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

List of Tables

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-T Hybrid GEMM and Direct Convolution Accelerator Template

FC Fully Connected

SAM Self Addresable Memory

HERO-T-Sim HERO-T simulator

TEMPO accelerator TEMPlate Optimizer.

CIGAR ConvoluTional neural network Statistics GAtheRer.

SAM Self Addressable SRAMs.

Acknowledgments

Here I wish to thank those who have supported me during the process of the thesis work....

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

This is a very abstract abstract.

Chapter 1

Introduction

1.1 AI and Edge Computing

Deep Neural Network (DNN)s currently represent the state of the art in complex regression and classification problems in image recognition, sequence to sequnce learning [11], speech recognition [2]. As such they are being deployed on both cloud platforms and edge devices at scale.

ConvoluTional Neural Network (CNN)s are a variant of DNNs that 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. A convolution layer operates on multi-demension tensors as part of a CNNs feature extraction portion of the network. (Citation needed). Convolution layers exhibit a significant amount of parallel behavior and data reuse. Additionally, the tight latency, throughout and energy constraints imposed on CNN's in various enviornments, particularly on edge devices has led to the proliferation of customized hardware accelerators for CNNs with particular emphasis on accelerating convolution layers.

1.2 Convolution accelerators

Citation Needed Prior work on CNN accelerators design can broadly be classified based on 1) their target execution platform Application Specific Integrated Circuit (ASIC)/Field Programmable Gate Array (FPGA) 2) Their target for acceleration, either entire layers of the network or specifically convolution layers and finally 3) Their mathematical interpretation of the convolution operation as either fundementally a matrix operation post reorganization of its input tensor or a conventional stencil based operation. In the next section, the strengths and weaknesses of the afformentioned approaches will be discussed.

1.3 Problem Definition

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 (ACC) generality can be decomposed into 1) Convolution generality, which can be defined as the range of support convolution layers and 2) Network generality, whcih can be defined as the types convolution network layers supported

FPGAs inherently have an advantage w.r.t architecture generality given their reconfigurable nature. FPGA-centric approaches incorporate the architecture of a target CNN network into their architecture compilation process [15]. This allows FPGA-based architectures to tackle network generality by adding layer-spcific accelerators (provided they are available) at compile time, as well as tailor Convolution (Conv) ACC primitives to the target network in order to provide the appropriate amount of Conv generality and performance. The disadvantage of FPGA based architectures is 1) The need to recompile the architecture prior to deployment of a new CNN with possibly no support for a new CNN without recompilation 2) Inferior performance and energy efficiency compared to a hardened architecture. One could harden a design produced by an FPGA-based architecture compilation process. However, this will produce a design optimized for only one network and may not be general enough. To the best of this author's knoweldge, no FPGA-based ACC compilation processes incorporate more than one 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 (CITE TPU). Additionally, they tackle convolution generality by either 1) Reinterpreting convolution layers as GEMM operations (CITE TPU and GEMMINI) or 2) Creating a general enough Conv ACC capable of supporting a wide range of Conv layer dimensions directly (CITE EYERISS). Given the pace of development in DNNs, new layers like (CITE ATTENTION IS ALL YOU NEED)'s self attention layer can arise and become integral to improving DNN model performance [3]. These new layers may not be fully compatabile with the chosen accelerator primitives in the ASIC-based design approach and as a result may only be partially accelerated. Additionally, ASIC-based designs must balance their dediction of on-chip resources to convolution vs other resource intensive layers (e.g Fully Connected (FC) 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 effiicency [15]. Finally, supporting a wide range of convolutions can come at the cost of reduced performance/ energy efficiency for the statistical common case of convolutions layer dimensions in a wide range of CNNs.

1.4 Solution Overview

There exists a convolution accelerator architectural template that offers 1) Improved performance/ energy efficiency for the common case of convolution layers accross a broad range of CNNs 2) Operational flexibility to be able to compute a wide variety of convolution layers under various and possibly uncommon configurations 3) The ability to partially or fully accelerate computationly intensive layers that currently exist in the literature and ones that may arise in the future 4) The configurability that enables the architectural template to be modified based on available compute resources and a target library of networks. To satisfy the afformentioned requirements, in this thesis, the following is proposed:

1. A Hybrid GEMM and Direct Convolution Accelerator Template (HERO-T)
2. Self Addressable SRAMs (SAM) a novel on-chip memory primitive capable of orchestrating energy efficient on chip data movements within HERO-T
3. (ConvolutIon statIstics GAthere) CIGAR a tool that enables HERO-T's data driven design of HERO-T
4. accelerator TEMPlate Optimizer (TEMPO) a tool that optimizes HERO-T based on a target CNN library and available compute resources
5. HERO-T simulator (HERO-T-Sim) a cycle SystemC model of HERO-T combined with a python evaluation frontend that assesses a concrete HERO-T instance's performance and energy efficiency on a target CNN library

1.5 Thesis Structure

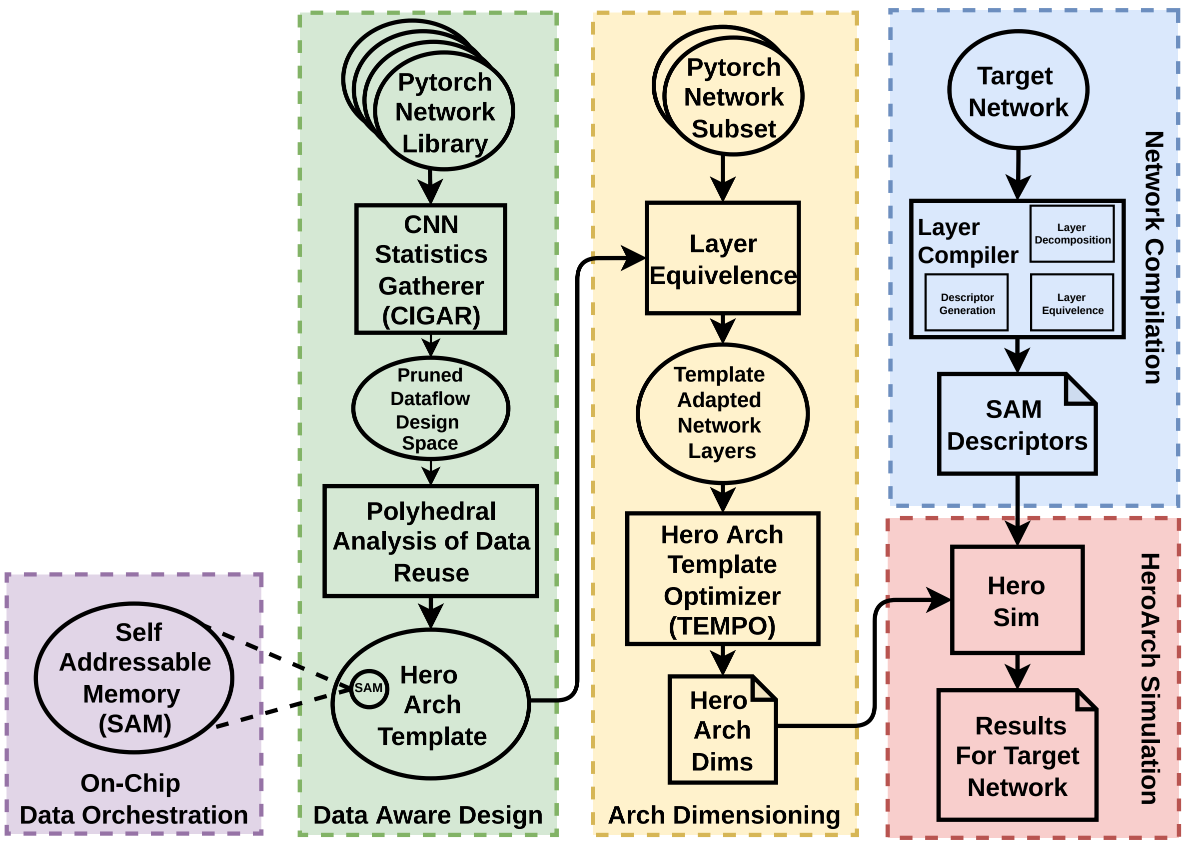


Figure 1.1: Visual illustration of this thesis's content

Chapter 2

Background

To motivate the architecture design space introduce in chapter (REFERENCE chapter) I first need to 3) show how to convert a convolution operation into a GEMM 4) Introduce dataflow exploration through direct representation if implementing an accelerates that accelerates convolutions directly 5) Introduce hardware + based on reuse behavior in the Dataflow dataflow -¿ describes communication -¿ describes datareuse 6) Introduce polyhedral model to evaluate temporal reuse behavior In this chapter I breakdown what convolutions are mathetmatically

Figure 2.1

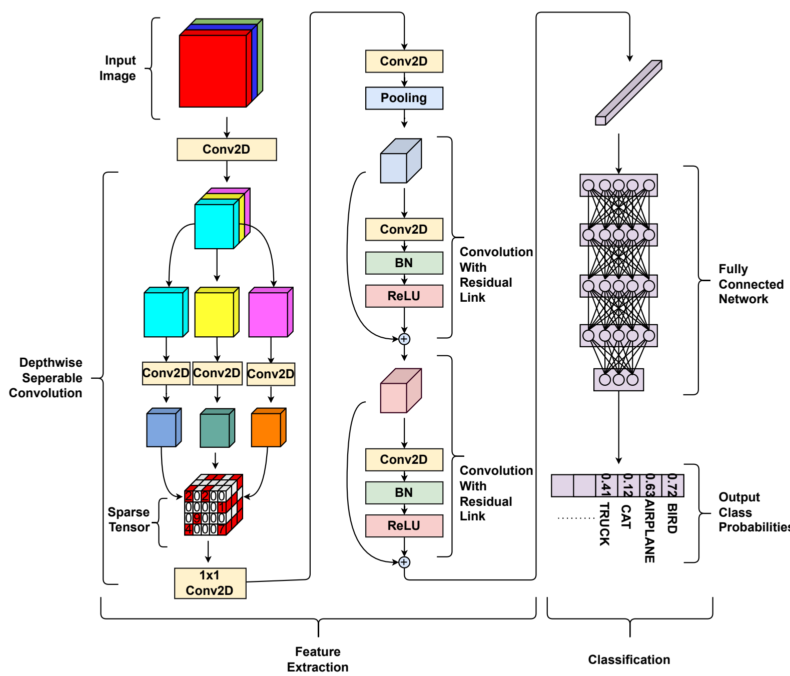


Figure 2.1: Example Image Classfication CNN

2.1 The Math behind convolutions

1) explain what convolutions are mathematically

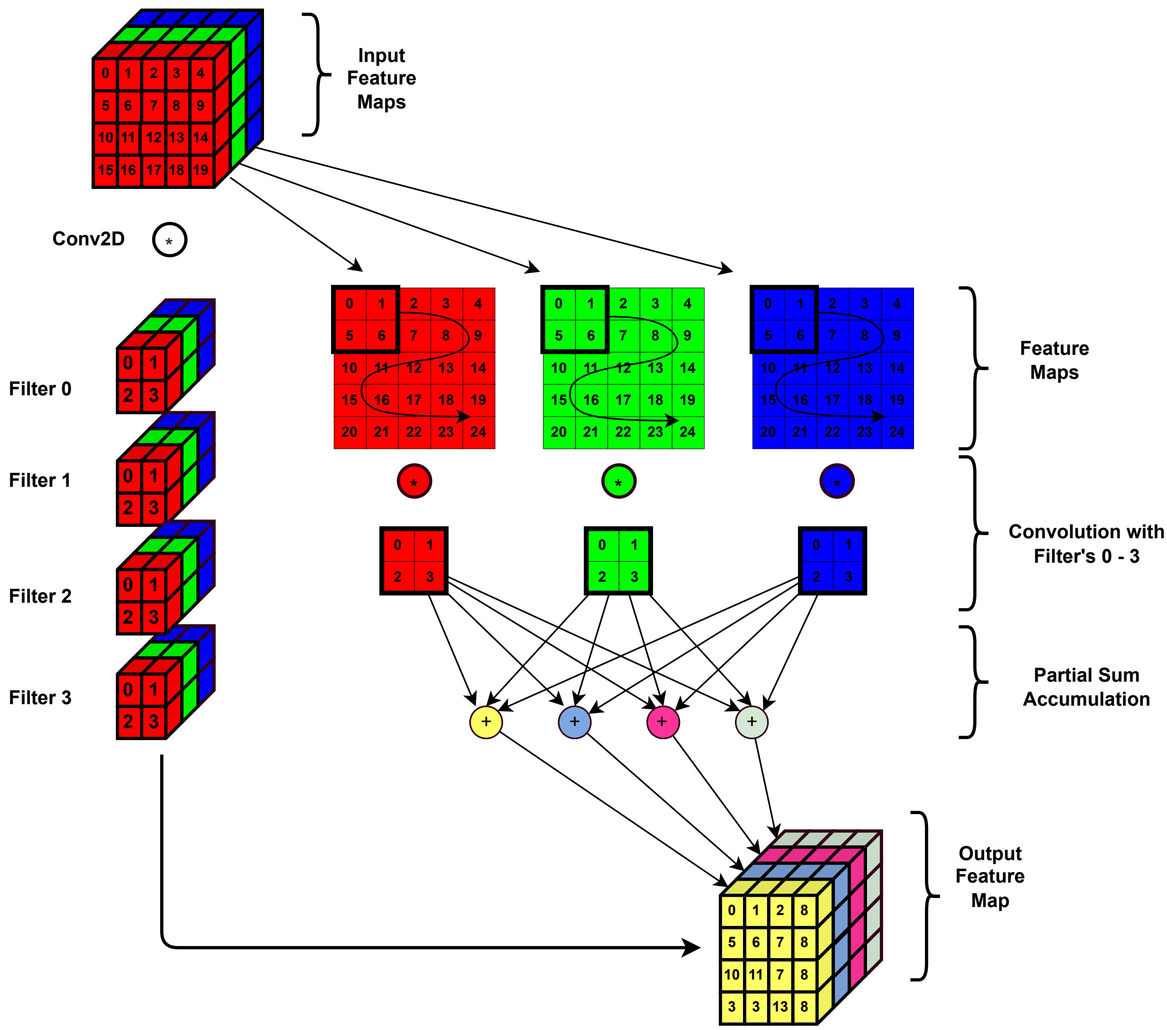


Figure 2.2: Convolution Operation Illustrated

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

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

2.2 Loop Based Representation Of Convolutions

Direct naive implementation of **??** with tensors Equation 2.1 as a series of loops is given below Listing 2.2

Listing 2.1: Convolution implemented as nested loops

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

2.3 Convolution as general matrix multiplication

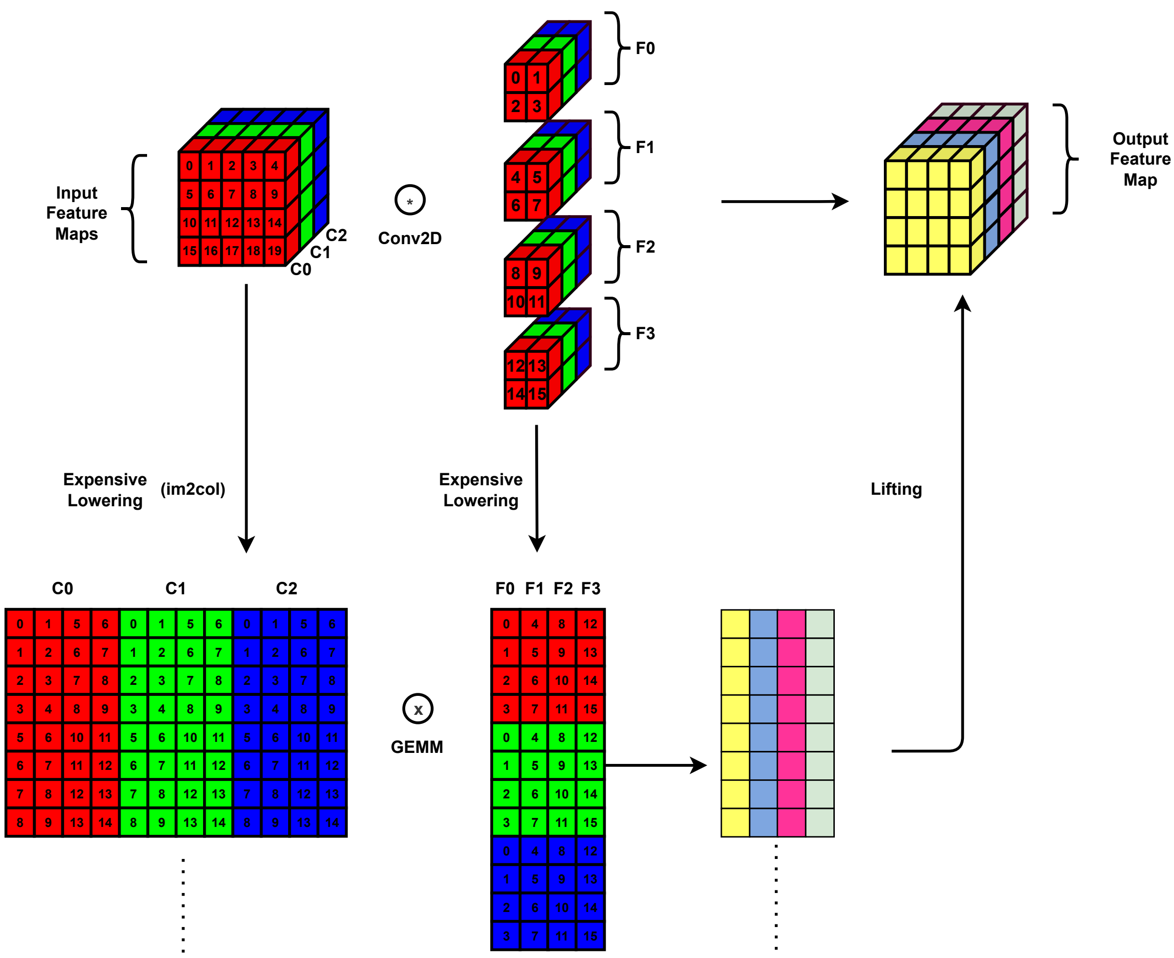


Figure 2.3: Im2Col Illustrated

Convolution can be converted into a GEMM

Popular techniques like im2col (illustrated in Figure 2.3) can be used to convert any convolution operation into GEMM (listed as expensive lowering [1]) but there are others techniques that don't produce such massive input feature maps

If willing to increase complexity of lowering of ifmap and weight tensors after GEMM we can reduce bloat in ifmap with balanced lifting/ loading in [1]. Analytical expressions adapted from [1] are given below with the inclusion of lowering in the presence of multiple filters.

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

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

In balanced lowering, we first lower the ifmap and weights using expression (2.3) and (2.4).

$$O\hat{Fmap} = I\hat{Fmap}.We\hat{i}ght \tag{2.5}$$

Then a GEMM is performed (2.5)

$$\begin{gathered} O\hat{Fmap} \in R^{nm\times FK} \xrightarrow{BalancedLifting} OFmap \in R^{m\times m\times F} \\ OFmap[f,r,c] = (\sum_{j=0}^{K-1} O\hat{Fmap}[cn+r+j,j+fK]) \\ \forall f,r,c \in [0,F-1],[0,m-1],[0,m-1] \end{gathered} \tag{2.6}$$

Followed by a lift operation on the output $O\hat{Fmap}$ matrix using (2.6)

Illustration of this data transformation is presented in Figure 2.4

Pros of lowering/ lifting

Conv as GEMM Offers the most flexibility because it's insensitive to changes in convolution parameters

It also allows leaves us with a GEMM accelerator which is useful for many other NN layers e.g Linear/ Self attention/ LSTM

Con is that it still causes bloat in ifmap and additionaly complexity/ latency of lifting/ lowering. Which in the balanced case is m^2K [1]

2.4 Implementing convolutions in hardware

2.4.1 The dataflow taxonomy

In the dataflow design space, from [13] dataflows can be represented using the direct convolution nested loop structure combined with unroll pragmas. Listing **??** shows a generic implementation of a single convolution layer as a loop structure under a weight stationary dataflow

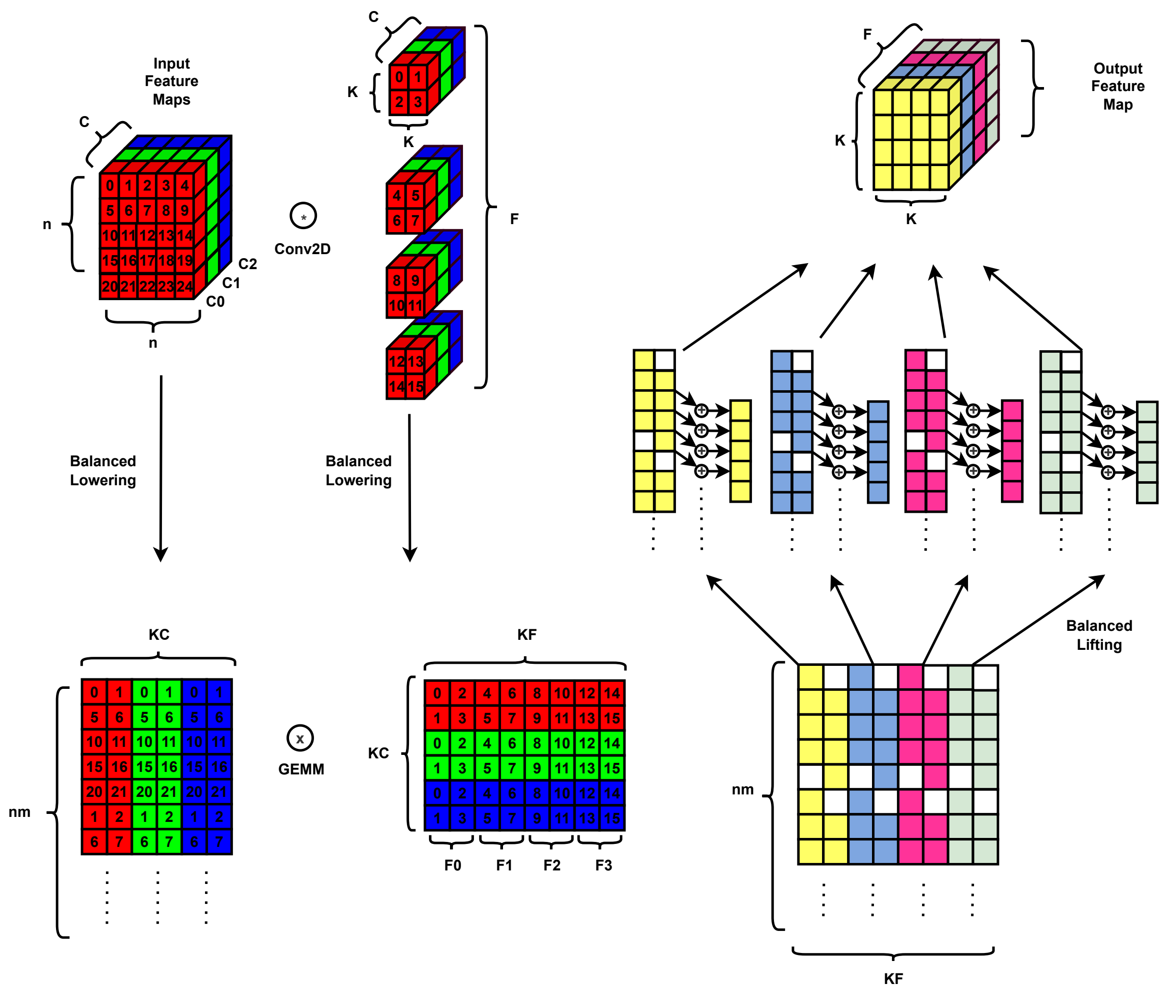


Figure 2.4: Balanced Lowering/Lifting Illustrated

configuration. What defines that dataflow is 1) loop unroll targets 2) loop order 3) the unroll factors of the unrolled loops. 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. From listing **??** 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. To summarize, from the loop representation of convolution accelerator dataflows we have three design space dimensions, 1) Loop unroll targets 2) Loop unroll factors 3) Loop spatial mapping. Given the size of this dataflow design space we will use CIGAR and Tempo to derive the common case for convolution layers and limit the scope of the dataflow design space.

CNN layer loops + loop ordering + Loop tilings can express different accelerators dataflows like row stationary, weight stationary, etc

Listing 2.2: 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
#pragma UNROLL Y_T
        for(int y = 0; y < Y; y+=Y_T) // Output feature map row
#pragma UNROLL X_T
            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 dataflow we can derive hw implementation based on communication pattern and reuse behavior of the individual data elements accessed in the convolution layer.

2.4.2 The Hardware Implementation taxonomy

Figures in Figure 2.5 show different reduction/ multicast schemes based on reuse behavior of data elements (IFmap, OFmap, Weights) apparent in the dataflow. The space of available schemes is not limited to those presented in Figure 2.5 though. In [6] a hardware taxonomy illustrated in figure **??**.

Depending on the dataflow described using the dataflow taxonomy (1) loop ordering (2)

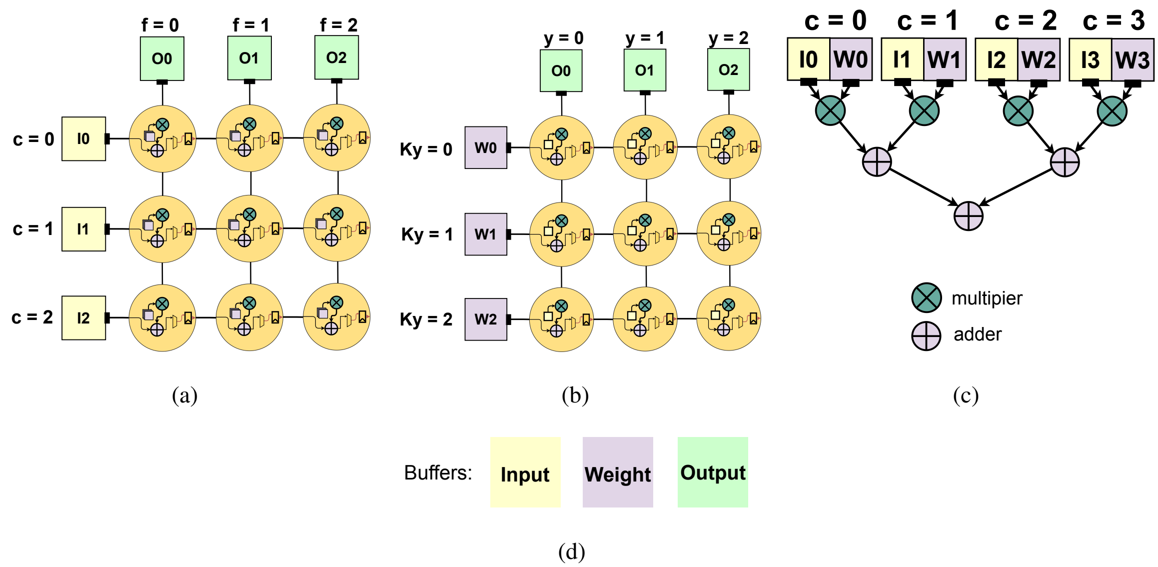


Figure 2.5: Illustration of different dataflow implementations adapted from [13] (a) blah (b) blah (c) blah (d) blah

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 [6], we can classify the available hardware implementation options based on the the type of reuse is spatial (reuse distance = 0) or temporal (reuse distance ¿ 0) that exists for a given data element accessed in the dataflow. Within a reuse type, 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 the reuse. To deduce the type of reuse and overall communicaiton behavior for each data element in any dataflow we can use the polyhedral model to detect temporal reuse. Spatial reuse detection can be inferred directly from the loops.

2.5 Analysis of data reuse with the polyhedral model

[7] used polyehdral model to analyse reuse within stencil based applications described as nested loops. One important element in their approach is their program written in iscc that can determine temporal reuse of data elements

There's the polyhedral extraction tool out there but unfortunately there's no way to encode parallelism or loop unrolling in it without relying on compiler pragmas.

below is an example of this reuse applied to the gemm loops

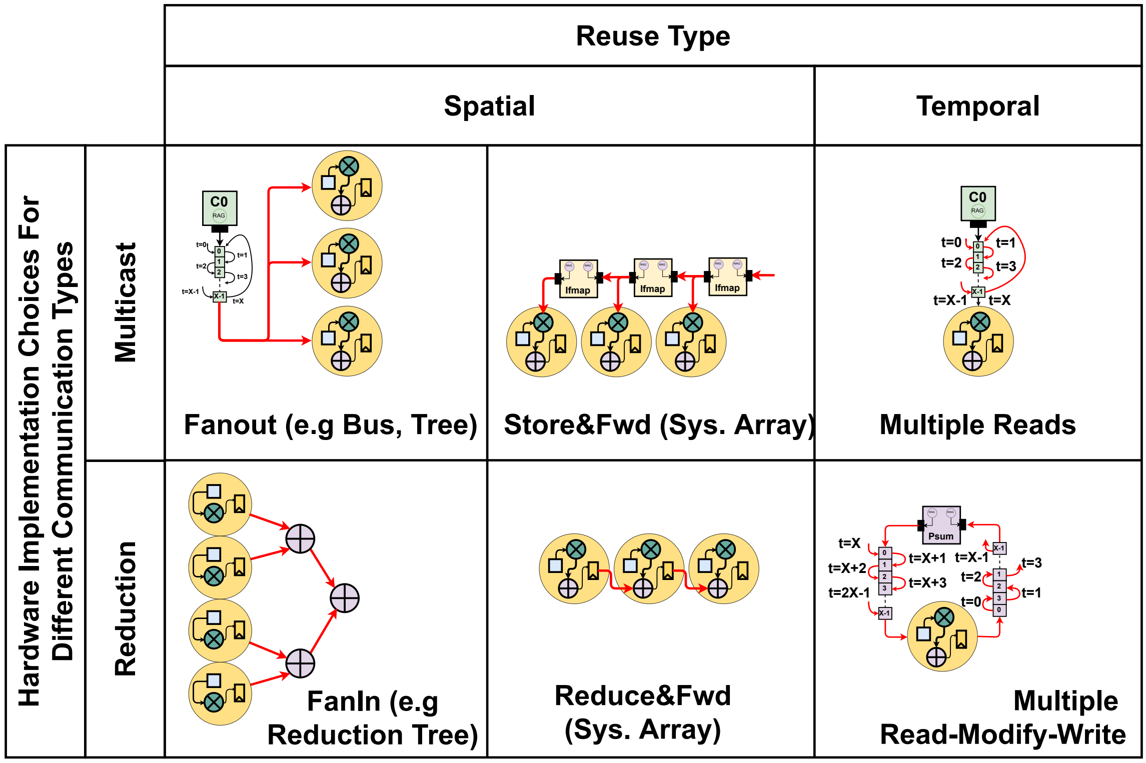


Figure 2.6: Hardware Implementation Taxonomy adapted from [6]

2.6 Related work

2.6.1 Convolution accelerator architectures

DNN accelerator generator GEMMINI doesn't optimize for direct convolutions

GEMM Based

TPU

Direct

Reduction tree based

Maeri

Systolic array based

Eyeriss

2.6.2 Convolution accelerator generation techniques

ASIC Based generators

King Kong

Horowitz

Reduction tree based

Maeri

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 [14] 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

Ideally dataflow design space exploration should be hardware implementation agnostic. We can enumerate the size of the design space by examining the scope of the afformentioned 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 upperbounds of the indexes in the loop representation $max(F).max(C).max(Y).max(X).max(KY).max(KX)$ and the upperbound 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 apprioriate 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 affornentioned 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

Psudocode 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 $model^{stats}_{dict}$ 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.

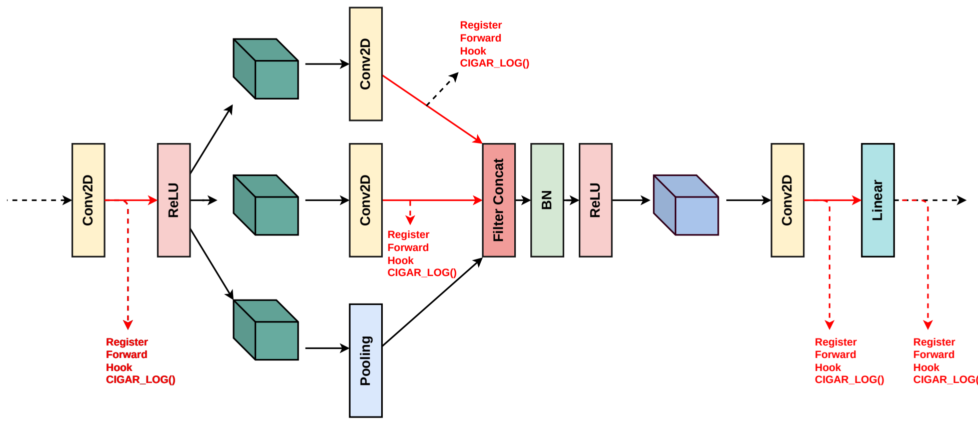


Figure 3.1: CIGAR attachment of forward hooks to all model convolution layers

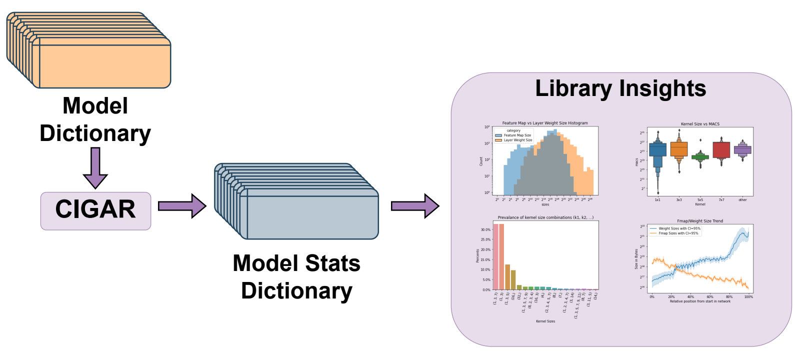


Figure 3.2: CIGAR extraction of convolution layers

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

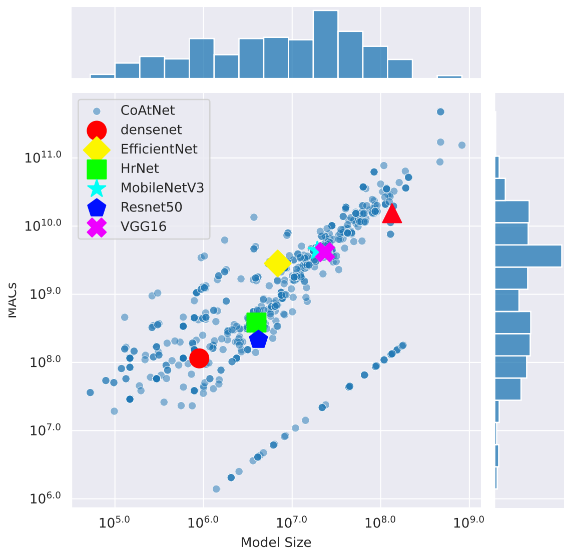
Input: $model_{dict}$

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

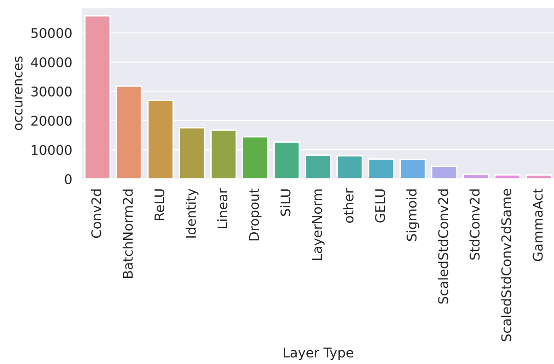
```
function ATTACH_COLLECTION_HOOKS(model, collector)
    hooks ← []
    for layer ∈ model.named_modules() do
        if type(layer) is conv2d or type(layer) is linear then
            hooks.push(layer.register_forward_hook(collector.layer_collector))
        end if
    end for
    return hooks
end function

function COLLECT_LIBRARY_STATISTICS(model_dict)
    model_dict^stats ← {}
    collector ← Collector()
    for (model_name, model) ∈ model_dict do
        input_img_tensor ← transform(open('default.jpg'), model)
        hooks ← Attach_Collection_Hooks(model, collector)
        model.forward(input)
        model_dict^stats[model_name] ← collector.model_stats()
        collector.reset()
        hooks ← 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 diversity



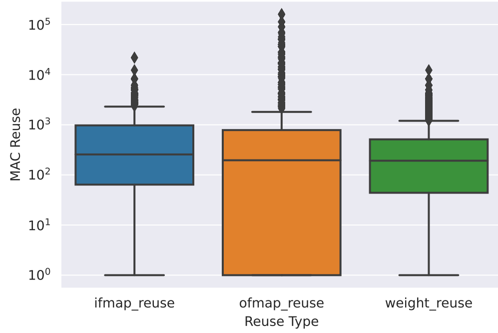
(a)



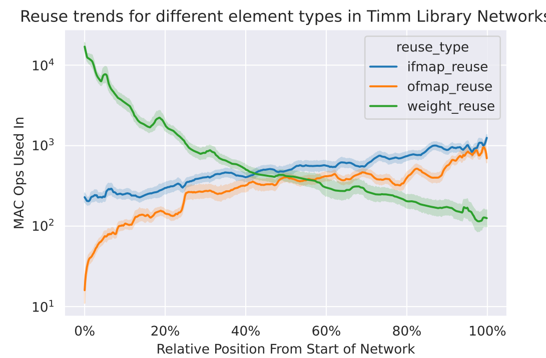
(b)

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

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 [9]. 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 tranformer self attention layers with convolution layers like CoAtNet [12]. 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 [8] or the PyTorch Image Models (timm) package in [10].



(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

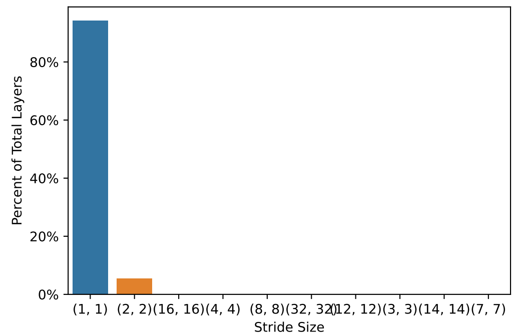
3.1.2.1 Loop Unroll Targets

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 elmements. 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 accross 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 analyse 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 throught this overlap which are quite prevelant in many moder 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 indirect 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 indirect mode is given in section 3.2. Note that convolution layers with non (1, 1) strides are executed under indirect 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 asscessing the overall performance and energy efficiency of an accelerator implementing the afformentioned 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

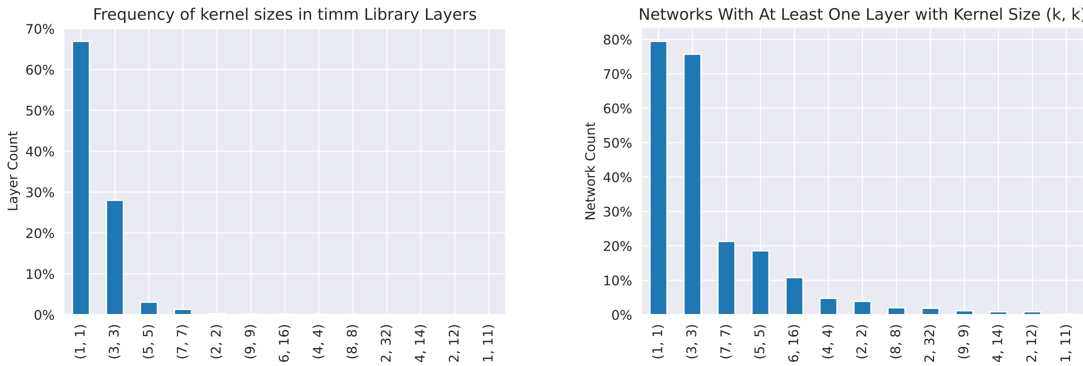


(a)

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

difficult to answer.

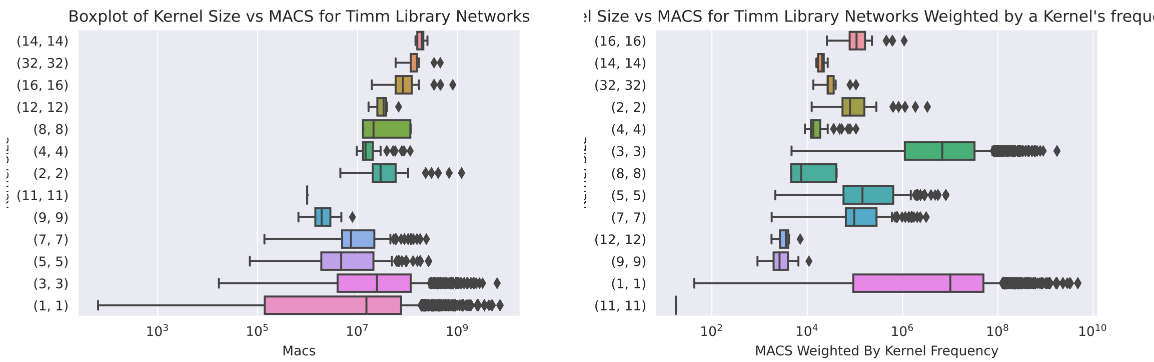
From Figure 3.6.a (1, 1) and (3, 3) kernel sizes dominate in comparison to all other kernel sizes. This renders the choice of keeping KY and KX loops folded impractical because if support is extended to an arbitrary K x K kernel while KY and KX loops are folded the onboard storage for weights would then need to be at least K^2 where K is the upperbound of kernels supported directly to avoid excessive weight fetches from DRAM. Unfortunatly, 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 underutilization 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 similair 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 prevalance of 7x7 convolutions originates from the historical use of resnet [4] as a feature extractor for a significant portion of networks analyzed by CIGAR.



(a) (b)

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 negligble 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 verticle 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.

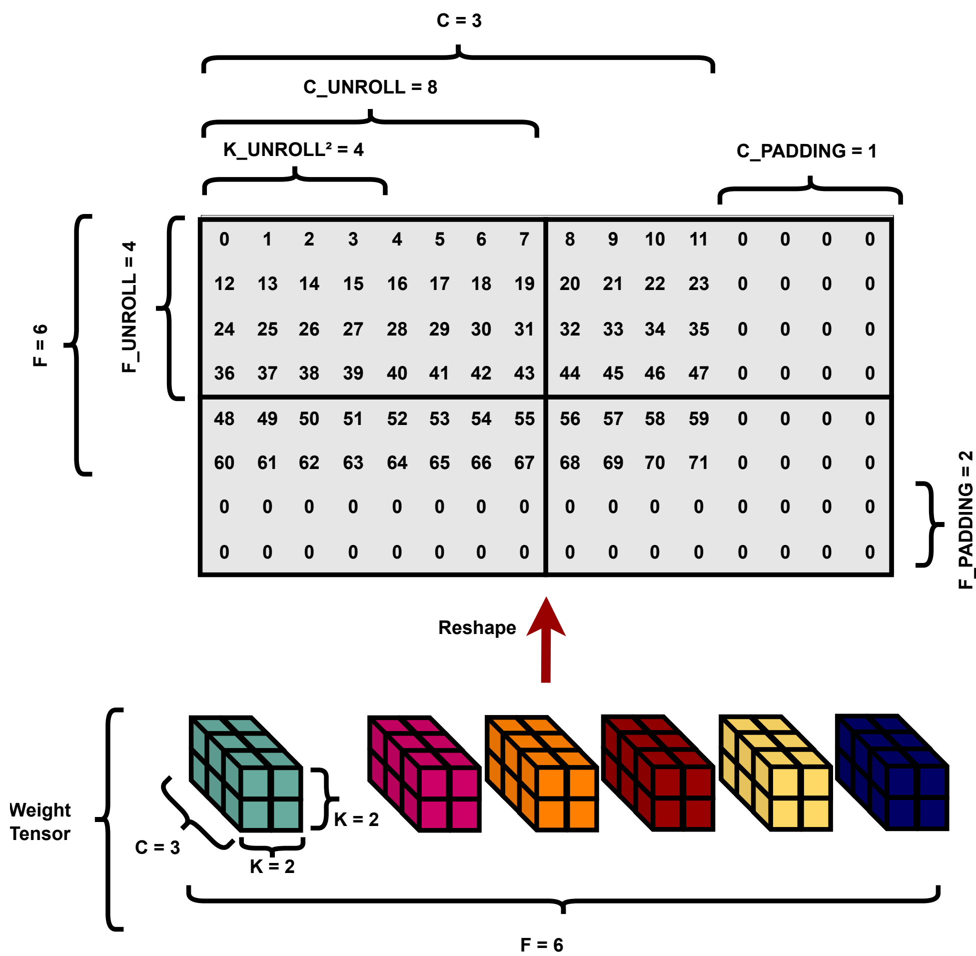


Figure 3.8: An illustration of weight tiling by loop unroll factors

3.2 Enabling Indirect mode through layer equivelence

Indirect 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 equivelence 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 [1]. This approach to supporting arbitary convolution operations would still require a matrix multiplication accelerator. To support matrix multiplication within a convolution accelerator we can either 1) reporpose existing compute and memory hardware on chip to perform both GEMM and convolutions operations or 2) we can establish a mathematical equivelence 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 reporposing 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 equivelent layers that are supported directly. In subsection 3.2.1, a proof for the equivelence between GEMM and (1, 1) Convolutions is given. Additionally, in subsection 3.2.2 the afformentioned proof will be used in tandem with the approach in [1] to provide support for arbitrary convolution operations. Finally, a discussion of layer dimentionality for unsupported layers that have been converted to equivelent supported layers is given in subsection 3.2.3.

3.2.1 Functional equivelence between GEMM and (1, 1) Convolutions

We can establish the functional equivelence between GEMM and (1, 1) convolutions with the following proof. An illustration of this proof is given in Figure 3.9. Given two matricies $A \in \mathbb{R}^{Z \times C}$ 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.

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

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 $\hat{R} \in \mathbb{R}^{F \times Z}$

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

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

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

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

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

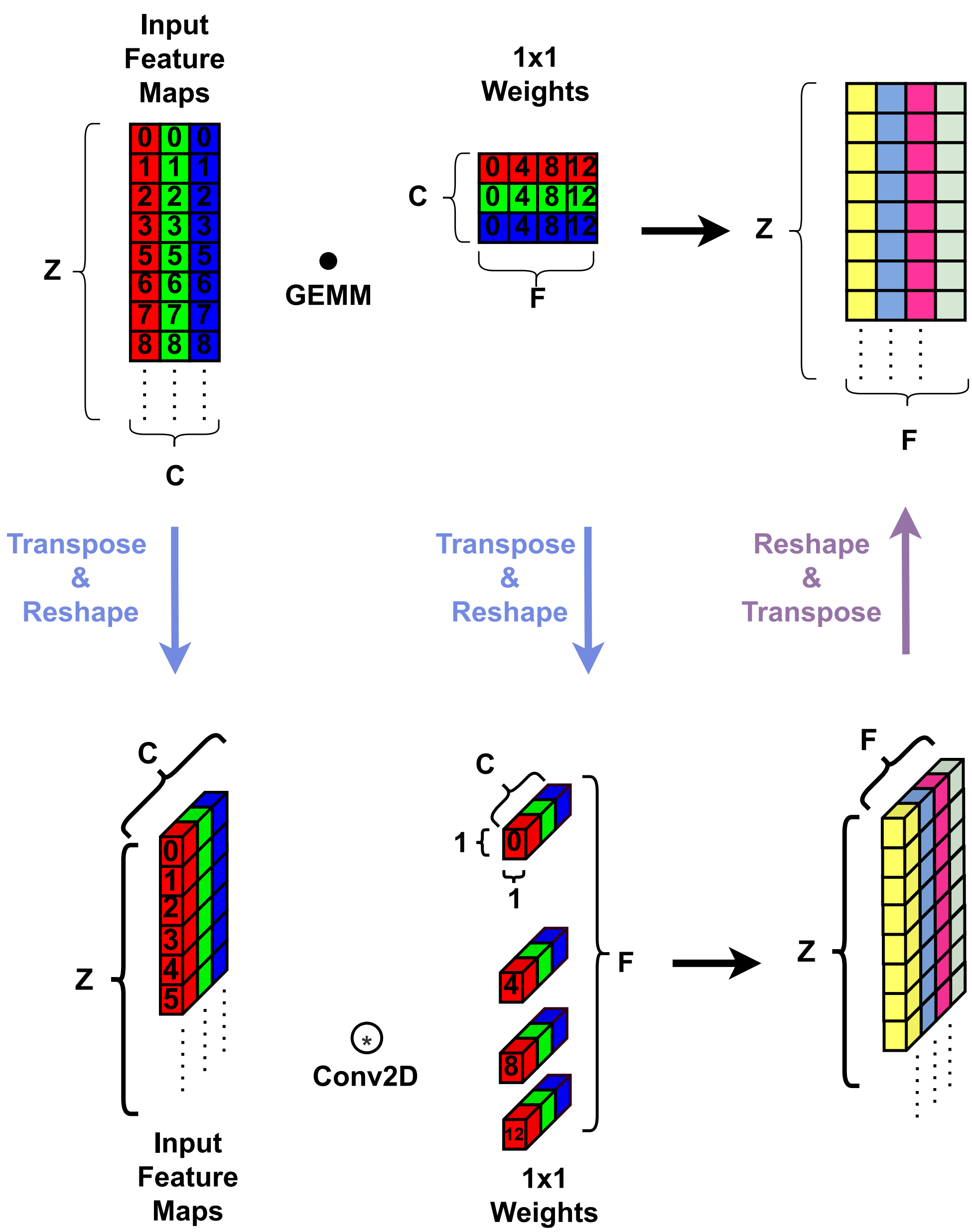


Figure 3.9: GEMM and (1, 1) Convolution Equivelence

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 [1]. 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 Equation 2.3 and Weights using Equation 2.4. This results in two matricies IFmap and Weights in Equation 3.6. Lowering should be performed if the kernel size of the Weight tensor is unsupported $K' \notin \{SupportedKernels\}$.

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

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 dimentions of size 1 to both IFmap and Weights. The transformations yields two new tensors $I\hat{Fmap}$ and $We\hat{i}ght$.

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

After performing the transformations in Equation 3.7 the output $OFm\hat{a}p_{prelift}$ can be calculated after performing a (1, 1) convolution in Equation 3.8 using the $I\hat{Fmap}$ and $We\hat{i}ght$ tensors.

$$OFm\hat{a}p_{prelift} \in R^{K'F\times nm\times 1} = I\hat{Fmap} * We\hat{i}ght \tag{3.8}$$

Finally we can lift $OFm\hat{a}p_{prelift}$ 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 Equation 2.6 to get the final OFmap in Equation 3.9.

$$\begin{aligned} OFm\hat{a}p_{prelift} \in R^{K'F\times nm\times 1} &\xrightarrow{Reshape} OFmap_{prelift} \in R^{K'F\times nm} \\ OFmap_{prelift}^T \in R^{nm\times FK} &\xrightarrow{BalancedLifting} OFmap \in R^{F\times m\times m} \end{aligned} \tag{3.9}$$

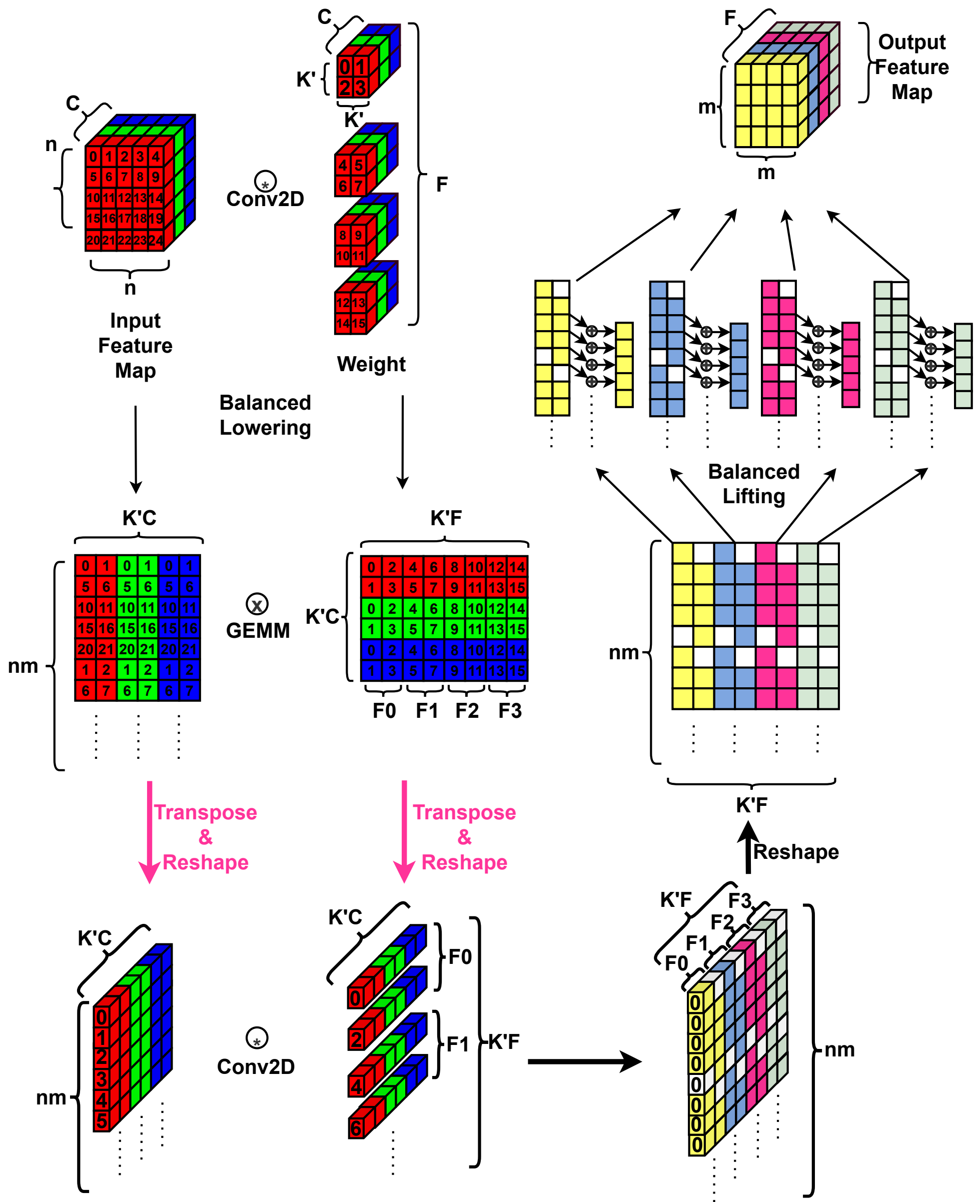


Figure 3.10: Illustration of approach Conv2Gemm2Conv approach

3.2.3 Layer Dimensionality

Wheather 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.

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

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

3.2.4 Accelerator Spatial Axis mapping

When determining axis mapping, F and C loops are assumed to be bound to an accelerator's verticle 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 veticle 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 verticle 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 $KY = KX = K = 1$.

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

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

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 matricies are transposed and reshaped. All reshape operations discussed in this chapter add a dimention 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 matricies 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 [6]. 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 [6]. To perform this deduction we will use the polyhedral model to analyse 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 analysed 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 2.2. These statements represent Processing Engine (PE)s performing MAC operations concurrently. MAC statement instances can be distinguished from eachother 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 coresponding 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

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

Each MAC statement is composed of three seperate 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 analysed concurrently using the same loop representation as they only differ based on whether we set the width loop upperbound 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 analyse temporal reuse in the dataflow represented in Listing 3.1 by adapting the approach in [7] to the afformentioned 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 ommited 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 iterationtions 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 [6], 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 (equivelent to number of Filters), and Weights. The beginings of a hardware template derived from the afformentioned temporal reuse behavior of the different memories referenced in the

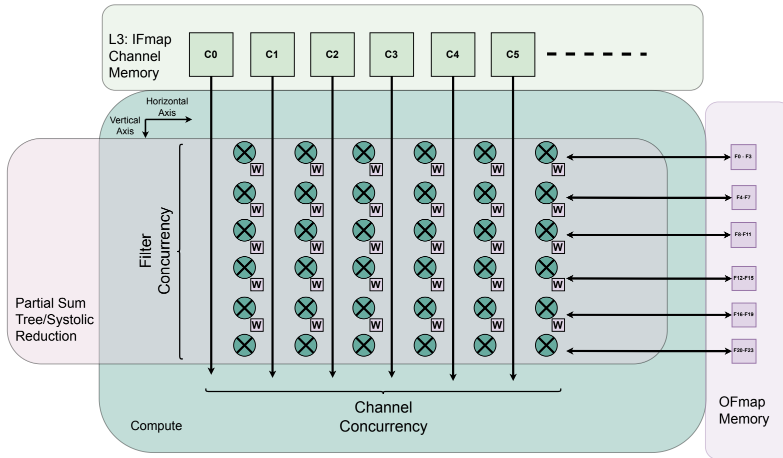


Figure 3.11: Initial hardware template incorporating buffers IFmap and OFmap temporal reuse

convolution dataflow can be seen in 3.11. In 3.11 the template is broken into 3 major componenets. 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

```
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};
IFMAP_XY:=({S[y, x, ky, kx] -> IF[y+ky][x+kx]})*ID;
IFMAP_REUSE_XY:=(IFMAP_XY.(IFMAP_XY^-1))*(ID_XY<<ID_XY);
IFMAP_XY_REUSE;
[Y, X, KY, KX] -> {
   S[y, x, ky, kx] -> S[y', x', ky' = (y - y') + ky, kx' = (x - x') + kx] :
      ... y' > y and (y + ky) -KY < y' <= (y + ky) and (x + kx) -KX < x'
         <= (x + kx) ...;
```

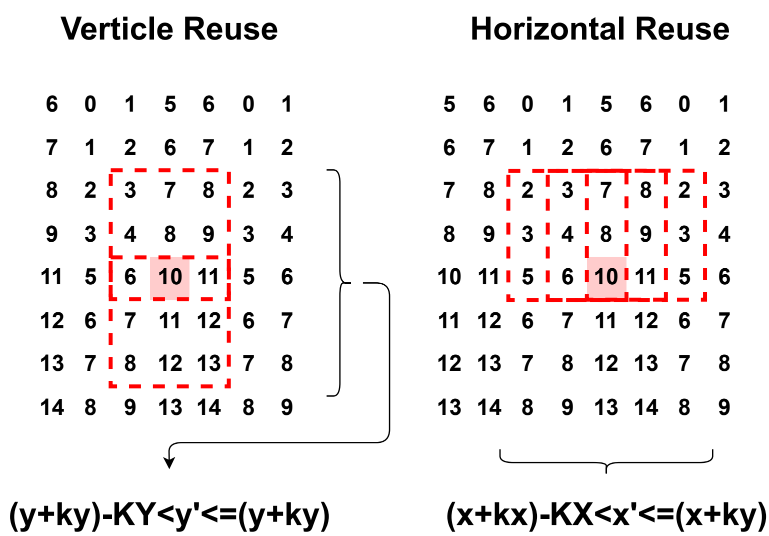


Figure 3.12: IFmap Reuse Behavior w.r.t individual feature map channels

```
}
```

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 verticle 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 accross 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 [7] 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.

[7] 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 [7] 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 [7] 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 [7] 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 afformentioned 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.

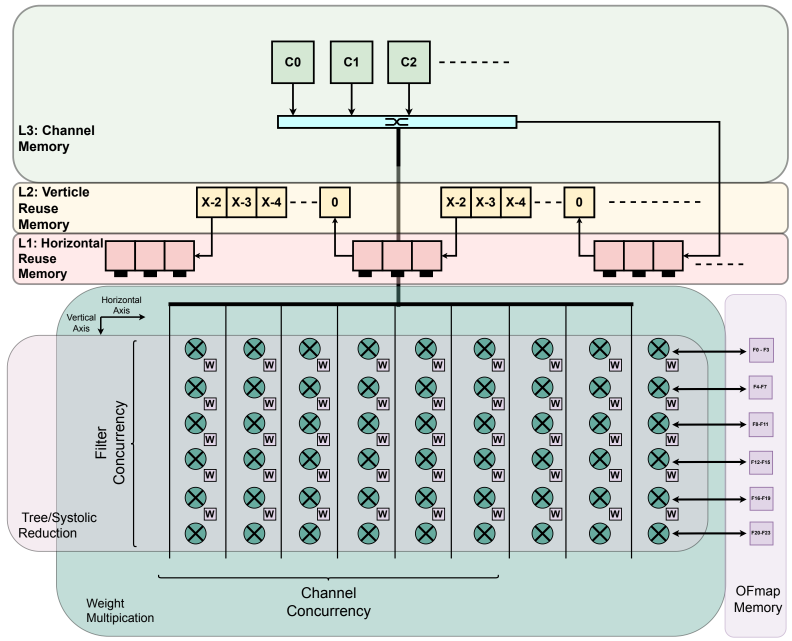


Figure 3.13: Hardware template incorporating a reuse chain for reuse within an IFmap channel

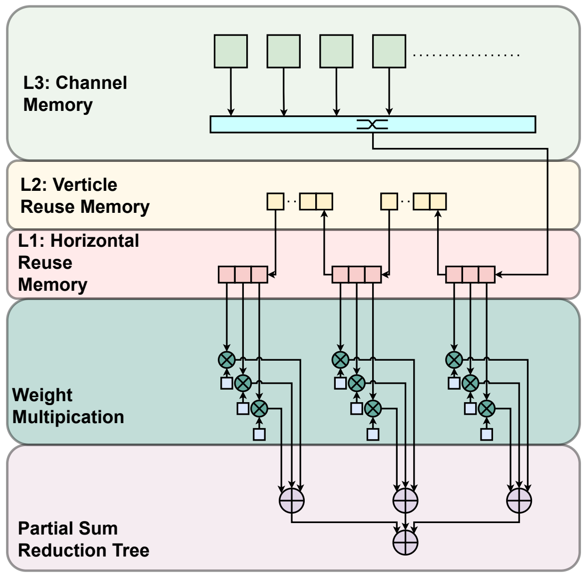
3.3.2 Spatial Reuse Analysis

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

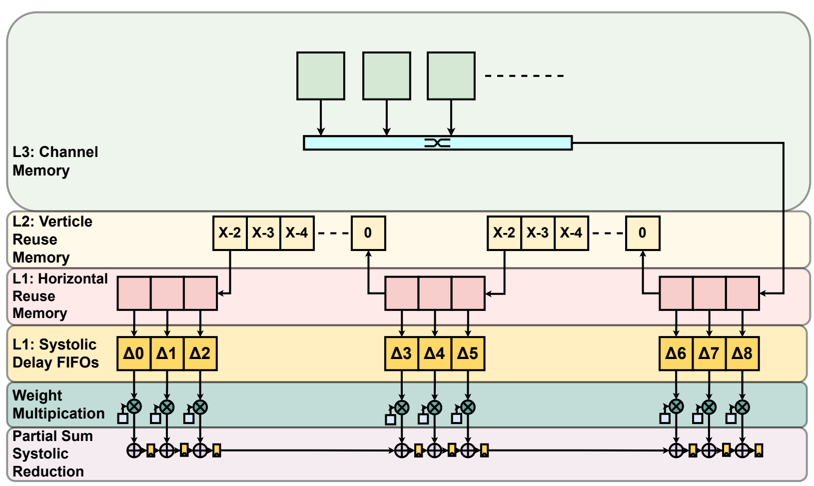
Listing 3.7: Spatial reuse in fully unrolled kernel loops

```
for(int f = 0; f < F; f+=F_T) // Filter loop
    for(int c = 0; c < C; c+=C_T) // Channel loop
        for (int y = 0; y < Y; y++) // FeatureMap Height
            for(int x = 0; x < X; x++) // FeatureMap Width
            {
                O[f+0][y][x]  += W[f+0][c+0][0][0] * \
                                    I[c+0][y+0][x+0]; // j=0
                O[f+0][y][x]  += W[f+0][c+0][0][1] * \
                                    I[c+0][y+0][x+1]; // j=1
                O[f+0][y][x]  += W[f+0][c+0][0][2] * \
                                    I[c+0][y+0][x+2];  // j=2
                O[f+0][y][x]  += W[f+0][c+0][1][0] * \
                                    I[c+0][y+1][x+2];  // j=3
                ...
                O[f+1][y][x]  += W[f+1][c+0][0][0] * \
                                    I[c+0][y+0][x+0]; // j=C_T*KY_T*KX_T
                O[f+1][y][x]  += W[f+1][c+1][0][1] * \
                                    I[c+0][y+0][x+1]; // j=C_T*KY_T*KX_T+1
                ...
                O[f+F_T-1][y][x] += W[f+F_T-1][c+C_T-1][KY_T-1][KX_T-1] * \
                                      I[c][y+KY_T-1][x+KX_T-1];
                                                // j=F_T*C_T*KY_T*KX_T-1

            }
```

Applying the taxonomy in Figure 7.2 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

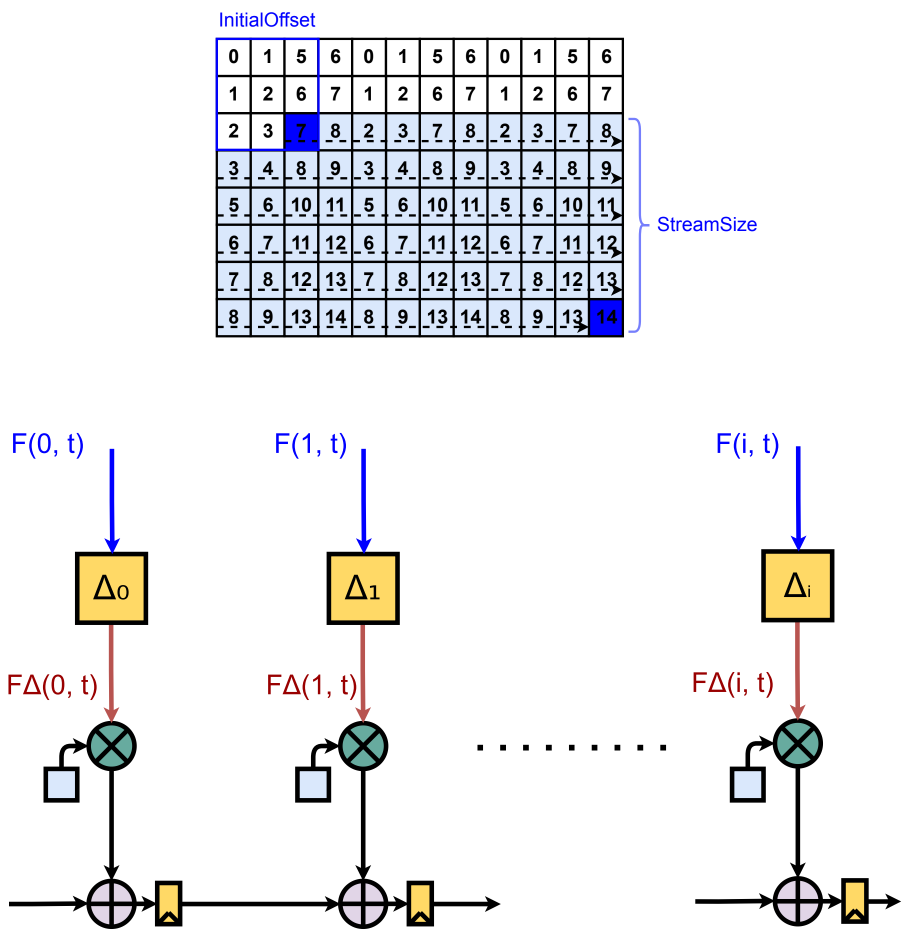


Figure 3.15: Reinterpretation of IFmap memory hierarchy outputs as a stream function

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 $A_\Delta(i,t)$.

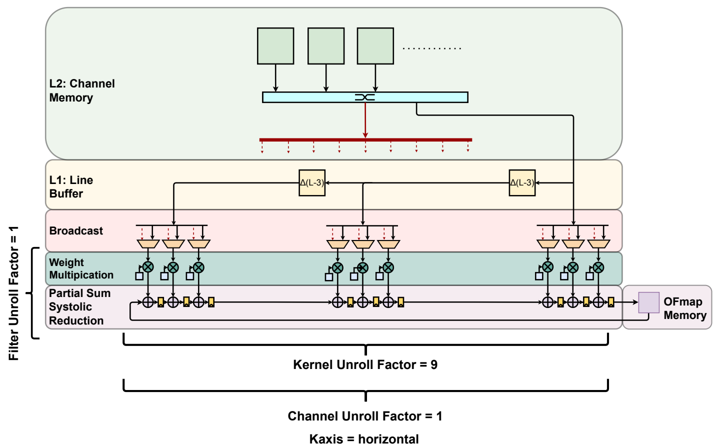


Figure 3.16: Using a systolic reduce and forward to calculate OFmaps

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

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

$$\tag{3.28}$$

Substituting the InitialOffset function in A_{Δ} 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 accross K groups of ports as well as accross 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-anyinterconnect 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 beenfit 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 accross 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.

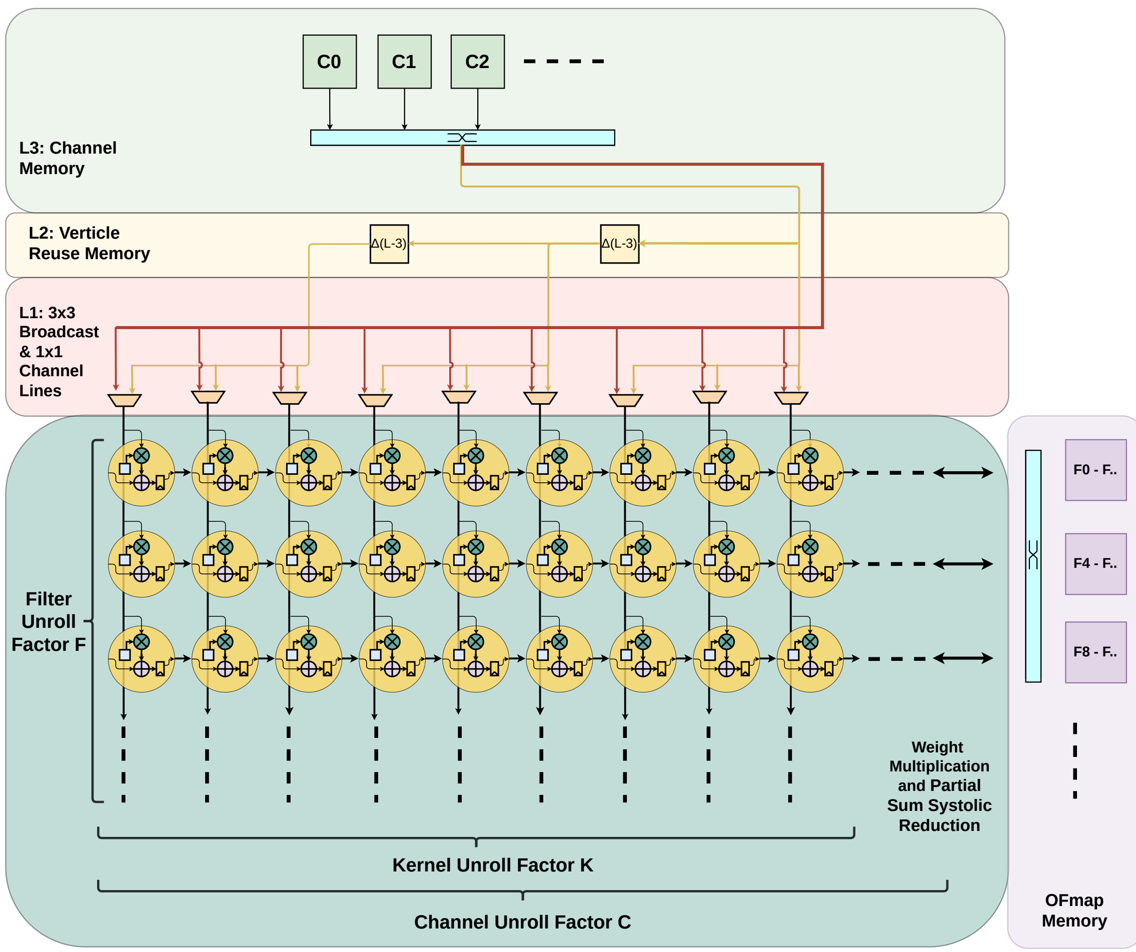


Figure 3.17: Hardware Implementation Taxonomy adapted from [6]

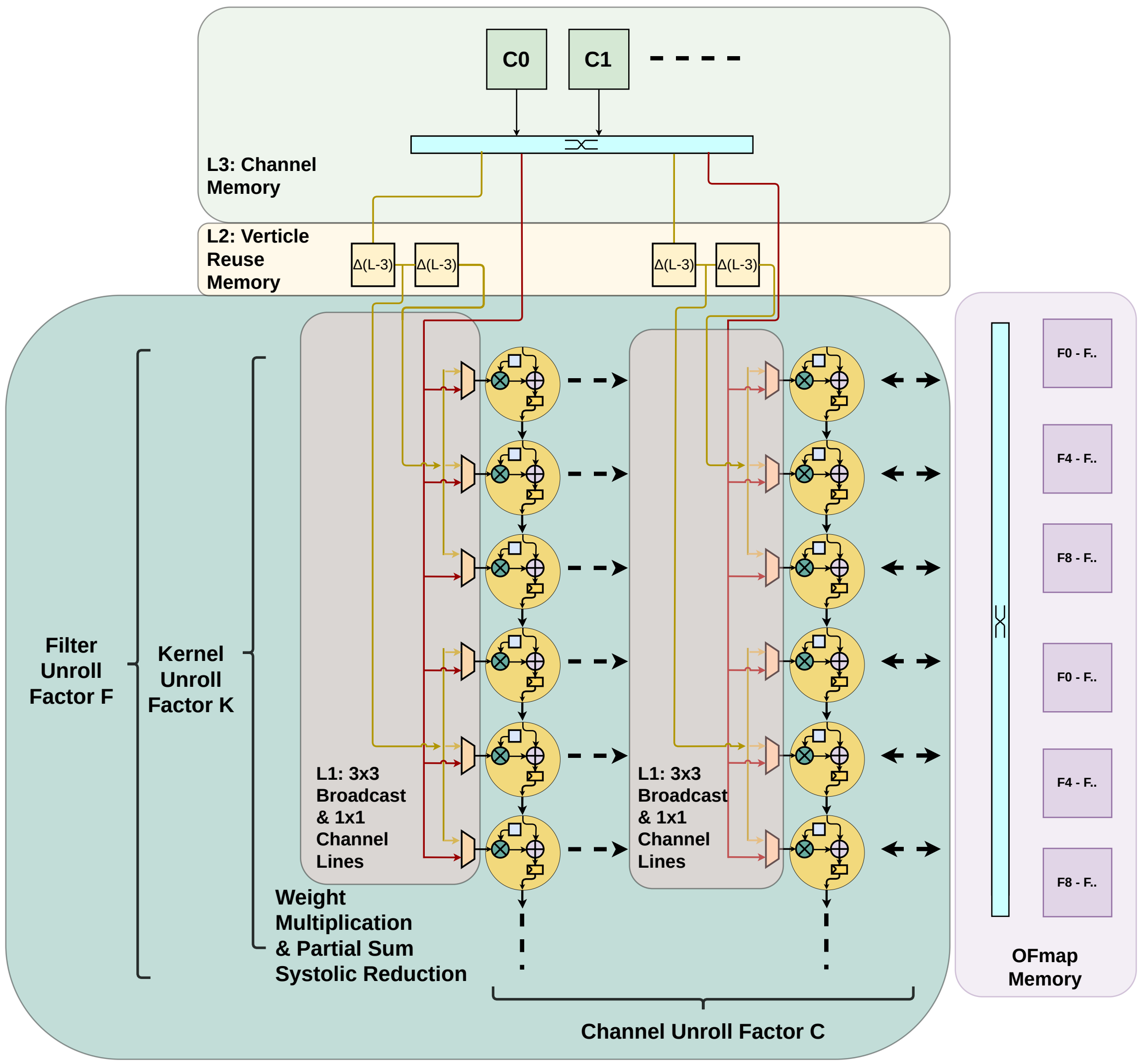


Figure 3.18: Hardware Implementation Taxonomy adapted from [6]

Chapter 4

Architecture Dimensioning

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 is the unroll factors for F and C loops 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 paremeters 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 upperbounds for the afformentioned unrolled loops. However, as discussed earlier, some combinations of loop upperbounds 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 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.

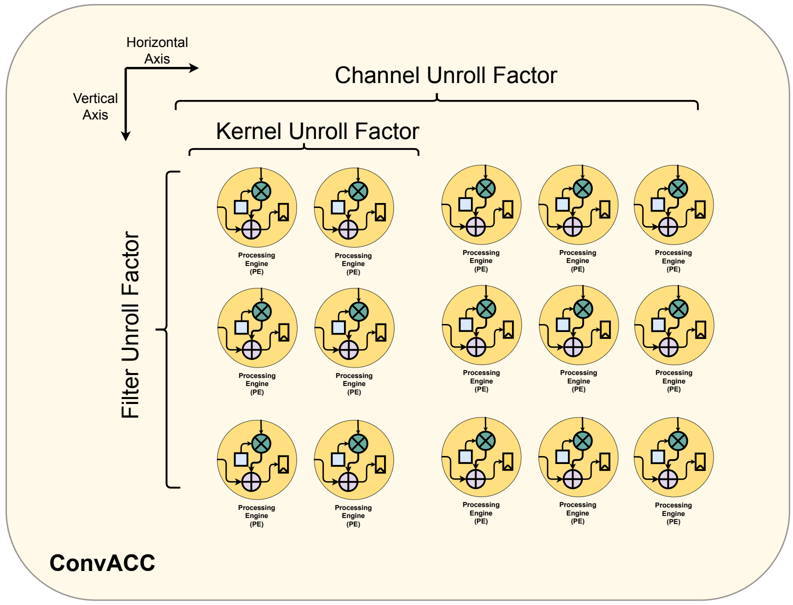


Figure 4.1: GEMM and (1, 1) Convolution Equivelence

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 $model^{stats}_{dict}$ 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 $model^{stats}_{dict}$. An architcture 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) equivelent layers based on the equivelence method that discussed in section 3.2.

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

Algorithm 2 TEMPO

Input: $model^{stats}_{dict}$, obj_fn, $kernels_{supported}$, pe_{budget}

Output: $template^{opt}_{config}$

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

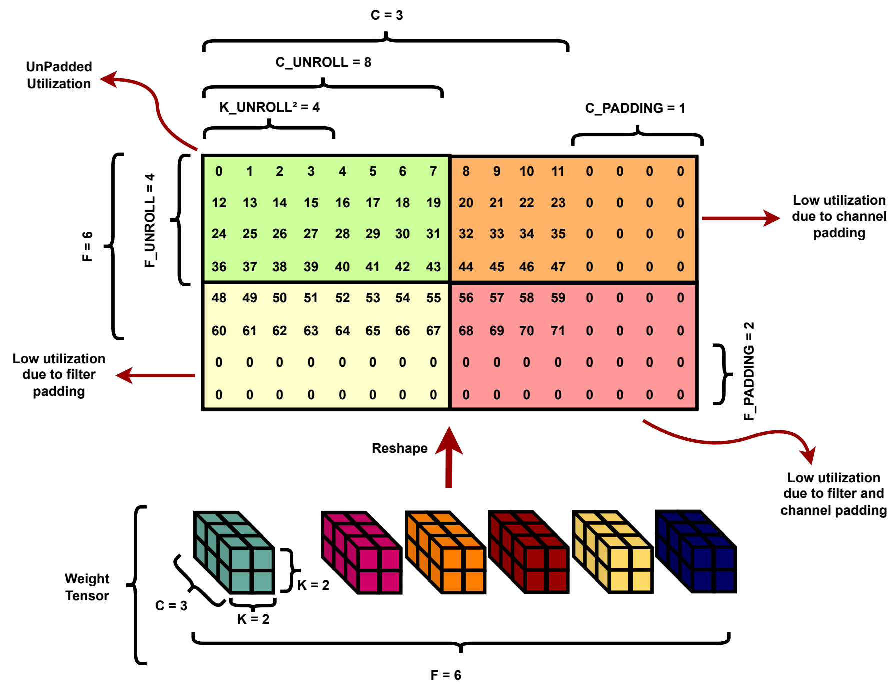


Figure 4.2: GEMM and (1, 1) asd Equivelence

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 evalute all 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 $\lceil \frac{\hat{F}}{F_{eff}} \rceil = 2$ verticle 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 occurence in the layer as reflect in Equation 4.3. For brevity each of the utilization equations are multiplied by their frequency of occurence in the same equation.

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

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 occurence depends on the number of unpadded tiles in the layer $\lfloor \frac{\hat{F}}{F_{eff}} \rfloor \lfloor \frac{\hat{C}}{C_{eff}} \rfloor$.

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

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 occurence is given in Equation 4.5.

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

If the allocation of PEs for channel loops exceeds avaliable 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 $utilization^{Padded}_{ChannelTiles}$ is then a function of how many PEs are active in the tile $\frac{(\hat{C} \bmod C_{eff})F_{eff}K^2_{unroll}}{Count_{pe}}$ multipled by the frequency of occurence. $\lfloor \frac{\hat{F}}{F_{eff}} \rfloor$. If $\hat{C} \bmod C_{eff} = 0$ then there are no padded channel tiles so $utilization^{Padded}_{ChannelTiles} = 0$.

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

If the allocation of PEs for filter loops exceeds avaliable 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 paremeters 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 incured 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 necssary 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 incoporated 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 paremeters 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

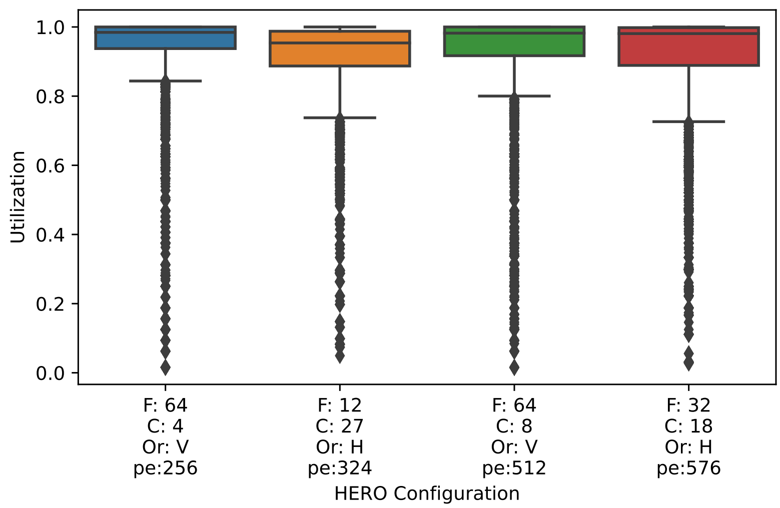


Figure 4.3: Utilization results for different optimal configurations found using TEMPO

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.

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. Lower-

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

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

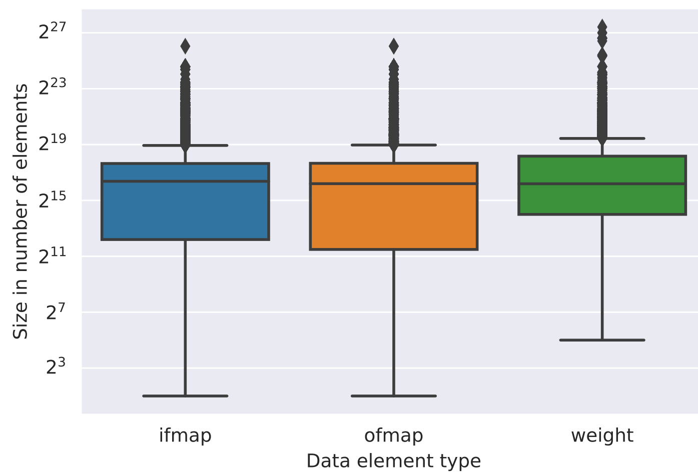


Figure 4.4: Boxplot of storage required for different data elements assuming no lowering

ing and lifting operations required under indirect 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. Table 4.2 shows the total storage in number of elements assumed by this work.

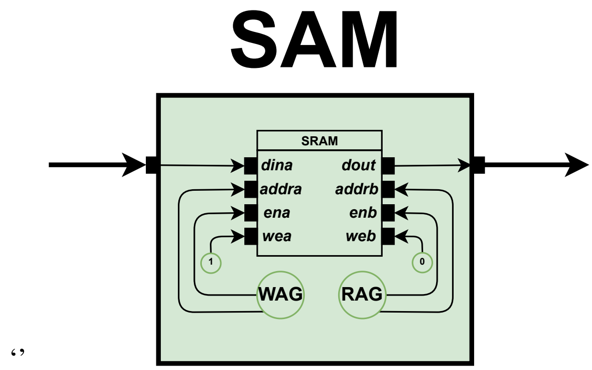
Data Element Type	On-Chip Storage
Weights	N/A
IFmap L3	2^{20}
IFmap L2	2^{9}
OFmap	2^{20}

Table 4.2: Assumed on-chip storage for different data elements in a convolution operation

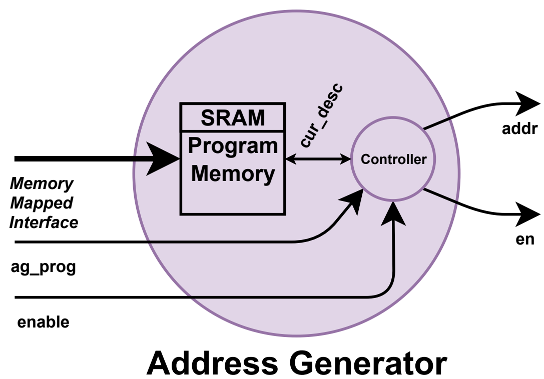
Chapter 5

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 SAMs, 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.



(a)



(b)

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.

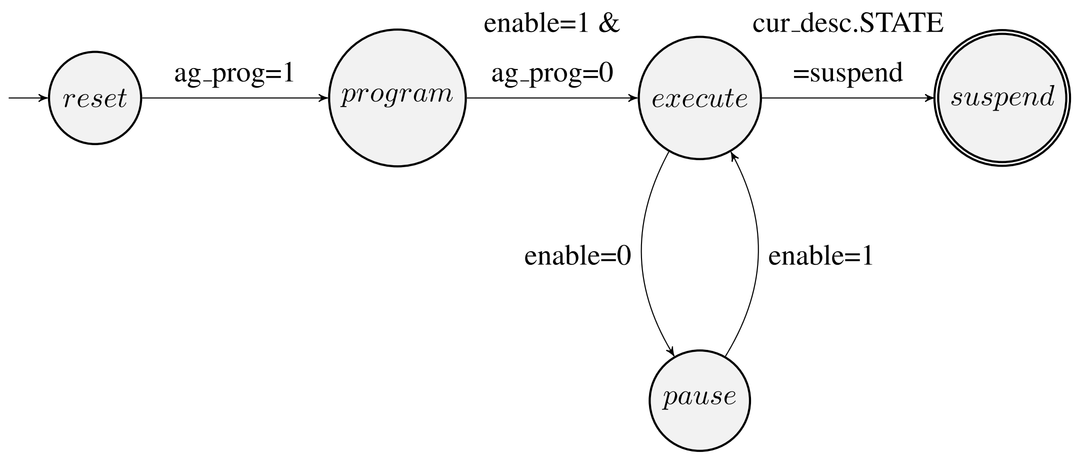


Figure 5.2: Address generator Finite State Machine

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 [5] 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 [5] 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

```
struct Descriptor
{
    DescriptorType type;
    unsigned int start;
    unsigned int x_count;
    int x_modify;
    unsigned int y_count;
    int y_modify;
};
```

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

```
    en = 0;
    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.

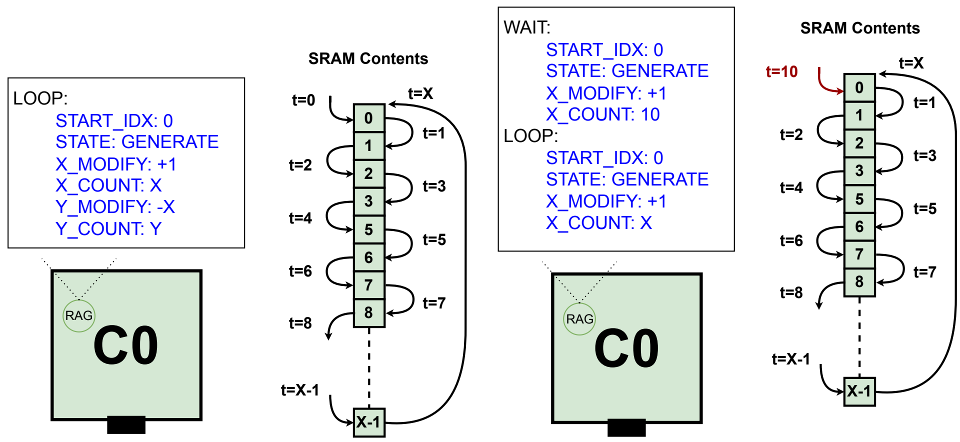
Listing 5.3: Descriptor as a set of loops

```
en = 1;
addr = start;
for(int y = 0; y < y_count; y++)
    for(int x = 0; x < x_count; x++)
        addr += x_modify;
    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.

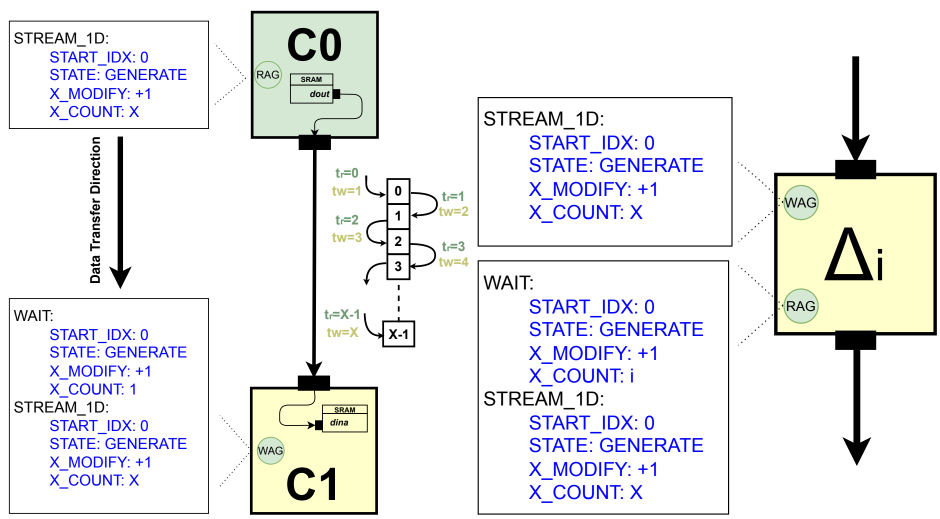
More complicated memory operations can be performed via the implicit coordination of multiple address generators across SAMs or within the same SAM. An illustration of that coordination is presented in Figure 5.4. In Figure 5.4.a a data transfer between two SAMs is achieved using one read address generator 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 Δ_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 Δ_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

6.1 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 sublayers 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 accross channel and filter axis for input and output feature maps. Decomposition can occur by decomposing large fmaps along the width and height dimentions however that complicates the process of aggregating sub layers. This is due to the potential overlaps of kernel windows accross decomposition boundaries. This issue arises specifically with (3, 3) kernels under direct mode in HERO. In layers with (1, 1) kernel sizes no overlaps accross 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.

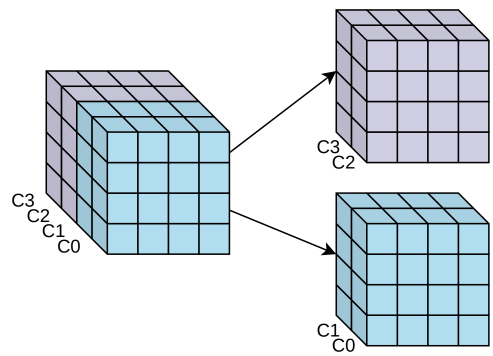


Figure 6.1: Illustration of layer decomposition's effect on Ifmap tensors

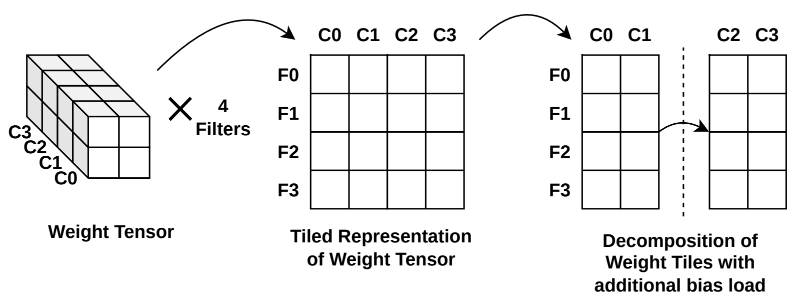


Figure 6.2: Illustration of layer decomposition's effect on weight tiling

Decomposition due to large ofmaps follows the same logic as in the ifmap case. An illustration of ofmap based decompsotion 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 when processing channels and filters. In some cases it may cause a significant reduction of PE utilization. In cases of low utilization due to layer decomposition other forms of concurrency may need to be supported beyond the filter/ channel/ kernel concurrency chosen by HERO. An exploration of other forms of concurrency is left as part of future work.

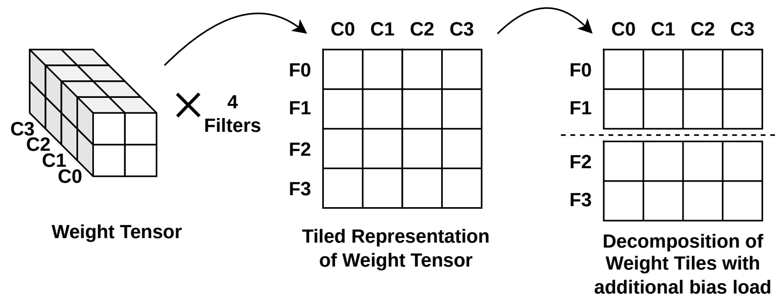


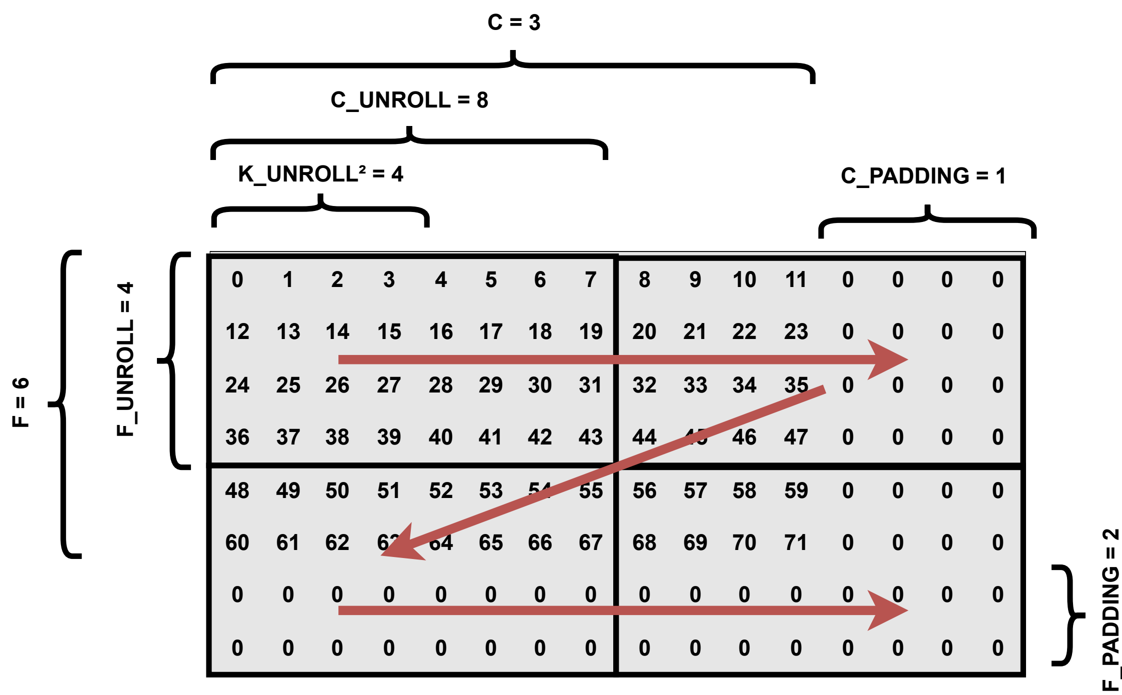
Figure 6.3: Illustration of layer decomposition's effect on weight tiling

6.2 Layer Scheduling

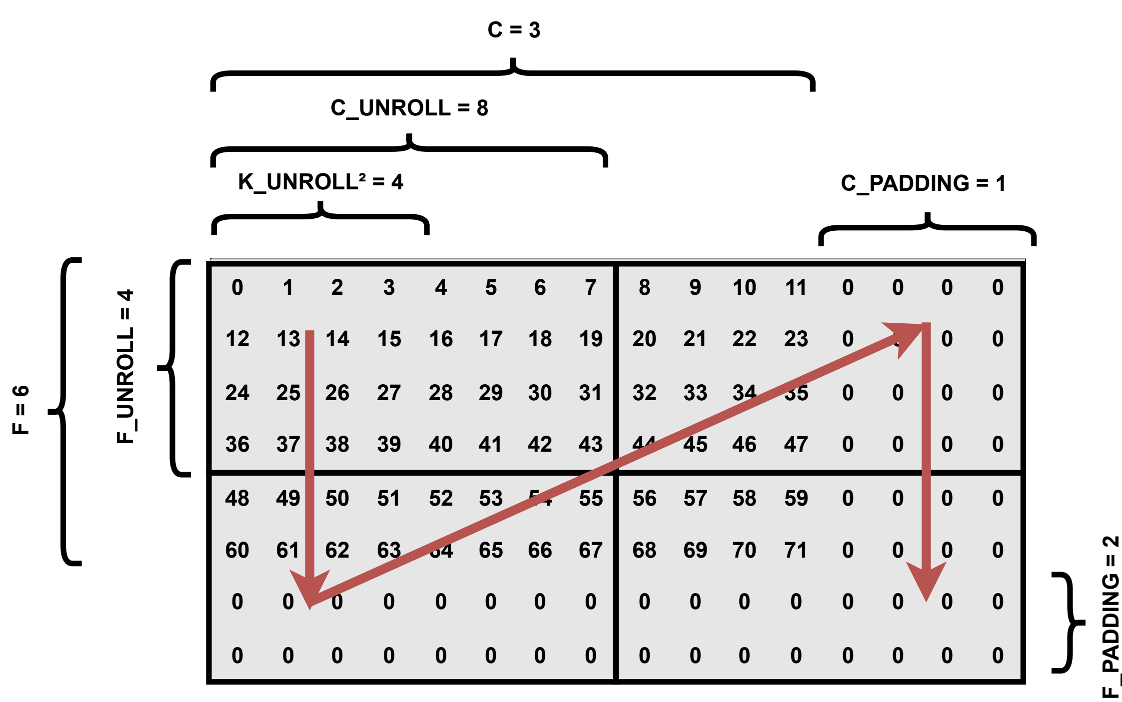
On-chip storage requriements for ifmaps and ofmaps are influenced by the ordering of tiles in the tiling representation of weights discussed in chapter 3. The order of processing weight tiles is equivelent to the ordering of the F, and C loops in the loop based representation of the convolution operation 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. Conversly, 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 6.4.

An illustration of this architectural tradeoff is present in Figure 6.5.c. Note that depending on the tile scheduling chosen, different HERO template parameters affect the scaling of different on-chip memories. For example, under ASAP, input featuremap memory requriements remain unchanged with any template parameters, however, ofmap memory scales with the architecture's filter unroll factor. Under ALAP ofmap memory remains unaffected by any of HERO's template paramaters, however, ifmap memory scales with the architecture's channel unroll factor. An illustration of how both memories scale with tempalate paramaters is given in Figure 6.5.a & Figure 6.5.b.

In this work ASAP scheduling is always assumed given the relatively poor scaling of ofmap memory with architecture template paramaters. This poor scaling is due to the necessity of storing ofmap elements using higher precision values to prevent numeric overflow. Note that in allowing different tile schedules may sometimes negate the need for layer decomposition. The interaction between layer decomposition and layer scheduling is left as part of future work.

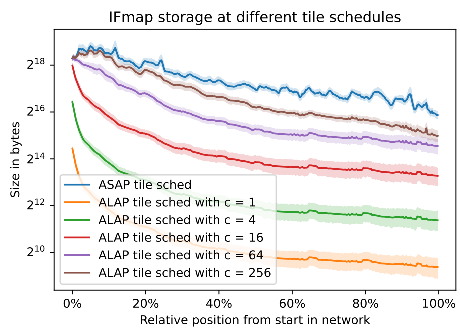


(a)

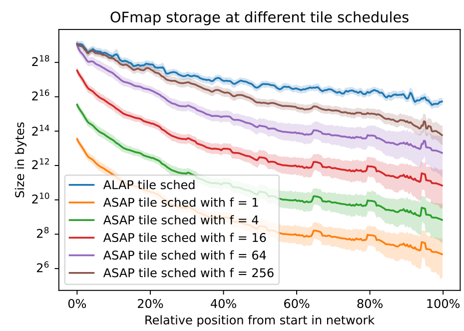


(b)

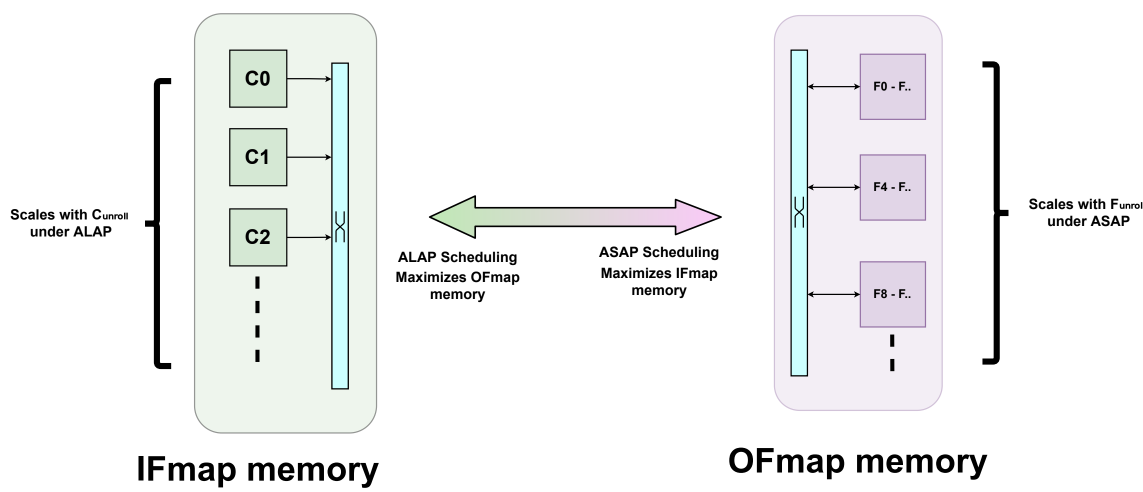
Figure 6.4: Illustration of different tiling schedules (a) ASAP scheduling (F, C) (b) ALAP scheduling (C, F)



(a)



(b)



(c)

Figure 6.5: Illustration of storage tradeoff between OFmaps and IFmaps depending on tile scheduling (loop ordering) and accelerator template parameters (loop unroll factors) (a) IFmap storage (b) OFmap storage (c) archictural illustration

6.3 Descriptor Program Generation

We can generate descriptor programs for the SAMs present in in the final architecture template 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 template. 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 Indirect mode) we will discuss the descriptor programs required for both of these modes to take place independently of the other. Note that indirect mode is just a data transformation operation followed by a (1, 1) convolution operation as discuseed in section 3.2. Indirect mode data transformations (lowering/ lifting) are assumed to be performed off chip. Moving these transformations on-chip is left as part of future work. 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 6.6

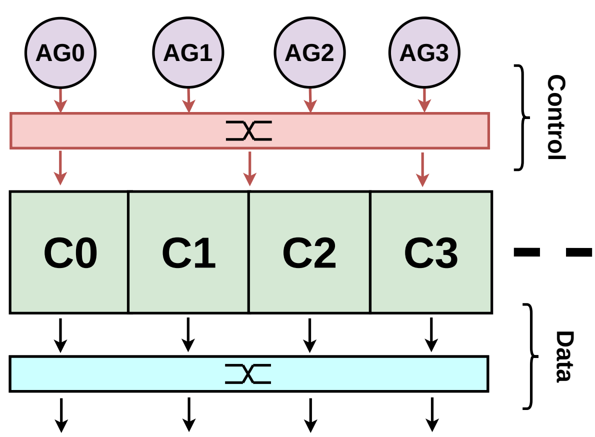


Figure 6.6: Illustration of interconnects for control and data in on-chip featuremap memories

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 superceeded 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.

6.3.1 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 6.7 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 6.7.

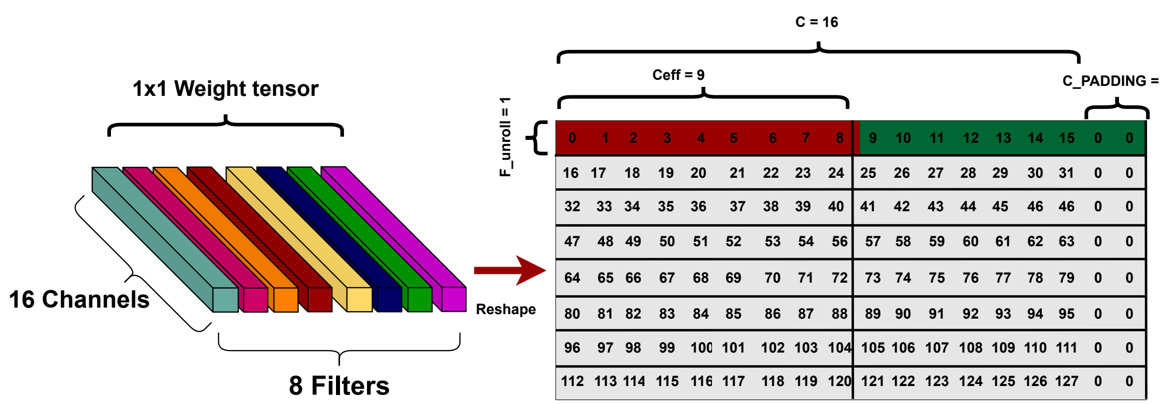


Figure 6.7: Illustration of weight tiling under (1, 1) convolution

For each weight in the tiles of Figure 6.7 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 section 6.2). 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 6.8. Note that in cases where channel featuremaps exceed the size of one bank, layer decomposition spreads the feature map accross 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 6.8. Additionally, they are assumed to just forward any partial sums/ output feature maps recieved from their input to their output with no modifications.

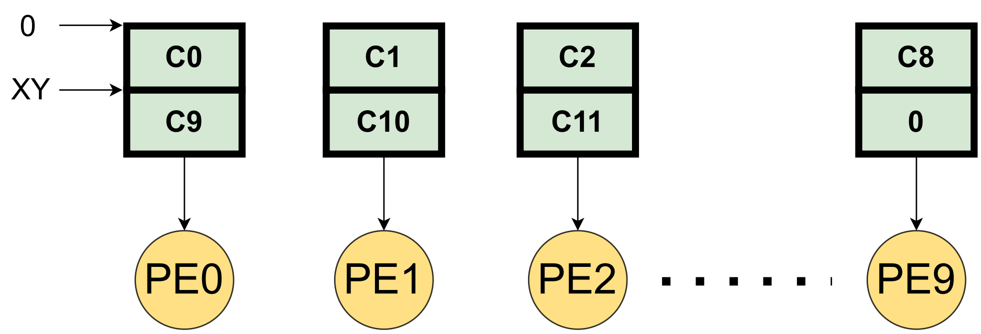


Figure 6.8: Illustration of channel banking in Ifmap L3 on-chip SRAMs

Since (1, 1) convolutions involve entire channel feature maps streamed into PEs with no reuse within a feature map occuring 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 afformentioned memories is given in Figure 6.9. In Figure 6.9 hirarchy layers unused by the computatation as well as some PEs have been ommited to for brevity.

In Figure 6.9 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 begining of their descriptor programs. The delay defined by the wait descriptor for each SAM depends on the index of the PE that

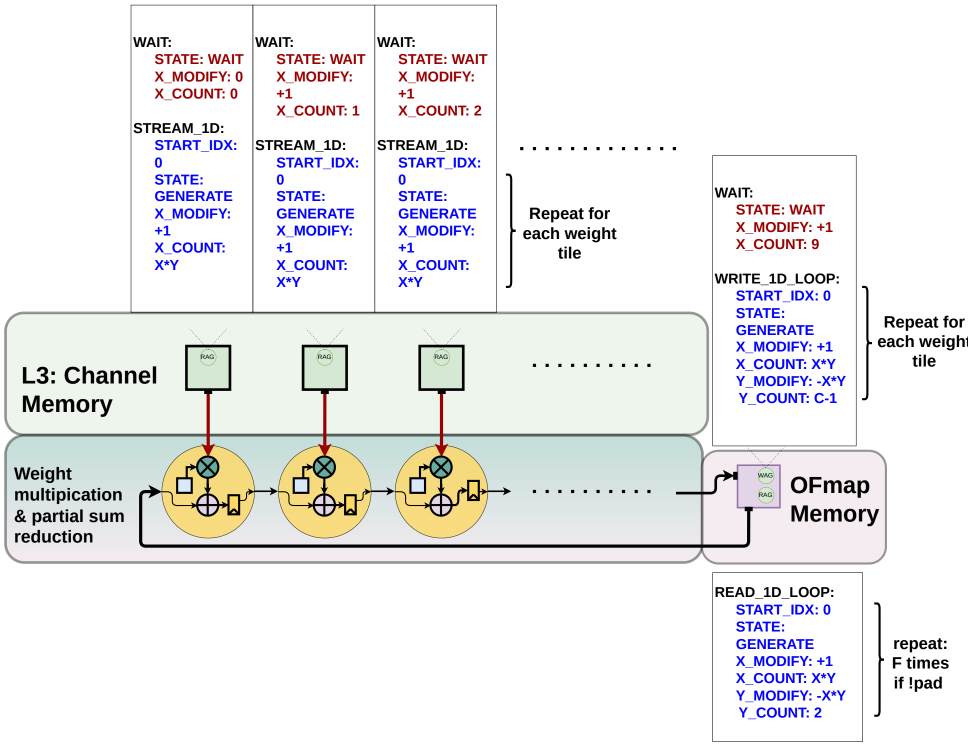


Figure 6.9: Illustration of (1, 1) convolution scheduling

the SAM is connected to. So the first PE's corresponding SAM has no read delay so the x_count 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 begining 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. Optimizating 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 propogate through the PEs to reach the OFmap. The descriptor required for the write address generator are similair to the the read ones with the exception of an additional wait descriptor that gives the first partial sum/ Ofmap time to propogate 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 similair to the read generate descriptor.

6.3.2 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 similair to the (1, 1) convoltuion 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 propogate down to the set of PEs each L3 featuremap bank is connected to. Once this setup phase concludes the rest of the featuremap is 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. An illustration of this behavior is given in Figure 5.4.b.

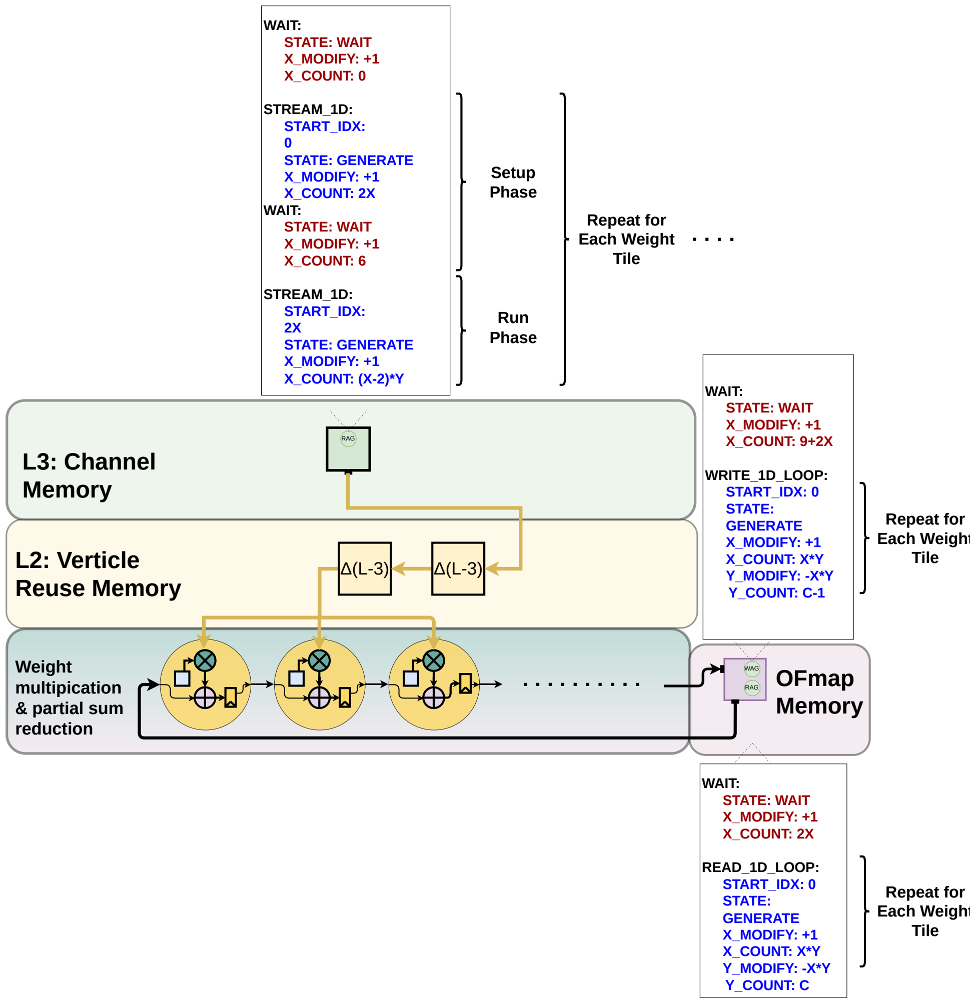


Figure 6.10: Illustration of (1, 1) convolution scheduling

Chapter 7

HERO Architecture Simulation

7.1 Simulation Enviornment

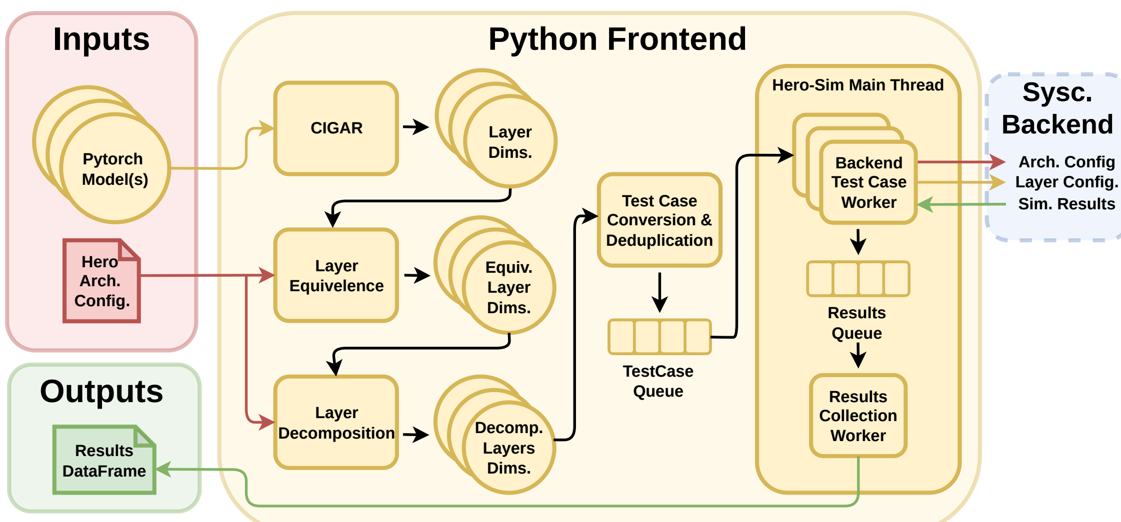


Figure 7.1: Hardware Implementation Taxonomy adapted from [6]

7.2 Experimental Results

Prominent model's convolution layers were are assessed using the simulation platform namely resnet-50 mobilenetv3 and vgg16. Performance for each of the three conrete architectures suggested by tempo is reported below.

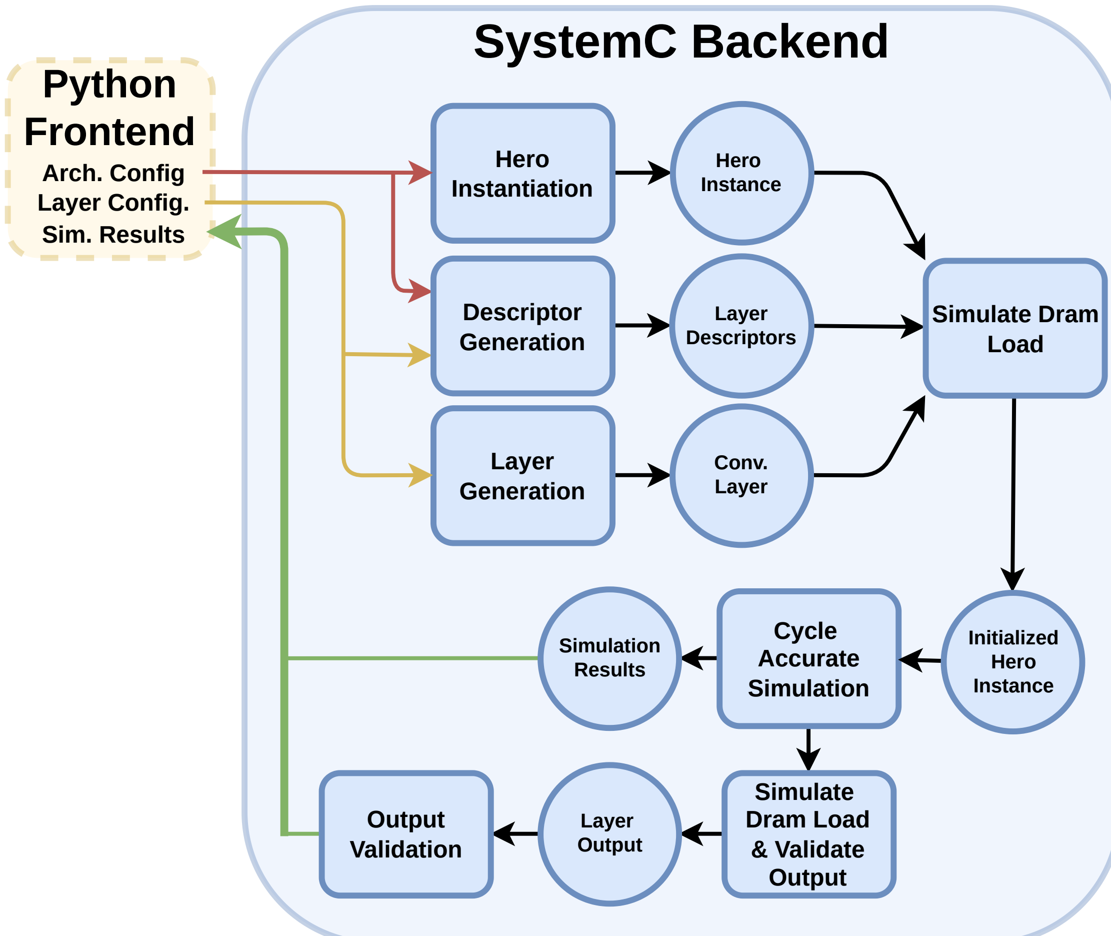


Figure 7.2: Hardware Implementation Taxonomy adapted from [6]

7.2.1 Latency and Speedup over CPU Baseline

7.2.2 Energy

7.2.3 Utilization

7.2.4 Per network results

7.2.5 DRAM Bandwidth

7.2.6 Descriptor program scaling

7.2.7 Area

Chapter 8

Conclusion

Writing a long manuscript is easy ... only if one starts early enough.

Bibliography

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

[2] 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.

[3] 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.

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

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

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

[7] 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.

[8] 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.

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

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

[11] 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.

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

[13] 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.

[14] 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.

[15] 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.