lifting nomenclature arises from reduction of the IFmap and Weight tensor dimensionalities from 3D to 2D and vice versa for the OFmap. The expensive descriptor arises from the substantial increase in memory allocation required from lowering the IFmap input into two dimensions as a result of data duplication.

Expensive lowering and lifting "flattens" the IFmap by positioning a hypothetical stencil where the real stencil would be positioned in the convolution operation and collects all the IFmap elements present in that stencil into one row of an IFmap matrix. The stencil's dimensions are proportional to the kernel size in the weight tensor. This collection processes is repeated for every real stencil position present in the original convolution operation. The stencil is moved through the IFmap tensor following a sliding window pattern. Since stencil positions are usually very close to each other (depending on the stride size of the layer) a substantial amount of IFmap elements are reused between stencil positions. The act of lowering the contents of each stencil position into one row results in the duplication of IFmap elements between consecutive rows in the IFmap matrix. The weight kernels for each filter are flattened vertically into a weight matrix. The OFmap matrix

is then produced by matrix multiplication between the IFmap and weight matrices. To recover the OFmap tensor a ”lifting” operation is performed by reshaping the output OFmap matrix into a 3D tensor. An illustration for expensive lowering/ lifting is given in Figure 2.3. In Figure 2.3 a 4x4x3 IFmap is lowered into a matrix using expensive lowering. Each row of the matrix represents the contents of a single stencil position. The weight tensor of size 2x2x3x4 is also lowered into a matrix. Each filter in the weight tensor is transformed into a column in the weight matrix. The multiplication of both of the IFmap and weight matrices produces the OFmap matrix. The lifting operation performed on the OFmap matrix reshapes the matrix into a 3D tensor by converting each column of the OFmap matrix into a channel in the OFmap tensor.

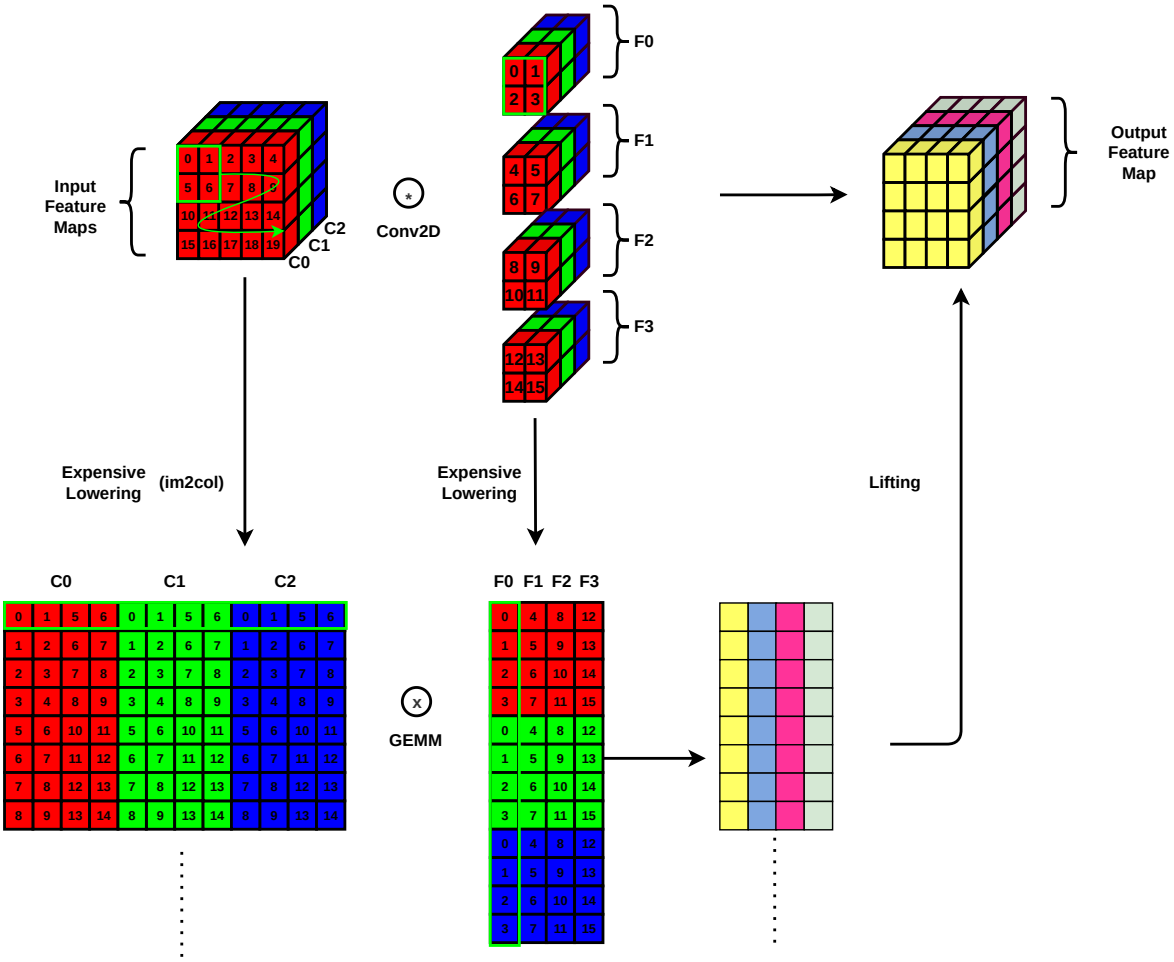



Figure 2.3: Im2Col (Expensive lowering/ lifting) Illustrated

An alternative approach to expensive lowering/ lifting is balanced lowering and lifting. Analytical expressions that describe balanced lowering/ lifting adapted from [2] are given in (2.3) - (2.6) with the inclusion of lowering in the presence of multiple filters to describe these data transformation operations. In addition, to supplement the analytical expressions balanced lowering presented Figure 2.4 is used to clarify the available expressions further.

In balanced lowering, we first lower the ifmap and weights using expression (2.3) and (2.4). Then a matrix multiplication is performed in (2.5) followed by a lifting operation. Unlike the lifting operation in the expensive lowering/ lifting transformation, lifting in the balanced transformation involves a series vector operation in addition to a reshape of the output OFmap matrix. Balanced lowering has the advantage of reduced data duplication in the IFmap and trades that off with increasing the number of FLOPs for lifting the OFmap. Figure 2.4 illustrates balanced lowering/ lifting on a 5x5x3 IFmap and a 2x2x3x4 weight tensor. The same stencil based intuitive explanation presented for expensive lowering/ lifting applies here with a few modification. For IFmap lowering, instead of using a full sized 2x2 stencil to collect the IFmap tensor contents a half sized 1x2 stencil is used. The stencil dimensions remain proportional to the weight tensor's kernel size. The sliding window pattern still applies when moving the stencil through the IFmap. However, instead of moving the stencil horizontally through IFmap then vertically we first move the stencil vertically then horizontally through the IFmap tensor. Each position of the stencil corresponds to one output row in the IFmap matrix. Lowering the weight tensor is similar to expensive lowering in that the contents of each filter are lowered into columns of the weight matrix. However, instead of each filter occupying 1 column of the weight matrix, each filter row and it's associated channels occupies 1 column in the weight matrix.

$$\begin{gathered} IFmap \in R^{C\times n\times n} \xrightarrow{BalancedLowering} \hat{IFmap} \in R^{nm\times KC} \\ \hat{IFmap}[cn+r,:] = vec(IFmap[:,r,c:c+K]) \\ \forall r,c \in [0,n-1],[0,m-1] \end{gathered} \tag{2.3}$$

$$\begin{gathered} Weight \in R^{F\times C\times K\times K} \xrightarrow{BalancedLowering} \hat{Weight} \in R^{KC\times FK} \\ \hat{Weight}[f*K:f*K+K,i] = vec(Weight[f,:,i,:]) \\ \forall f,i \in [0,F-1],[0,K-1] \end{gathered} \tag{2.4}$$

$$\hat{OFmap} = \hat{IFmap}.\hat{Weight} \tag{2.5}$$

$$\hat{OFmap} \in R^{nm \times FK} \xrightarrow{BalancedLifting} OFmap \in R^{m \times m \times F}$$

$$OFmap[f, r, c] = (\sum_{j=0}^{K-1} \hat{OFmap}[cn + r + j, j + fK]) \tag{2.6}$$

$$\forall f, r, c \in [0, F-1], [0, m-1], [0, m-1]$$

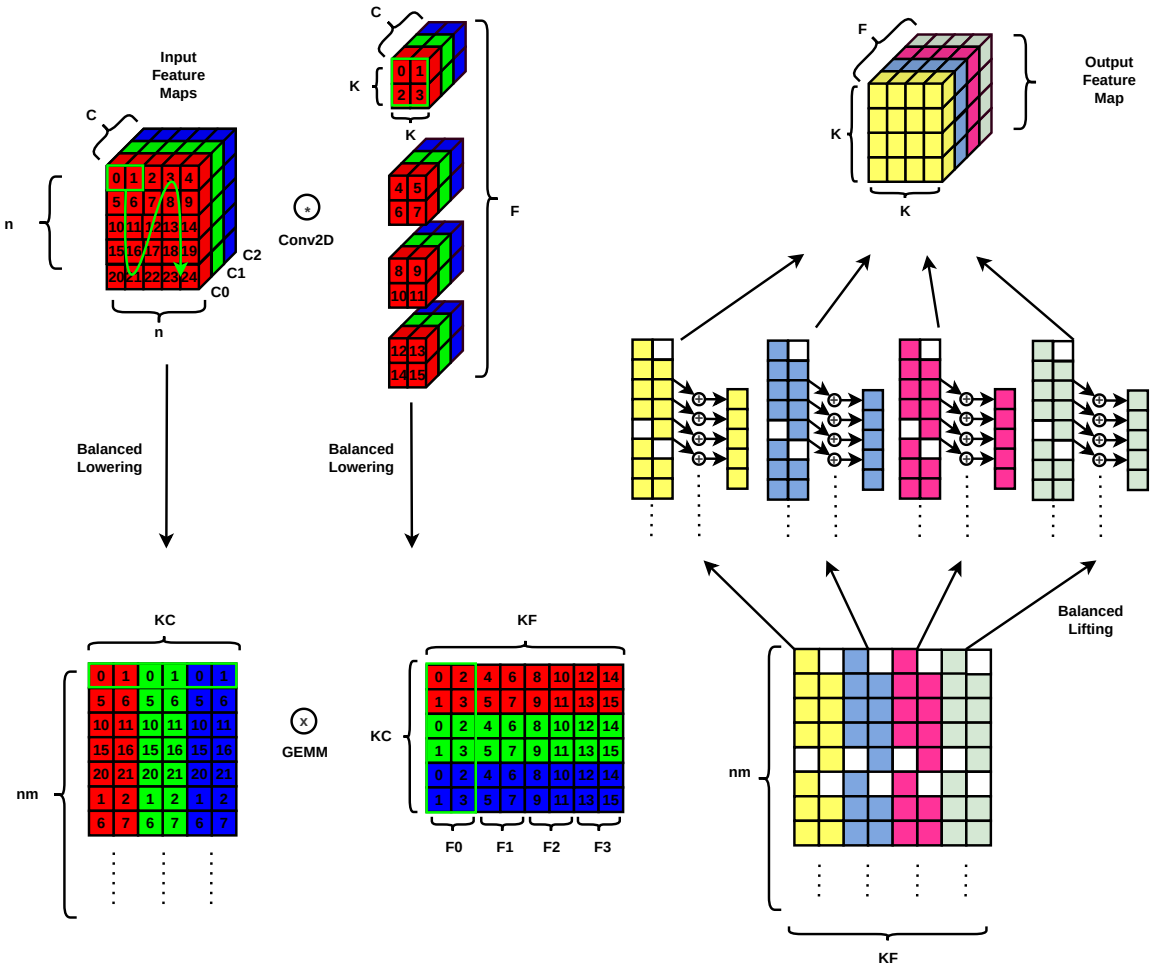



Figure 2.4: Balanced Lowering/Lifting Illustrated

The third (lowering/ expensive lifting) strategy further trades off the the size of the lowered matrices by increasing the complexity of lifting. However, this strategy diminished the gains from converting convolution layers to general matrix multiplication and therefore are not elaborated on in this thesis. In section 2.2 presents the dimensionalities and complexities of the three strate-

gies discussed in [2] using the dimensions of the IFmap, OFmap, and Weight tensors defined in Equation 2.1.

HERO takes advantage of the aforementioned data transformation approaches to convert unsupported convolution operations into GEMM operations. However, HERO is optimized for the common case of convolutions which allows it to run a subset of the convolution operations in the literature directly. HERO implements this subset of convolution operations as dataflow operations in hardware. As mentioned in the introduction of this chapter in order to explore the accelerator dataflow design space available to HERO we must first define the dataflow design space. This will be the focus of the next section.

## 2.3 Implementing Convolutions In Hardware

### 2.3.1 The Dataflow Taxonomy

From [24] dataflows can be defined using the direct convolution nested loop structure combined with unroll pragmas. Listing listing 2.2 expands on listing 2.1 by including unroll pragmas on all of the for loops. What defines that dataflow is 1) loop unroll targets (which loops are unrolled) 2) loop order 3) the unroll factors of the unrolled loops. These choices influence the stationarity of different data elements used in the convolution operation.

Listing 2.2: Convolution implemented as nested loops

```
1  #pragma UNROLL F_T
2  for(int f = 0; f < F; f+=F_T) // Filter loop
3  #pragma UNROLL C_T
4      for(int c = 0; c < C; c+=C_T) // Channel loop
5  #pragma UNROLL Y_T
6          for(int y = 0; y < Y; y+=Y_T) // Output feature map row
7  #pragma UNROLL X_T
8              for(int x = 0; x < X; x+=X_T)  // Output feature map col
9  #pragma UNROLL KY_T
10                 for(int ky = 0; ky < KY; ky+=KY_T)  // Kernel row
11 #pragma UNROLL KX_T
12                     for(int kx = 0; kx < KX; kx+=KX_T)  // Kernel col
13                         O[f][y][x] += I[c][y+ky][x+kx]*W[f][c][ky][kx];
```

Listing 2.3 shows a generic implementation of a single convolution layer with all layers unrolled except the output feature map loops. The pragmas define the unroll targets and the con-

stants in the pragmas define the unroll factors. Weight elements within a kernel remain stationary throughout the computation of an output feature map until a new tile of channels C_T is loaded into the accelerator. Once the weights within a particular channel and filter group are used to produce an output feature map they are discarded and are only loaded again when computing the same layer for a new input image. The choice of loop unroll targets, factors and loop ordering in listing 2.3 creates a common dataflow in the literature called weight stationary.

Listing 2.3: Convolution implemented as nested loops

```
#pragma UNROLL F_T
for(int f = 0; f < F; f+=F_T) // Filter loop
#pragma UNROLL C_T
    for(int c = 0; c < C; c+=C_T) // Channel loop
        for(int y = 0; y < Y; y+=Y_T) // Output feature map row
            for(int x = 0; x < X; x+=X_T)  // Output feature map col
#pragma UNROLL KY_T
                for(int ky = 0; ky < KY; ky+=KY_T)  // Kernel row
#pragma UNROLL KX_T
                    for(int kx = 0; kx < KX; kx+=KX_T)  // Kernel col
                        O[f][y][x] += I[c][y+ky][x+kx]*W[f][c][ky][kx];
```

From listing 2.3 we can see that from the loop unroll targets and loop unroll factors there are many other possible dataflow configurations available to us outside of weight stationary. Additionally, since accelerators are generally limited to two spatial axis the loops of the convolution operation can be mapped to two spatial axis. If we allow multiple convolution loops under some kernel unroll factor KY_T/KX_T to be unrolled and mapped to the same accelerator spatial axis we can influence the effective unroll factors when performing different convolutions of different kernel sizes other than KY_T/KX_T. The choice of which loops are mapped to which spatial axis is an additional design dimension alongside loop unrolling.

The figures in Figure 2.5 demonstrate hardware implementations for different dataflow configurations that result from various loop unroll selections. However, it is important to note that the space of available hardware schemes is not limited to those presented in Figure 2.5. A discussion on the space of possible hardware schemes that can be achieved given different dataflow configurations will be presented in the following section.

With the dataflow design space defined by [24] a data driven exploration of the dataflow design space becomes possible. Combining network configuration data provided by CIGAR with [24] dataflow taxonomy will enables us to define HERO's dataflow for the common case of con-

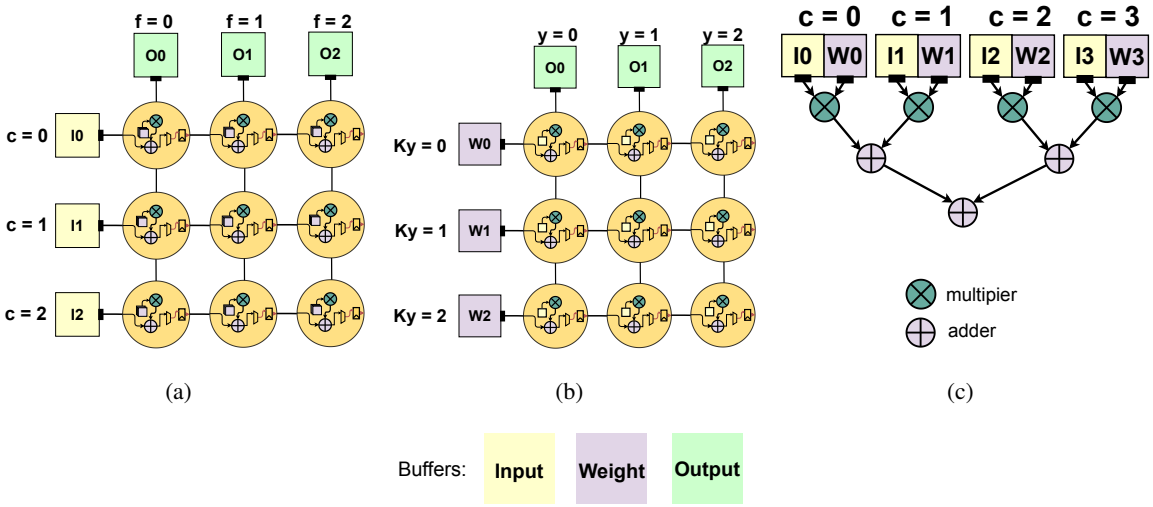



Figure 2.5: Illustration of different dataflow implementations (adapted from[24]) arising from (a) Unrolling F and C loops (b) unrolling Ky and Y loops (c) unrolling C loops

volutions in the literature. However, a dataflow choice needs to be matched with an appropriate hardware implementation. In the next section an accelerator hardware implementation taxonomy is introduced from [13]. The available hardware implementation options in the taxonomy are determined by communication and reuse behavior of the different data elements in a chosen dataflow. The hardware taxonomy presented in the next section will be used in tandem with the dataflow taxonomy to define HERO's implementation in hardware.

### 2.3.2 The Hardware Implementation taxonomy

The full hardware taxonomy from [13] is illustrated in figure Figure 2.6. Depending on the dataflow selected using the dataflow taxonomy (1) loop ordering (2) unroll targets (2) loop unroll factors. The implementation options are derived based on the type of reuse present in the dataflow. Following the hardware implementation taxonomy presented in in [13], we can classify the available hardware implementation options based on the the type of reuse is spatial where a data element is read and used in the same cycle or temporal where a data element is read in one cycle and reused after several cycles. Depending on the nature of the reuse, if it is read or read modify write, there are several options for supporting the communication inferred from that reuse type. To deduce the type of reuse and overall communication behavior for each data element in any dataflow we can use the

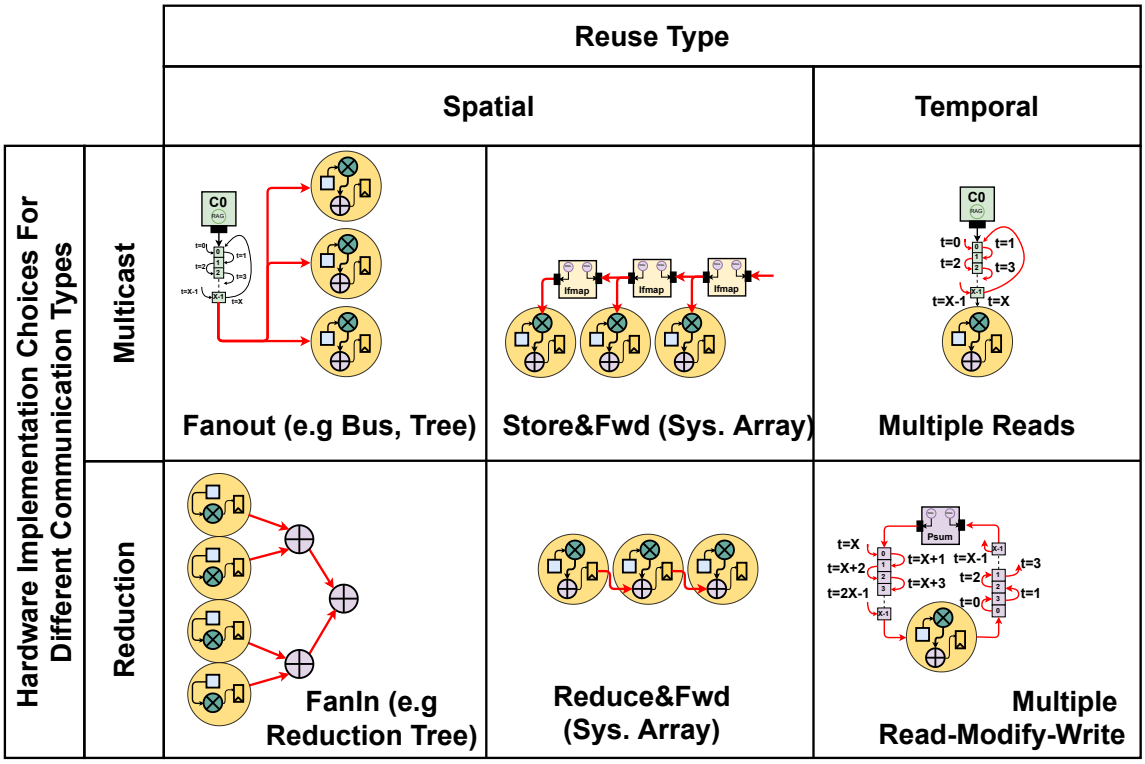



Figure 2.6: Hardware Implementation Taxonomy adapted from [13]

polyhedral model to detect temporal reuse. Spatial reuse detection can be inferred directly from the loops. The application of the polyhedral to infer temporal reuse behavior as well as the inference of spatial reuse directly from the convolution loops is discussed in chapter 3.

The hardware taxonomy presented in this section will be used in tandem with the dataflow taxonomy to define HERO's implementation in hardware. Note that the usage of this taxonomy will span both operational modes of HERO, direct convolution acceleration as well GEMM. What enables this taxonomy to be applied to GEMM is the overlap between GEMM and the depthwise convolution operation. A discussion of how these two operations overlap is presented in section 3.2.

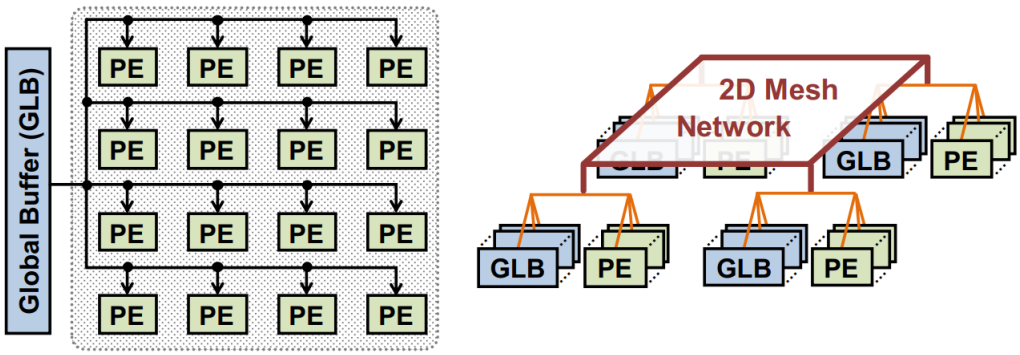
## 2.4 Related work

In this section, a comparison of HERO with other works in the literature is presented with emphasis on competing ASIC-based accelerator designs. It should be noted that the discussion is limited to ASIC-based implementations, as FPGA-based accelerators typically optimize the

accelerator's structure for a specific network through compile-time optimizations. Since HERO is intended as a static architecture for synthesis as an ASIC, comparisons with related architectures in the literature are limited to other ASIC-based accelerators.

Accelerator designs discussed in this section include 1)the Eyeriss V1 and V2 accelerators [7] which utilize a 2D array of processing elements (PEs) to support convolution operations. EyerissV1 employs a tiled architecture to support a wide range of convolutional layers and a specialized on-chip memory hierarchy to provide high memory bandwidth and low energy consumption. Eyeriss V2 which optimizes the Eyeriss V1 architecture for sparse and compact DNNs 2) the Tensor Processing Unit (TPU) [11] developed by Google uses an array of PEs to accelerate neural network computations. It is specifically optimized for Google's neural network workloads and provides high performance and energy efficiency. 3) the MAERI accelerator [14] which aims to support a wide variety of DNN layers and mappings by moving the complexity of data orchestration to runtime configuration of a flexible on-chip Network-on-Chip (NoC). It allows for efficient mapping of diverse layer types to the architecture and offers high performance and energy efficiency.

### 2.4.1 Eyeriss V1 and V2




(a) Original Eyeriss (b) Eyeriss v2

Figure 2.7: Illustration of Eyeriss V1 and Eyeriss v2's architecture from [7]

Eyeriss [7] is an accelerator for state-of-the-art deep convolutional neural networks (CNNs). It attempts to optimize for the energy efficiency of the entire system by minimizing data movement between the accelerator and DRAM thus reducing data movement energy. Eyeriss achieves these goals by using a the row stationary (RS) dataflow on a spatial architecture with 168 processing

elements.

EyerissV2 [6] improve on EyerissV1 by proposing an architecture optimized for sparse and compact DNNs. To account for the substantial variation between different CNN layer it introduces a highly flexible on-chip network, called hierarchical mesh, that can adapt to the different amounts of data reuse and bandwidth requirements of different data types. THe goal of the mesh is to improve the utilization of the computation resources.

EyerissV1 and EyerissV2 do not optimize for linear layers. Linear layers may not represent a substantial portion of a CNN network runtime, however, some models use linear layers exclusively. These models are underrepresented in vision based tasks however, they are present in NLP based tasks. An examples of these models includes the Transformer model [19]. This limits how general the Eyeriss architecture is when used for models outside of the vision domain. This work aims to build an accelerator that accounts for the importance of linear layers in those models.

### 2.4.2 Tensor Processing Unit

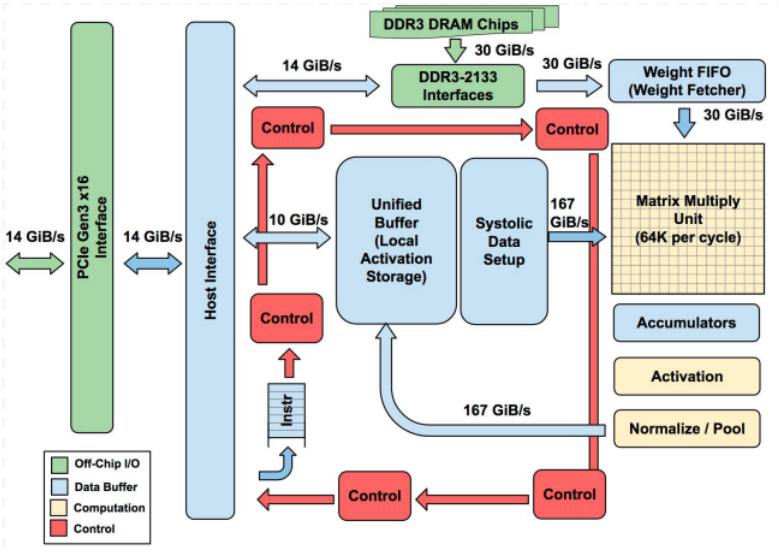



Figure 2.8: Block diagram of the TPU architecture from [11]

At the heart of a TPU is a 65,536 8-bit MAC matrix multiply unit that offers a peak throughput of 92 TeraOps/second (TOPS) and a large (28 MiB) software-managed on-chip memory [11]. The TPU emphasizes efficiency in computing matrix multiplication above all other operations. With the aforementioned transformations used to convert convolution layers to general ma-

trix multiplication the TPU is able to accelerate a wide variety of computation workloads given the generality of the operation it's hardware accelerates. This generality comes at the cost of efficiency in computing convolution layers given the overheads of the transformations discussed in the prior sections. This works aims to mitigate this inefficiency while maintaining support for a wide variety of computation workloads.

### 2.4.3 Maeri

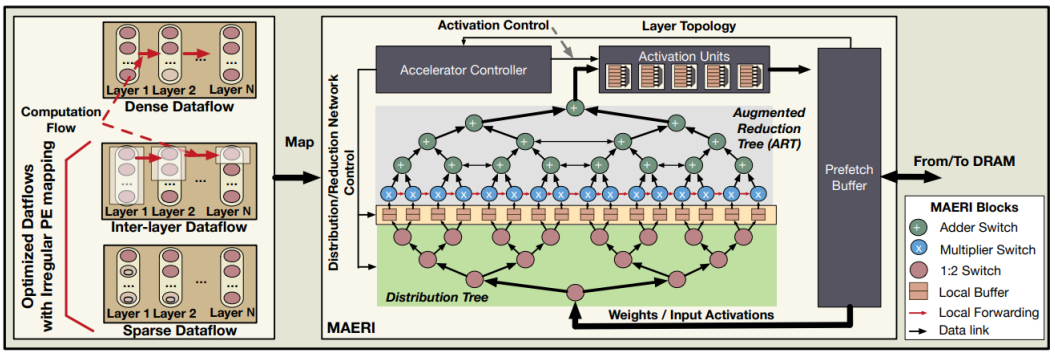



Figure 2.9: An overview of Maeri's architecture [14]

Since most DNN accelerators support only fixed dataflow patterns internally mapping arbitrary layer dataflows to these fabric efficiently is challenging. Mapping inefficiencies can lead to under utilization of the available compute resources. DNN accelerators need to be configurable internally to support the various dataflow patterns that could be mapped over them. To address this [14] introduces MAERI, a DNN accelerator built with a set of modular and configurable building blocks backed by a flexible on-chip network capable of supporting a myriad of DNN partitions and mappings.

MAERI aims to support a wide variety of DNN layers and mappings by moving the complexity of data orchestration to runtime configuration of a flexible on-chip NOC. The drawback of this approach is the complexity of the NOC as well as the area it occupies on-chip. This work aims to combine the runtime flexibility of data orchestration on-chip through the use of a novel data orchestration primitive called self addressable memory (SAMs) discussed in chapter 5 with the small area footprint of a fixed on-chip fabric optimized through a novel optimizer presented in chapter 4.

# Chapter 3

# Data-Aware Accelerator Design

As discussed in chapter 2, any convolution accelerator can be reduced into it's dataflow and hardware implementation choices. Based on the dataflow exploration approach in [25] dataflows are explorable through the nested loop structure that makes up a convolution layer introduced in chapter 2. The design space dimensions of dataflows is comprised of:

- Loop unroll targets (which loops are unrolled and which are not)
- Loop unroll factors for the loop unroll targets
- Mapping of loops to an accelerators spatial axis of which there are 2

We can enumerate the size of the design space by examining the scope of the aforementioned design space dimensions. Beginning with the choice of loop unroll targets. One can choose to unroll only one loop or all loops with varying unroll factors. Unrolling loops exposes opportunities for parallelism when executing unrolled loops on an accelerator. Therefore the number of possible combinations of loop unroll targets is $\sum_{l=1}^{6} \binom{6}{l}$ with a total of 6! possible orderings for said loops. The space of possible loop unroll axis mappings is $\binom{l}{\min(2,l)}$ depending on the chosen number of loops $l$ unrolled. The space of possible unroll factors is then dictated by the upper bounds of the indexes in the loop representation $\prod_l max(l) \forall l \in \{F, C, Y, X, KY, KX\}$ and the upper bound of the available processing engines $Count_{pe}$. Some combination of upper bounds are very unlikely to occur in real networks which limits the size of the design space for loop unroll factors. However, when considering the mapping of unrolled loops to an accelerator's spatial axis, the choice of unroll factors becomes more complicated. When two loops are unrolled in the same spatial axis, the effective unroll factor for each one of them may change when processing a layer with a different

convolution layer configuration than the one assumed when unrolling the loops. For example, consider the following situation. For a convolution layer with kernel size (3, 3) and channel count 32, if the kernel loops are unrolled fully with an unroll factor of 9, and the channel loops are unrolled partially with an unroll factor of 4, and they were both mapped to the same spatial axis, then the total number of processing engines allocated to that spatial axis would be 36. After allocating those PEs, if we attempt to execute a different convolution layer with a different configuration, for example, a (1, 1) convolution layer with 32 channels, the allocated PEs would be underutilized because the effective channel unroll factor would be 36 instead of 4 in the original configuration.

In this chapter we will first prune the dataflow design space dimensions in section 3.1 by determining the appropriate loop unroll targets and loop unroll factors using insights acquired from CIGAR a convolutional network analysis tool discussed in subsection 3.1.1. Then, given the complexity of exploring loop unroll factors and loop axis mapping, an automated accelerator Template optimizer TEMPO introduced in section 4.1 is used to explore the the aforementioned design space dimensions to produce utilization optimized accelerator template configurations.

## 3.1 Pruning the dataflow design space with CIGAR

### 3.1.1 CIGAR: The ConvolutIon statIstics GAtherer

#### 3.1.1.1 Algorithm

Pseudocode for CIGAR's algorithm is presented in algorithm 1. In algorithm 1, CIGAR begins by calling Collect_Library_Statistics after acquiring a dictionary of pytorch models $modell_{dict}$. Collect_Library_Statistics then instantiates an empty $model_{dict}^{stats}$ to be populated with model layer statistics . It then instantiates a collector object that acts as a container for collected model statistics. For each model, a new input image tensor is created based on the requirements of the model being analyzed. If no special transformations are required a default image tensor configuration is used where the width and the height dimension of the image is set 224x224 with 3 RGB channels. After an image tensor is created, Attach_Collection_Hooks is called to attach the collector object's extract_stats callback function or hook on each Conv2D layer present in the model's layers. Attach_Collection_Hooks returns an array to each attached hook to be later detached once the model under inspection is processed. An illustration of this process is given in Figure 3.1.

After all forward hooks are attached to the model, a forward pass of the model is performed. Model layer statistics are added to the $model_{dict}^{stats}$ and the collector's internal layer statis-

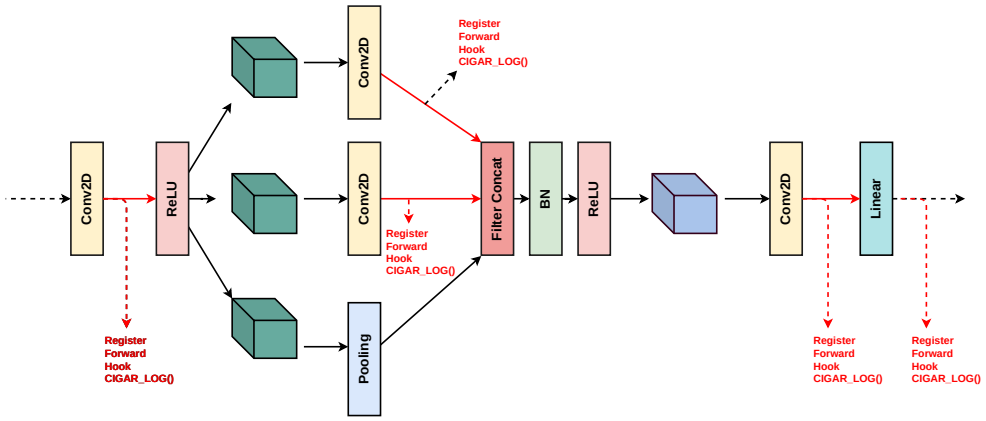



Figure 3.1: CIGAR attachment of forward hooks to all model convolution layers

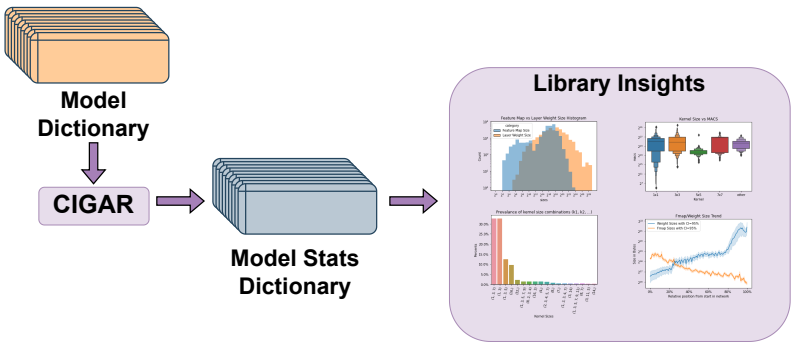



Figure 3.2: CIGAR extraction of convolution layers

tics tracker is reset. The layer statistics collected for convolution layers are 1) the kernel sizes 2) strides 3) any additional padding 4) the number of convolution groups 5) kernel dilation. For linear layers the input and output feature sizes are collected. For both layer types, input feature map dimensions are collected. After processing all of the models in $model_{dict}$, a $model_{dict}^{stats}$ is returned for further analysis used to derive the necessary library insights for pruning the dataflow design space. An illustration of that process is available in Figure 3.2. New layers can be analyzed by CIGAR provided that the collector is updated to be able to collect statistics from different layer types and Attach_Collection_Hooks is allowed to attach the collector's callback function to the newly supported layer.

**Algorithm 1** CIGAR

**Input:** $model_{dict}$

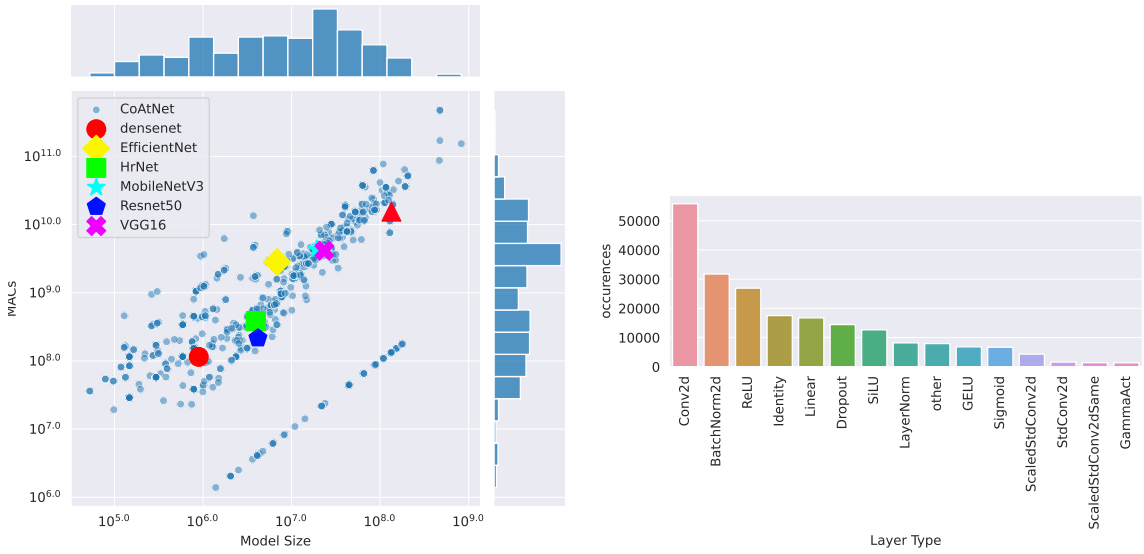
**Output:** $model_{dict}^{stats}$

**function** ATTACH_COLLECTION_HOOKS($model, collector$)
    $hooks \leftarrow []$
    **for** $layer \in model.named\_modules()$ **do**
        **if** $type(layer)$ is $conv2d$ or $type(layer)$ is $linear$ **then**
            $hooks.push(layer.register\_forward\_hook(collector.layer\_collector))$
        **end if**
    **end for**
    **return** $hooks$
**end function**

**function** COLLECT_LIBRARY_STATISTICS($model_{dict}$)
    $model_{dict}^{stats} \leftarrow \{\}$
    $collector \leftarrow Collector()$
    **for** $(model_{name}, model) \in model_{dict}$ **do**
        $input\_img\_tensor \leftarrow transform(open('default.jpg'), model)$
        $hooks \leftarrow Attach\_Collection\_Hooks(model, collector)$
        $model.forward(input)$
        $model_{dict}^{stats}[model_{name}] \leftarrow collector.model\_stats()$
        $collector.reset()$
        $hooks \leftarrow Detach\_Collection\_Hooks(hooks)$
    **end for**
    **return** $model_{dict}^{stats}$
**end function**

#### 3.1.1.2 Neural Network Library Explored

A diverse range of networks were explored by CIGAR for dataflow design space pruning. The diversity of networks is reflected in the diversity of model types, layer types, model sizes, and the number of MACs in the network. An illustration of the model sizes vs number of MAC




(a) (b)

Figure 3.3: Illustration of CIGAR's library diversity based on (a) Model sizes and number of MACS (b) Model layer types

diversity is presented in Figure 3.3. From Figure 3.3 it is clear that a wide range of models were selected as part of the CIGAR library explored. The library includes smaller models like squeezenet and mobilenetv2 as well as larger models like VGG16 [18]. In terms of layer diversity the library includes conventional networks with both convolution layers and linear layers as well as newer more exotic networks that combine transformer self attention layers with convolution layers like CoAtNet [23]. An illustration of that layer type diversity is reflected in figure Figure 3.3.b. A total of 695 networks where explored. The full list of networks explored is available in the appendix of this thesis. All models explored by CIGAR were implemented in pytorch and provided by either torchhub in [17] or the PyTorch Image Models (timm) package in [20].


(a) (b)

Figure 3.4: Exploration of data element reuse behavior in convolution layers of models from the TIMM library, a) shows overall reuse behavior as a boxplot b) shows reuse behavior trends within models with multiple convolution layers

### 3.1.2 Applying CIGAR to prune the dataflow design space

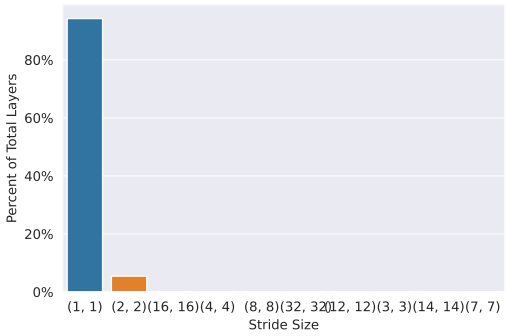
#### 3.1.2.1 Loop Unroll Targets

The choice of loop unroll targets affects the stationarity of the data elements in the convolution operation. For example, assuming a convolution layer with a kernel size of (2, 2), if we unroll the F, C, KY, KX loops by a factor of 2, weight element batches of size 16 will be loaded on to the chip in order to compute the output feature map. These weight will remain stationary until all input featuremap elements are loaded and consumed to evaluate the relevant partial sums associated with the filter weights loaded. In this scenario, the stationarity of the weight elements exceeds that of the input featuremap and output featuremap elements. We can determine determine what loop targets should be unrolled based on the stationarity of each date type present in the convolution operation. A data element that exhibits a high degree of stationarity should remain on chip for as long as possible in order to minimize excessive reloads from off chip memory. We can use the number of MAC operations a data element participates in as a surrogate for stationarity. A data elements element reused across many MAC operations should be kept on chip for as long as possible to avoid excessive reloading of that element from of-chip dram. Using CIGAR we can analyze data element reuse behavior in all models of the TIMM library for the three data elements present in the convolution operation, input feature maps elements (IFmaps), output feature map elements (OFmaps), and weight elements. The results from this analysis are present in Figure 3.4.

From Figure 3.4.a the reuse behavior between all three data elements is comparable with the exception OFmap reuse having a much lower 0.25 quantile. IFmap elements have a slightly higher reuse with the median MAC operations performed per element load equal to 256 MACs. Weight and OFmap elements exhibit lower reuse at 192 and 196 MACs per load. Reuse trends within networks show a general shift from high weight reuse to high IFmap and OFmap reuse depending on the relative position from start within a network. Weight reuse is initially almost 2 orders of magnitude higher than IFmap and OFmap reuse, however, since the shift in reuse behavior happens relatively early in most networks, higher IFmap reuse exceeding weight and OFmap reuse persists for more layers within an network. These findings indicate that all elements can benefit from stationarity depending on the network and even the layer position within a network. For an accelerator with a fixed dataflow the choice of dataflow and hence which loops to unroll is heavily influenced by the target networks expected to run on the accelerator. The most flexible dataflow in this case is a weight stationary dataflow in which the F, C, KY and KX loops are the unroll targets. This is due to the overlap between weight stationary under (1, 1) convolutions and GEMM discussed in subsection 3.2.1, the choice of a weight stationary dataflow lends itself well to GEMM given the similarities in the loop structure with regards to F and C loops for both applications. Furthermore, a weight stationary based dataflow allows support for linear layers through this overlap which are quite prevalent in many modern networks as seen in Figure 3.3.b. Given the flexibility of weight stationary, F, C, KY and KX loops will be the loop unroll targets. This choice of dataflow effectively creates two operational modes. A direct mode where convolution operations with kernels that are supported directly are executed on the accelerator and an GEMM mode where kernels that are not supported directly are supported by lowering/ lifting followed by conversion of the proceeding GEMM operation into a (1, 1) convolution. A full explanation of GEMM mode is given in section 3.2. Note that convolution layers with non (1, 1) strides are executed under GEMM mode. Supporting convolution layers with non (1, 1) strides directly is left as part of future work. From Figure 3.5 (1, 1) strides are represent 95% of convolution layers so the implications of indirect support of non (1, 1) strides will likely be negligible when assessing the overall performance and energy efficiency of an accelerator implementing the aforementioned operation modes.

#### 3.1.2.2 Loop Unroll Factors

There exists significant variation with regards to the kernel sizes present in the TIMM library. This makes the question of unroll factors and axis mapping for the KY and KX loops more




(a)

Figure 3.5: Percentage of total layers in the TIMM library's networks that have a stride size (k, k)

difficult to answer.

From Figure 3.6.a (1, 1) and (3, 3) kernel sizes dominate in comparison to all other kernel sizes. This renders the choice of keeping KY and KX loops folded impractical because if support is extended to an arbitrary K x K kernel while KY and KX loops are folded the onboard storage for weights would then need to be at least $K^2$ where $K$ is the upper bound of kernels supported directly to avoid excessive weight fetches from DRAM. Unfortunately, for 80% of the layers in the network, that additional storage area would be significantly underutilized by a factor of $\frac{1}{K^2}$ due to the overrepresentation of 1x1 kernels. To mitigate this under utilization of onboard memory for weights, KY and KX loops need to be unrolled fully. However, this begs the question, what kernel sizes should be assumed when unrolling the KY and KX loops? Any kernel sizes assumed when unrolling KY and KX loops become kernel sizes that are supported directly. Kernel sizes that are not assumed when unrolling KY and KX loops can be supported indirectly through a lowering/lifting approach similar to the ones discussed in chapter 2. This means that (1, 1) kernels are assumed when unrolling KY and KX loops, hence they have to be supported directly. Other kernel sizes to support directly can be derived from Figure 3.6. In Figure 3.6.b many networks contain at least 1 convolution layer that is not (1, 1) or (3, 3). For example a 7x7 kernel exists in around 20% of networks. The reason for the prevalence of 7x7 convolutions originates from the historical use of resnet [10] as a feature extractor for a significant portion of networks analyzed by CIGAR.

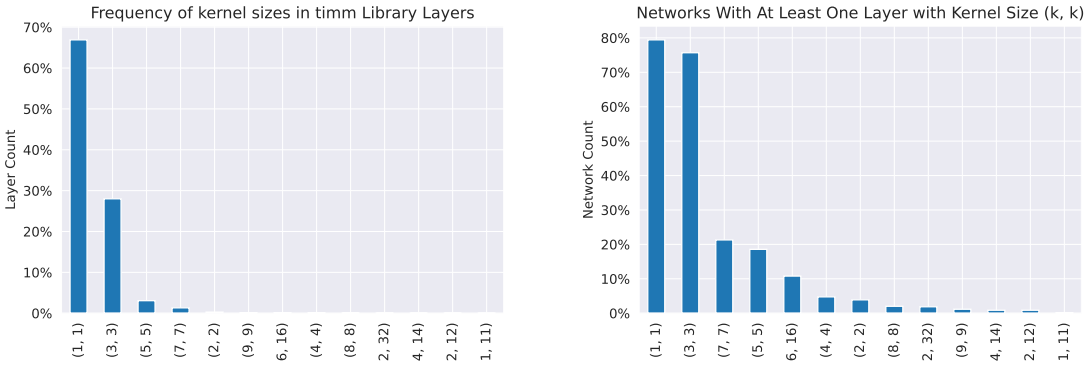
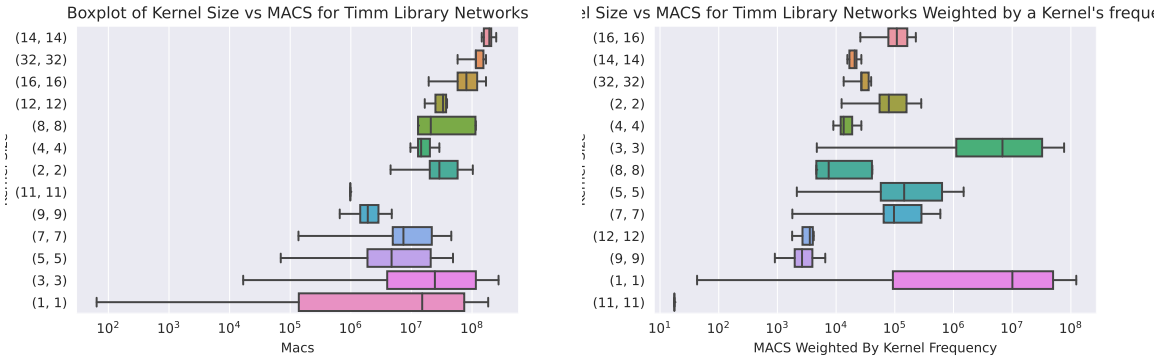



Figure 3.6: (a) Percentage of total layers in the TIMM library's networks that have a kernel size (k, k) (b) The percentage of networks in TIMM that have at least one kernel of size (k, k)

In Figure 3.7.a, when adjusting for the the number of MACs present in layers where these kernel sizes exist, (1, 1) and (3, 3) kernels share a similar computational burden on the network with (1, 1) having a much wider spread. 7x7 kernels have a much tighter spread but they still represent a similar computational burden to (1, 1) and (3, 3) kernels in networks where they are present. Adjusting for kernel frequency in Figure 3.7.b, (1, 1) and (3, 3) kernels dominate all other kernel sizes in terms of number of MACs in most network layers. From Figure 3.7 it is clear that (1, 1) and (3, 3) kernels need to be supported directly while all other kernels need to be supported indirectly through a lifting/lowering approach like those discussed in chapter 2. This limits the space of possible unroll factors for the loop unroll targets and thus prunes the dataflow design space. Supporting kernels indirectly will lead to an expansion of the IFmap due to the duplication introduced by lowering, however that expansions is negligible given the relative infrequency of non (1, 1) and (3, 3) layers. What remains of the dataflow design space under weight stationary is 1) the unroll factors for F and C loops and 2) the axis mapping for F, C, KY and KX loops for the 2 spatial axis of a convolution accelerator. To explore the what remains of the dataflow design space an automated dataflow exploration tool will be introduced and discussed in chapter 4.




(a) (b)

Figure 3.7: (a) Kernel size vs number of layer MACs (b) Kernel size vs number of layer MACs adjusted by kernel frequency

Depending on the chosen unroll factors, the architecture implementing the unroll factors in effect tiles the weight tensor and processes it tile by tile in the convolution operation. An illustration of this concept is present in Figure 3.8. Loop unroll factors determine PE allocation Tiling of a weight tensor arises from the processing of filters, channels and kernels in batches whose size depend on the unroll factors. Padding of a weight tensors is performed wherever the chosen PE binding for filter or channel loops exceeds the number of channels and filters being processed in the tile. In Figure 3.8 a weight tensor of dimension $R^{6\times3\times2\times2}$ is tiled with $F_{unroll} = 4$, $C_{unroll} = 8$, $K_{unroll} = 2$ with kernel loops mapped to the horizontal axis alongside channel loops. Additional padding in the horizontal and vertical axis is added given excess allocation of PEs in both spatial axis in all tiles except the top left one. With this representation, the accelerator processes the weight tensors as a series of tiles to produce an output featuremap. This representation of the weight tensor as a series of tiles processed by the architecture is useful when considering the scheduling of a convolution operation in a network. Tiling of weights and their effect on scheduling is discussed in chapter 6.

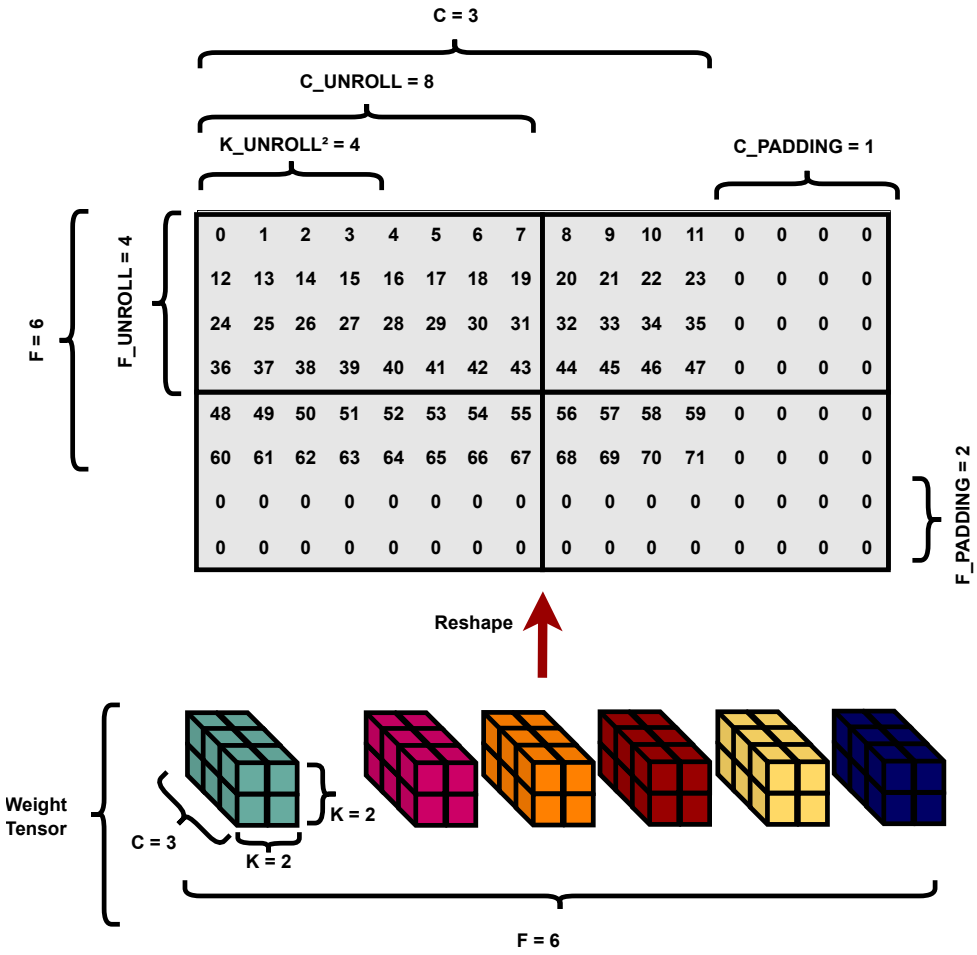



Figure 3.8: An illustration of weight tiling by loop unroll factors

## 3.2 Enabling GEMM mode through layer equivalence

GEMM mode extends support to convolution layers with kernel sizes that are not directly supported by an convolution accelerator. It does this by establishing a layer equivalence relationship between unsupported layers and supported layers. Convolution operations can be converted into general matrix multiplication through the use of lowering and lifting techniques like Im2col discussed in [2]. This approach to supporting arbitrary convolution operations would still require a matrix multiplication accelerator. To support matrix multiplication within a convolution accelerator we can either 1) repurpose existing compute and memory hardware on chip to perform both GEMM and convolutions operations or 2) we can establish a mathematical equivalence between GEMM operations and Conv operations by reinterpret GEMM into a special case of convolution, namely convolutions with a (1, 1) kernel size. This avoids the overhead of repurposing existing convolution hardware to support GEMM. Once this relationship between GEMM and (1, 1) Convolutions is defined, layers that are unsupported directly can be converted into equivalent layers that are supported directly. In subsection 3.2.1, a proof for the equivalence between GEMM and (1, 1) Convolutions is given. Additionally, in subsection 3.2.2 the aforementioned proof will be used in tandem with the approach in [2] to provide support for arbitrary convolution operations. Finally, a discussion of layer dimensionality for unsupported layers that have been converted to equivalent supported layers is given in subsection 3.2.3.

### 3.2.1 Functional equivalence between GEMM and (1, 1) Convolutions

We can establish the functional equivalence between GEMM and (1, 1) convolutions with the following proof. An illustration of this proof is given in Figure 3.9. Given two matrices $A \in \mathbb{R}^{Z \times C}$ and $B \in \mathbb{R}^{C \times F}$, let $R \in \mathbb{R}^{Z \times F} = A.B$. A different way to express the matrix multiplication $A.B$ is Equation 3.1.

$$R[z][f] = \sum_{c=0}^{C-1} A[z][c] \times B[c][f] \quad \forall z \in [0, n^2 - 1] \tag{3.1}$$

Transposing $A$ and $B$ yields $\hat{A} \in \mathbb{R}^{C \times Z}$ and $\hat{B} \in \mathbb{R}^{F \times C}$. Using the identity $(A.B)^T = B^T.A^T$ we can rewrite Equation 3.1 as Equation 3.2 where $\hat{R} \in \mathbb{R}^{F \times Z}$

$$\hat{R}[f][z] = \sum_{c=0}^{C-1} \hat{B}[f][c] \times \hat{A}[c][z] \quad \forall z \in [0, Z - 1] \tag{3.2}$$

We can reshape $\hat{A}$ and $\hat{B}$ using Equation 3.3 into 3D tensors by adding an additional dimension of size 1 for $\hat{A}$ and 2 additional dimensions of size 1 for $\hat{B}$.

$$\hat{A} \xrightarrow{Reshape} \hat{A} \in \mathbb{R}^{C \times Z \times 1} \quad \hat{B} \xrightarrow{Reshape} \hat{B} \in \mathbb{R}^{F \times C \times 1 \times 1} \tag{3.3}$$

Applying Equation 3.3 to Equation 3.2 yields Equation 3.4 where $\hat{R} \in \mathbb{R}^{F \times Z \times 1}$ remains the transposed output of $A.B$.

$$\hat{R}[f][z][0] = \sum_{c=0}^{C-1} \hat{A}[c][z][0] * \hat{B}[f][c][0][0] \quad \forall z \in [0, Z - 1] \tag{3.4}$$

Adding kernel summations to Equation 3.4 yields Equation 3.5 which is equivalent to a (1, 1) convolution of stride 1. To recover $R$ from $\hat{R}$ we can reshape $\hat{R}$ by removing the last dimension and then transpose it.

$$\hat{R}[f][z][0] = \sum_{c=0}^{C-1} \sum_{k_x=0}^{1} \sum_{k_y=0}^{1} \hat{A}[c][y + ky][x + kx] * \hat{B}[f][c][k_y][k_x] \quad \forall y \in [0, Z - 1] \wedge x = 0 \tag{3.5}$$

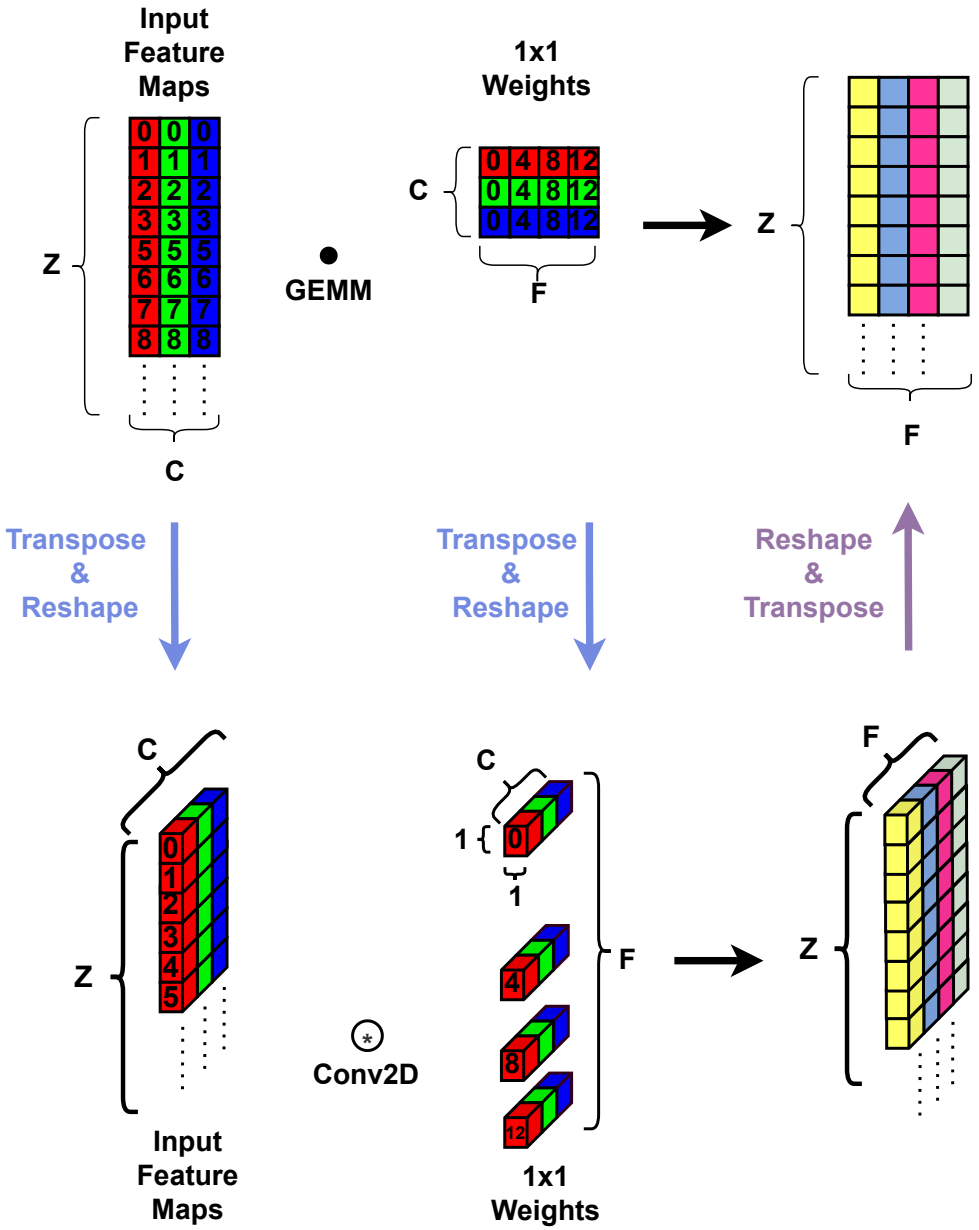



Figure 3.9: GEMM and (1, 1) Convolution Equivalence

### 3.2.2 Layer Equivalence

We can support previously unsupported kernel sizes by combining the GEMM to (1, 1) Conv conversion in subsection 3.2.1 with any tensor lowering/lifting approach in [2]. Lowering converts a convolution into a GEMM operation, and the approach in subsection 3.2.1 reinterprets that operation as another (1, 1) Convolution. This approach allows any convolution accelerator that can support (1, 1) convolution operations with asymmetric IFmaps to support any arbitrary convolution operation. A visual illustration for this technique is presented in Figure 3.10. To demonstrate this approach we begin by lowering both IFmap using **??** and Weights using Equation 2.4. This results in two matrices IFmap and Weights in Equation 3.6. Lowering should be performed if the kernel size of the Weight tensor is unsupported $K' \notin \{SupportedKernels\}$.

$$\begin{aligned} IFmap \in R^{C\times n\times n} &\xrightarrow{BalancedLowering} \hat{IFmap} \in R^{nm\times K'C} \\ Weight \in R^{F\times C\times K'\times K'} &\xrightarrow{BalancedLowering} \hat{Weight} \in R^{K'C\times K'F} \end{aligned} \tag{3.6}$$

After lowering both tensors, we apply the transformations in Equation 3.7 to reinterpret the anticipated GEMM operation that occurs after lowering into a (1, 1) convolution operation. The transformations are composed of a transpose operation followed by a reshape operation that appends additional dimensions of size 1 to both IFmap and Weights. The transformations yields two new tensors $\hat{IFmap}$ and $\hat{Weight}$.

$$\begin{aligned} \hat{IFmap}^T \in R^{K'C\times nm} &\xrightarrow{Reshape} \hat{IFmap} \in R^{K'C\times nm\times 1} \\ \hat{Weight}^T \in R^{K'F\times K'C} &\xrightarrow{Reshape} Weight \in R^{K'F\times K'C\times 1\times 1} \end{aligned} \tag{3.7}$$

After performing the transformations in Equation 3.7 the output $\hat{OFmap}_{prelift}$ can be calculated after performing a (1, 1) convolution in Equation 3.8 using the $\hat{IFmap}$ and $\hat{Weight}$ tensors.

$$\hat{OFmap}_{prelift} \in R^{K'F\times nm\times 1} = \hat{IFmap} * \hat{Weight} \tag{3.8}$$

Finally we can lift $\hat{OFmap}_{prelift}$ by first reshaping it into a 2D matrix by dropping the last dimension and then transposing it. After that, we can apply balanced lifting in **??** to get the final OFmap in Equation 3.9.

$$\begin{aligned} \hat{OFmap}_{prelift} \in R^{K'F\times nm\times 1} &\xrightarrow{Reshape} OFmap_{prelift} \in R^{K'F\times nm} \\ OFmap^T_{prelift} \in R^{nm\times FK} &\xrightarrow{BalancedLifting} OFmap \in R^{F\times m\times m} \end{aligned} \tag{3.9}$$

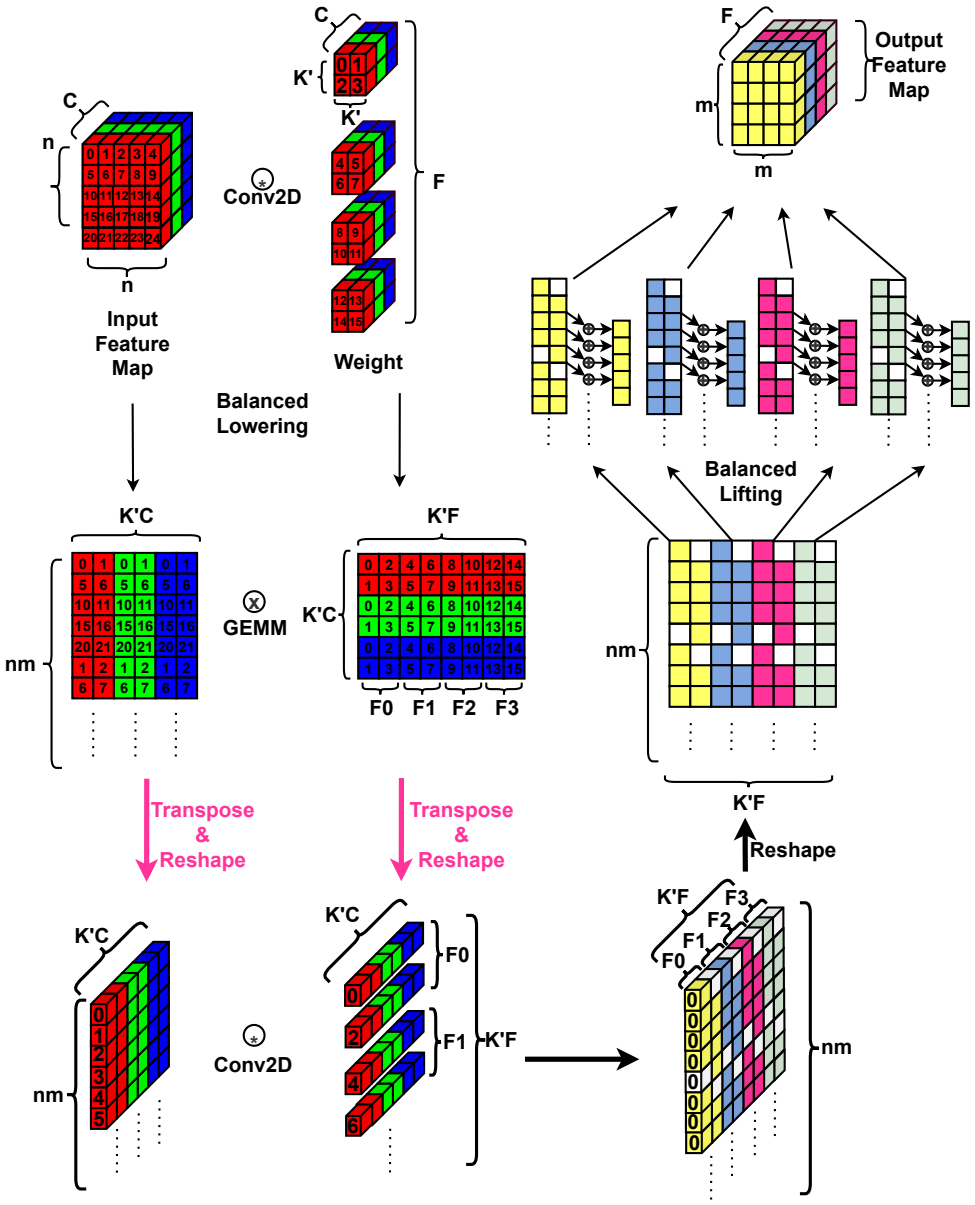



Figure 3.10: Illustration of approach Conv2Gemm2Conv approach

### 3.2.3 Layer Dimensionality

Whether or not a convolution layer's kernel size is supported directly has an effect on the assumed dimensionalities of the IFmap and Weight tensors as well as the kernel loops unroll factor K. Kernels are assumed to be symmetric so both kernel KY and KX loops and share a single unroll factor $K_{unroll}$. If the convolution layer's kernel size is supported directly no changes are assumed to have been made to the dimensionality of the IFmap. The kernel unroll factor is then equal to the kernel size of the layer in accordance with the conclusions drawn from subsection 3.1.2. However, if a convolutions layer's kernel size is not supported directly, the layer's kernel size is converted to (1, 1) as seen in Equation 3.10 and lowering is assumed to have been performed on IFmap and Weight tensors in accordance with the approach discussed in subsection 3.2.2. For a convolution layer with IFmap dimensionality $R^{C\times n\times n}$, Weight dimensionality $R^{F\times C\times K\times K}$ and OFmap dimensionality $R^{F\times m\times m}$ the layer's new filter count $\hat{F}$, channel count $\hat{C}$, and IFmap channel size $\hat{Z}$ values are reflected in Equation 3.11.

$$K_{unroll} = \begin{cases} K & K \in \{SupportedKernels\} \\ 1 & K \notin \{SupportedKernels\} \end{cases} \tag{3.10}$$

$$\begin{aligned} \hat{C} &= \begin{cases} C & K \in \{SupportedKernels\} \\ CK & K \notin \{SupportedKernels\} \end{cases} \\ \hat{F} &= \begin{cases} F & K \in \{SupportedKernels\} \\ FK & K \notin \{SupportedKernels\} \end{cases} \\ \hat{Z} &= \begin{cases} m^2 & K \in \{SupportedKernels\} \\ nm & K \notin \{SupportedKernels\} \end{cases} \end{aligned} \tag{3.11}$$

### 3.2.4 Accelerator Spatial Axis mapping

When determining axis mapping, F and C loops are assumed to be bound to an accelerator's vertical and horizontal spatial axis. Utilizing both axis results in better overall on-chip area utilization assuming conventional 2 dimensional constraints for chip fabrication. Mapping all loops to the same axis can provide the most flexibility with regards to the allocation of PEs as a resource to process different filters, channels and kernels. However, this complicates on chip connectivity. KY and KX loops can then be bound to either the horizontal or vertical axis of an accelerator based

on the variable $K_{axis}$. Depending on which axis KY and KX loops are mapped, the effective unroll factors for F and C loops ($F_{eff}$ and $C_{eff}$) are changed. If the unrolled kernel loops share the same axis as the C loops, the effective $C_{unroll}$ factor for the C loops is then $\lfloor \frac{C_{unroll}}{K_{unroll}} \rfloor$ which means the effective C unroll factor decreases depending on the size of the kernel unroll factor. This decrease in effective unroll factor arises from the fact that, within the same axis as the C loops, compute resources are allocated to process a single $K^2_{unroll}$ kernel. This results in a decrease of compute resources available to process other channels concurrently. The same logic applies to F loops if the Kernel loops are mapped to the same axis vertical axis as they are. This idea is presented in Equation 3.12 and Equation 3.13. In both equations $F_{unroll}$ and $C_{unroll}$ are the unroll factors for F and C loops assuming $KY = KX = K = 1$.

$$C_{eff} = \begin{cases} \lfloor \frac{C_{unroll}}{K^2_{unroll}} \rfloor & K_{axis} = horizontal \\ C_{unroll} & K_{axis} = Vertical \end{cases} \tag{3.12}$$

$$F_{eff} = \begin{cases} \lfloor \frac{F_{unroll}}{K^2_{unroll}} \rfloor & K_{axis} = Vertical \\ F_{unroll} & K_{axis} = Horizontal \end{cases} \tag{3.13}$$

### 3.2.5 Overhead of lowering and lifting

Lowering and lifting introduce additional overheads with regards to latency and tensor sizing. The latency for performing balanced lowering and lifting is $m^2K$ for a layer with a Weight tensor $\mathbb{R}^{F \times C \times K \times K}$ and an OFmap tensor $\mathbb{R}^{F \times m \times m \times}$. While lowering and lifting can be performed by a convolutions accelerator, in this thesis it is assumed that a software processor on the same chip performs these operations. Lowering also introduces duplicate data elements in the IFmap tensor thus increasing it's overall size.

To enable GEMM operations using the approach in subsection 3.2.1 both input and output matrices are transposed and reshaped. All reshape operations discussed in this chapter add a dimension of size 1 to the data and they incur no data reorganization overhead. Additionally, all transpose operations are assumed to be performed during transfer to and from accelerator on-chip and thus incur no latency penalty. A discussion of how transfers to and from on-chip memory can mask the latency of transposing matrices is left as part of future work along with incorporating lowering and lifting into the accelerator.

## 3.3 Exploring The Hardware Implementation Design Space

Based on the conclusions derived from section 3.1, weight stationary is the most flexible dataflow choice given the overlap between 1x1 convolutions and GEMM discussed in section 3.2. This gives rise to a weight stationary dataflow based accelerator with two operational modes, direct Mode where a subset of possible kernel sizes are supported and GEMM mode where all other kernel sizes and strides are supported in via lowering/ lifting based approach. In this section we will determine an appropriate hardware implementation for this accelerator using the hardware implementation taxonomy from [13]. Based on the reuse and communication behavior of the different elements (IFmap, OFmap and weights) in a convolution operation using weight stationary we can infer the appropriate hardware implementation for on-chip communication and memory from [13]. To perform this deduction we will use the polyhedral model to analyze temporal reuse in subsection 3.3.1 and spatial reuse in subsection 3.3.2 of data elements in a convolution operation. Based on the analyzed reuse behavior an initial hardware implementation for HERO will be given and further improved after applying a simplification of the on-chip memory hierarchy in subsection 3.3.3. The final hardware implementation for HERO will be given in section 3.4. Note that the implementation in section 3.4 will serve as template to be further tuned based on a library of target networks in chapter 4.

### 3.3.1 Temporal Reuse Analysis

Unrolling convolution dataflow loops yield multiple instances of the Multiply and Accumulate (MAC) statement present in the original convolution nested loops in Listing 2.1. These statements represent Processing Engine (PE)s performing MAC operations concurrently. MAC statement instances can be distinguished from each other based on the memory access offsets that exist in them as a result of unrolling filter, channel and kernel loops. For unroll factors F_T for filters, C_T for channels, KY_T and KX_T for kernels each statement will have a corresponding access offset based on the statement index $j \in [0, F\_T * C\_T * KY\_T * KX\_T]$ for each of the data elements (IFmap, OFmap and Weights) accessed in the loop body. Each MAC statement at index j is characterized by a set of access offsets Fj, Cj, KYj, KXj used by the memory accesses in the MAC statement. Applying the unroll factors and distinguishing each MAC statement based on it's statement index j yields the loop configuration in Listing 3.1.

Listing 3.1: Fully unrolled convolution dataflow loops

```
1    for(int f = 0; f < F; f+=F_T) // Filter loop
2        for(int c = 0; c < C; c+=C_T) // Channel loop
3            for (int y = 0; y < Y; y++) // FeatureMap Height
4                for(int x = 0; x < X; x++) // FeatureMap Width
5                        ...
6                        /* For all j in [0, F_T*C_T*KY_T*KX_T[ */
7                        O[f+Fj][y][x] += W[f+Fj][c+Cj][KYj][KXj]* \
8                                         I[c][y+KYj][x+KXj]
9                        ...
```

Each MAC statement is composed of three separate memory accesses for IFmap, OFmaps and weights. For each of those memory access has a temporal index (it's location in time) defined by the iteration domain vector $[f, c, y, x, ky, kx]$. A mapping exists between each iteration domain vector and MAC statement's memory accesses. Temporal reuse analysis for each of the memory accesses in the MAC statements is performed on the loops in Listing 3.1. The different operational modes (Indirect/ Direct) are analyzed concurrently using the same loop representation as they only differ based on whether we set the width loop upper bound to 1, and set the kernel loops upper bounds to 1. Since kernel loops are always unrolled fully this sets KY_T and KX_T to 1. We can analyze temporal reuse in the dataflow represented in Listing 3.1 by adapting the approach in [16] to the aforementioned dataflow iteration domain and access functions. Given iteration domain restrictions imposed by the polyhedral model, Listing 3.2 assumes unroll factors F_T = C_T = 4. Setting F_T and C_T to concrete values does not overall reuse behavior during temporal reuse analysis.

Listing 3.2: Polyhedral analysis of reuse in iscc for convolution loops

```
1  // Define iteration domain for all accessed data elements
2  ID:=[F, C, Y, X] -> { S[f, c, y, x] : 0<=f<F and 0<=c<C and f mod 4=0 and c
       mod 4=0, 0<=y<Y and 0<=x<X};
3  // Define access functions for each data element
4  IFmap:=([Cj, KYj, KXj] -> {S[f, c, y, x] -> IF[c+Cj][y+KYj][x+KXj]})*ID;
5  OFmap:=([Fj] -> {S[f, c, y, x] -> PS[f+Fj][y][x]})*ID;
6  WEIGHT:=([Fj, Cj, KYj, KXj] -> {S[f, c, y, x] -> W[f+Fj][c+Cj][KYj][KXj]})*
       ID;
7  // Evaluate temporal reuse
8  IFmap_REUSE:=(IFmap.(IFmap^-1))*(ID<<ID);
9  OFmap_REUSE:=(OFmap.(OFmap^-1))*(ID<<ID);
10 WEIGHT_REUSE:=(WEIGHT.(WEIGHT^-1))*(ID<<ID);
```

In Listing 3.2, the iteration domain for the loops in Listing 3.1 is converted into it's set representation in line 2 where for some access statement S the loop iteration vector [f, c, y, x,] is bound by the upper and lower bounds [0, F], [0, C], [0, Y], [0, X] respectively. These bounds are represented by the associated parameters passed to the iteration domain set assignment in line 2.

Each memory accessed for IFmaps, OFmaps and weights in each MAC statement has an associated memory access function.

Each instance of the loop iteration vector [f, c, y, x] is mapped to a memory access for each of the memories in lines 4-6. Access offsets used in the memory access functions are passed as parameters based on the convention established in Listing 3.1. This mapping creates multiple temporal instances for each memory access in each MAC statement instance. For example, for example, the OFmap access that occurs at iteration vector [f = 2, c= 1, y = 0, x = 1] is a different temporal instance of the same OFmap access at [f = 1, c = 1, y = 0, x = 1]. Two accesses that access the same index but at different iteration vectors are different temporal instances of the same access. After applying the operation in lines 8-10, we can determine the temporal reuse behavior of the accessed memories in the convolution loops. Listing 3.3 shows the reuse behavior for each memory. Original iteration domains constraints are omitted for brevity. The operation in lines 8-10 map all iteration domains to all proceeding iteration domains that access the same memory locations for each of the data elements.

Listing 3.3: Polyhedral analysis results w.r.t data elements in convolution loops

```
IFmap_REUSE;
[F, C, Y, X, Cj, KYj, KXj]->{
    S[f, c, y, x] -> S[f', c' = c, y' = y, x' = x] :
        ... f' > f and 0 <= f' < F ...
    }
OFmap_REUSE;
[F, C, Y, X, Fj]->{
    S[f, c, y, x] -> S[f' = f, c', y' = y, x' = x] :
        ... c' > c and 0 <= c' < C ...
    }
WEIGHT_REUSE;
[F, C, Y, X, Fj, Cj, KYj, KXj] -> {
    S[f, c, y, x] -> S[f' = f, c' = c, y', x'] :
        ... y' > y and 0 <= y' < Y and 0 <= x' < X ...;
}
```

Listing 3.3 shows the temporal reuse behavior in memory accesses. For each of the memories accessed (IFmap, OFmap and Weights) there exists a set of reuse (IFmap_REUSE, OFmap_REUSE and WEIGHT_REUSE) maps that map each iteration vector of an access to all the proceeding iteration vectors where that same access occurs. From the above listing we can see that, in the set of IFmap reuse maps (IFmap_REUSE), IFmap channels are reused temporally with respect to filter loops. For a given IFmap accessed at channel c, that channel is accessed again when computing the output for all proceeding filter loop iteration f' where $f' > f$. The absence of other mappings in the set of reuse maps IFmap_REUSE shows that 1) this reuse behavior holds at any arbitrary iteration vector [f, c, y, x] and 2) this reuse behavior depends only on the filter loop. For the set OFmap reuse maps (OFmap_REUSE), for an OFmap access at iteration vector [f, c, y, x], it is accessed again at loop iteration f'=f, c', y'=y, x'=x where $c' > c$. For (WEIGHT_REUSE) Weights exhibit temporal reuse w.r.t feature map width and height, the X and Y loops.

Applying the hardware taxonomy in [13], IFmap exhibits temporal reuse, multicast communication given their repeated read only behavior. OFmap exhibits temporal reuse, reduction communication given their read-modify-write behavior. Weights exhibit temporal multicast communication. Given the limited implementation options derivable from temporal reuse we can comfortably define the appropriate connectivity and memory hierarchies for IFmaps channels, OFmaps channels (equivalent to number of Filters), and Weights. The beginnings of a hardware template derived from the aforementioned temporal reuse behavior of the different memories referenced in

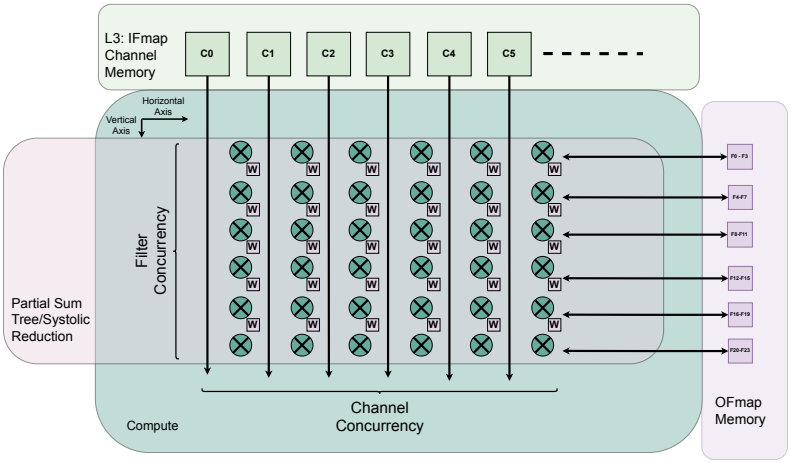



Figure 3.11: Initial hardware template incorporating buffers IFmap and OFmap temporal reuse

the convolution dataflow can be seen in 3.11. In 3.11 the template is broken into 3 major components. The first is the IFmap memory hierarchy currently with only 1 level. The 2nd component is the compute portion of the template where partial sums are computed and aggregating into OFmap data elements. Finally the 3rd component which is the OFmap memory that stores OFmap partial sums until they are aggregated into OFmap pixels and are written back to memory.

In addition to the temporal reuse behavior exhibited across IFmap channels, temporal reuse exists within individual IFmap channels due to the stencil based access pattern arising from the X, Y, KY, KX loops in the dataflow. That temporal reuse is affected by the decision to fully unroll kernel loops which causes temporal reuse to exist between unrolled different PEs processing the same kernel. Proof of the existence of that temporal reuse is given in the polyhedral analysis in Listing 3.4.

Listing 3.4: Analysis of IFmap channel reuse

```
1  ID_XY:=[Y, X, KY, KX] -> { S[y, x, ky, kx] : 0<=ky<KY and 0<=kx<KX and 0<=y<
       Y and 0<=x<X};
2  IFmap_XY:=({S[y, x, ky, kx] -> IF[y+ky][x+kx]})*ID;
3  IFmap_REUSE_XY:=(IFmap_XY.(IFmap_XY^-1))*(ID_XY<<ID_XY);
4  IFmap_XY_REUSE;
5  [Y, X, KY, KX] -> {
6      S[y, x, ky, kx] -> S[y', x', ky' = (y - y') + ky, kx' = (x - x') + kx] :
7          ... y' > y and (y + ky) -KY < y' <= (y + ky) and (x + kx) -KX < x'
               <= (x + kx) ...;
```

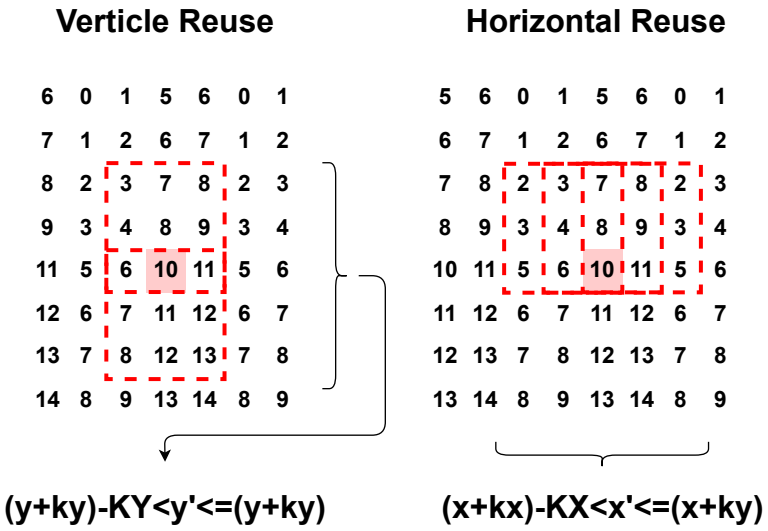



Figure 3.12: IFmap Reuse Behavior w.r.t individual feature map channels

8 }

Within an individual IFmap channel, temporal reuse is exhibited w.r.t X and Y loops. Given the complexity of the domain constraints of IFmap_XY_REUSE in Listing 3.4, an illustration of the reuse behavior is available in Figure 3.12. In Figure 3.12, individual pixels within the kernel are reused based on the position of the sliding window or stencil of the convolution in an IFmap channel. There are two primary directions where that reuse is exhibited, vertical and horizontal with an IFmap channel. The loops that control the vertical and horizontal stencil position in the IFmap are the Y and X loops in the dataflow. Because kernel loops are fully unrolled, the temporal reuse exhibited in Listing 3.4 occurs across different PEs processing the unrolled kernel. To determine the appropriate memory infrastructure to support that stencil based access pattern, we can apply the technique in [16] to construct a reuse chain that moves reused data between different PEs. The advantage of using a reuse chain is that the temporal reuse that exists within an IFmap channel is relegated to a smaller memory with lower memory access cost.

[16] constructs a reuse chain for applications with a sliding window access pattern that connects each unrolled kernel port with it's neighbors using a FIFO or a shift register. If the temporal reuse distances between the accesses of neighboring PE ports are constant [16] uses a shift register, otherwise they use a FIFO. The reuse distance between accesses of neighboring ports are then converted into storage of the same size. So if two ports share the same data but with a lag

of 2 iterations in the iteration domain they're operating in, then a shift register of size 2 can be placed between them. Similar to the sliding window application explored in [16] the reuse distances between the PEs processing the unrolled kernel in the convolution dataflow are also constant. To determine the reuse distances necessary between ports we can apply the analysis in Listing 3.5 adapted from [16] to determine the sizing of the buffers in the reuse chain for IFmap accesses within a channel. The analysis in Listing 3.5 assumes a kernel size of (3, 3) based on the conclusions of subsubsection 3.1.2.2. Note that (1, 1) kernels exhibit no temporal reuse within the kernel loops.

Listing 3.5: Determining buffer sizes in 3x3 convolutions

```
1   ID:=[IFmap_Y, IFmap_X] -> {S[y,x]:y>=0 and x>=0 and y<=IFmap_Y-3 and x<=
        IFmap_X-3};
2   A0:=[IFmap_Y, IFmap_X] -> {S[y,x]->A[y+0,x+0]}*ID;
3   A1:=[IFmap_Y, IFmap_X] -> {S[y,x]->A[y+0,x+1]}*ID;
4   A2:=[IFmap_Y, IFmap_X] -> {S[y,x]->A[y+0,x+2]}*ID;
5   A3:=[IFmap_Y, IFmap_X] -> {S[y,x]->A[y+1,x+0]}*ID;
6   ...
7   A8:=[IFmap_Y, IFmap_X] -> {S[y,x]->A[y+2,x+2]}*ID;
8
9   R10:=(lexmin ((A1.A0^-1)*(ID<<ID)));
10  R21:=(lexmin ((A2.A1^-1)*(ID<<ID)));
11  R32:=(lexmin ((A3.A2^-1)*(ID<<ID)));
12  ...
13  R87:=(lexmin ((A8.A7^-1)*(ID<<ID)));
```

In Listing 3.5, the iteration domain for the YX loops are defined as functions of the IFmap dimensions passed as parameters (line 1). The unrolled kernel loop IFmap accesses are then described using access maps that map the iteration vector [y,x] to the associated IFmap access (lines 2-7). Notice that the accesses are described as constant offsets added to access iterators y and x. These constants represent the kernel loop iterators ky, and kx that are now unrolled. For each neighboring pair of ports accessing the IFmap we can determine the reuse behavior in (lines 9-10). Operations in lines (9-13) map iterations where a port accesses a data element in IFmap with the earliest next iteration in which the neighboring port accesses that same data element. The distance between the accesses is then used as the reuse buffer size. The results of the analysis are presented in Listing 3.6.

Listing 3.6: Polyhedral analysis of reuse in iscc for convolution loops

```
1  R10;
2  $1 := [IFmap_Y, IFmap_X] -> {
3      S[y, x] -> S[y' = y, x' = 1 + x] :
4          0 <= y <= -3 + IFmap_Y and 0 <= x <= -4 + IFmap_X
5  }
6  R21;
7  $2 := [IFmap_Y, IFmap_X] -> {
8      S[y, x] -> S[y' = y, x' = 1 + x] :
9          0 <= y <= -3 + IFmap_Y and 0 <= x <= -4 + IFmap_X
10 }
11 R32;
12 $3 := [IFmap_Y, IFmap_X] -> {
13     S[y, x] -> S[y' = 1 + y, x' = -2 + x] :
14         0 <= y <= -4 + IFmap_Y and 2 <= x <= -3 + IFmap_X
15 }
16 ...
17 R87;
18 $8 := [IFmap_Y, IFmap_X] -> {
19     S[y, x] -> S[y' = y, x' = 1 + x] :
20         0 <= y <= -3 + IFmap_Y and 0 <= x <= -4 + IFmap_X
21 }
```

In 3.6, reuse distances between neighboring ports depend on the relationship between the ports and whether their access offsets are in the same row of the stencil or not. If two neighboring ports have unequal ky offsets the reuse distance between them is IFmap_X-3. If two neighboring ports have an equal ky offset the reuse distance is 1. An example of the first case is lines 11-15 where the reuse distance between port 2 and port 3 is IFmap-3. The evidence of that is that for any data accessed at port 3 with iteration vector y, x that same data is accessed at port 2 at iteration vector [y+1, x-2]. Based on the lexicographic ordering of iteration vector [y, x] and [y+1, x-2], the distance between those two vectors is IFmap_X-3, or in terms of OFmap dimensions X-1. Applying the same analysis to two ports in the same row (R10, R21, R45, R87, ...) yields a reuse distance of 1 as evidence by the iteration vectors of access [y, x] and [y, x+1] in all of the aforementioned neighboring port pairs.

Applying the results of the analysis in Listing 3.6 with the previous template Figure 3.11 results in the updated template Figure 3.13.

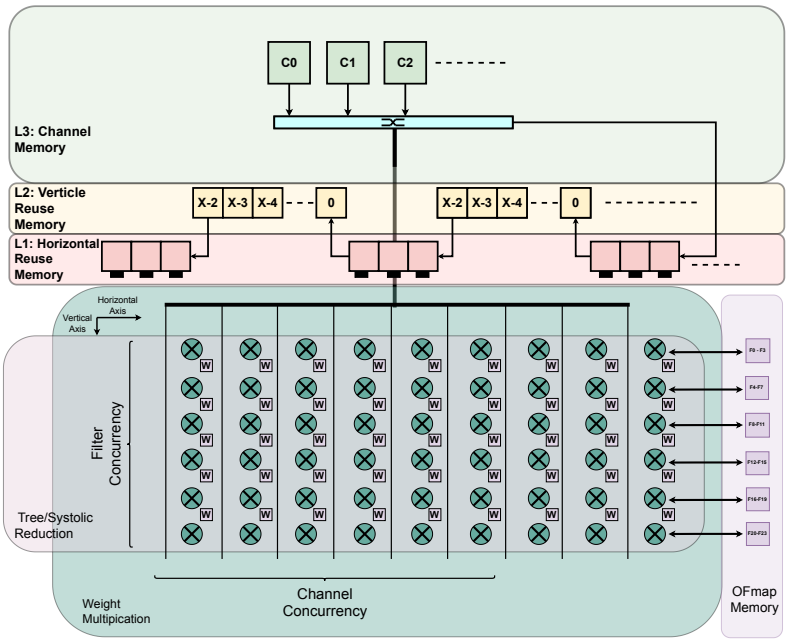



Figure 3.13: Hardware template incorporating a reuse chain for reuse within an IFmap channel

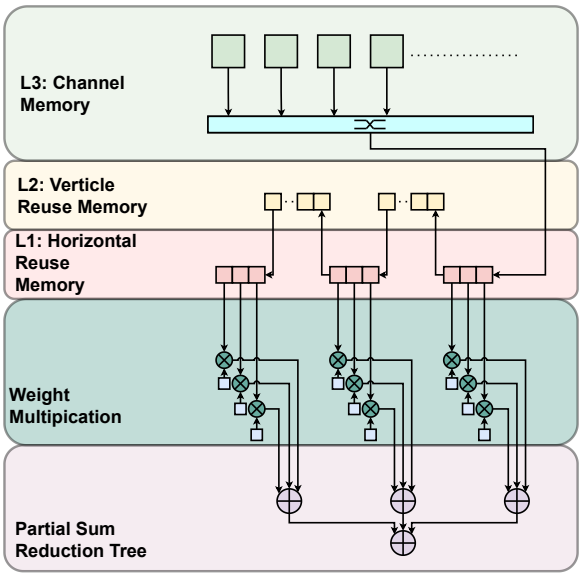
### 3.3.2 Spatial Reuse Analysis

Each MAC statement in the unrolled loop body has an associated j index. In the loop body there exists duplicate memory accesses across individual MAC statements. Those duplicate accesses are highlighted in Listing 3.7 and they are the origin of spatial reuse in the dataflow. IFmaps exhibit spatial reuse with multicast communication w.r.t to filter loops. OFmap exhibit spatial reuse with reduction communication w.r.t to channel loops. Weights exhibit no spatial reuse

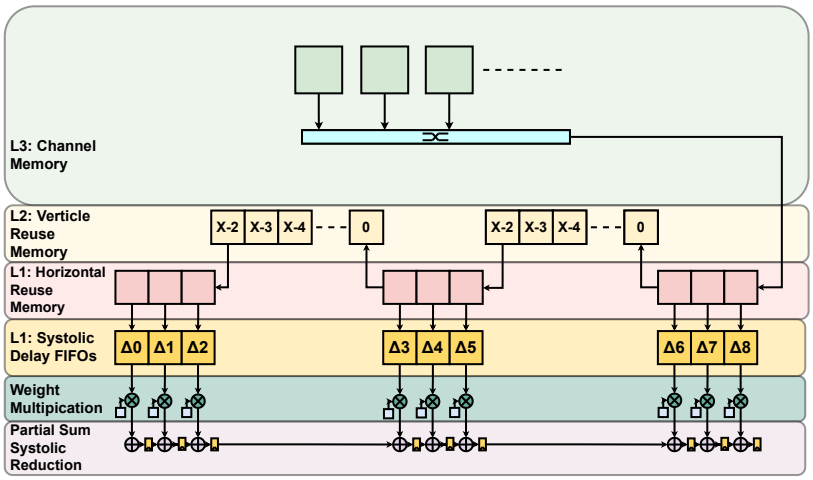
Listing 3.7: Spatial reuse in fully unrolled kernel loops

```
1  for(int f = 0; f < F; f+=F_T) // Filter loop
2      for(int c = 0; c < C; c+=C_T) // Channel loop
3          for (int y = 0; y < Y; y++) // FeatureMap Height
4              for(int x = 0; x < X; x++) // FeatureMap Width
5              {
6                  O[f+0][y][x] += W[f+0][c+0][0][0] * \
7                                      I[c+0][y+0][x+0]; // j=0
8                  O[f+0][y][x] += W[f+0][c+0][0][1] * \
9                                      I[c+0][y+0][x+1]; // j=1
10                 O[f+0][y][x] += W[f+0][c+0][0][2] * \
11                                     I[c+0][y+0][x+2];  // j=2
12                 O[f+0][y][x] += W[f+0][c+0][1][0] * \
13                                     I[c+0][y+1][x+2];  // j=3
14                 ...
15                 O[f+1][y][x] += W[f+1][c+0][0][0] * \
16                                     I[c+0][y+0][x+0]; // j=C_T*KY_T*KX_T
17                 O[f+1][y][x] += W[f+1][c+1][0][1] * \
18                                     I[c+0][y+0][x+1]; // j=C_T*KY_T*KX_T+1
19                 ...
20                 O[f+F_T-1][y][x] += W[f+F_T-1][c+C_T-1][KY_T-1][KX_T-1] * \
21                                         I[c][y+KY_T-1][x+KX_T-1];
22                                                 // j=F_T*C_T*KY_T*KX_T-1
23
24             }
```

Applying the taxonomy in Figure 2.6 to data elements that are spatially reused, IFmap channels that are spatially reused across unrolled filter loops can be broadcast with a bus. The reuse chain discussed in subsection 3.3.1 can be thought of as a Store&Forward scheme to deliver individual IFmap channel data elements to the PEs for reduction into OFmaps.Weights reused for channel iteration and are discarded. They exhibit no spatial reuse, just temporal. Therefore they should be kept in small on chip buffers, preferably close to the computation they are used in. OFmap exhibit spatial reuse across concurrent channels as well as temporal reuse across channel sets as discussed in subsection 3.3.1. A reduction tree as in Figure 3.14.a or a systolic array reduce and fwd as in Figure 3.14.b are both possible assuming no restrictions arising from synthesis. Combining the reuse chain derived in subsection 3.3.1 with the required systolic delays yields a simplification to the L1 memory present in Figure 3.13. This simplification is discussed in subsection 3.3.3.




(a)




(b)

Figure 3.14: Illustration of different partial sum reduction styles assuming kernel size is (3, 3) (a) Tree Reduction (b) Systolic array reduction

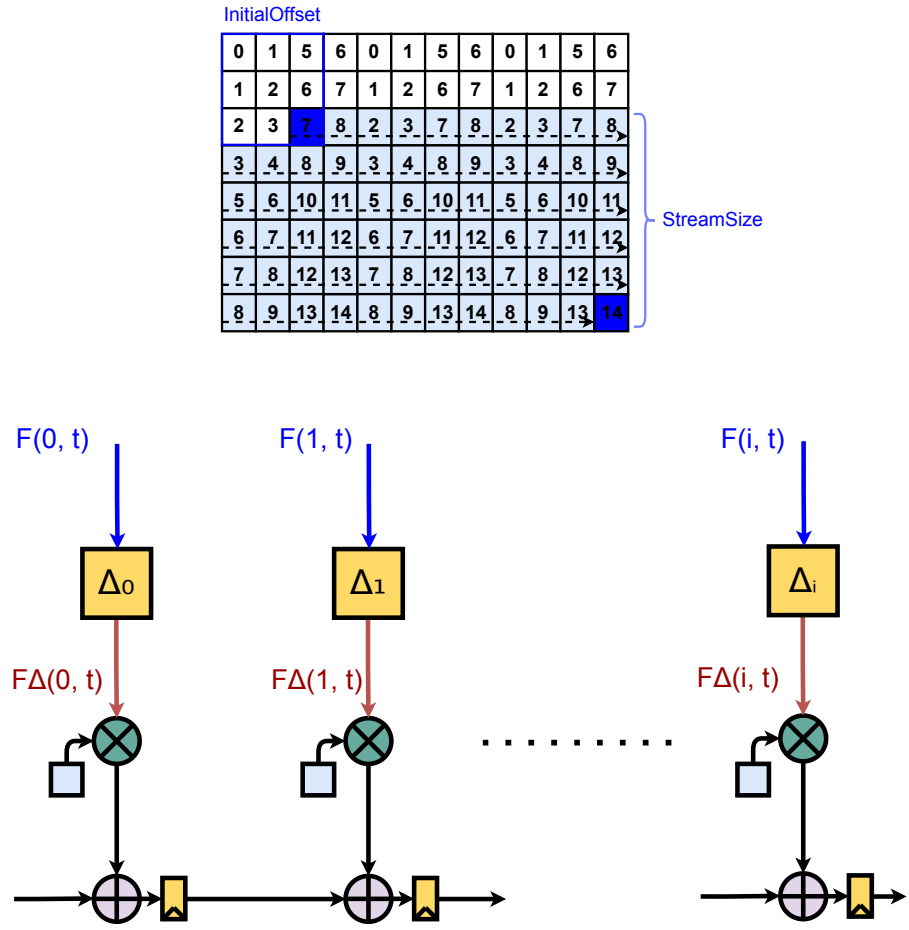



Figure 3.15: Reinterpretation of IFmap memory hierarchy outputs as a stream function

### 3.3.3 Simplifying the memory hierarchy

We can reinterpret the accesses made by the IFmap memory hierarchy in Figure 3.13 as a stream function $F(i, t)$ whose output produces element from an IFmap channel. The variable $i$ is the port index of the IFmap hierarchy and $t$ is the time in cycles since the beginning of the convolution operation. A representation of this reinterpretation of the accesses made in the IFmap memory hierarchy can be seen in Figure 3.15. Since it's always assumed that in direct mode kernel loops are unrolled fully the number of ports into the IFmap memory hierarchy is always a multiple of $K^2$ where $K$ is the size of the kernel being processed in direct mode.

$$IFmap \in R^{C\times n\times n} \xrightarrow{Reshape} IFmap \in R^{1\times Cn^2} \tag{3.14}$$

$$Weight \in R^{F\times C\times K\times K} \tag{3.15}$$

$$F(i,t) = \begin{cases} IFmap_{A(i,t)} & 0 <= t < StreamSize \\ 0 & else \end{cases} \tag{3.16}$$

$$StreamSize = n(n-K) + (n-K) \tag{3.17}$$

$$A(i,t) = InitialOffset(i) + t \tag{3.18}$$

Each data element streamed from the IFmap depends on an access function that also takes the same variables $i$ and $t$. Depending on the port index $i$ the access function for each port is composed of an initial offset in the IFmap and the current cycle count $t$. A total of $StreatSize$ elements are streamed the IFmap memory hierarchy. The stream size is a function of the IFmap dimensions and the kernel Size.

$$InitialOffset = C_i n^2 + Y_i n + X_i \tag{3.19}$$

$$C_i = \lfloor \frac{\lfloor \frac{i}{K} \rfloor}{K} \rfloor \tag{3.20}$$

$$Y_i = (\lfloor \frac{i}{K} \rfloor) \bmod K \tag{3.21}$$

$$X_i = i \bmod K = (i - \lfloor \frac{i}{K} \rfloor K) \tag{3.22}$$

The initial offset function defines the initial index offset in the IFmap tensor where stream begins from for each port $i$. It can be decomposed into three main offsets. A channel offset $C_i$, a row offset $Y_i$ and a column offset $X_i$.

$$F_\Delta(i,t) = \begin{cases} IFmap_{A_\Delta(i,t)} & \Delta_i <= t < \Delta_i + StreamSize \\ 0 & else \end{cases} \tag{3.23}$$

$$\Delta_i = i \tag{3.24}$$

$$A_\Delta(i,t) = A(i,t) - \Delta_i \tag{3.25}$$

Under this new streaming based interpretation of the accesses in the IFmap memory hierarchy, the delay elements in the systolic reduction scheme in Figure 3.14.b are represented as time shifts in the stream function $F(i,t)$. These time shifts are represented in the new delayed access function $A_\Delta(i,t)$.

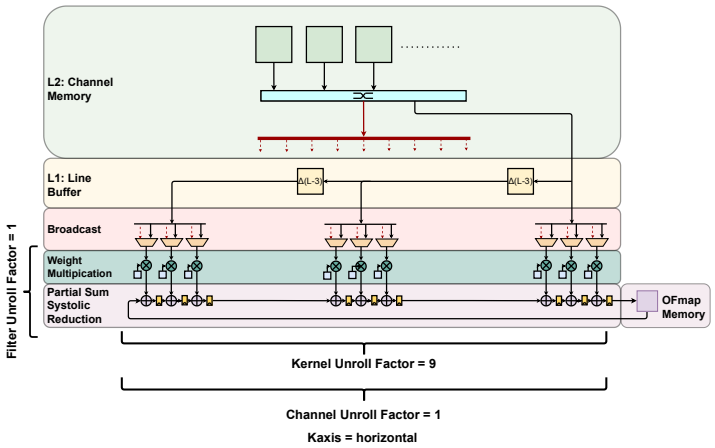



Figure 3.16: Using a systolic reduce and forward to calculate OFmaps

$$A_{\Delta}(i,t) = C_i n^2 + Y_i n + (i - \lfloor \frac{i}{K} \rfloor K) + t - i \tag{3.26}$$

$$A_{\Delta}(i,t) = \lfloor \frac{\lfloor \frac{i}{K} \rfloor}{K} \rfloor^2 + (\lfloor \frac{i}{K} \rfloor) \bmod K + \underbrace{(-\lfloor \frac{i}{K} \rfloor K)}_{X'_i} + t \tag{3.27}$$

$$\tag{3.28}$$

Substituting the InitialOffset function in $A_{\Delta}$ allows us to simplify the column offset. This yields a new column offset $X'_i$. The final delayed access function's initial offset becomes insensitive to changes in the port index $i$ that are not multiples of $K$. This allows us to remove the lowest layer memory along with the systolic array delays in Figure 3.13 and replace both layers with just a series of broadcast buses that span consecutive $K$ groups of IFmap ports provided that we relax the start time constraints to the $\lceil \frac{i}{K} \rceil$ for each group of ports $\lfloor \frac{i}{K} \rfloor$. Delays in accessing IFmap data elements across $K$ groups of ports as well as across $K^2$ groups of ports accessing different channels still remain. This simplification of the IFmap memory hierarchy by removing the systolic delays still requires complex delayed reads from the IFmap hierarchy which necessitates smart SRAMS whose access times can be programmed. A discussion of these smart memories is presented in chapter 5. The final hardware implementation with the added IFmap memory hierarchy optimization discussed in this section is given in Figure 3.16. Figure 3.16 shows the broadcast busses for every group of 3 processing engines as well as the (1, 1) vertical broadcast busses highlighted in red.

## 3.4 HERO: A Hybrid GEMM and Direct Conv. Accelerator

After applying the simplification in subsection 3.3.3 we arrive at the final HERO template architecture variants in Figure 3.17 and Figure 3.18. Both figures illustrate templates with unroll factors for F, C loops undefined. Both variants in each of the figures represent two different spatial axis mappings for the unrolled kernel loops. Depending on the choice of axis mapping the effective channel concurrency available (in the horizontal case in Figure 3.17) and the effective number filter concurrency available (in the vertical case in Figure 3.18) for (1, 1) convolutions will change. A flexible any-to-any interconnect that allows arbitrary bank access is assumed to exist for both L3 IFmap memory and OFmap memory. Arbitrary access to any IFmap and OFmap bank enables flexible distribution of IFmap and OFmap data across multiple banks. The benefit of this flexible distribution will be discussed in chapter 6. In addition to arbitrary IFmap bank access in Figure 3.17, the IFmap interconnect enables broadcasting of IFmap pixels vertically to all filters rows in HERO as well as broadcasting IFmap pixels across groups of PEs for (3, 3) kernel computations. The choice of which spatial axis mapping and F and C unroll factors is discussed further in chapter 4 where these HERO template parameters are optimized based on the layer configurations present in the TIMM Library's networks.

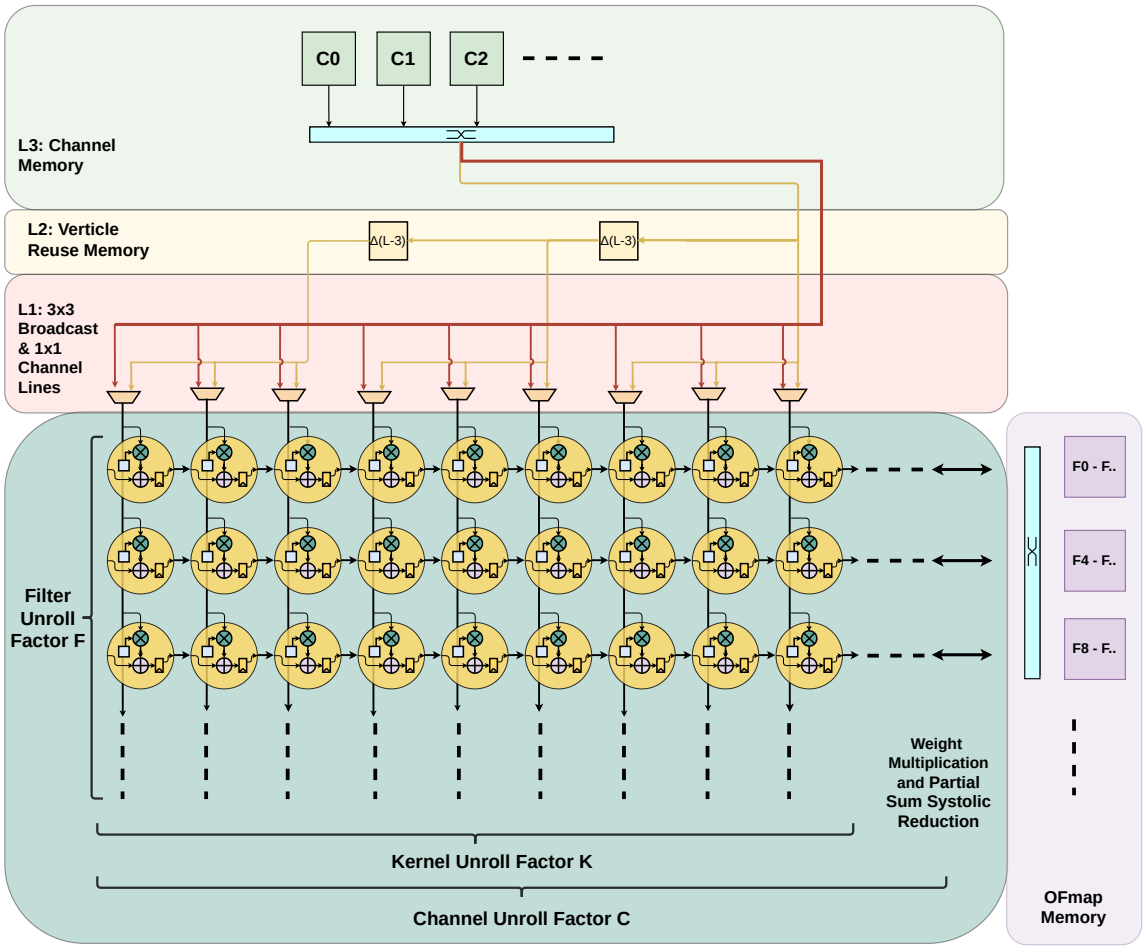



Figure 3.17: Hardware Implementation Taxonomy adapted from [13]

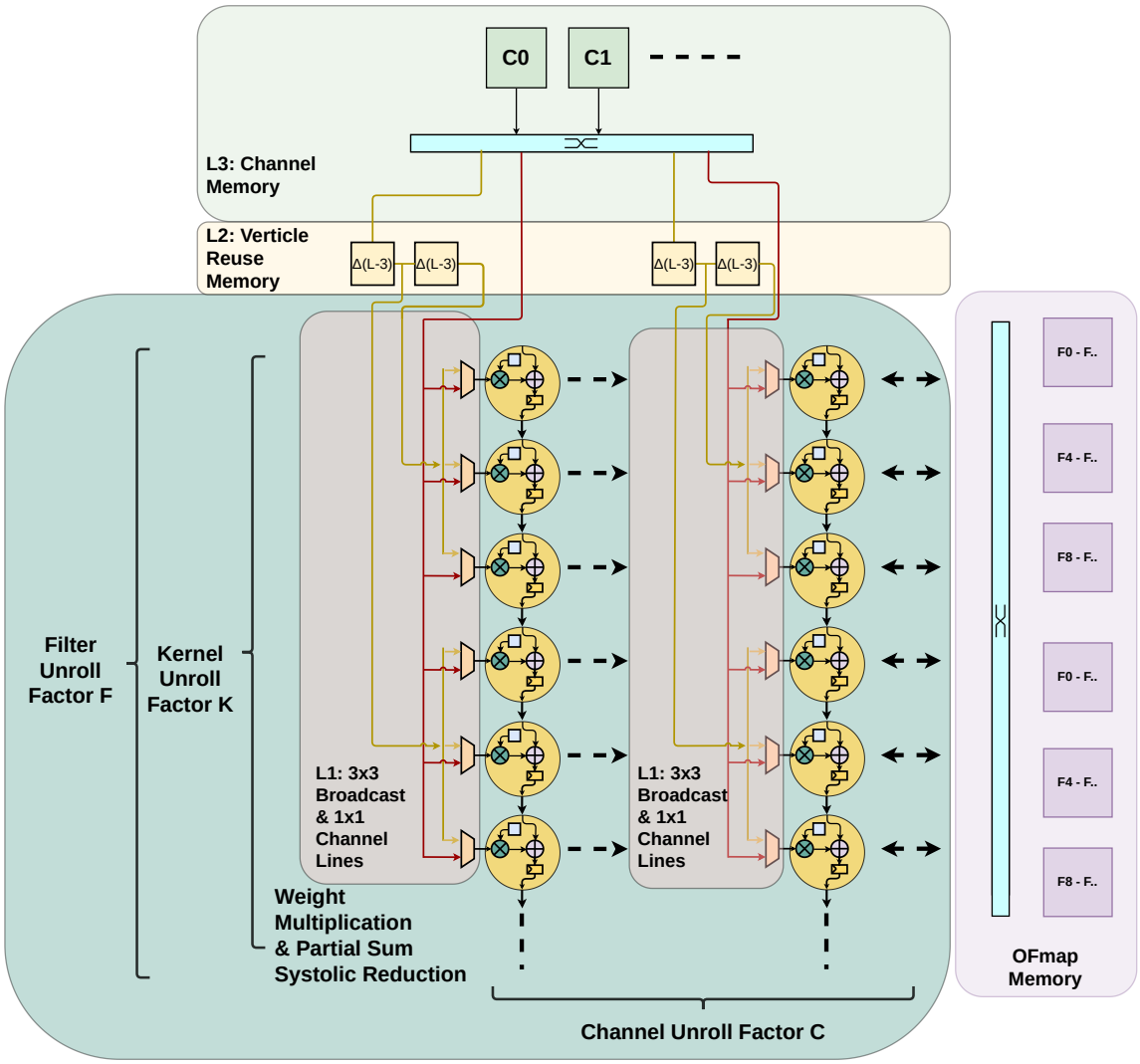



Figure 3.18: Hardware Implementation Taxonomy adapted from [13]

# Chapter 4

# Architecture Dimensioning

From the previous chapter, portions of the dataflow design space remain undefined. Additionally, sizing of different memory banks in HERO's memory hierarchy remain undefined. To explores what remains of the dataflow design space this chapter introduces TEMPO. TEMPO is an optimizer for HERO templates takes the following as an input

- A a library of networks written in pytorch
- A processing engine budget
- An objective function defined using a mixture of analytical models developed for estimating latency, utilization and memory access counts

From the inputs TEMPO returns the optimal PE allocation for the different dimensions of HERO. To determine the appropriate sizes for HERO's on chip memory (given a concrete PE allocation) CIGAR's model dim collector is used to collect statistics on the number of elements in the different input/ output tensors of convolution layers. A discussion of TEMPO's algorithm is presented in section 4.2. The various analytical models used to estimate power and performance metrics when running supported layers in the TIMM library are presented in section 4.3. Results for different utilization optimal configurations of HERO under various PE constraints are presented in section 4.4.

## 4.1 Exploring what remains of the dataflow design space with TEMPO

From the previous section we have concluded that loops F, C, KY and KX are all unroll targets, and KY and KX loops should be unrolled assuming that (1, 1) and (3, 3) kernels are supported directly. What remains of the dataflow design space are the unroll factors for F and C loops

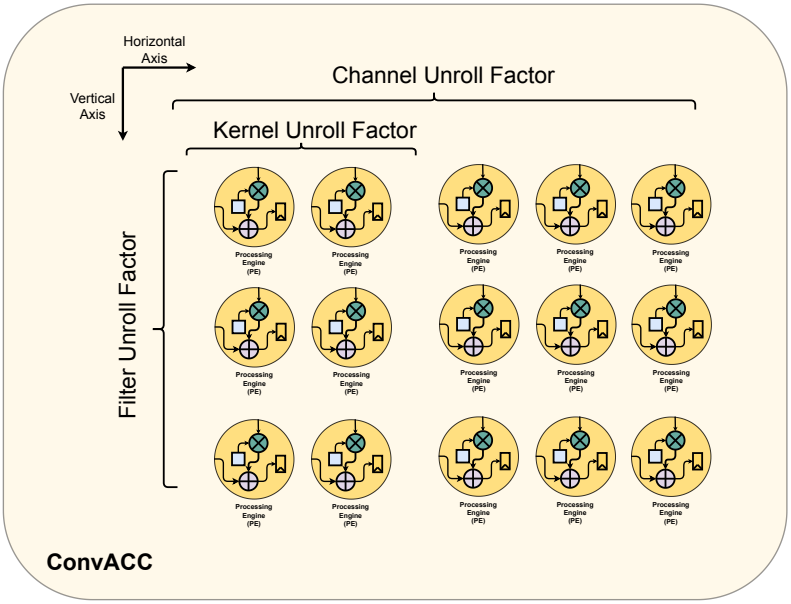



Figure 4.1: GEMM and (1, 1) Convolution Equivalence

as well as the accelerator spatial axis mapping for all unrolled loops F, C, KY, KX. From this point these parameters will be referred to as HERO template parameters from which a concrete accelerator instance can be defined. These accelerator template parameters define the number of processing engines allocated to process channels, filters, and kernels concurrently in the a HERO instance. An illustration of this PE allocation is present in Figure 4.1. The space of possible unroll factors is as large as the space of possible loop upper bounds for the aforementioned unrolled loops. However, as discussed earlier, some combinations of loop upper bounds are unlikely in real networks. Additionally, spatial axis mapping affects the effective unroll factors when executing different convolution layers than the ones assumed when unrolling said loops. This further expands the design space of a possible template parameters. To effectively explore the space of loop unroll factors and accelerator spatial axis mapping we introduce accelerator TEMPlate Optimizer (TEMPO), a dataflow exploration and analysis tool used to optimize an accelerator's weight stationary dataflow based on a target CNN library as well as an arbitrary objective function. A discussion of TEMPO's algorithm model is presented in section 4.2 as well as it's analytical models for utilization, latency and access counts, in section 4.3. Additionally, results of running TEMPO on TIMMS's library of networks is presented in section 4.4. Finally a brief discussion on on-chip memory hierarchy sizing will be given in section 4.5.

## 4.2 TEMPO Algorithm

TEMPO's algorithm is presented in algorithm 2. TEMPO explores the space of possible filter and channel unroll factors as well as kernel axis mappings by exhaustively iterating through the space of possible values. TEMPO expects the following inputs.

- Processed $model_{dict}^{stats}$ from TIMM
- An objective function $obj\_fn$
- Maximum $pe_{budget}$
- Set of kernels supported directly $kernels_{supported}$

TEMPO then produces an optimal filter, channel unroll factors and kernel axis mapping that maximizes the given objective function under the $pe_{budget}$ constraint specified. In algorithm 2 TEMPO effectively runs an exhaustive search using an objective function $obj\_fn$ and a set of layers from a model library. The objective function is a function that evaluates an architectures score when executing layers in the model library $model_{dict}^{stats}$. An architecture score is based on any of the metrics that will be discussed in section 4.3. Prior to the search being performed algorithm 2 converts all layer not supported directly in the model to (1, 1) equivalent layers based on the equivalence method that discussed in section 3.2.

$$\begin{aligned} \underset{f_{unroll}, c_{unroll}, k_{axis}}{\operatorname{argmax}} \quad & obj\_fn(f_{unroll}, c_{unroll}, k_{axis}, ModelLibrary) \\ \text{subject to} \quad & \\ F_{unroll}.C_{unroll} <= Pe_{budget} & \end{aligned} \tag{4.1}$$

**Algorithm 2** TEMPO

**Input:** $model_{dict}^{stats}$, $obj\_fn$, $kernels_{supported}$, $pe_{budget}$

**Output:** $template_{config}^{opt}$

**function** TEMPO_RUN($model_{dict}^{stats}$, $obj\_fn$, $kernels_{supported}$, $pe_{budget}$)
  $max_{score} \leftarrow -\inf$
  $template_{opt} \leftarrow nil$
  $\hat{model}_{dict}^{stats} \leftarrow convert\_all\_unsupported\_layers(model_{dict}^{stats}, kernels_{supported})$
  **for** $f_{unroll} \leftarrow factors(pe_{budget})$ **do**
    **for** $k_{axis} \leftarrow \{Verticle, Horizontal\}$ **do**
      $c_{unroll} \leftarrow \lfloor \frac{pe_{budget}}{f_{unroll}} \rfloor$
      $template_{score} \leftarrow obj\_fn(f_{unroll}, c_{unroll}, k_{axis}, \hat{model}_{dict}^{stats})$
      **if** $max_{score} < template_{score}$ **then**
        $max_{score} \leftarrow template_{score}$
        $template_{opt} \leftarrow template_{config}$
      **end if**
    **end for**
  **end for**
  **return** $template_{config}^{opt}$
**end function**

## 4.3 TEMPO analytical model

On-chip memory constraints are ignored for each of the metrics evaluated using TEMPO's analytical model. This means that TEMPO's results are the most accurate when there are no constraints for on-chip memory. On-chip memory constraints may limit available concurrency in a layer which will cause PEs to be underutilized. To clarify why this happens, assume we have a convolution layer with a kernel size of (k, k) and an ifmap of size 2 MB. Additionally, assume a single channel of that ifmap is 512 KB. If the available on-chip ifmap memory is constrained to 512 KB the accelerator can only process one channel at a time despite the existence of 4 channels in the ifmap tensor. If there are many PE's dedicated to processing channels concurrently then PE utilization will suffer due to the single channel restriction mentioned earlier. If constraints for on-chip memory are required then an additional layer decomposition step is necessary in order to properly evaluate all

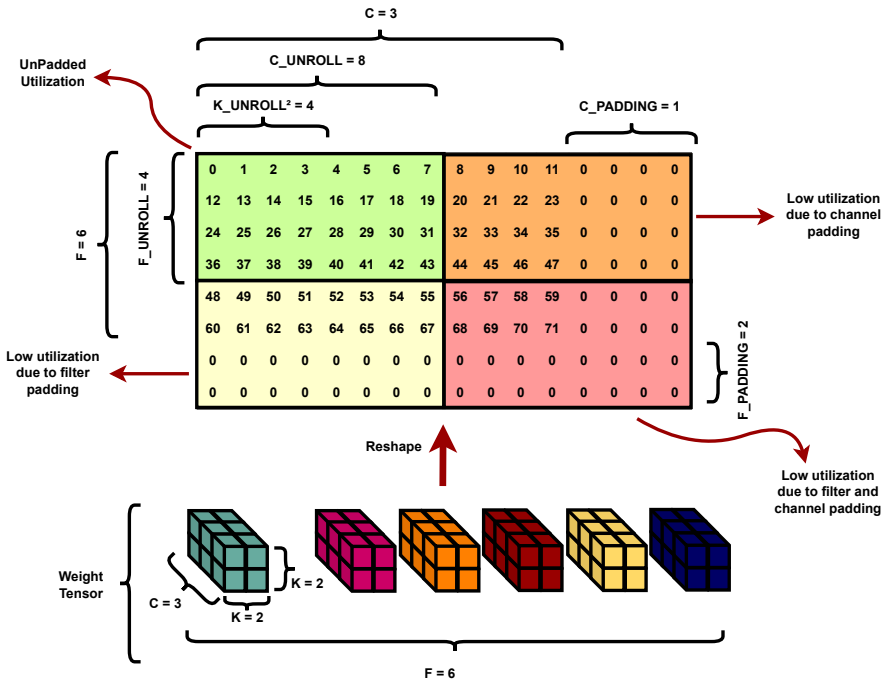



Figure 4.2: GEMM and (1, 1) asd Equivalence

metrics that can be calculated using TEMPO's analytical model. The inclusion of on-chip memory constraints in TEMPO's analytical model are left as part of future work.

### 4.3.1 Utilization

TEMPO models accelerator utilization based on how the template parameters (loop unroll factors and loop axis mapping) tile and pad a convolution layer's stationary weight tensor. Kernel axis mapping is assumed to be fixed in HERO. An illustration of TEMPO's utilization model in action is present in Figure 4.2. In Figure 4.2 a layer with a weight tensor of dimensionality $R^{6\times3\times2\times2}$ is tiled and padded based on the template parameters $C_{unroll} = 8$, $F_{unroll} = 4$, $K_{unroll} = 2$ and axis mapping $K_{axis} = horizontal$. Based on these template parameters the effective filter and channel unroll factors are $F_{eff} = 4$, $C_{eff} = 2$. These unroll factors create $\lceil \frac{\hat{C}}{C_{eff}} \rceil = 2$ horizontal tiles and $\lceil \frac{\hat{F}}{F_{eff}} \rceil = 2$ vertical tiles assuming padding has been applied. The weight tensor in Figure 4.2 is then reshaped into a 2D matrix of dimensionality $R^{8\times16}$ with additional padding. The total number of tiles is then reflected in Equation 4.2.

$$Count_{Tiles} = \lceil \frac{\hat{F}}{F_{eff}} \rceil \lceil \frac{\hat{C}}{C_{eff}} \rceil \tag{4.2}$$

Since HERO processes the weight tensor in tiles, utilization is calculated on a per-tile basis. There are two different types of tiles, padded and unpadded, each with their own utilization calculation. Layer utilization is then an average of the utilizations of each tile type weighted by their frequency of occurrence in the layer as reflect in Equation 4.3. For brevity each of the utilization equations are multiplied by their frequency of occurrence in the same equation.

$$LayerUtilization = \frac{utilization_{Tiles}^{UnPadded} + utilization_{Tile(s)}^{Padded}}{Count_{Tiles}} \tag{4.3}$$

The first tile type is the unpadded tile illustrated in Figure 4.2 as the green tile. Utilization is calculated using Equation 4.4. In this tile utilization is assumed to be 1 and it's frequency of occurrence depends on the number of unpadded tiles in the layer $\lfloor \frac{\hat{F}}{F_{eff}} \rfloor \lfloor \frac{\hat{C}}{C_{eff}} \rfloor$.

$$utilization_{Tiles}^{UnPadded} = 1. \lfloor \frac{\hat{F}}{F_{eff}} \rfloor \lfloor \frac{\hat{C}}{C_{eff}} \rfloor \tag{4.4}$$

The second tile type is the padded tile of which there are three variations depending on the reason for padding the tile. The utilization for all padded tiles weighted by their frequencies of occurrence is given in Equation 4.5.

$$\begin{aligned} utilization_{Tile(s)}^{Padded} = & \ utilization_{ChannelTiles}^{Padded} \\ & + utilization_{FilterTiles}^{Padded} \\ & + utilization_{ChannelAndFilterTiles}^{Padded} \end{aligned} \tag{4.5}$$

If the allocation of PEs for channel loops exceeds available channels to be processed in the tile, then that tile will be padded. The padding in that tile results in reduced PE utilization. An illustration of that padded tile variation is present in Figure 4.2 as the orange tile. The calculation for the weighted utilization in that tile variation is given in equation Equation 4.6. To determine if a a padded channel exists or not we can check if $\hat{C} \bmod C_{eff} > 0$ is true. If that condition is true, padded channel tiles exist in the layer and their weighted $utilization_{ChannelTiles}^{Padded}$ is then a function of how many PEs are active in the tile $\frac{(\hat{C} \bmod C_{eff}) F_{eff} K_{unroll}^2}{Count_{pe}}$ multiples by the frequency of occurrence. $\lfloor \frac{\hat{F}}{F_{eff}} \rfloor$. If $\hat{C} \bmod C_{eff} = 0$ then there are no padded channel tiles so $utilization_{ChannelTiles}^{Padded} = 0$.

$$utilization_{ChannelTiles}^{Padded} = \begin{cases} \frac{(\hat{C} \bmod C_{eff}) F_{eff} K_{unroll}^2}{Count_{pe}} \cdot \lfloor \frac{\hat{F}}{F_{eff}} \rfloor & \hat{C} \bmod C_{eff} > 0 \\ 0 & \hat{C} \bmod C_{eff} = 0 \end{cases} \tag{4.6}$$

If the allocation of PEs for filter loops exceeds available filters to be processed in the tile, then that tile will be padded. This another variation of a padded tile and the weighted utilization for that tile variation is calculated using Equation 4.7 and is illustrated in Figure 4.2 as the yellow tile.

$$utilization^{Padded}_{FilterTiles} = \begin{cases} \frac{C_{eff}(\hat{F} \bmod F_{eff})K^2_{unroll}}{Count_{pe}} . \lfloor \frac{\hat{C}}{C_{eff}} \rfloor & \hat{F} \bmod F_{eff} > 0 \\ 0 & \hat{F} \bmod F_{eff} = 0 \end{cases} \tag{4.7}$$

Finally the last padded tile variation is the tile padded due to the excess allocated of PEs for both filter and channel loops. This type of tile is illustrated in Figure 4.2 as the red tile. To determine if a tile like this exists we can evaluate the condition $\hat{F} \bmod F_{eff} > 0 \wedge \hat{C} \bmod C_{eff} > 0$ is true. If it there exists exactly one tile where utilization is reduced due to excess allocation of PEs for filter and channel loops. The equation to calculate weighted utilization in this padded tile variation is given in Equation 4.8.

$$utilization^{Padded}_{Channel\&FilterTile} = \begin{cases} \frac{(\hat{C} \bmod C_{eff})(\hat{F} \bmod F_{eff})K^2_{unroll}}{Count_{pe}} & \hat{F} \bmod F_{eff} > 0 \wedge \hat{C} \bmod C_{eff} > 0 \\ 0 & else \end{cases} \tag{4.8}$$

### 4.3.2 Latency

Estimating latency follows the same tiling model discussed the previous section. The latency of executing a layer based on the template parameters chosen is given in Equation 4.9. Latency is a function of the number of tiles present in the layer multiplied by the number of cycles spent processing a single IFmap channel $\hat{Z}$ plus the additional latency incurred due to lowering lifting depending on the support for the layer's kernel size. Latency for lowering and lifting is given in Equation 4.10. If the kernel is supported directly, no additional lowering and lifting penalties are incurred, otherwise penalties are calculated based on the number of operations necessary to lower the IFmap and Weight tensors plus the number of operations to lift the OFmap. Lowering and lifting are assumed to be performed by a software based co-processor. The latencies associated with lowering and lifting can be eliminated if these operations are incorporated into the processor however, that is left as part of future work.

$$Latency = \hat{Z}.Count_{Tiles} + Latency_{Lowering} + Latency_{Lifting} \tag{4.9}$$

$$Latency_{Lowering} = Latency_{Lifting} = \begin{cases} 0 & K \in \{SupportedKernels\} \\ m^2K & K \notin \{SupportedKernels\} \end{cases} \tag{4.10}$$

### 4.3.3 Memory access counts

Following the tiling model discussed earlier, memory access counts are calculated based on how the template parameters tile the layer's weight tensor. Access counts for IFmaps are given in Equation 4.11, OFmap access counts are giving in Equation 4.12 and finally weight access counts are given in Equation 4.13.

$$IFmap^{AccessCount} = ((\hat{Z}K^2_{unroll})(\lfloor\frac{\hat{C}}{C_{eff}}\rfloor C_{eff} + \hat{C} \bmod C_{eff}))\lceil\frac{\hat{F}}{F_{eff}}\rceil \tag{4.11}$$

$$OFmap^{AccessCount} = 2 * \hat{Z}(\lfloor\frac{\hat{F}}{F_{eff}}\rfloor F_{eff} + \hat{F} \bmod F_{eff})\lceil\frac{\hat{C}}{C_{eff}}\rceil \tag{4.12}$$

$$\begin{aligned} Weight^{AccessCount} = \hat{Z}((&C_{unroll}F_{unroll})(\lfloor\frac{\hat{C}}{C_{eff}}\rfloor\lfloor\frac{\hat{F}}{F_{Feff}}\rfloor) \\ &+ (C_{unroll}F_{eff})(\lfloor\frac{\hat{C}}{C_{eff}}\rfloor\hat{F} \bmod F_{eff}) \\ &+ (C_{eff}F_{unroll})(\lfloor\frac{\hat{F}}{F_{eff}}\rfloor\hat{C} \bmod C_{eff}) \\ &+ (C_{eff}F_{eff})(\hat{F} \bmod F_{eff} * \hat{C} \bmod C_{eff})) \end{aligned} \tag{4.13}$$

## 4.4 TEMPO results

$$obj\_fn \leftarrow average(\{LayerUtilization(layers) | \forall layers \in m, \forall m \in \{ModelLibrary\}\}) \tag{4.14}$$

Since TEMPO expects an objective function to maximize based on any or a combination of all the discussed metrics in section 4.3 Equation 4.14 defines an objective function based solely on the average layer utilization metric over the entire set of convolution layers in TIMM's model library. Layer utilization is evaluated based on the discussion in subsection 4.3.1. Results for optimal configurations are given in Figure 4.3. Figure 4.3 shows a boxplot of average layer utilizations under different utilization optimal configurations found with TEMPO. Median utilization achieved for the architecture with 576 PEs was 98% however outlier utilization can drop to as low as 2%. As stated earlier, TEMPO does not consider any restrictions for on-chip memory. This may affect utilization results due to limited available concurrency in a layer. Regardless, the configurations suggested by TEMPO in Figure 4.3 without on-chip memory constraints are a good starting point for generating results from a cycle accurate model of HERO.

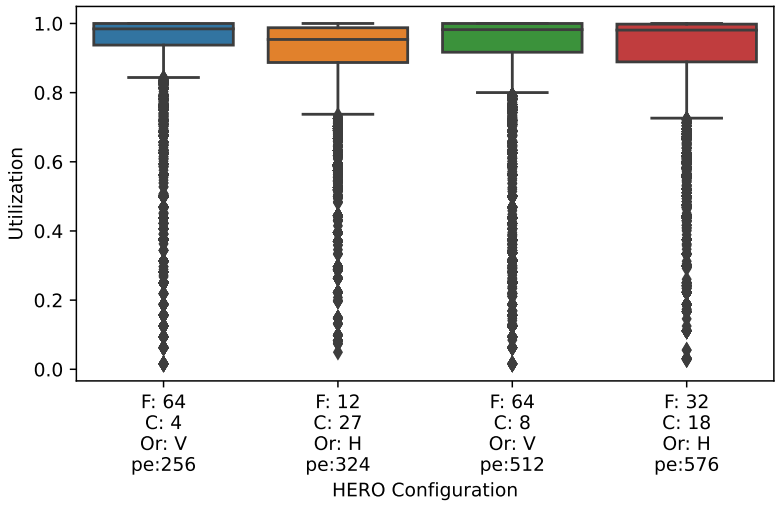



Figure 4.3: Utilization results for different optimal configurations found using TEMPO

## 4.5 Memory Hierarchy Sizing

To determine the necessary sizes of on-chip memories we need to first look at the sizing behavior of the different data elements in a convolution operation (IFmaps, OFmaps and weights). Note that all discussions of storage requirements are precision agnostic. All storage requirement results are given in number of elements. Figure 4.4 is a boxplot of the sizes (in number of elements) for the storage requirements of different data element types present in the convolution layers of the TIMM library networks. The median storage requirements for all elements is given in Table 4.1. From both figure and table, note the similarity in storage requirements of all data elements. Lowering and lifting operations required under GEMM mode only increase median storage requirements by a factor of 1.01X and 1.02X for ifmap and OFmap respectively. To support as much as 85% of convolution layers in TIMM without requiring layer decomposition (to be discussed in chapter 6) we only need to allocate 1 MB of storage for IFmap memory (L3) and OFmap memory in either of the HERO architectures presented in section 3.4. Additionally since L2 storage in the ifmap hierarchy scales with the width of ifmap tensors, it's assumed that the maximum ifmap tensor width will not exceed 512 elements. Note that there are no storage requirements for weight storage due to the choice of weight stationary dataflow made in chapter 3. **??** shows the total storage in number of elements assumed by this work.

On-chip storage requirements for IFmaps and OFmaps are influenced by the ordering

| Data Element Type | Median Size |
|---|---|
| Weights | $2^{16.614}$ |
| IFmap | $2^{16.614}$ |
| OFmap | $2^{16.192}$ |

Table 4.1: Table of median storage requirements for data elements in convolution layers of networks in the TIMM Library

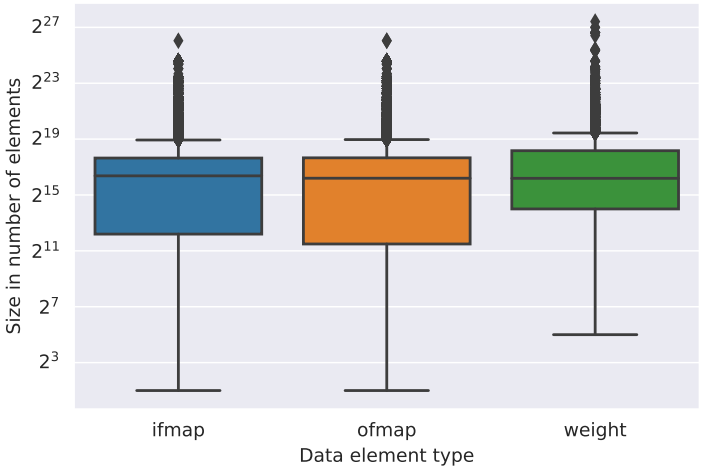
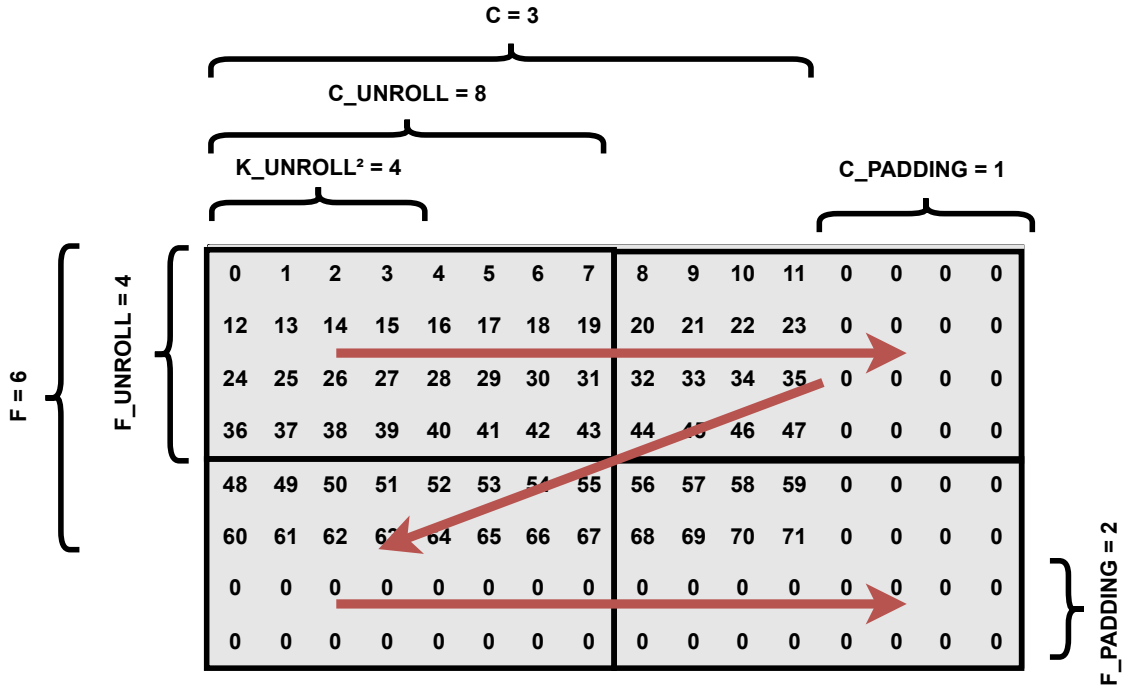



Figure 4.4: Boxplot of storage required for different data elements assuming no lowering

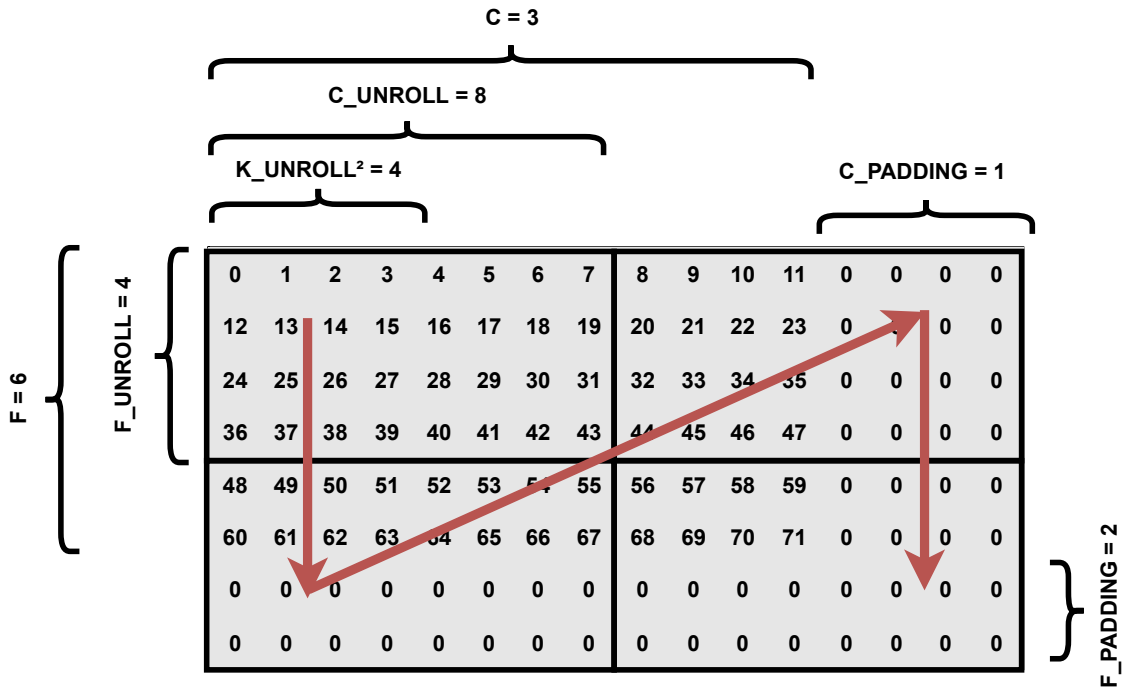
of tiles in the tiling representation of weights discussed in chapter 3. The order of processing weight tiles is equivalent to the ordering of the F, and C loops in the loop based representation of the convolution operation discussed in chapter 3. Processing tiles in F, C order (ASAP) results in retiring output featuremaps as soon as possible while retaining input featuremaps for as long as possible. Conversely, processing tiles in C, F order (ALAP) retains output featuremaps for as long as possible while retiring IFmaps as soon as possible. An illustration of ASAP and ALAP tile schedules in available in Figure 4.5.

An illustration of this architectural tradeoff is present in Figure 4.6.c. Note that depending on the tile scheduling chosen, different HERO configuration parameters affect the scaling of different on-chip memories. For example, under ASAP, input featuremap memory requirements remain unchanged with any configuration parameters, however, OFmap memory scales with the architecture's filter unroll factor. Under ALAP OFmap memory remains unaffected by any of HERO's configuration parameters, however, ifmap memory scales with the architecture's channel unroll factor. An illustration of how both memories scale with configuration parameters is given in Figure 4.6.a & Figure 4.6.b.

In this work ASAP scheduling is always assumed given the relatively poor scaling of OFmap memory with architecture configuration parameters. This poor scaling is due to the necessity of storing OFmap elements using higher precision values to prevent numeric overflow.

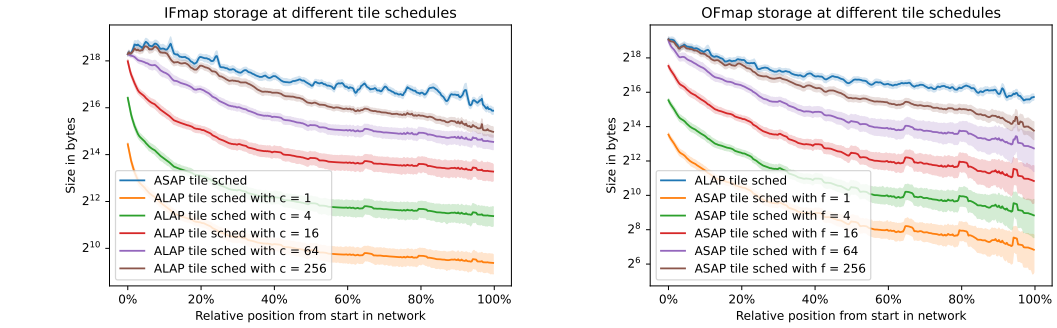



(a)

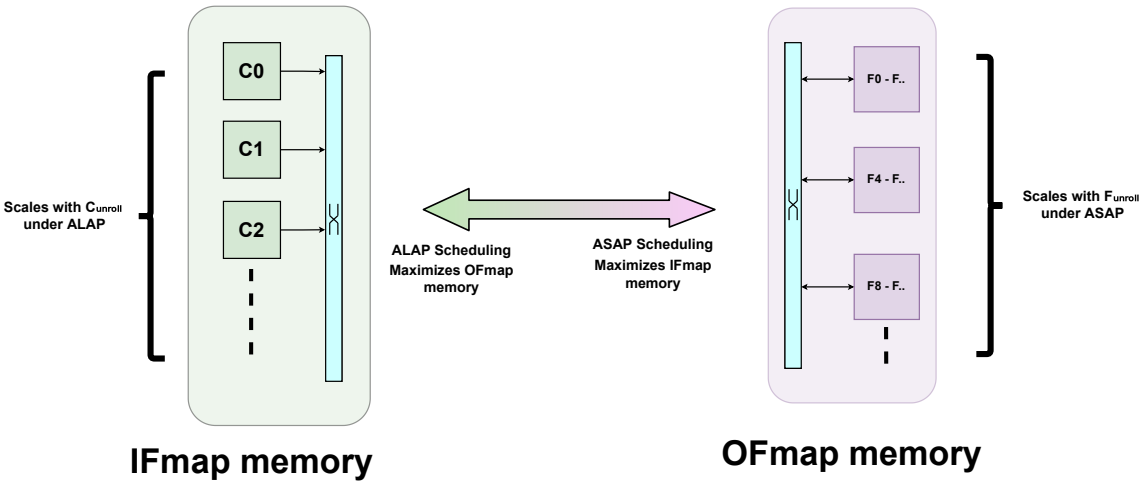



(b)

Figure 4.5: Illustration of different tiling schedules (a) ASAP scheduling (F, C) (b) ALAP scheduling (C, F)




(a) (b)




(c)

Figure 4.6: Illustration of storage tradeoff between OFmaps and IFmaps depending on tile scheduling (loop ordering) and accelerator configuration parameters (loop unroll factors) (a) IFmap storage (b) OFmap storage (c) architectural illustration

# Chapter 5

# On-Chip Data Orchestration

To coordinate IFmap reads, OFmap read-modify-writes and Weight reads based on the final implementation in section 3.4 we need smart programmable memories that can 1) perform timed reads and writes between themselves and PEs and 2) perform timed data transfers between themselves and other programmable memories. In this chapter, we introduce Self Addressable SRAMs (SAM)s, a programmable memory primitive that can execute descriptor based programs. Depending on the composition of these descriptor based programs, timed reads and writes can be made by on-chip SAMs to and from processing engines. Additionally, with sufficient connectivity between SAMs as well as implicit coordination between different SAM programs we can orchestrate timed data transfers between SAMs. In this chapter we first discuss the structure of a SAM in section 5.1 followed by the functional behavior of a SAMs address generator controller in section 5.2. Finally we introduce descriptor based programs in section 5.3, specifically the different types of descriptors available in subsection 5.3.1 as well as the different types of memory transactions possible using coordinating descriptor based programs in subsection 5.3.2.

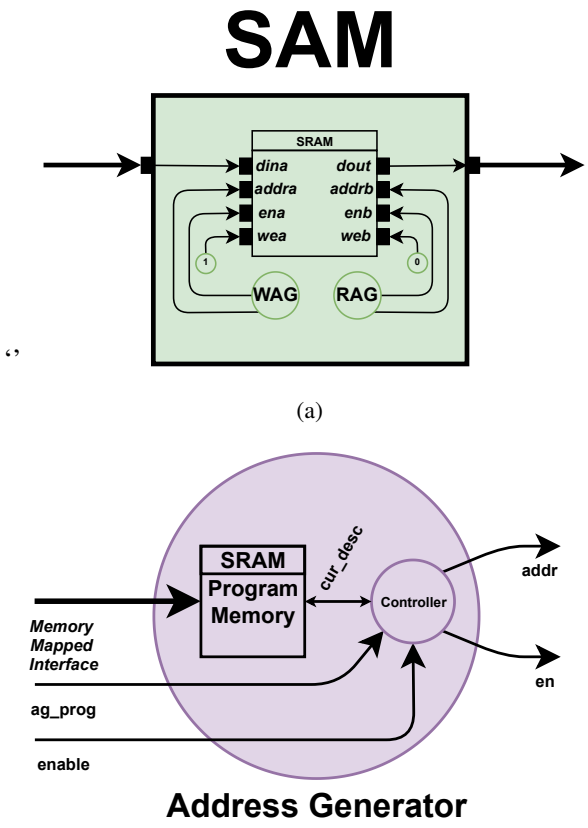



Figure 5.1: (a) SAM structure (b) Address generator structure

## 5.1 Structure of a SAM

In Figure 5.1.a SAMs are composed of an address generator and a data sram. Address generators are attached to ports of an SRAM. They control the address and enable ports for each SRAM port. The port behavior (read or write) is set by a memory mapped register attached to the write enable pins of each port. Address generators are programmable modules within SAMs that generate address streams based on descriptor programs. These address streams are then fed to the SAMs data SRAM. In Figure 5.1.b, address generators are composed of a controller attached to a program memory SRAM. Depending on the sizing requirements of the descriptor programs, program memory SRAMs can be replaced with register files containing all relevant descriptors. SAM address generators are equipped with an external memory mapped interface to allow transfer of descriptor programs from an a software based co-processor.

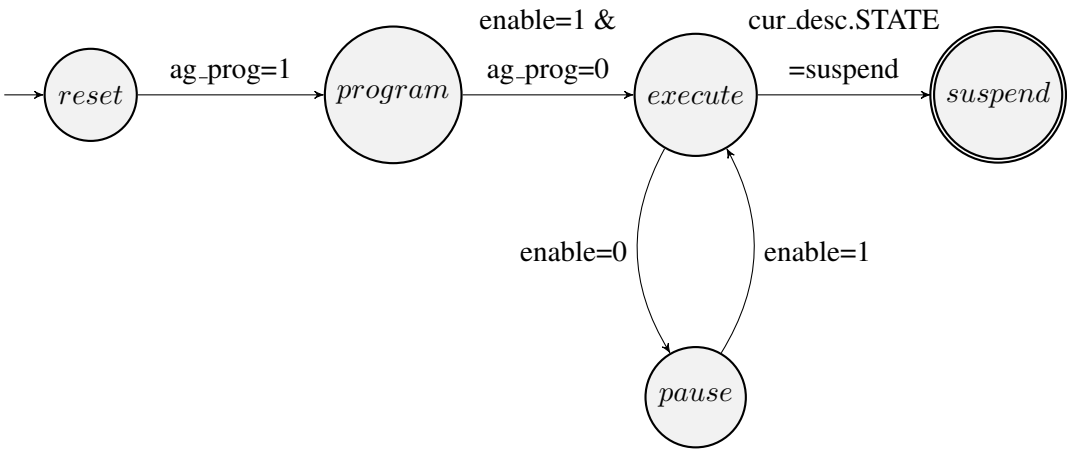



Figure 5.2: Address generator Finite State Machine

## 5.2 Address generator controller

The finite state machine of address generator controllers is presented in Figure 5.2. After an initial reset, the controller waits until the ag_prog signal is asserted thus indicating that the generator is in the program state and is awaiting to receive a descriptor based program from the external memory mapped interface. Once the program is confirmed to be have been written by the external interface the ag_prog signal can be de-asserted followed by the assertion of the enable signal. When that occurs the controller transitions into the execute state in which it loads the first descriptor and executes it. If the enable signal is de-asserted for any reason the controller enters a pause state. When the enable signal is re-asserted the controller goes back to the execute state. Once a descriptor is retired, the controller reads the next descriptor from the program memory and begins executing it without leaving the execute state. If the controller's cur_desc pointer points to a suspend descriptor execution terminates and the controller enters a suspend state.

## 5.3 Descriptor based programs

Descriptor based programs are inspired by [12] where the authors illustrate different ways to program a model Blackfin processor's DMA using various descriptor configurations. The main difference between this work's approach to descriptors and [12] is the inclusion of timing and hybrid access/timing descriptors that allow more complicated memory transactions to occur between SAMs.

### 5.3.1 Descriptor Types

Before discussing descriptor based programs we must first discuss the properties of individual descriptors. Each descriptor can be represented as a struct as depicted in Listing 5.1. In a single descriptor, the type field describes the type of the descriptor. There are three different types of descriptors. Generate descriptors used for generating address streams. Wait descriptors that pause execution of descriptor based programs for a set number of cycles. Lastly, suspend descriptors used to mark the termination of a descriptor based program.

Listing 5.1: Descriptor Struct

```
1 struct Descriptor
2 {
3     DescriptorType type;
4     unsigned int start;
5     unsigned int x_count;
6     int x_modify;
7     unsigned int y_count;
8     int y_modify;
9 };
```

Each descriptor can be thought of as a self contained address stream generation program. The C code representation for the generate and wait descriptor types is given in Listing 5.3 and Listing 5.2. Suspend descriptors are the simplest of the different descriptor types. All fields except the type field are set to 0 in the descriptor struct. The state field is set to some predetermined value that represents the SUSPEND state.

In both generate and wait c code listings, the output signals from the address generator ”en” and ”addr” are referred to as global variables. In Listing 5.2, the wait descriptor is represented as a for loop that runs for x_count iterations while the SRAM port enable pin is de-asserted. The address output signal is left undefined as it has no effect when the SRAM port enable pin is de-asserted. The wait descriptor is used to synchronize different descriptor programs across SAMs as well as create timed writes and reads to and from SAMs.

Listing 5.2: Descriptor as a set of loops

```
1     en = 0;
2     for(int x = 0; x < x_count; x++);
```

In Listing 5.3, generate descriptors use the y_count, and x_count fields in the descriptor struct to define the upper bounds for two nested loops within which an addr variable is incremented by x_modify in the inner loop and y_modify in the outer loop. The ”addr” output signal is initialized with the contents of the start field and the ”en” signal is asserted for the duration of the descriptors execution.

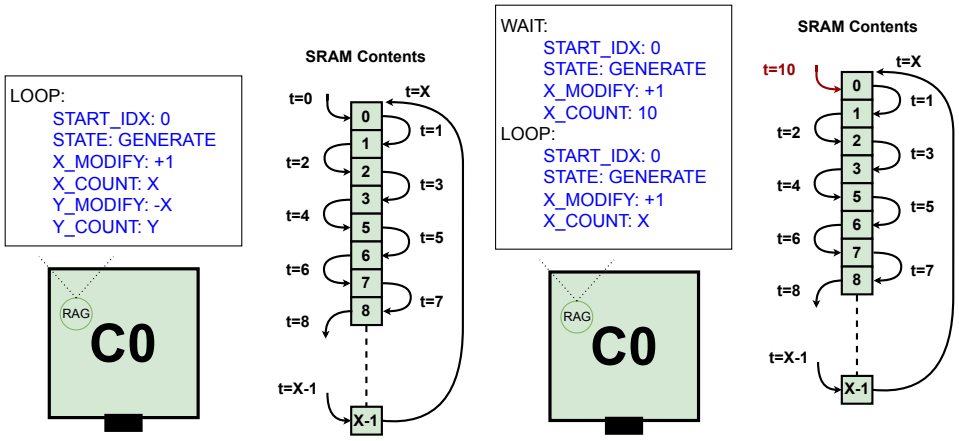
Listing 5.3: Descriptor as a set of loops

```
1 en = 1;
2 addr = start;
3 for(int y = 0; y < y_count; y++)
4     for(int x = 0; x < x_count; x++)
5         addr += x_modify;
6     addr += y_modify;
```

### 5.3.2 Creating timed memory operations with descriptor programs

Depending on the composition of different descriptor programs we can create timed memory operations with SAMs. An illustration of some of the possible timed operations involving single address generators is given in Figure 5.3. In Figure 5.3.a the contents of C0 are read in a loop. This is achieved by setting the y_modify variable to -X to reset ”addr” to the start idx 0. In Figure 5.3.b a wait descriptor is inserted prior to the loop descriptor to introduce a delay in the start time of the loop descriptor.

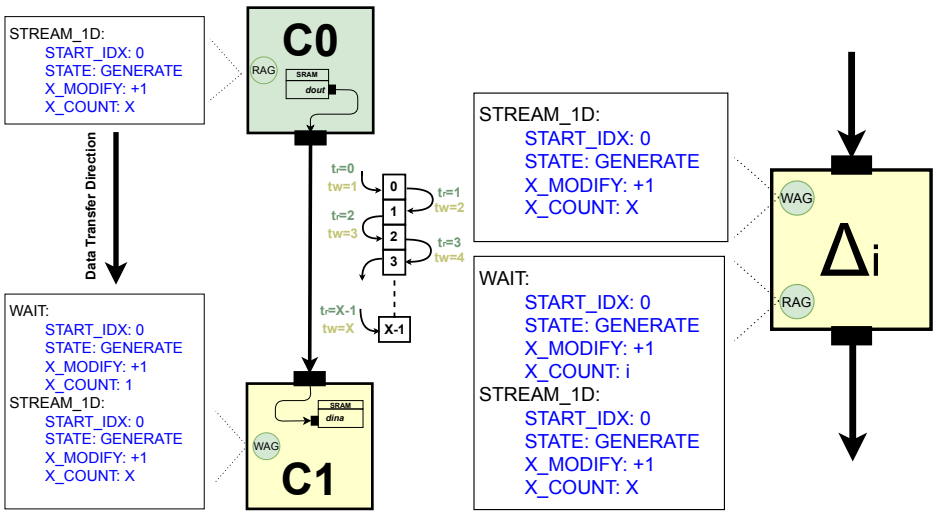
More complicated memory operations can be performed via the implicit coordination of multiple address generators across SAMs or within the same SAM. An illustration of that coordination is presented in Figure 5.4. In Figure 5.4.a a data transfer between two SAMs is achieved using one read address generator in C0 and one write address generator in C1 as well a connection between the dout pins of C0 and din pins of C1. The read address generator executes a generate descriptor that reads out the contents of the SAM. The write address generator waits for 1 cycle then executes a write operation to store the contents of the C0 in C1. These two descriptor programs across two SAMs implicitly coordinate with the inclusion of that wait descriptor. They are each unaware of the program executed by the other. Similarly this implicit coordination can occur between address generators in the same SAM. In Figure 5.4.b a read and a write address generator coordinate to creates a variable sized FIFO that reads out data received by the SAM after a delay $\Delta_i$. This delay is introduced using similar descriptor programs as in Figure 5.4.a. In Figure 5.4.b




(a) (b)

Figure 5.3: Illustration of different descriptor based programs with single address generators (a) Loop program (b) Delayed loop program

the read address generator waits for $\Delta_i$ cycles before starting to read the contents written by the write address generator.




(a) (b)

Figure 5.4: Illustration of different descriptor based programs with dual address generators (a) Memory to memory transfer (b) variable sized FIFO

# Chapter 6

# Network Compilation

Network compilation is driven by Network Compiler For Hero (EMPIRE). EMPIRE compiles arbitrary pytorch models into descriptors for SAMs in HERO. EMPIRE has two phase. The first phase applies layer transformation phase that converts unsupported network layers into layers supported by HERO. Phase one uses uses the layer equivalence approach discussed in Equation 3.11 combined with a layer decomposition technique discussed in this chapter. Phase two is the descriptor generation phase where layers are converted into descriptors for on-chip SAMs. Based on the conclusions from chapter 3 the operations that we need to compile descriptors for are 1x1 and 3x3 convolutions. As such, phase two's discussion of descriptor generation is constrained to generating descriptors for these two operations. Phase one is discussed in section 6.1, and phase two is discussed in section 6.2.

## 6.1 Phase one: Layer Transformation

### 6.1.1 Layer Decomposition

Given that chip area may be restricted, a hero instance may have insufficient on-chip storage available to hold ifmap and OFmap memory on chip during layer processing. To minimize excessive reloads from DRAM we need a way to decompose a larger layer into sub-layers that can fit in the accelerator's on-chip memory. Decomposition can occur due to either large ifmap or large OFmaps or in some cases both. In this work it's assumed that decomposition occurs across channel and filters for input and output feature maps. Decomposition can occur by decomposing large featuremaps along the width and height dimensions however that complicates the process of

aggregating sub layers. This is due to the potential overlaps of kernel windows across decomposition boundaries. This issue arises specifically with (3, 3) kernels under direct mode in HERO. In layers with (1, 1) kernel sizes no overlaps across decomposition boundaries can occur due to the small kernel size.

Depending on the cause of layer decomposition, the dimensionality of either the input or output featuremaps may change. Beginning with the case of decomposition due to large ifmap tensors, Figure 6.1 shows how an ifmap of size (4, 4) with 4 channels is decomposed into two separate ifmap tensors each with 2 channels. Each sub layer consists of half the ifmap tensor. Sub layers are processed sequentially. When processing the second sub layer a bias is assumed to be loaded in which contains the partial sums from the first sub layer's output. Once the second sub layer's output is computed the final OFmap for the filter being processed becomes available. An illustration of how ifmap decomposition affects weight tiling is available in Figure 6.2. When decomposing a large featuremap along the channel axis, the weight tensor has to also be decomposed. Using the tiling representation of weight tensors, ifmap based decomposition halves the size of tiles being processed by the architecture.

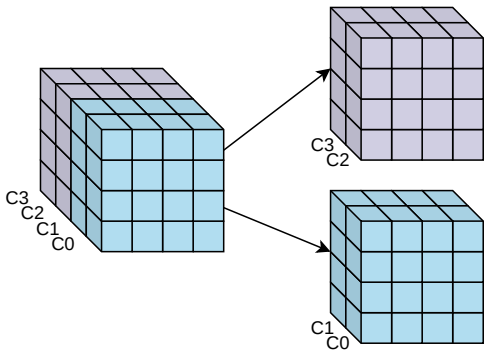



Figure 6.1: Illustration of layer decomposition's effect on Ifmap tensors

Decomposition due to large OFmaps follows the same logic as in the ifmap case. An illustration of OFmap based decomposition is available in Figure 6.3. If on-chip memory is insufficient for storing partial sums of large OFmaps from different the layer will be decomposed into sub layers that will process only a portion of the available filters in the layers. OFmap based layer decomposition is also used when processing depthwise convolution layers given that they are not supported directly by HERO.

Decomposition primarily affects PE utilization since it may limit available concurrency

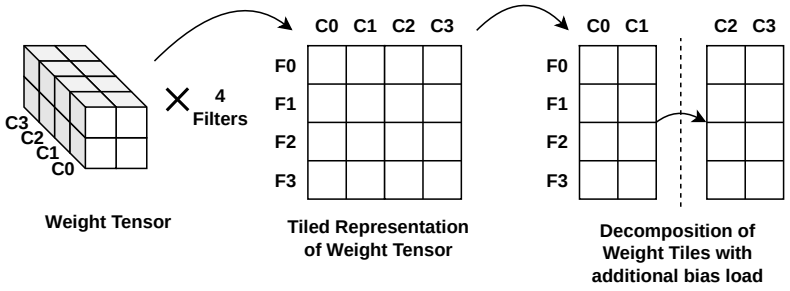



Figure 6.2: Illustration of channel based layer decomposition's effect on weight tiling

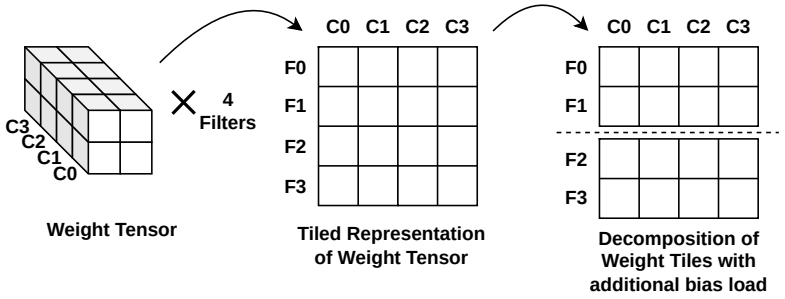



Figure 6.3: Illustration of filter based layer decomposition's effect on weight tiling

when processing channels and filters. A layer with a large IFmap for example may have to be decomposed into two sub-layers were each has a lower overall channel count that would fail to saturate available PEs dedicated for channel concurrency. The same logic applies in layers with OFmaps that have high channel counts failing to saturate available PEs for filter concurrency. This type of PE under-utilization is orthogonal to under-utilization due to overall IFmap/OFmap channel sizes being larger than the size of one IFmap L3/ OFmap bank. This forces some of the available banks to be idle since they cannot output their partial feature maps to available PEs until the appropriate time in the accelerators computation lifecycle. Scenarios of low utilization cause by layer decomposition and/or insufficient individual bank sizes can be mitigated by supporting other forms of concurrency beyond the filter/ channel/ kernel concurrency chosen by HERO. An exploration of other forms of concurrency within HERO is left as part of future work.

## 6.2 Phase Two: Descriptor Program Generation

We can generate descriptor programs for the SAMs present in in the final architecture configuration for HERO discussed in section 3.4. These descriptor programs will be used to perform memory operations necessary for the convolution operation to take place in the final architectural configuration. It's assumed that all on-chip SRAMs present in the HERO architecture are SAMs capable of being programmed with descriptor based programs. Since we have two operational modes based on the findings of chapter 3 (direct and GEMM mode) we will discuss the descriptor programs required for both of these modes to take place independently of the other. Note that GEMM mode is just a data transformation operation followed by a (1, 1) convolution operation as discussed in section 3.2. GEMM mode data transformations (lowering/ lifting) are assumed to be performed off chip. This leaves us with effectively two operations we need to generate descriptors for, (1, 1) and (3, 3) convolutions. For each operation, descriptor based programs need to be generated to perform the necessary data transfer operations between on-chip memories and PEs. It's assumed that all on-chip SRAMs in HERO are SAMs capable of being programmed with descriptor based memories. For now it's assumed that there exists two flexible interconnects for routing control signals between address generators and on-chip SRAMS. One between all address generators and banks in Ifmap L3 and another between all address generators and OFmap banks. These control interconnects are different from the data interconnects that allow routing of data from banks to arbitrary output ports connected PEs. An illustration of this interconnect scheme is available in Figure 6.4

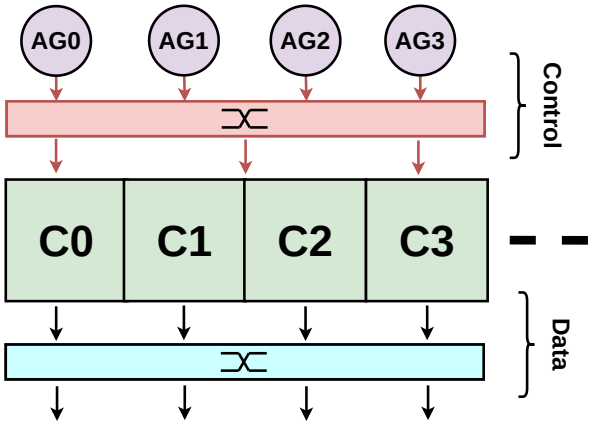



Figure 6.4: Illustration of interconnects for control and data in on-chip featuremap memories

This flexible interconnect for routing control signals allows address generators of SAMs

to be able to send read/write requests to different SRAMs connected to the same interconnect which enables arbitrary access of featuremap banks. This solution is likely not scalable to larger instances of HERO and will be superseded by statically scheduled control interconnects in future work. Additionally, all interactions between SAMs and DRAM are left as part of future work. IFmap and OFmap data is assumed to be read from and written to DRAM before and after the operation of the SAM programs discussed in thi section. Latencies and energy penalties associated with DRAM will be considered in chapter 7 but the descriptor programs discussed in this chapter are DRAM agnostic.

### 6.2.1 1x1 convolution programs

To illustrate how SAMs can be programmed to perform a (1, 1) convolution we will use a simplified version of HERO illustrated in Figure 6.5 and a (1, 1) convolution layer with 16 channels and 8 filters as a driving example. The simplified version of HERO assumed that there are a total of 9 processing engines with all 9 mapped to the accelerator horizontal spatial axis which enables creates an effective an unroll factor of 9 at kernel sizes (1, 1). Based on the tiling discussion in subsubsection 3.1.2.2, the architecture tiles the weight tensor into 16 tiles (padding included) as illustrated in Figure 6.5.

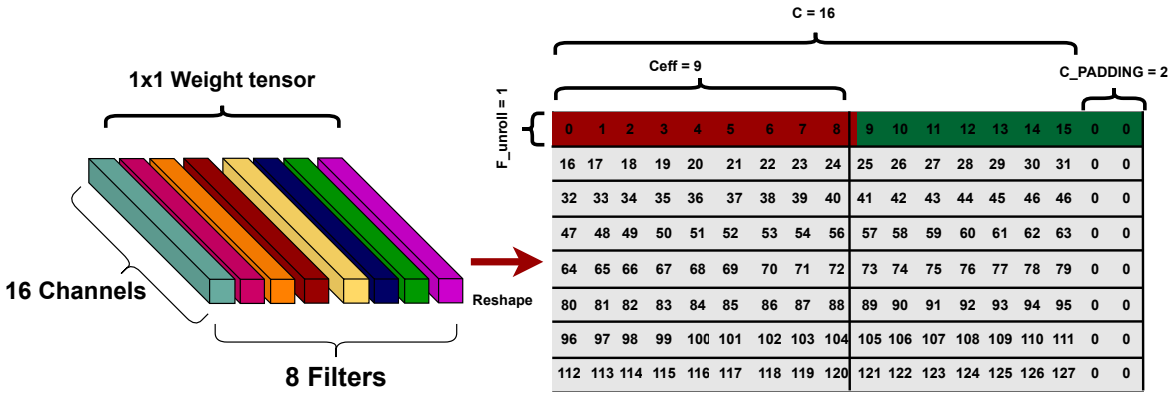



Figure 6.5: Illustration of weight tiling under (1, 1) convolution

For each weight in the tiles of Figure 6.5 the channel feature map corresponding to that weight has to be streamed into the PE processing the output of that weight. For example, tile highlighted in red needs channels C0-C8 streamed into the PEs storing those weights for processing.

After the red tile is processed, the green tile is loaded into the PEs weight buffer (assuming ASAP scheduling as seen in **??**). Channels C9-C15 then needs to be streamed into the PEs holding the weights corresponding to those channels. This means that channel memories may need to hold multiple channels that are streamed out depending on the index of the tile being processed. An illustration of how channel feature maps are stored on the channel SAMs is present in Figure 6.6. Note that in cases where channel featuremaps exceed the size of one bank, layer decomposition spreads the feature map across multiple banks. This causes a reduction in available channel concurrency which leads to low PE utilization. PEs holding 0 valued weights due to padding have no corresponding channel data so nothing is streamed into them as reflected in the 0 padding of the last channel SAM attached to the 9th PE in Figure 6.6. Additionally, they are assumed to just forward any partial sums/ output feature maps received from their input to their output with no modifications.

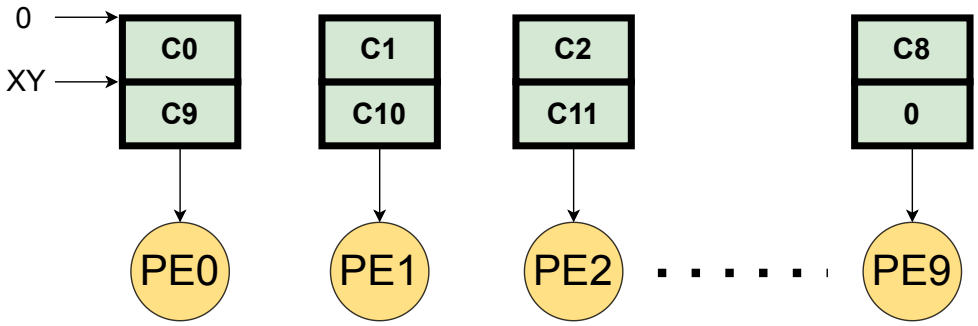



Figure 6.6: Illustration of channel banking in Ifmap L3 on-chip SRAMs

Since (1, 1) convolutions involve entire channel feature maps streamed into PEs with no reuse within a feature map occurring as in the (3, 3) case, the only memories that need to be programmed are the L3 channel memories and the OFmap memories. An illustration of the required descriptor programs for the aforementioned memories is given in Figure 6.7. In Figure 6.7 hierarchy layers unused by the computation as well as some PEs have been ommiad to for brevity.

In Figure 6.7 channel memories can be implemented with SAMs. For each channel SAM there are two descriptors types that appear frequently in their programs, a wait descriptor and generate descriptor. Channel SAMs need to perform timed reads due to the systolic delays arising from the systolic reduction of partial sums into output feature maps. Therefore, an initial wait instruction due to the systolic delay required by each SAM is inserted at the beginning of their descriptor programs. The delay defined by the wait descriptor for each SAM depends on the index of the PE that the SAM is connected to. So the first PE's corresponding SAM has no read delay so the x_count

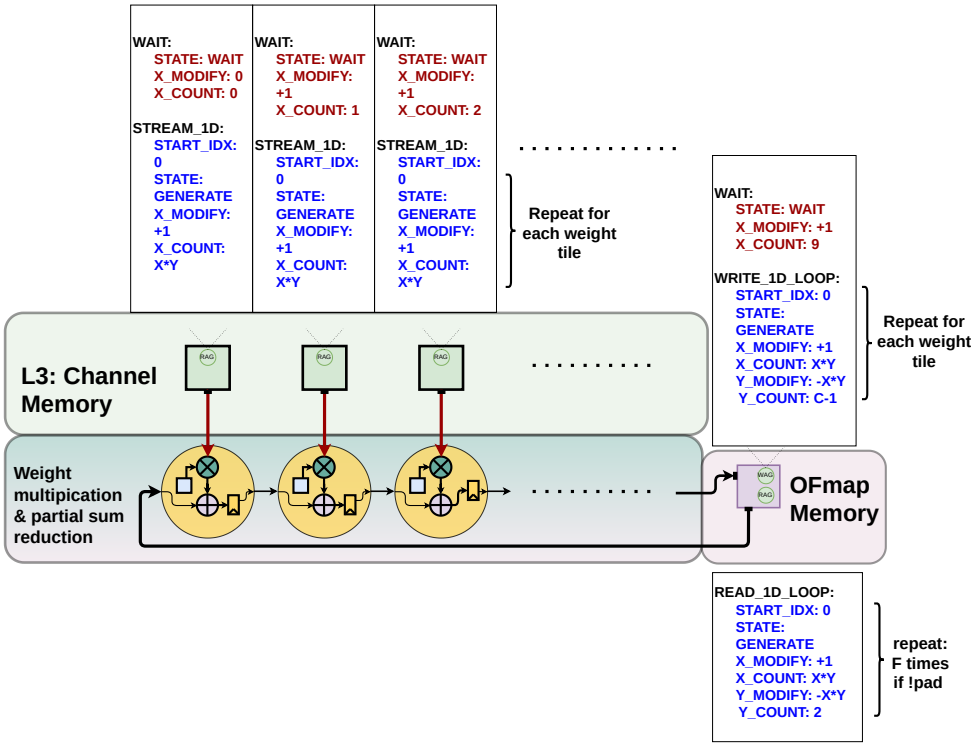



Figure 6.7: Illustration of (1, 1) convolution scheduling

variable in the wait descriptor is set to 0. The next PE's channel SAM has a delay of 1 so it's initial weight descriptor's x_count is set to 1. After the initial wait descriptor, each channel SAM needs to stream out a feature map. Depending on the index of the tile being processed, each SAM streams out a different IFmap. What distinguishes each IFmap stored on a channel SAM from another is it's start index. For example, for the first tile highlighted in red, the first PE streams out the IFmap beginning at start index 0 with a generate descriptor. The size of that IFmap is assumed to by $XY$ where $X$ and $Y$ are the width and height of the IFmap. The generate descriptor increments the internal address "addr" $XY$ times with "addr" starting at 0. The corresponding generate descriptor that manipulates the "addr" like that is a generate descriptor with an x_count of $XY$ and a start index of 0. When the first PE begins processing the second tile, it reads out the IFmap stored at index $XY$ or more generally $MOD(tile_{idx}, 2).XY$ since each filter has 2 tiles. This generate descriptor is repeated for each tile in the weight tensor assuming that no padding. If a PE is storing a 0 valued weight due to padding the generate descriptor is replaced with a wait descriptor with an x_count of $XY$. Optimizations descriptors to reduce code size is left as part of future work.

For the OFmap SAM two address generators are required due to the read modify write

nature of OFmaps. The read port begins reading the layer bias immediately and streams it into the first PE as part of partial sum reduction. All later reads from the OFmap SAM are for partial sums that have yet to be accumulated into OFmaps. The read descriptor required for streaming in bias/ partial sums is a generate descriptor that streams out the contents of OFmap in a loop. It achieves this by setting x_count to $XY$ to stream out partial sums of size $XY$ and y_count to 2 which the number of tiles in a filter. To reset the "addr" index of the read descriptor the y_modify is set to $-XY$. This generate descriptor is repeated 8 times where 8 is the number of filters present processed by the PEs assuming no filter padding. Assuming ASAP tile scheduling, OFmaps are written to DRAM as soon as they are completed and are not kept on chip.

The write port waits for $C_{UNROLL}$ number of cycles to write the first partial sums that will eventually become OFmaps once the filter being processed concludes. IFmaps of $XY$ size less than 9 (the number of PEs in the horizontal axis) will cause additional delay cycles to be introduced via wait descriptors to allow partial sums to propagate through the PEs to reach the OFmap. The descriptor required for the write address generator are similar to the the read ones with the exception of an additional wait descriptor that gives the first partial sum/ OFmap time to propagate through the systolic reduction. The delay required by the wait descriptor is equal to the number of PEs present in the horizontal axis. After the wait descriptor comes a generate descriptor that writes the partial sum output/ OFmap output into the OFmap SAM in a loop. The write generate descriptor is repeated 8 times as well assuming no padding similar to the read generate descriptor. All descriptor based programs in the 1x1 convolution program are terminated with suspend descriptors.

### 6.2.2 3x3 convolution programs

Since (3, 3) convolution operations involve the ifmap L2 vertical reuse memory, data transfer operations between L3 an L2 have to be performed by the descriptor based programs in memories of both layers. Thankfully, the programs for L3 memories are similar to the (1, 1) convolution case where each memory streams a series of ifmap channels after a systolic delay. The main difference between (3, 3) and (1, 1) L3 programs is is the existence of an initial setup phase where the first two lines of a featuremap are transferred into the L2 followed by a delay that allows partial sums to propagate down to the set of PEs each L3 featuremap bank is connected to. Once this setup phase concludes the rest of the featuremap is streamed out in the run phase. These two phases are repeated for every weight tile in the layer. The L2 memory acts as a series of programmable FIFOs that emit featuremaps after a delay determined by the width of the featuremap. They also have an

additional setup phase that enables them to receive the first two lines from L3 channel memories and then wait until partial sums begin propagating to the OFmap memory before entering their run phase. An illustration of this behavior is given in Figure 5.4.b. All descriptor programs in the 3x3 convolution program are terminated with suspend descriptors.

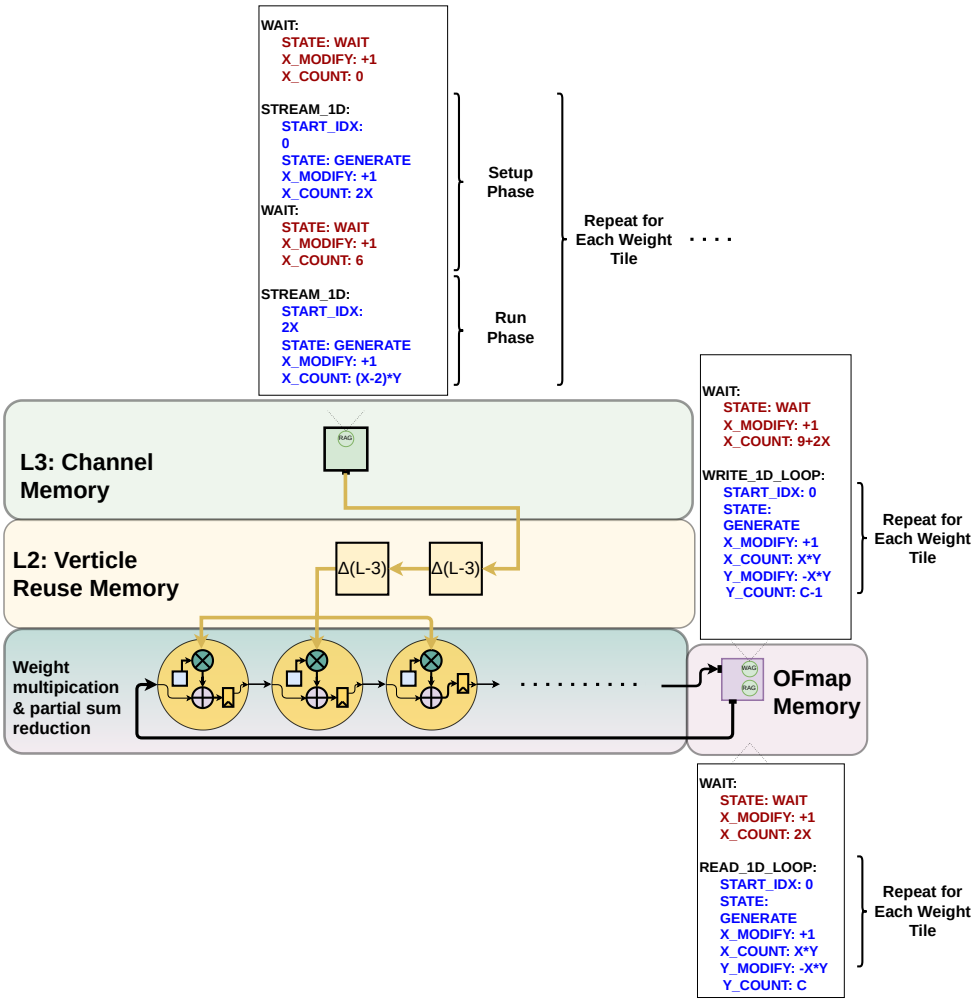



Figure 6.8: Illustration of (3, 3) convolution scheduling

# Chapter 7

# HERO Architecture Simulation

To assess the performance of the HERO architecture a cycle accurate simulation environment was implemented using SystemC and python. The environment was used to a single configurations of the HERO architecture on real networks implemented in pytorch from the TIMM library. The simulator also enables a study of the HERO architecture by allowing different configurations of HERO to be passed in during simulation. This study is left as part of future work. The simulator is composed of two parts. 1) A python based frontend that can analyze the performance results of any pytorch network on a configuration of HERO. 2) A SystemC based backend for cycle accurate simulations of HERO running convolution/linear layers. In this chapter we will first discuss the simulation environment and it's components in section 7.1. The results from simulating all 695 networks from TIMM using a small HERO instance are given section 7.2.

## 7.1 Simulation Environment

### 7.1.1 Python Frontend

The python frontend depicted in Figure 7.1, expects a list of pytorch models and a dictionary defining the configuration of a HERO instance as inputs. The configuration specifies the unroll factors for F, C, the list of kernels sizes supported in direct mode, and the on-chip storage dedicated for input and output feature map reuse, as well as the maximum line size in any IFmap tensor. The output of the frontend is a pandas dataframe containing the results from the backend simulation wof the network(s) on the specified hero instance. After the frontend receives the list of pytorch models, CIGAR extracts the dimensions of all supported layers (Conv2D and Linear). After which two layer

transformation operations are performed. The first converts convolution layers that are unsupported directly into equivalent (1, 1) convolutions as described in section 3.2. Additionally, linear layers are reinterpreted as (1, 1) convolutions based on the proof derived from subsection 3.2.1. The HERO arch. config being evaluated is used to determine what layers are supported directly After all layers have been converted to equivalent layers that are supported by HERO, a layer decomposition step is performed to decompose layers that have featuremaps that are two large to fit in the HERO's on-chip memory. After all layer transformations are performed, the layers are converted to test cases and a testcase deduplication operation is used to limit the number of simulation runs required to simulate the network(s) on HERO. After the testcase queue is generated it gets passed off to several worker threads that manage independent backend instances that simulate the different layers of the input network(s) concurrently. Note that since we're only concerned with the architecture metrics produced by the simulation the layers being simulated don't use real weights and featuremaps from a forward inference pass of the original network. Instead equivalent layer weights and featuremaps are instantiated in the backend to avoid unnecessary transfer of data between the front and backend.

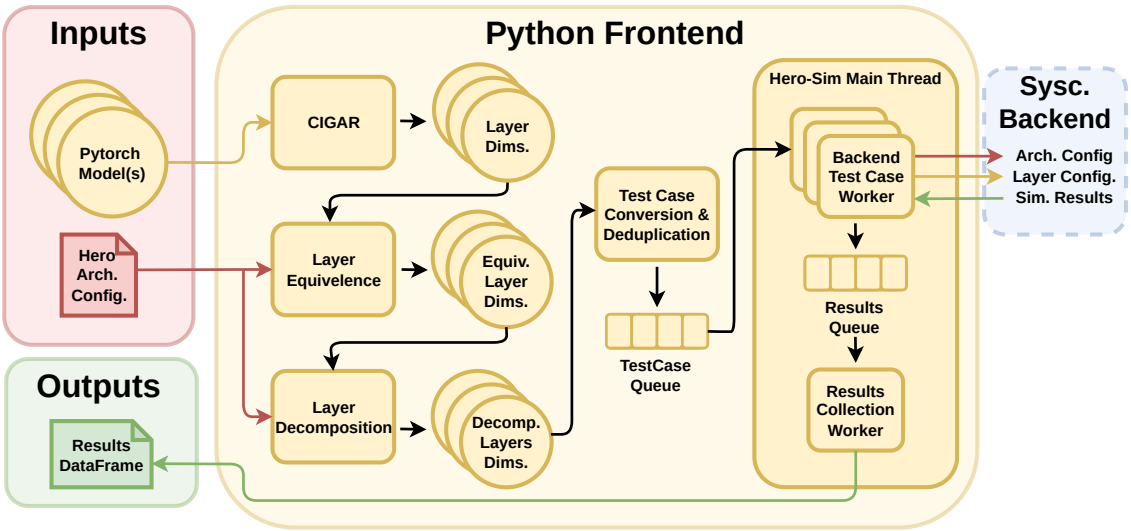



Figure 7.1: Illustration of the simulation environment's python frontend

### 7.1.2 SystemC backend

The SystemC backend depicted in Figure 7.2, expects all inputs to be passed in as command line arguments. The main inputs are 1) architecture configuration and 2) layer configuration needed for simulation by the frontend. Sim results are generated and sent back to the frontend us-

ing protobufs. The backend first instantiates a HERO instance using the architecture configuration passed as input. Then it generates an equivalent layer based on the input layer configuration. Finally using both layer and architecture configuration it generates the SAM descriptors required to perform all data movement operations on-chip. A dram load is then simulated in zero time to transfer the layer data and descriptors to the on-chip memories of the newly constructed HERO instance. Once the initialized HERO instance is ready, the cycle accurate simulation starts and continues until all SAMs reach a suspend descriptor. Then a dram load operation is again performed in zero time to transfer the results from the HERO instance for validation. After the simulated dram load, a protobuf is constructed, then populated with the result of output validation and simulation results (assuming validation success) and sent back to the frontend for further analysis and aggregation.

The backend simulates interactions between the architecture and DRAM functionally, in zero-time to avoid the complexity of DRAM and SAM interactions. The interaction between SAMs and DRAM are left as part of future work.

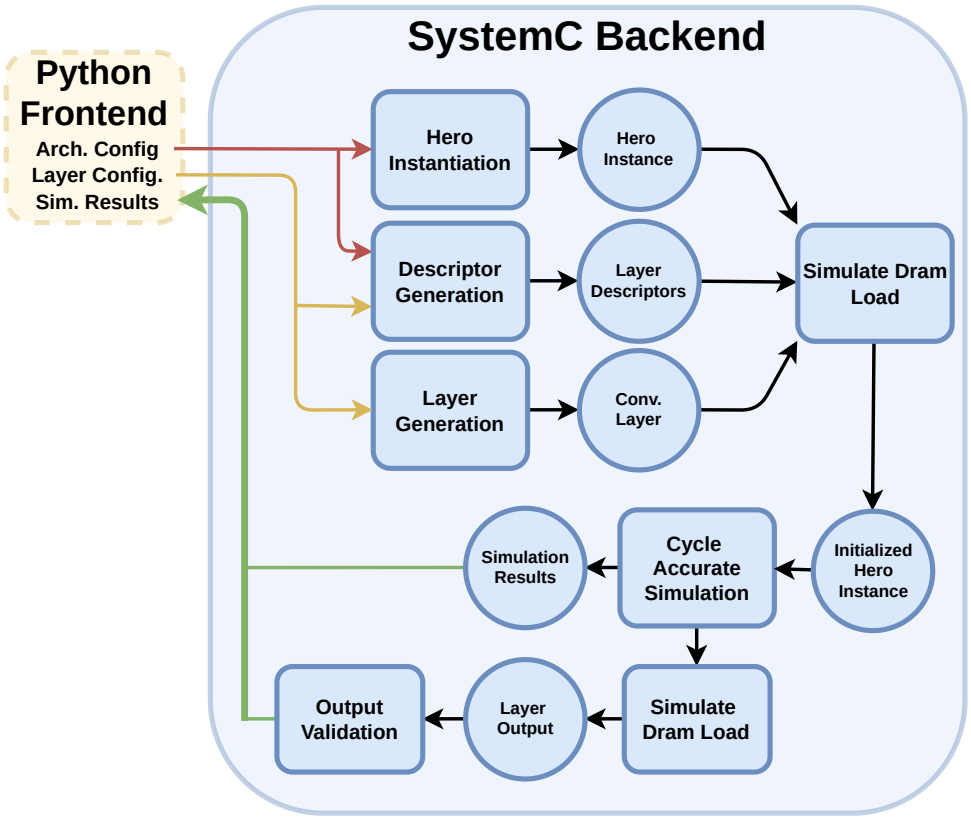



Figure 7.2: Illustration of the simulation environment's SystemC backend

## 7.2 Experimental Results

The configuration for the simulated architecture is given in Table 7.1 based on the dimensions discovered in section 4.4 and the sizing conclusions from section 4.5. Note that other larger HERO variants could have been chosen for this work, however, to minimize simulation time and to explore a variant of HERO fit for more constrained embedded environments the configuration in Table 7.1 were chosen. Note that sizes for on-chip storage are given in the aggregate. The number of banks for IFmap L3 and OFmap storage scales with the chosen $F_{unroll}$ and $C_{unroll}$ factors. As a result, the size of individual IFmap L3 and OFmap bank is proportional with the overall size of IFmap L3 and OFmap on-chip storage but inversely proportional with the chosen unroll factors. The CPU baseline used in this work is an AMD 5950X processor. To minimize simulation time the frontend takes advantage of the similarity between layer configurations across networks in the timm library. The total number of both convolution and linear layers post layer configuration deduplication is reduced from 182773 to 6048. This enables a simulation time of less than 1.5 hours for all 695 Networks.

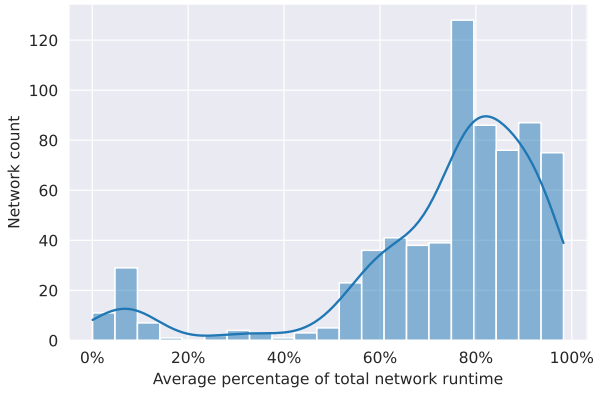



Figure 7.3: Percent of computation represented by layers studied in the TIMM library

The layers simulated (convolution and linear) represent a substantial portion of total network computation as show in Figure 7.3. To determine percentage of compute Pytorch can be used to estimate the total number of MAC operations used by natively supported layers. Unfortunately many networks use custom layers constructed with pytorch primitives. For each of these layers an analytical expression would need to be created to calculate the number of operations in these layers. For simplicity the runtime of these layers on a CPU were used to estimate computational demand.

| Config. Param. | On-Chip Storage in Bytes |
|---|---|
| Weight Storage | 16 B / PE (8 bit precision) |
| IFmap L3 Storage | 1 MB (8 bit precision) |
| IFmap L2 Storage | 512 B (8 bit precision) |
| OFmap Storage | 2 MB (16 bit precision) |
| $C_{unroll}$ | 18 |
| $f_{unroll}$ | 32 |
| Directly Supported Kernels | (1, 1), (3, 3) |
| Assumed CLK Speed | 1 Ghz |

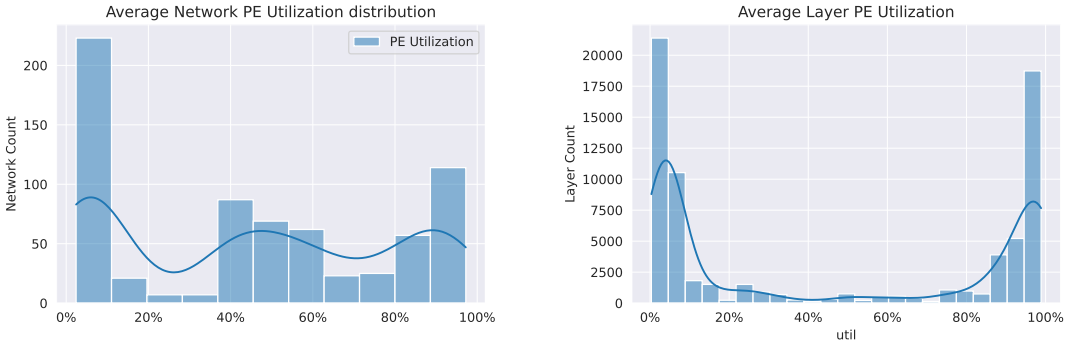
Table 7.1: HERO configuration used for analysis

In Figure 7.3, the horizontal access defines the average percentage of total network runtime taken up by supported layers. The vertical axis defines the number of networks where convolution and linear layers take up x% of the computation where x is defined on the horizontal axis. From Figure 7.3 it can be seen that the convolution and linear layers represent the majority of network runtime in the TIMM library.

### 7.2.1 Utilization

Utilization can be used as a surrogate for how well layers map to the selected HERO architecture based on the dataflow optimizations discussed in chapter 4. From figure Figure 7.4 we can see that some networks benefit substantially from the architecture while others don't. Network level distribution of average layer utilization is generally flat with about a third of networks not benefiting from the architecture due to low utilization. Low utilization is defined as average PE utilization throughout a layer's computation that's less than 50%.

At the layer level we can see a more pronounced disparity in how much the architecture benefits some layers over others. The next series of figures will explore the cause of this disparity in utilization. If the cause of low utilization in most layers is layer size as defined by number of operations then utilization would generally scale with number of operations. Figure Figure 7.5 shows that the expected trend of utilization scaling with MACS generally holds. However for a portion of the layers on the bottom right utilization is low while MACs ops are high. This low utilization would be especially concerning if combined with a speedup factor $< 1X$ over CPU baseline for that layer. One possible cause of low utilization, combined with high MAC OPs and low speedup could be a high layer memory footprint. To examine the effect of layer memory footprint




(a) (b)

Figure 7.4: Average layer utilization distributions by a) Network, b)Layer

as a cause for low utilization, a scatter plot of layer feature map size vs utilization is presented in Figure 7.6 for layers where speedup was ¡1 compared to the CPU baseline.

From Figure 7.6 it's clear that layer memory footprint does not correlate with low utilization. What does correlate low utilization is layer type. The majority of low utilization layers are depthwise convolution layers as in figure Figure 7.7.a. Depthwise layers were not considered during dataflow DSE in chapter 3. Since they weren't an optimization target the frontend decomposes depthwise layers into individual groups of convolutions and since depthwise layers are convolution layers where each IFmap channel is convolved with a kernel independently of the others, all PEs dedicated to concurrent channel processing are underutilized. From Figure 7.7.a there a few non grouped convolution layers with low low utilization and low speedup. These layers when examined more closely in Figure 7.7.b and they appear to have lower available channel/ filter concurrency as defined by the number of channels/ filters in a layer relative to the available effective channel/ filter concurrency of the architecture. In both cases of depthwise/grouped convolutions and low channel/filter count non-grouped convolutions, low utilization and low speedup can be remedied by exploiting other forms of concurrency in the architecture other that Filter/Channel/Kernel concurrency chosen by HERO.

Paradoxically, Figure 7.6 shows a cluster with layers with both high utilization and low speedup relative to the CPU baseline. When examining the Channel and Filter counts for these layers in Figure 7.8.a it can be seen that they are generally multiple orders of magnitude greater than the available effective channel/ filter concurrency. In effect these layers suffer from the opposite problem of the previous layers. The resources allocation defined in Table 7.1 is too low to

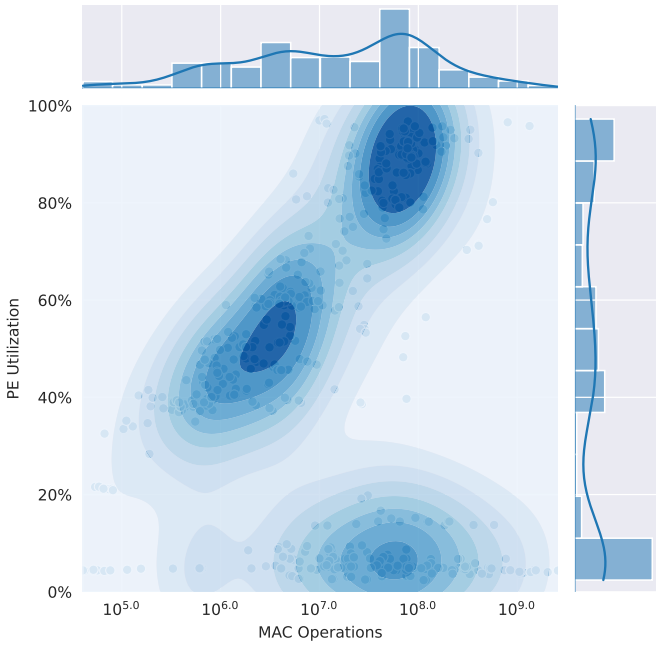



Figure 7.5: Shaded contour plot of average layer utilization vs number of MACs

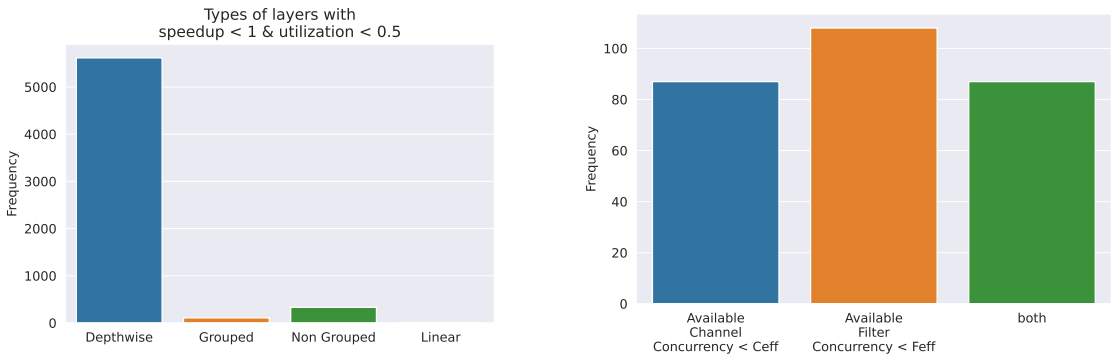
satisfy the available filter and channel concurrency in these high utilization but low speedup layer. The impact of lowering and lifting on increasing the number of MAC operations of a layer cannot be understated. When comparing the CPU baseline vs HERO, the CPU baseline has the inherent advantage of a larger and more complex memory hierarchy that enables more complicated access patterns to occur. This advantage allows CPUs to support arbitrary layers without relying on lowering/ lifting data transformations which can dramatically increase the effective number of MAC operations in a layer as seen in Figure 7.8.b where the median MAC factor increase as a result of the balanced lowering/ lifting scheme chosen by HERO is 3.8X over the original number of MACs present in the layer. In short HERO has to perform more operations for the same layers relative to CPU. To remedy this issue, more layers need to be supported directly by HERO.

### 7.2.2 Latency and speedup over CPU Baseline

The majority of the 695 Networks in the Timm library experience a speedup over CPU baseline on HERO with mean speedup being 6.8X over CPU baseline as show in Figure 7.9.a. This network level speedup is limited by layers that do not benefit from running HERO due to poor mapping to the architecture as discussed in subsection 7.2.1. In fact close to 12% of layers


(a) (b)

Figure 7.6: Scatter plot of utilization vs layer a) IFmap size and b) OFmap size. The dotted blue line represents the on-chip memory allocated for both types of feature maps.




(a) (b)

Figure 7.7: Illustration of the causes of low utilization due to A) Layer type and B) Layer dimensions

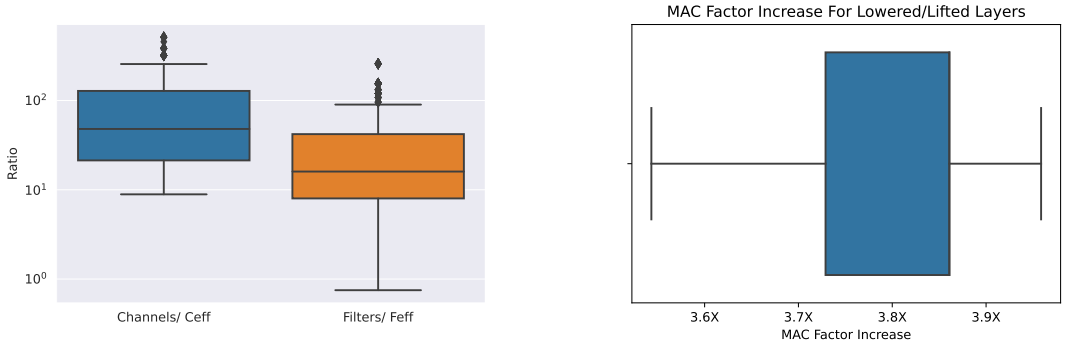



Figure 7.8: A) Barplot of the ratios between layer channel/ filters vs effective available channel/ filter concurrency for layers with speedup $< 1$ and utilization $> 95\%$ b) Barplot of the MAC factor increase due to lowering/lifting for layers with speedup $< 1$ and utilization $> 95\%$

experience a <1X speed relative to CPU baseline as see in Figure 7.9.b. Offloading these poorly supported layers to the CPU yields Figure 7.9.c where the mean speedup over CPU baseline is 30X. Note that these results are restricted to not only the simulated config in Table 7.1 but the AMD 5950X cpu used for comparison. The 5950X is a workstation CPU with a TDP greatly in excess of what's expected in the small embedded environments the simulated configuration in Table 7.1 of HERO is expected to run in.

Figure 7.9.d shows the FPS for layers supported by HERO. The median FPS of supported layers is 91 FPS with a median speedup of 4.87X over CPU baseline. Since HERO's support is restricted to convolution and linear operations these FPS results should serve only as an upper bound estimate for the actual FPS when running these networks on HERO. An exploration of including other unsupported layers into HERO is left as part of future work.

### 7.2.3 DRAM Bandwidth

Figure 7.10 show the histograms of the load/ store and combined load store DRAM bandwidth necessary to keep the simulated HERO instance from stalling. The median required combined DRAM bandwidth is reflected in Table 7.2. The maximum combined bandwidth for any network in the Timm library is 18.3 GiB/s (19.65 GB/s) which is within the ddr4 specification for PC4-21300 DDR4 SDRAM modules [21] capable of a 21.3 GB/s transfer rate.

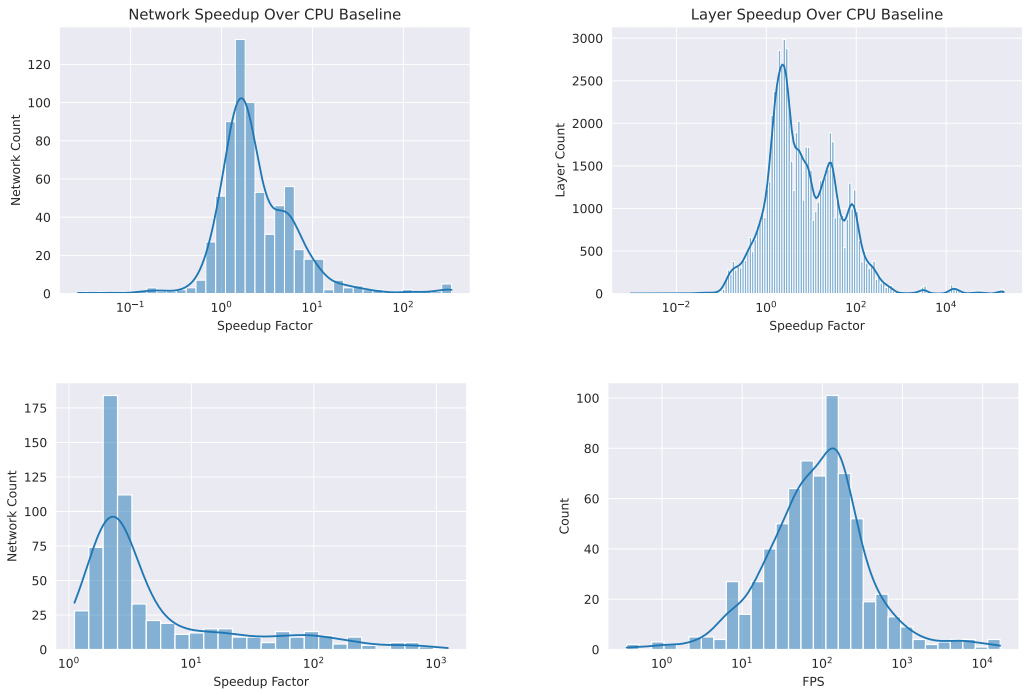



Figure 7.9: a) Histogram of average network speedup over CPU baseline b) Histogram of layer speedup over CPU baseline c) Histogram of average network speedup with offloading of poorly mapped layers with low utilization and low speedup c) Histogram of FPS upper bounds for simulated networks

| Bandwidth Type. | Bandwidth in GiB/s |
|---|---|
| Load | 3.466823 |
| Store | 1.411312 |
| Combined | 4.940233 |

Table 7.2: Median bandwidth requirements for simulated networks

| Operation | Energy Calculation |
|---|---|
| DRAM Access | 160pJ/B |
| On-Chip SRAM Access | 50+0.022$\sqrt[2]{S}$ |
| MAC | 767uJ/op |

Table 7.3: Energy model from [9]

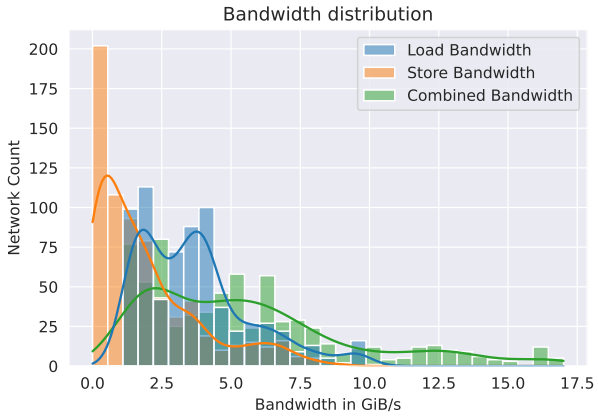



Figure 7.10: Hardware Implementation Taxonomy adapted from [13]

### 7.2.4 Energy

To estimate energy usage during HERO's operation the energy model from [9] was used. The cost of on-chip and off-chip communication is not considered as part of this work. A breakdown of the energy used by different operations from the energy model in [9] is given in Table 7.1. Note that while DRAM access and MAC operations are given as constants multiplied by number of accesses and number of operations, the access energy for On-Chip sram is a function of the square root of size of the on-chip SRAM in bits. From Figure 7.11 it's clear that DRAM energy dominates during an inference of most networks. The energy consumed by on-chip accesses of OFmap memory banks is predictably lower than accesses for IFmap banks given that there's more of them and they are each smaller memories in comparison to IFmap banks (see Table 7.1). Reuse chain energy is largely negligible. MAC energy is non negligible and is similar to weight memory access energy. Table 7.4 shows a numerical breakdown of each of the different operations and the median energy they consumed during inferences passes of networks in the Timm library.

Figure 7.12 shows a histogram of the number of inferences/J of energy consumed. From Figure 7.12.a and Figure 7.12.b DRAM energy cost is, unsurprisingly, highly influential on inferences/J. The median inferences/J jumps from 57 Inference/J to 17094 Inferences/J. It's clear that further reducing off-chip data movement by exploiting more types of concurrency is required.

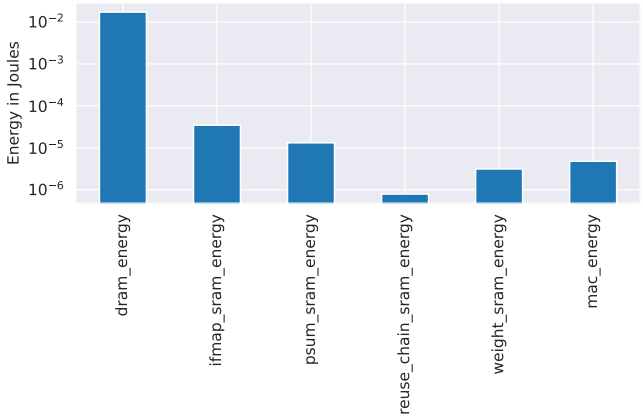



Figure 7.11: Median network energy breakdown by operation type, note that the vertical axis uses a log scale

| Operation | Energy Calculation |
|---|---|
| DRAM access | 1.725008e-02 J |
| IFmap SRAM access | 3.461726e-05 J |
| OFmap SRAM access | 1.316624e-05 J |
| Reuse Chain SRAM access | 7.892650e-07 J |
| Weight Register access | 3.130283e-06 J |
| MAC operation | 4.778069e-06 J |

Table 7.4: Energy model from [9]

### 7.2.5 Area

To estimate on-chip area for the configurations in Table 7.1 the area model from [9] was used. The area model from [9] is presented in Table 7.5. Figure 7.13 shows the area model applied to the configuration in Figure 7.13 where the vast majority of on-chip area is consumed by IFmap and OFmap SRAM banks. OFmap memory requires close to double the area for IFmap memory given the increase in precision required by accumulation. The total estimated on-chip area required for this configuration of HERO is 0.34 $mm^2$ using the area model in Table 7.5 adapted from [9].

### 7.2.6 Per network results

In addition to the library results presented in the previous subsections this subsections shows the results of running two networks, Resnet50 and Mobilenetv3 on HERO. Resnet50 is a popular image recognition model used as a backbone within many networks within the Timm li-

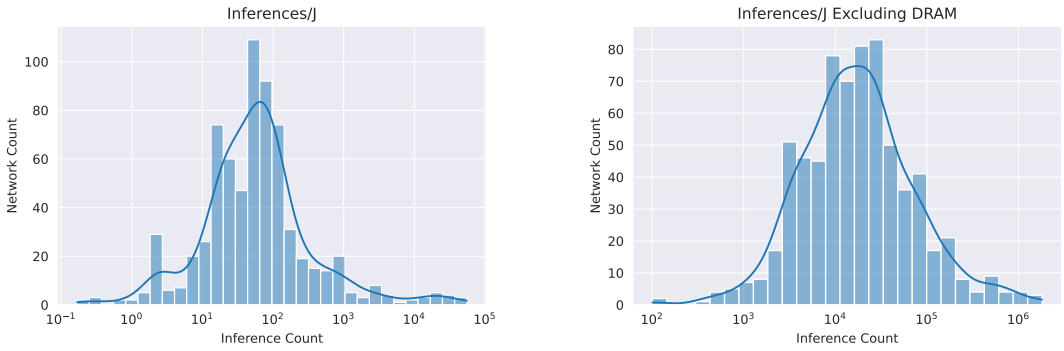



Figure 7.12: a) Inferences/J when factoring in DRAM access cost b) Inferences/J excluding DRAM access cost

| Unit | On-Chip area |
|---|---|
| MAC | 16 $um^2$ |
| SRAM bit | 0.013 $um^2/bit$ |

Table 7.5: Area model from [9] assuming a 14nm technology node

brary. MobilnetV3 is an image recognition library developed for low power edge devices. For each network a table is shown with each layer's configuration. Accompanying the layer breakdown tables are a series of figures illustrating each network layer's speedup, latency, pe utilization, energy, bandwidth requirements, and MAC factor increase (in situations were lowering is required). Note that since each run of HERO is deterministic, layers with identical configurations are only plotted once. This an important fact to consider when observing layers with underwhelming performance on HERO. If they appear infrequently their impact on overall network performance is limited.

#### 7.2.6.1 ResNet50

From Table 7.6, Resnet50 lacks depthwise layers. Hence when run on HERO the majority of layers fully utilize the available PEs as seen in Figure 7.14. The exception to this high utilization is conv_0. Due to it's low channel count and unsupported kernel types conv_0 is lowered which substantially increases the MAC count of the layer. Additionally, because of the large kernel size lowering inflates the IFmap size necessitating a layer decomposition step that causes PEs to be underutilized. This causes an overall reduction in layer speedup. The influence of lowering extends to energy consumption with a few key layers (conv_6, conv_12, conv_18) consuming the bulk of

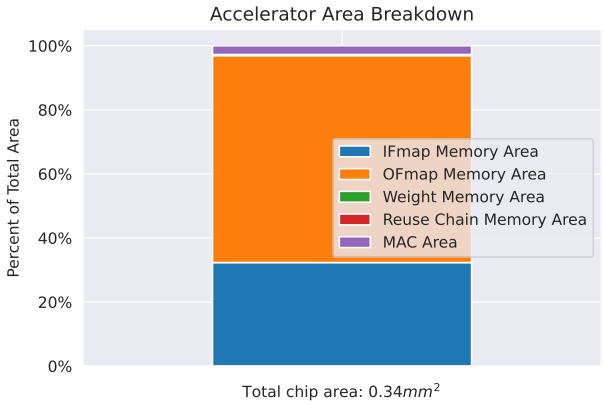



Figure 7.13: Area breakdown of HERO in the configuration specified in Table 7.1

on-chip energy because of IFmap tensor size increase. Thankfully these layers occur infrequently in the network and as a result the majority of layers do not require lowering. The overall estimated upper bound FPS for resnet50 is 62.7 FPS at 61 Inferences/J and the upper bound of dram bandwidth required to prevent HERO from stalling falls well below the PC4-21300 DDR4 SDRAM modules assumed in the energy model discussed in subsection 7.2.4.

#### 7.2.6.2 MobilenetV3

From Table 7.7, unlike Resnet50, Mobilenetv3 has depthwise layers (conv_1, conv_6, conv_12, ...) that map poorly to HERO. Depthwise layers are supported by running each convolution group sequentially which greatly diminishes the concurrency advantage of HERO. Depthwise layers suffer from PE under utilization and substantially reduced speedup over CPU baseline. Depthwise layers may also require lowering which further reduces their energy efficiency on HERO. In fact, the layers with the most on-chip energy consumption are all depthwise layers that have been lowered. Thankfully the impact of these poorly supported depthwise layers is limited given their relative infrequency. As a result the overall estimated upper bound FPS for MobilenetV3 is 928.9 FPS at 328.4 Inferences/J and the upper bound of dram bandwidth required to prevent HERO from stalling falls well below the PC4-21300 DDR4 SDRAM modules assumed in the energy model discussed in subsection 7.2.4.

| Name | Count | IFmap | Cin | Cout | K | Stride | Padding | Groups | Bias |
|---|---|---|---|---|---|---|---|---|---|
| conv_0 | 1 | (224, 224) | 3.0 | 64.0 | (7, 7) | (2, 2) | (3, 3) | 1.0 | False |
| conv_1 | 1 | (56, 56) | 64.0 | 64.0 | (1, 1) | (1, 1) | (0, 0) | 1.0 | False |
| conv_2 | 3 | (56, 56) | 64.0 | 64.0 | (3, 3) | (1, 1) | (1, 1) | 1.0 | False |
| conv_3 | 4 | (56, 56) | 64.0 | 256.0 | (1, 1) | (1, 1) | (0, 0) | 1.0 | False |
| conv_4 | 2 | (56, 56) | 256.0 | 64.0 | (1, 1) | (1, 1) | (0, 0) | 1.0 | False |
| conv_5 | 1 | (56, 56) | 256.0 | 128.0 | (1, 1) | (1, 1) | (0, 0) | 1.0 | False |
| conv_6 | 1 | (56, 56) | 128.0 | 128.0 | (3, 3) | (2, 2) | (1, 1) | 1.0 | False |
| conv_7 | 4 | (28, 28) | 128.0 | 512.0 | (1, 1) | (1, 1) | (0, 0) | 1.0 | False |
| conv_8 | 1 | (56, 56) | 256.0 | 512.0 | (1, 1) | (2, 2) | (0, 0) | 1.0 | False |
| conv_9 | 3 | (28, 28) | 512.0 | 128.0 | (1, 1) | (1, 1) | (0, 0) | 1.0 | False |
| conv_10 | 3 | (28, 28) | 128.0 | 128.0 | (3, 3) | (1, 1) | (1, 1) | 1.0 | False |
| conv_11 | 1 | (28, 28) | 512.0 | 256.0 | (1, 1) | (1, 1) | (0, 0) | 1.0 | False |
| conv_12 | 1 | (28, 28) | 256.0 | 256.0 | (3, 3) | (2, 2) | (1, 1) | 1.0 | False |
| conv_13 | 6 | (14, 14) | 256.0 | 1024.0 | (1, 1) | (1, 1) | (0, 0) | 1.0 | False |
| conv_14 | 1 | (28, 28) | 512.0 | 1024.0 | (1, 1) | (2, 2) | (0, 0) | 1.0 | False |
| conv_15 | 5 | (14, 14) | 1024.0 | 256.0 | (1, 1) | (1, 1) | (0, 0) | 1.0 | False |
| conv_16 | 5 | (14, 14) | 256.0 | 256.0 | (3, 3) | (1, 1) | (1, 1) | 1.0 | False |
| conv_17 | 1 | (14, 14) | 1024.0 | 512.0 | (1, 1) | (1, 1) | (0, 0) | 1.0 | False |
| conv_18 | 1 | (14, 14) | 512.0 | 512.0 | (3, 3) | (2, 2) | (1, 1) | 1.0 | False |
| conv_19 | 3 | (7, 7) | 512.0 | 2048.0 | (1, 1) | (1, 1) | (0, 0) | 1.0 | False |
| conv_20 | 1 | (14, 14) | 1024.0 | 2048.0 | (1, 1) | (2, 2) | (0, 0) | 1.0 | False |
| conv_21 | 2 | (7, 7) | 2048.0 | 512.0 | (1, 1) | (1, 1) | (0, 0) | 1.0 | False |
| conv_22 | 2 | (7, 7) | 512.0 | 512.0 | (3, 3) | (1, 1) | (1, 1) | 1.0 | False |
| linear_0 | 1 | (1, 1) | 2048.0 | 1000.0 | N/A | N/A | N/A | N/A | True |

Table 7.6: Resent50 layer breakdown

| Name | Count | IFmap | Cin | Cout | K | Stride | Padding | Groups | Bias |
|---|---|---|---|---|---|---|---|---|---|
| conv_0 | 1 | (224, 224) | 3.0 | 16.0 | (3, 3) | (2, 2) | (1, 1) | 1.0 | False |
| conv_1 | 1 | (112, 112) | 16.0 | 16.0 | (3, 3) | (2, 2) | (1, 1) | 16.0 | False |
| conv_2 | 1 | (1, 1) | 16.0 | 8.0 | (1, 1) | (1, 1) | (0, 0) | 1.0 | True |
| conv_3 | 1 | (1, 1) | 8.0 | 16.0 | (1, 1) | (1, 1) | (0, 0) | 1.0 | True |
| conv_4 | 1 | (56, 56) | 16.0 | 16.0 | (1, 1) | (1, 1) | (0, 0) | 1.0 | False |
| conv_5 | 1 | (56, 56) | 16.0 | 72.0 | (1, 1) | (1, 1) | (0, 0) | 1.0 | False |
| conv_6 | 1 | (56, 56) | 72.0 | 72.0 | (3, 3) | (2, 2) | (1, 1) | 72.0 | False |
| conv_7 | 1 | (28, 28) | 72.0 | 24.0 | (1, 1) | (1, 1) | (0, 0) | 1.0 | False |
| conv_8 | 1 | (28, 28) | 24.0 | 88.0 | (1, 1) | (1, 1) | (0, 0) | 1.0 | False |
| conv_9 | 1 | (28, 28) | 88.0 | 88.0 | (3, 3) | (1, 1) | (1, 1) | 88.0 | False |
| conv_10 | 1 | (28, 28) | 88.0 | 24.0 | (1, 1) | (1, 1) | (0, 0) | 1.0 | False |
| conv_11 | 1 | (28, 28) | 24.0 | 96.0 | (1, 1) | (1, 1) | (0, 0) | 1.0 | False |
| conv_12 | 1 | (28, 28) | 96.0 | 96.0 | (5, 5) | (2, 2) | (2, 2) | 96.0 | False |
| conv_13 | 2 | (1, 1) | 96.0 | 24.0 | (1, 1) | (1, 1) | (0, 0) | 1.0 | True |
| conv_14 | 2 | (1, 1) | 24.0 | 96.0 | (1, 1) | (1, 1) | (0, 0) | 1.0 | True |
| conv_15 | 1 | (14, 14) | 96.0 | 32.0 | (1, 1) | (1, 1) | (0, 0) | 1.0 | False |
| conv_16 | 2 | (14, 14) | 32.0 | 192.0 | (1, 1) | (1, 1) | (0, 0) | 1.0 | False |
| conv_17 | 2 | (14, 14) | 192.0 | 192.0 | (5, 5) | (1, 1) | (2, 2) | 192.0 | False |
| conv_18 | 2 | (1, 1) | 192.0 | 48.0 | (1, 1) | (1, 1) | (0, 0) | 1.0 | True |
| conv_19 | 2 | (1, 1) | 48.0 | 192.0 | (1, 1) | (1, 1) | (0, 0) | 1.0 | True |
| conv_20 | 2 | (14, 14) | 192.0 | 32.0 | (1, 1) | (1, 1) | (0, 0) | 1.0 | False |
| conv_21 | 1 | (14, 14) | 32.0 | 96.0 | (1, 1) | (1, 1) | (0, 0) | 1.0 | False |
| conv_22 | 1 | (14, 14) | 96.0 | 96.0 | (5, 5) | (1, 1) | (2, 2) | 96.0 | False |
| conv_23 | 1 | (14, 14) | 96.0 | 40.0 | (1, 1) | (1, 1) | (0, 0) | 1.0 | False |
| conv_24 | 1 | (14, 14) | 40.0 | 120.0 | (1, 1) | (1, 1) | (0, 0) | 1.0 | False |
| conv_25 | 1 | (14, 14) | 120.0 | 120.0 | (5, 5) | (1, 1) | (2, 2) | 120.0 | False |
| conv_26 | 1 | (1, 1) | 120.0 | 32.0 | (1, 1) | (1, 1) | (0, 0) | 1.0 | True |
| conv_27 | 1 | (1, 1) | 32.0 | 120.0 | (1, 1) | (1, 1) | (0, 0) | 1.0 | True |
| conv_28 | 1 | (14, 14) | 120.0 | 40.0 | (1, 1) | (1, 1) | (0, 0) | 1.0 | False |
| conv_29 | 1 | (14, 14) | 40.0 | 240.0 | (1, 1) | (1, 1) | (0, 0) | 1.0 | False |
| conv_30 | 1 | (14, 14) | 240.0 | 240.0 | (5, 5) | (2, 2) | (2, 2) | 240.0 | False |
| conv_31 | 1 | (1, 1) | 240.0 | 64.0 | (1, 1) | (1, 1) | (0, 0) | 1.0 | True |
| conv_32 | 1 | (1, 1) | 64.0 | 240.0 | (1, 1) | (1, 1) | (0, 0) | 1.0 | True |
| conv_33 | 1 | (7, 7) | 240.0 | 72.0 | (1, 1) | (1, 1) | (0, 0) | 1.0 | False |
| conv_34 | 3 | (7, 7) | 72.0 | 432.0 | (1, 1) | (1, 1) | (0, 0) | 1.0 | False |
| conv_35 | 2 | (7, 7) | 432.0 | 432.0 | (5, 5) | (1, 1) | (2, 2) | 432.0 | False |
| conv_36 | 2 | (1, 1) | 432.0 | 112.0 | (1, 1) | (1, 1) | (0, 0) | 1.0 | True |
| conv_37 | 2 | (1, 1) | 112.0 | 432.0 | (1, 1) | (1, 1) | (0, 0) | 1.0 | True |
| conv_38 | 2 | (7, 7) | 432.0 | 72.0 | (1, 1) | (1, 1) | (0, 0) | 1.0 | False |
| conv_39 | 1 | (1, 1) | 432.0 | 1024.0 | (1, 1) | (1, 1) | (0, 0) | 1.0 | True |

Table 7.7: Mobilenetv3 layer breakdown

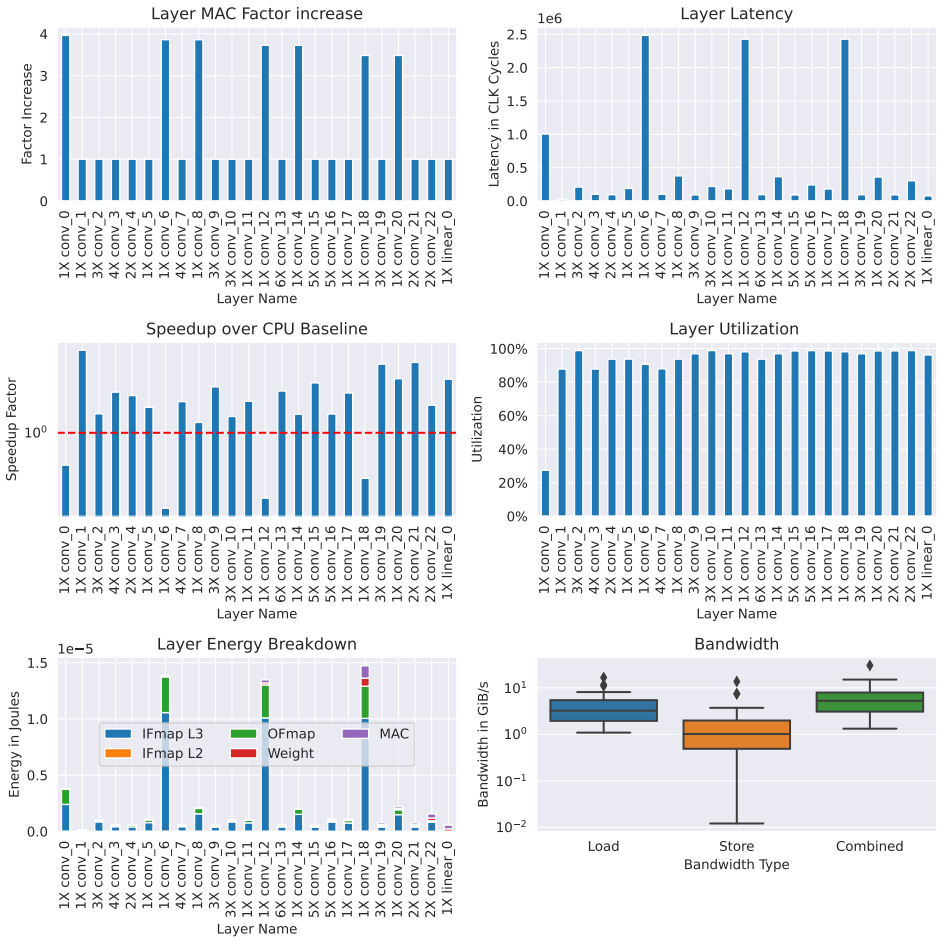



Figure 7.14: Resnet50 performance results on HERO with the configuration specified in Table 7.1

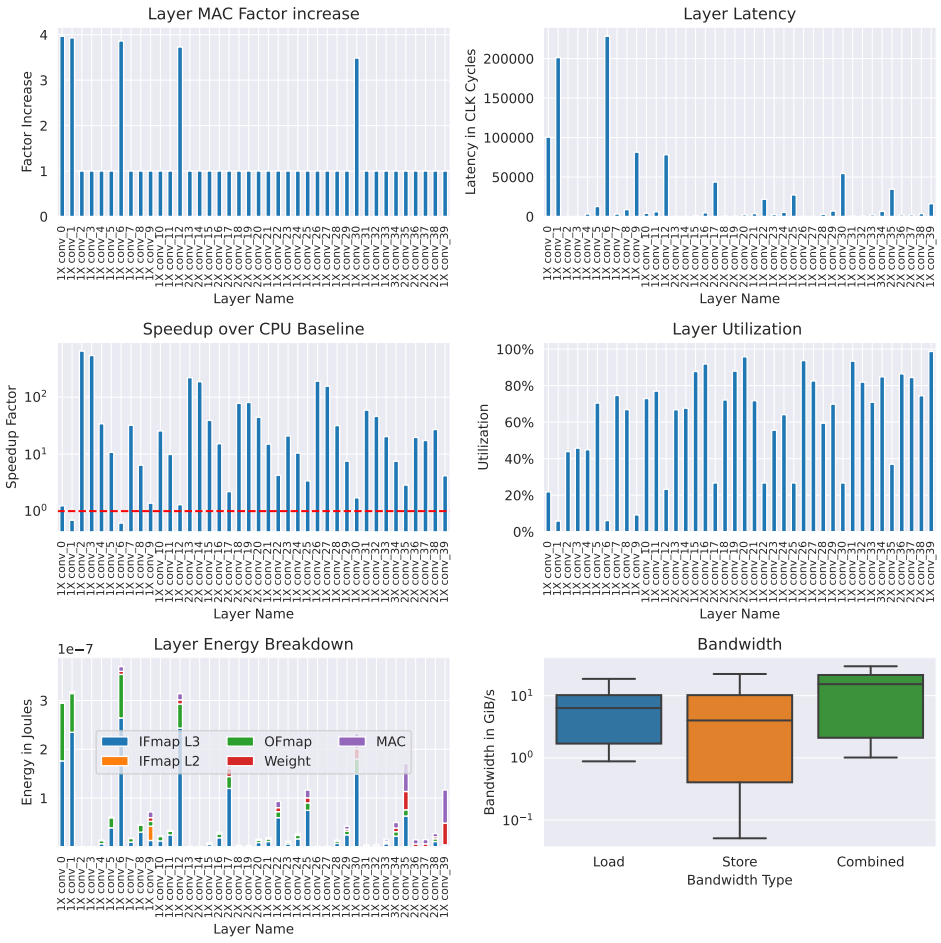



Figure 7.15: Hardware Implementation Taxonomy adapted from [13]

# Chapter 8

# Conclusion

In this thesis, a Hybrid GEMM and Direct Convolution Accelerator (HERO) was introduced as a neural network accelerator that preserves computation generality by supporting matrix multiplication while maintaining computational efficiency when running the common configurations of the convolution operation. The design of HERO was derived from a data-aware design process, where a ConvolutIon statIstics GAtherer (CIGAR) tool was used to identify the common case of the convolution operation that needed to be supported by HERO directly. To optimize HERO's configurations, a HERO Accelerator TEMPlate Optimizer (TEMPO) was introduced. TEMPO was driven by an analytical model that maximizes on-chip PE utilization. Furthermore, a novel descriptor-driven on-chip memory primitive called Self-Addressable Memory (SAM) was presented, which enables energy-efficient on-chip data movements within HERO. To convert arbitrary Pytorch models to SAM descriptors, a HERO layer compiler was discussed. To evaluate the performance and energy efficiency of various HERO configurations, a cycle-accurate simulation platform driven by a SystemC simulation backend and a Python evaluation frontend was developed. Finally, an analysis of an optimal HERO configuration when running 695 networks in the TIMM library was presented.

HERO was found to perform well on a wide variety of network configurations. It achieved a median FPS of 91 FPS with a median speedup of 4.87X over CPU baseline. The estimated bandwidth required for the configuration of HERO studied was 19.65GiB which is within the PC4-21300 DDR4 specification. With that configuration of DRAM the median inferences/J is 57. The total on-chip area is estimated at 0.34 $mm^2$.

In terms of utilization, the optimal configuration found by TEMPO enables some networks to benefit substantially from HERO while others less so. Instances of poor layer mapping to HERO

include depthwise and group convolution layers that were not considered when developing HERO as well as lowered layers that required more compute resources than what was available in the configuration studied.

To further the development of HERO explicit support of depthwise and grouped convolution layers is necessary along with an exploration of alternative forms of concurrency within HERO. Additionally, more convolution layer configurations need to be supported directly in HERO, specifically convolution layers with a stride size greater than 1. Along with increasing convolution support, support for other layer types like Batch normalization layers and activation layers needs to be included in HERO to minimize data movement between the CPU and HERO. Similarly, lowering and lifting transformation required to extend support to arbitrary convolution layers needs to be incorporated into HERO to minimize off-chip data movement. Finally a larger scale exploration of different HERO architectures using the developed simulation platform is required to validate and refine TEMPO's analytical model.

# Bibliography


[1] cublas. `https://github.com/clMathLibraries/clBLAS`. Accessed: 2016-05-14.

[2] Firas Abuzaid, Stefan Hadjis, Ce Zhang, and Christopher Ré. Caffe con troll: Shallow ideas to speed up deep learning. *CoRR*, abs/1504.04343, 2015.

[3] Dario Amodei, Rishita Anubhai, Eric Battenberg, Carl Case, Jared Casper, Bryan Catanzaro, Jingdong Chen, Mike Chrzanowski, Adam Coates, Greg Diamos, Erich Elsen, Jesse H. Engel, Linxi Fan, Christopher Fougner, Tony Han, Awni Y. Hannun, Billy Jun, Patrick LeGresley, Libby Lin, Sharan Narang, Andrew Y. Ng, Sherjil Ozair, Ryan Prenger, Jonathan Raiman, Sanjeev Satheesh, David Seetapun, Shubho Sengupta, Yi Wang, Zhiqian Wang, Chong Wang, Bo Xiao, Dani Yogatama, Jun Zhan, and Zhenyao Zhu. Deep speech 2: End-to-end speech recognition in english and mandarin. *CoRR*, abs/1512.02595, 2015.

[4] L Susan Blackford, Antoine Petitet, Roldan Pozo, Karin Remington, R Clint Whaley, James Demmel, Jack Dongarra, Iain Duff, Sven Hammarling, Greg Henry, et al. An updated set of basic linear algebra subprograms (blas). *ACM Transactions on Mathematical Software*, 28(2):135–151, 2002.

[5] Nicolas Carion, Francisco Massa, Gabriel Synnaeve, Nicolas Usunier, Alexander Kirillov, and Sergey Zagoruyko. End-to-end object detection with transformers. *CoRR*, abs/2005.12872, 2020.

[6] Yu-Hsin Chen, Joel S. Emer, and Vivienne Sze. Eyeriss v2: A flexible and high-performance accelerator for emerging deep neural networks. *CoRR*, abs/1807.07928, 2018.

[7] Chen, Yu-Hsin and Krishna, Tushar and Emer, Joel and Sze, Vivienne. Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks. In *IEEE In-*

*ternational Solid-State Circuits Conference, ISSCC 2016, Digest of Technical Papers*, pages 262–263, 2016.

[8] Jason Cong and Bingjun Xiao. Minimizing computation in convolutional neural networks. In Stefan Wermter, Cornelius Weber, Włodzisław Duch, Timo Honkela, Petia Koprinkova-Hristova, Sven Magg, Günther Palm, and Alessandro E. P. Villa, editors, *Artificial Neural Networks and Machine Learning – ICANN 2014*, pages 281–290, Cham, 2014. Springer International Publishing.

[9] William J. Dally, Yatish Turakhia, and Song Han. Domain-specific hardware accelerators. *Commun. ACM*, 63(7):48–57, jun 2020.

[10] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.

[11] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, 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, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance analysis of a tensor processing unit. *SIGARCH Comput. Archit. News*, 45(2):1–12, jun 2017.

[12] David Katz and Rick Gentile. *Embedded Media Processing*. Newnes (Elsevier), 2006.

[13] Hyoukjun Kwon, Michael Pellauer, and Tushar Krishna. MAESTRO: an open-source infrastructure for modeling dataflows within deep learning accelerators. *CoRR*, abs/1805.02566, 2018.

[14] Hyoukjun Kwon, Ananda Samajdar, and Tushar Krishna. Maeri: Enabling flexible dataflow mapping over dnn accelerators via reconfigurable interconnects. *SIGPLAN Not.*, 53(2):461–475, mar 2018.

[15] Qiaoyi Liu, Dillon Huff, Jeff Setter, Maxwell Strange, Kathleen Feng, Kavya Sreedhar, Ziheng Wang, Keyi Zhang, Mark Horowitz, Priyanka Raina, and Fredrik Kjolstad. Compiling halide programs to push-memory accelerators. *CoRR*, abs/2105.12858, 2021.

[16] Wim Meeus and Dirk Stroobandt. Data reuse buffer synthesis using the polyhedral model. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 26(7):1340–1353, 2018.

[17] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[18] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition, 2015.

[19] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *CoRR*, abs/1706.03762, 2017.

[20] Ross Wightman. Pytorch image models. `https://github.com/rwightman/pytorch-image-models`, 2019.

[21] Wikipedia. List of interface bit rates — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/w/index.php?title=List%20of%20interface%20bit%20rates&oldid=1108239877`, 2022. [Online; accessed 24-September-2022].

[22] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, Lukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, Keith Stevens, George Kurian, Nishant Patil, Wei Wang, Cliff Young, Jason Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, Greg Corrado, Macduff Hughes, and Jeffrey

Dean. Google's neural machine translation system: Bridging the gap between human and machine translation. *CoRR*, abs/1609.08144, 2016.

[23] Weijian Xu, Yifan Xu, Tyler Chang, and Zhuowen Tu. Co-scale conv-attentional image transformers.

[24] Xuan Yang, Mingyu Gao, Jing Pu, Ankita Nayak, Qiaoyi Liu, Steven Bell, Jeff Setter, Kaidi Cao, Heonjae Ha, Christos Kozyrakis, and Mark Horowitz. DNN dataflow choice is overrated. *CoRR*, abs/1809.04070, 2018.

[25] Xuan Yang, Mingyu Gao, Jing Pu, Ankita Nayak, Qiaoyi Liu, Steven Bell, Jeff Setter, Kaidi Cao, Heonjae Ha, Christos Kozyrakis, and Mark Horowitz. DNN dataflow choice is overrated. *CoRR*, abs/1809.04070, 2018.

[26] Chen Zhang, Guangyu Sun, Zhenman Fang, Peipei Zhou, Peichen Pan, and Jason Cong. Caffeine: Toward uniformed representation and acceleration for deep convolutional neural networks. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 38(11):2072–2085, 2019.