Technische Universität München
TUM School of Computation, Information and Technology

HW-Acceleration for Edge-AI

Sebastian Siegfried Prebeck

Vollständiger Abdruck der von der TUM School of Computation, Information and Technology der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Ingenieurwissenschaften (Dr.-Ing.)

genehmigten Dissertation.

Vorsitz:		Prof. Dr. Andreas Herkersdorf
Prüfer*innen der Dissertation:	1.	Hon.-Prof. Dr.-Ing. Wolfgang Ecker
	2.	Prof. Dr.-Ing. Georg Sigl

Die Dissertation wurde am 03.05.2023 bei der Technischen Universität München eingereicht und durch die TUM School of Computation, Information and Technology am 20.11.2023 angenommen.

Abstract

The utilization of embedded devices in IoT (Internet of Things) applications constitutes a cost and energy efficient solution to make devices smart. With increasing demand for applications, which rely on AI (Artificial Intelligence)-inference on the edge, the limited platform resources of embedded general purpose processors delimit the success. Available more powerful platforms exceed the requirements for such applications by magnitudes, similar to the permitted cost and energy budget. The aim of this thesis is the definition of HW (Hardware) architectures and tools to close this gap. Therefore, an analysis of operations involved in NN (Neural Network)-inference execution is done, where their diversity, complexity and resource utilization is broken down. The thesis further investigates on RISC-Vs NN-execution capability and specific bottlenecks. Moreover, NN-model features and properties are identified, which can be exploited for an optimized platform support. In this thesis an approach is proposed, which extends a 5-stage RISC-V processor with streaming modules, designed for collecting and provisioning NN input data and inference results. It addresses the data path bottleneck and ensures an efficient deployment of NN-model properties, e.g. sparsity and packing. The thesis shows that the diversity of operations involved in NN-model inference execution is very limited. Nevertheless a high demand for data movement and computation is distinct. A dedicated accelerator plugin offloads the processor from massive and inefficient data movements. Additionally, it releases processor time valuable for other tasks or such that cannot efficiently be accelerated with the developed method, e.g. pooling or depthwise convolution. A vector dot-product unit integrated via custom ISA (Instruction Set Architecture) extension causes a significant compute performance increase when handling reduced precision model data, but significantly contributes to on-chip area. Memory is found to be the most crucial component in terms of area and energy demand. The findings are summarized as follows. Proper platform resource balancing covering data path and compute is of utmost importance for high utilization and satisfactory efficiency. To fulfill the strict memory limitations of edge platforms, model compression is a feasible method. A processor coupled solution allows to account for the irregular compute patterns, without adding complexity to the accelerator module covering major amount of high redundancy operations. This ensures high utilization of the accelerator components at low complexity. Additionally, this approach ensures sustainability, since the coupled processor allows to handle any kind of operations, even such which aren't developed at time being, although with performance losses. For the development of efficient applications the three domains HW (Hardware), SW (Software) and AI have been considered in parallel. Missing any of them, due to a lack of knowledge infers sever disadvantages in the efficiency of the application. Each individual of the mentioned domains has to be aware of the others. With the proposed solution the addressed performance gap is closed.

Zusammenfassung

Der Einsatz eingebetteter Steuerungsprozessoren in IoT (Internet of Things) Anwendungen erfordert kosten- und energieeffiziente Lösungen, um die eingesetzten Geräte smart zu machen. Mit zunehmender Nachfrage an Anwendungen, die auf KI (Künstliche Intelligenz) im Egde setzten, wird die eingeschränkte Leistungsfähigkeit von eingebetteten Universalprozessoren zum Problem für deren Erfolg. Verfügbare leistungsfähigere Plattformen übersteigen die Anforderungen für solche Anwendung um Größenordnungen, ebenso aber auch die Budgets für Kosten und Energie. Das Ziel dieser Arbeit ist es HW (Hardware)-Architekturen und Werkzeuge zu entwickeln, um diese Lücke schließen zu können. Dazu werden alle Operationen analysiert, die in direktem Zusammenhang mit der Berechnung von NN (Neuronalen Netzen)-Inferenzen stehen, wobei deren Vielfalt, Komplexität und Ressourcennutzung aufgeschlüsselt werden. Darüber hinaus untersucht diese Arbeit die Ausführungsfähigkeiten und Engpässe, die ein RISC-V Prozessor bei der Berechnung einer NN-Inferenz mit sich bringt. Darüber hinaus werden Merkmale und Eigenschaften von NN-Modellen identifiziert, die für eine optimierte Plattformunterstützung ausgenutzt werden können. In dieser Arbeit wird ein Ansatz vorgeschlagen, der einen 5-stufigen RISC-V Prozessor mit Streaming-Modulen erweitert, die für die Bereitstellung von NN-Eingangsdaten und die Berechnung der Inferenz konzipiert sind. Dies adressiert Engpässe im Datenpfad und gewährleistet die Ausnutzung der Eigenschaften von NN-Modellen, z.B. Sparsity und Packing. Diese Arbeit zeigt, dass die Vielfalt der Operationen, die bei der Ausführung der Inferenz eines NN-Modells beteiligt sind, sehr begrenzt ist. Nichtsdestotrotz ist ein hoher Bedarf an Datentransfers und Berechnungen zu verzeichnen. Das Beschleuniger-Plugin entlastet den Prozessor von intensiven und ineffizienten Datentransfers. Zusätzlich wird wertvolle Prozessorzeit für andere Aufgaben frei, die mit der vorgestellten Methode nicht effizient beschleunigt werden können, z.B. Pooling oder Depthwise Convolution. Eine Skalar-Produkt Einheit, die über eine benutzerdefinierte ISA (Instruktionssatz)-Erweiterung integriert ist, führt zu einer signifikanten Steigerung der Rechenleistung bei der Verarbeitung von Modelldaten mit reduzierter Präzision, trägt aber auch erheblich zur On-Chip Fläche bei. Der Speicher erweist sich als die einflussreichste Komponente in Bezug auf Fläche und Energiebedarf. Die Ergebnisse lassen sich wie folgt zusammenfassen. Um eine hohe Auslastung und eine effiziente Ausführung zu gewährleisten, ist es wichtig eine Balance zwischen den Plattformressourcen zu finden, die einerseits dem Datenpfad und andererseits den Inferenzberechnungen zugeordnet sind. Modellkomprimierung stellt eine Methode dar, die der Erfüllung der strikten Speicherbeschränkungen auf Edge-Plattformen dient. Eine prozessorgekoppelte Lösung ermöglicht es, unregelmäßige Berechnungsmuster zu berücksichtigen, ohne die Komplexität des Beschleunigers zu erhöhen, der stattdessen einen großen Teil redundanter Operationen abdeckt. Dies ermöglicht eine hohe Auslastung der Beschleunigerkomponenten bei gleichzeitig geringer Komplexität. Zusätzlich sichert dieser Ansatz die Nachhaltigkeit, da es der gekoppelte Prozessor erlaubt, jede Art von Operationen zu verarbeiten. Auch jene sind damit abgedeckt, die zur Zeit noch nicht bekannt sind, wenn auch unter Leistungseinbußen. Für die Entwicklung effizienter Anwendungen wurden die drei Bereiche HW, SW und KI gleichzeitig betrachtet. Das Fehlen der Betrachtung eines dieser Bereiche aufgrund mangelnder Kenntnis hat schwerwiegende Nachteile auf die Ausführungseffizienz einer Anwendung. Jeder einzelne der genannten Bereiche muss in den jeweils anderen berücksichtigt sein. Mit der vorgeschlagenen Lösung wird die eingangs angesprochene Lücke geschlossen.

Contents

List of Figures

List of Tables

1. Introduction

The research domain of AI (Artificial Intelligence) is an ever growing field of methodologies for creating intelligent machines, which are enabled with the capabilities to achieve goals in a human like strategy. John McCarthy, a computer scientist coined this term in the 1950s (McCarthy 2007). The research field is active since decades, but recent groundbreaking achievements enabled by DNNs (Deep Neural Networks) made it popular.

1.1. Overview

Figure 1.1 depicts the taxonomy of DNNs in the context of the AI domain. A high level subcategory of AI is called ML (Machine Learning). Arthur Samuels work "Some Studies in Machine Learning Using the Game of Checkers" published in 1959 differentiates ML from other AI subcategories through an automated learning mechanism (Samuel 1959). Instead of handcrafted explicit coding of the program behavior, ML is coded in a way to ensure freedom for automatic definition of its behavior based on samples, i.e. learning.

The following classifications of AI subcategories follows (Sze et al. 2017). Within the domain of ML, another group of methodologies can be identified, which comprises all brain-inspired approaches. One can understand the brain as a "machine", capable of learning and taking decisions based on learned patterns. Therefore, it seems natural to make use of the available knowledge about brains and try to imitate its basic principles and functionalities for the purpose of ML. The brain is not entirely understood, but the available information is sufficient to extract a working model for the purpose of AI. The main building block constitutes a neuron, with dendrites and an axon. Every neuron receives signals from multiple other neurons via dendrites and creates another signal on its axon which depends on those dendrites signals. The axons

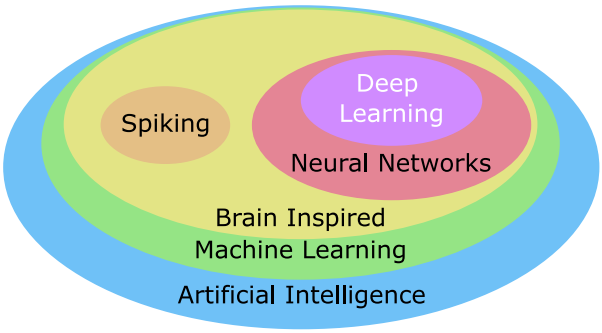


Figure 1.1.: Taxonomy of DNN in the field of AI, adopted from (Sze et al. 2017)

on their side are connected to dendrites via synapses, which scale the transferred signal. One can observe, the interconnection structure isn't reordered during learning, instead the synapses scaling adapts.

The category of brain-inspired AI comprises of two different approaches for neuron communication. The first one is "Spiking". Thereby, the communication between the neurons via axons and dendrites is not considered as a signal with constant value, but as spiking pulse. In extend to the spiking amplitudes, the transported information is encoded in the temporal dimension, i.e. length and time of appearance. The neuron computes the properties of the spike pulse at its axon on those three parameters by combining them.

For this thesis the second subcategory of brain-inspired AI is in focus. NNs (Neural Networks) don't consider the dendrites and axon communication as spike pulse based, but as signals with amplitude values instead. The neuron computes the weighted sum of the ingoing signal amplitudes. Hereby, the weights map to the synapses scaling value, which is introduced in the brain-inspired domain. Additionally, a non-linear function is applied on the weighted sum, to avoid linear dependency across layers, which would cause their collapsing.

The research of this thesis deals with DNNs. This group summarizes all NNs, which consist of more than three layers. Hereby, input and output layers are accounted within this number of layers, thus for an NN the number of hidden layers needs to exceed one to be considered as a DNN. Because of the increased number of layers, those NNs are considered as "deep". "Deep Learning" concurrently is the domain with the most active research and astonishing achievements. The increased number of layers allows to first extract low-level features, which are combined into higher-level features in subsequent layers. This allows a step-wise increase of the abstraction.

Yann LeCun is one of the pioneers of the DNN structure. LeNet-5 (Lecun et al. 1998) is the first DNN published, which was able to successfully recognize handwritten characters. From 2011 onward, when Microsoft successfully trained a DNN for speech recognition (Deng et al. 2013), research in the domain of DNN for new applications received more attention, due to increasing accuracy. In 2012 Alex Krizhevsky proposed AlexNet (Krizhevsky et al. 2017) designed for image recognition purposes, which achieves groundbreaking accuracy. Since then, many NNs with various shapes and parameters got developed for a wide variety of applications.

Why AI underwent such a rapid growth in the last decade, while experiencing a sluggish progress since it's first invention Sze explains by three limiting factors (Sze et al. 2017). NNs need to learn their behavior from training data. In order to create some well performing models, a huge amount of data is required. Earlier, researchers where lacking sufficient amount of data for the model training process. Thus, high error rates were resulting and the networks where considered useless. Due to services like Google and social media platforms, those valuable data becomes accessible and NNs get the chance to learn properly.

Another limiting factor were the available computational resources. Inference and especially training require the handling of a massive amount of data, which need to be stored and processed at proper platforms. Specialized platforms for such kind of applications were not devel-

oped at that time and the available general purpose platforms couldn't fulfill the high computation complexity of deep networks with many layers by far. Since general purpose platforms evolved, so that multi/many core CPUs (Central Processing Units) and GPUs (Graphics Processing Units) became more efficient and powerful, dealing with deep networks seemed tangible and proved valuable.

With increasing success, the community gained interest in this domain and frameworks and tools got increasingly developed supporting the research process. This simplifies development of NNs and opened the topic for a bigger group of contributors, which accelerate the progress further.

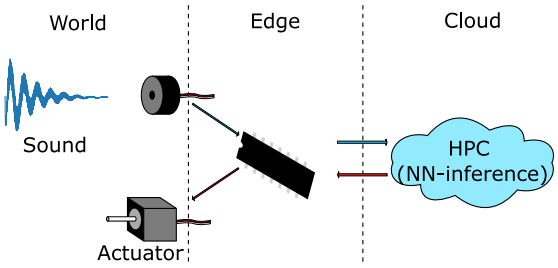
A status is reached, where intensive deployment of NNs in productive applications seems feasible. Some applications are already enabled with AI, today. Games use DNNs as smart players, cars make use of it in semi-autonomous driving and medicine deploys it for early cancer detection. In many of those enclosed tasks AI outperforms humans already by today.

The high accuracy of todays DNNs comes along with high computation complexity, which is covered by powerful platforms like GPUs or optimized high performance modules like Googles TPU (Tensor Processing Unit) (Jouppi et al. 2017). This aspect conflicts with the second goal of this thesis, i.e. edge applications.

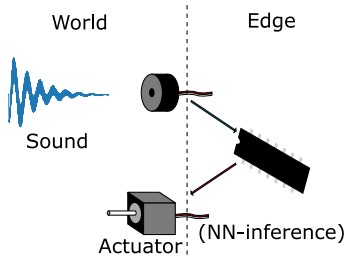
1.2. Motivation

In contrast to cloud platforms, the resource limitations at edge devices are rather strict. The manufacturing cost of the chips have to be kept low, because of the high quantity and low selling price of edge devices. On the other side, the granted energy budget is low, because those kind of devices usually run on battery and the battery life should be maximized for high user comfort. Additionally, the latency for the inference during application has to be small, since users expect response within a reasonable delay. The overlap between NN requirements and edge device capabilities doesn't seem to be significant at the first glance. Therefore, developers started to dispatch the actual inference computation of NNs to the cloud, where powerful platforms are available. Data collection and response is left to the edge platform still (refer to Figure 1.2a).

Nevertheless, this approach infers significant disadvantages. As shown in Figure 1.2a, the raw data collected via sensor is collected with the edge device first. Since the model is located at an HPC (High Performance Computer) at the cloud, raw data needs to be either directly transmitted or compressed and transmitted to that location via network. In any case, a transmission to the cloud is required and the bandwidth demand is high, because of the amount of sampling data. Edge devices are usually connected to the network via radio, since they are mobile. In low power platforms, radio is one of the main contributors to power consumption. Transmitting massive amount of data, reduces battery lifetime significantly. From a global perspective, transmitting collected data for myriads of applications from edge to cloud creates a huge bandwidth demand in the infrastructure and gains power consumption.



(a) Cloud enabled AI at the edge



(b) Accelerator enabled AI local to the edge

Figure 1.2.: Locality of NN-inference

From latency perspective, a cloud solution is not necessarily the fastest approach. HPC inferences are generally faster than executed at the edge platform, due to the availability of fast compute resources. But considering the delay introduced for cloud transmissions, inference at the edge becomes competitive (as depicted in Figure 1.2b).

Furthermore, relying on the cloud is not always a feasible solution. Some applications may also need to work reliably when in offline mode. Scenarios may be in autonomous cars, which are not allowed to quit their job when loosing radio connection, or application in regions with insufficient network coverage.

In some applications, data privacy is a concern and transmitting sensitive data to an unknown cloud server via network is critical. Inference at the edge vanishes this concern, since third party doesn't have access to the locally stored data at the edge device.

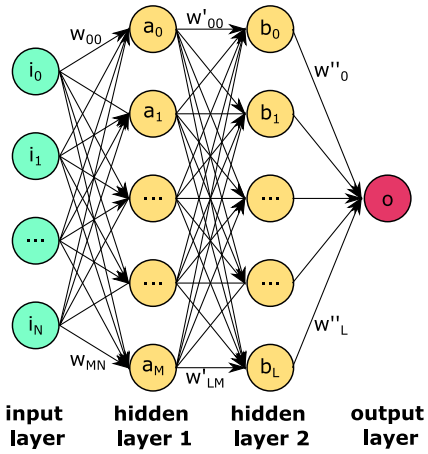


Figure 1.3.: NN-architecture

1.3. Background

In this section, different neural network architectures are introduced. Their individual advantages, disadvantages and differences are presented and referenced layer-types are introduced. Additionally, features and properties, which are applicable to individual layers are examined. Later in this section, state-of-the-art metrics are presented used for efficiency comparison of different applications executed on varying devices. The latter subsection introduces the two competing memory mapping formats used in state-of-the-art tools to store model data in memory.

1.3.1. Neural Network Architectures

AI has many facets and NNs have various shapes. In this thesis the focus is on DL (Deep Learning) and DNNs, a prominent subcategory of ML. The corresponding taxonomy is introduced in chapter 1. Every NN comprises input, hidden and output layers, as depicted in Figure 1.3. The "Input layer" is equivalent to the initial data provided to the network. The "Hidden layers" are intermediate layers between input and output layers. They cover the actual computation of the NN. The "Output layer" provides the result of the NN for a given input. The edges in the graph are assigned with weights, the nodes contain the activation value. The activations of the first hidden layer are computed according to (1.1). For the following layers, the computation is done analogous (Kelleher 2019).

$$a_m = \sum_{n=0}^{N} i_n w_{mn} \tag{1.1}$$

Categories of DNNs

There exist DNNs with certain properties, which allows to further classify them. Those are FFNNs (Fast Forward Neural Networks), RNNs (Recurrent Neural Networks) and CNNs (Convolutional Neural Networks). Their individual properties and differences are:

FFNNs also referred as ANNs (Artificial Neural Networks) are used in image classification tasks. Thereby, a two dimensional image is converted into a one dimensional array for processing. This conversion infers several issues to the complexity of the required network. First, the number of trainable parameters explodes, since parameter sharing isn't feasible. This is challenging during the training process as well as the inference with respect to resource requirements. Second, some of the spatial features are exclusively represented in the two dimensional format and get lost during conversion into a one dimensional array. Another issue is gradient related, caused by the deep layer structure common to all DNNs. For deep networks the gradient either explodes or vanishes, which makes the training process challenging. In contrast to the other DNNs introduced in this thesis, FFNNs have a significant difference. They offer no possibility to handle sequential information like videos or audio sequences.

RNNs use recurrent connections on neurons in hidden stages (shown in Figure 1.4b), which constitutes a significant difference to other DNNs (e.g. FFNNs shown in Figure 1.4a). This looping mechanism ensures proper capturing of sequential data. This peculiarity adds value for application on time dependent data, like text recognition or audio processing, since the output of a neuron does not only depend on the current input, but also on previous ones. In RNNs, parameter sharing is done implicitly, since multiple time steps reuse the same parameter set. This reduces the amount of parameters, which accelerates training and reduces the model footprint. Furthermore, it enhances cost efficiency especially for hardware. Nevertheless, RNNs and FFNNs have two similar issues. For deep networks the gradient diverges, which causes sever issues during training process. Furthermore, the network is only capable of dealing with one dimensional arrays, similar to FFNN. When applied to two dimensional data, the features exclusive to the two dimensional representation are lost.

CNNs are the most popular DNN subcategory. They are applicable to image and video processing. In contrast to the other DNN categories, they preserve the two dimensional spatial information. The underlying concept is based on filters (kernels), which are used to extract relevant features using convolution. CNNs learn relevant features automatically without manual and explicit specification. Due to the preservation of the two dimensional spatial information, object detection as well as localization and analysis of object relations within an image are typical applications. The filter based concept applies the same filter to multiple locations inside an image. Thereby, the filter parameters are shared, which increases the memory efficiency.

(a) FFNN neuron (b) RNN neuron

Figure 1.4.: NN types with different neuron connections

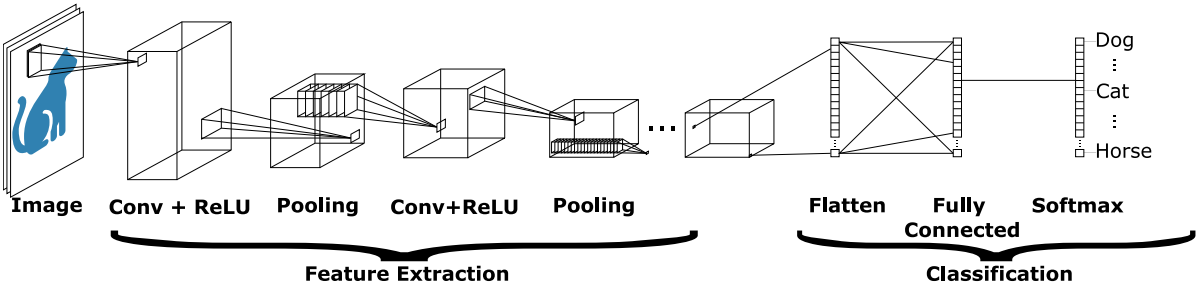


Figure 1.5.: CNN architecture (adopted from (MathWorks 2022))

Architecture of CNNs

The abstraction of a typical CNN network architecture is shown in Figure 1.5. The basic building blocks are categorized into "Feature Extraction" and "Classification". The features are extracted by applying a sequence of convolution, activation function and pooling layers to the input. Additionally, normalization commonly fused into a convolution layer is done. The subsequent classification deploys fully connected layers. The individual layers are introduced in section 1.3.1. (MathWorks 2022)

Famous DNNs

Various DNN models are available in the literature. They differ from each other via achievable accuracy, layer-depth, deployed layer types and other hyper-parameters. In this thesis, a selection of popular NNs is presented. The earliest model, which made it into application is LeNet-5. This DNN was proposed by LeCun (Lecun et al. 1998). It comprises seven layers, where two are of Conv (Convolutional) type and other two are of FC (Fully Connected) type. It was developed for the application of recognizing handwritten characters. Krizhevsky proposed AlexNet, a DNN with five Conv and three FC layers which achieved groundbreaking accuracy in image classification (Krizhevsky et al. 2017). With increasing research in this domain, the depth of the NNs became more attention and deeper networks like VGGNet with up to eight Conv and three FC layers where proposed (Simonyan & Zisserman 2014). In the same year, Szegedy presented GoogLeNet (Szegedy et al. 2015) a even deeper DNN using inception modules for image classification and detection, comprising of twentytwo layers and 12x fewer parameters than AlexNet. With increasing depth training becomes challenging, because of diverging back-propagation gradients. In order to create even deeper structures of up to 152 layers, a residual training framework was proposed together with ResNet (He et al. 2016).

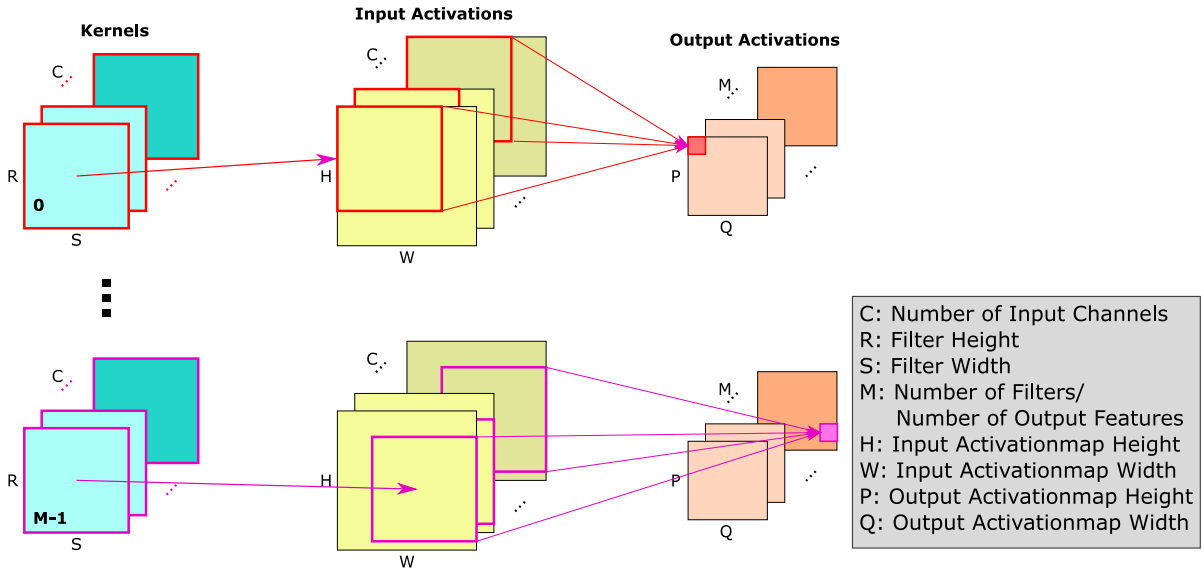
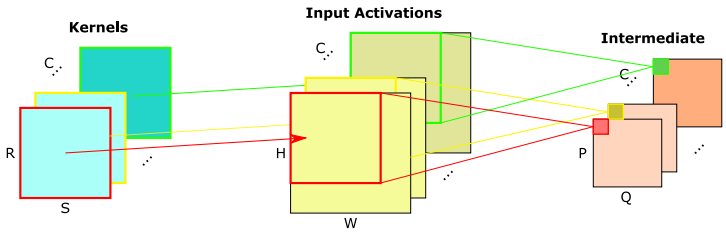


Figure 1.6.: Convolution layer operation

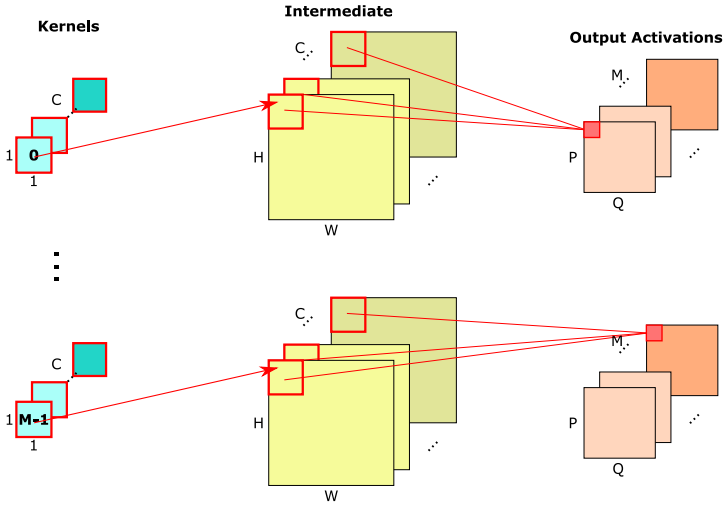
Layer Types

Previously, in this work different neural network architectures are introduced. These networks are composed of individual layers, which frequently are among the following:

Convolution Layer Convolution comprises a set of filters, which are small compared to the input activation feature map. Figure 1.6 depicts an example of a two dimensional convolution with multiple filters. The exact weights of the filters are learned during training, but the dimensions are predefined via so called hyper-parameters in advance. The depth of each filter is equivalent to the number of channels of the feature map (C). In two dimensional convolution the filter slides across width (W) and height (H) dimensions. One dimensional convolution can be understood as a special case of two dimensional convolution, where the width of filter (S) and feature map (H) are equivalent. Thus, the filter only slides across the feature map in the width dimension. In both cases, for every filter position during the filter sliding the dot product of the overlapping filter and activation window is calculated. In two dimensional convolution, this process produces a two dimensional output activation map. The depth of the output activation feature map (M) is determined by the number of applied filters. Every filter implicitly learns a different feature, e.g. edges in an image. As hyper-parameters, the receptive field, i.e. filter size (R and S), number of filters (M), stride and zero padding, are considered. The stride defines the sliding distance of the filter. Zero padding appends zeros to the borders of the actual input activation feature map. This method is applied, when the spatial feature map dimensions have to be preserved during convolution.



(a) Depthwise convolution



(b) Pointwise convolution

Figure 1.7.: Depthwise separable convolution operation, adopted from (Howard et al. 2017)

Depthwise Separable Convolution Layer Depthwise separable convolution is an alternative approach of computing the convolution operation. It separates the convolution operation into two independent convolutions, see Figure 1.7. The depthwise convolution (a) operates on per channel basis and doesn't affect the original depth (C). The pointwise convolution (b) increases the channel number of the output feature map by application of multiple kernels (M). The advantage of the depthwise separable convolution in contrast to the simple convolution is the reduced number of computations. This has the potential to accelerate the layer execution. The drawback becomes visible when dealing with small networks. With small filter dimensions the network is not able to learn features properly, because of the limited number of parameters. If handled correctly, the network significantly gains compute efficiency (Howard et al. 2017).

Fully Connected Layer Fully connected layers are a special case of convolution layers. Hereby, kernel and input activation feature map share the same dimensions. Each output activation element is a weighted sum of the input activation feature map. Figure 1.8 shows an example of a fully connected layer operation. The input and filters are treated as vectors with height (H) and width (S), respectively. The number of filters (M) determines the depth of the output activation feature map. Neither input activations are allowed to have multiple channels, nor kernel depth can be different from one. When connecting a multi-channel convolution layer

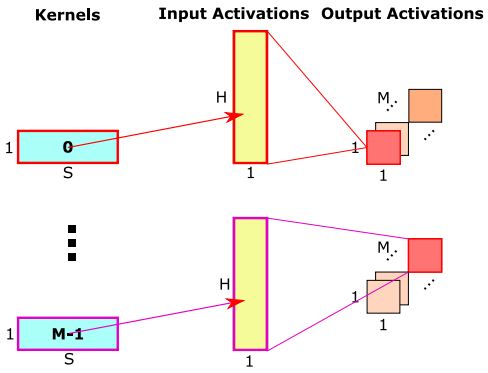
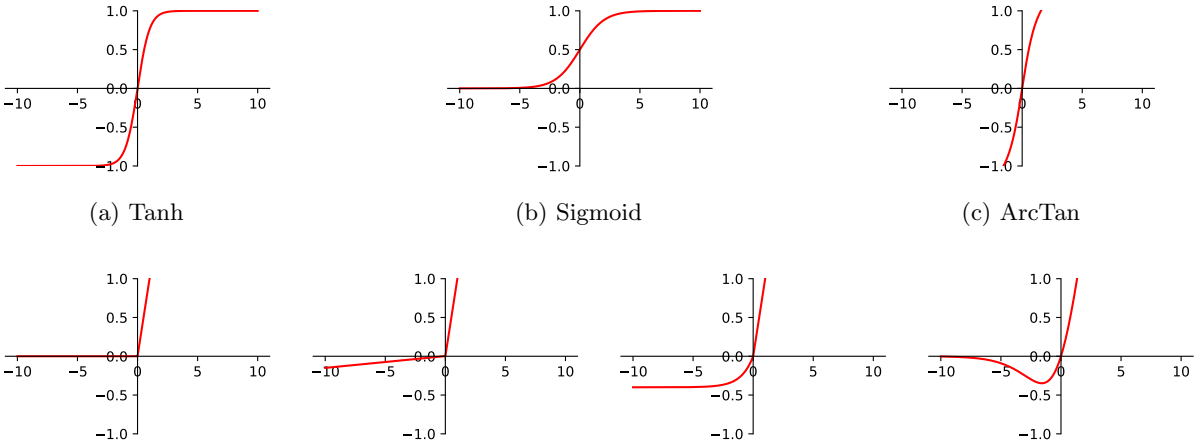


Figure 1.8.: Fully connected layer operation



(a) Tanh (b) Sigmoid (c) ArcTan

(d) ReLU (e) Leaky ReLU (f) exponential LU (g) Swish

Figure 1.9.: Various activation functions

to a fully connected layer, an intermediate flattening step is needed. In context of a CNN, fully connected layers are utilized to learn the nonlinear combination of the learned features, identified in the preceding convolution steps, for correct classification (Sze et al. 2020).

Non-linearity Layer The non-linearity is typically applied after Conv or FC layers to separate them from the succeeding layer. The non-linearity prevents folding of multiple linear operations (e.g. Conv or FC) into a single one. A multi-layer network of linear layers only without intermediate non-linearities collapses into a single linear layer. The non-linearity is introduced by so called activation functions, which are applied to every neuron. The non-linearity enables the network to describe complex non-linear input-output dependencies. Typical activation functions are shown in Figure 1.9. Sigmoid (b), TanH (a), ArcTan (c) were popular earlier. At time being, ReLU (d) is popular especially in CNNs. Due to its simple equation and implementation in HW, it provides fast training abilities. Apart from ReLU, leaky ReLU (e), exponential LU (f) and Swish function (g) are common.

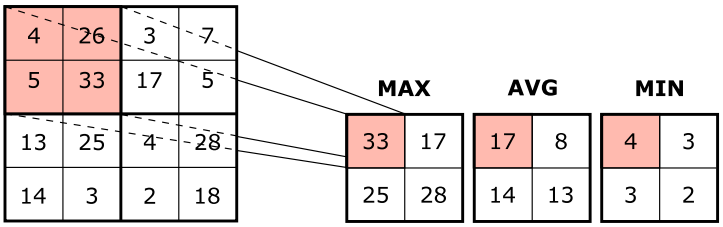


Figure 1.10.: Different pooling modes (adopted from (Wofk et al. 2019))

Pooling Layer Pooling layers are commonly inserted between convolution layers. Their purpose is the continuous reduction of spatial dimensions. This helps to control overfitting and reduces computation complexity as well as the parameter number. Pooling is operated channel-wise. The operation typically determines either the maximum, minimum or average of a spatial window within the feature map, as depicted in Figure 1.10. The window slides across width and height dimensions in non-overlapping frames. A stride equal to the pooling window dimensions ensures that every activation is treated once. Pooling is usually performed after non-linearity and supports the extraction of dominant features, which are rotational and positional invariant. Maximum pooling is more favorable than average pooling, due to lower implementation complexity and its well performance with respect to model accuracy. The pooling layer introduces two additional hyper-parameters to the model, which are stride and pooling window dimension (Sze et al. 2020).

Normalization Layer Normalization layers are located between convolution and pooling layers. They normalize the output of the previous layer and are meant to enhance the independent learning of layers while avoiding overfitting. This layer type is proven to not substantially contribute to efficient training. Therefore, they are infrequently used in modern networks. Equation (1.2) describes the normalization operation, where μ is the mean and σ the standard deviation with ϵ as a small constant to avoid numerical problems. γ and β are constants learned during training (Ioffe & Szegedy 2015, Sze et al. 2020).

$$y = \frac{x - \mu}{\sqrt{\sigma^2 + \epsilon}}\gamma + \beta \tag{1.2}$$

Model Properties

There exist methods in literature to optimize model footprint, energy per inference and execution latency. The following paragraphs introduce a collection of those methods and present a qualitative classification with respect to the impact on the inference.

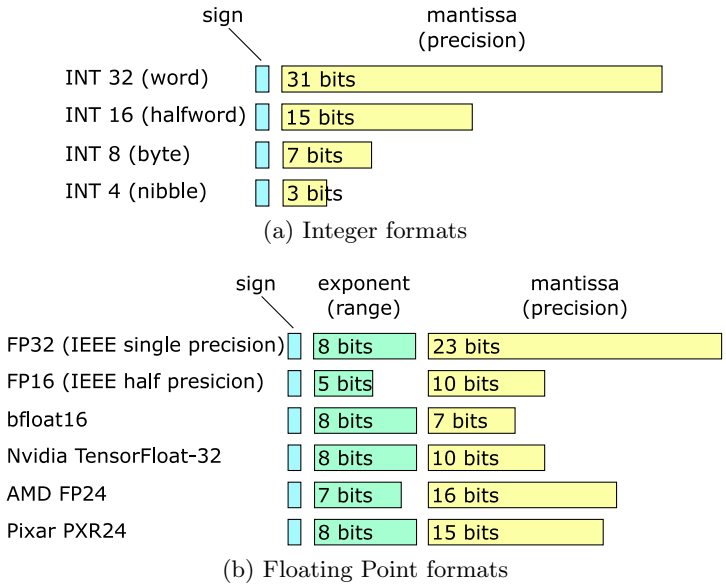
Sparsity Sparsity appears in activations and weights. It defines the repeated usage of a particular value (usually zero) with high probability of appearance. The sparsity ratio is defined by the percentage of sparse data within a bunch of datasets. From NN-computation

perspective, sparsity comprises two beneficial aspects. (1) It's useful for compression, which reduces the memory footprint. Sparse datasets don't need to be stored explicitly, instead storing their spatial location is sufficient. (2) The number of computations can be reduced, since multiplications with zero can be skipped. The multiplication result will become zero in any case and doesn't impact the final result of the dot product.

There exists one significant difference between weight and activation sparsity. In contrast to the weights, activations aren't known apriori. Therefore, forcing an increased sparsity in activations is challenging. Some methodologies to increase the sparsity ratio in activations are utilization of ReLU as activation function or operation on correlated data (e.g. video). The first sets values below a certain threshold to zero, which increases the number of zeros and the sparsity ratio. The drawback of utilizing activation sparsity is the need for additional HW to encode or compress the data during execution time. Sparsity in weights is observed naturally, since the number of weights is typically bigger than the domain range. The smaller the range, the higher the probability for zeros. Additionally, computationally more complex operations can be exploited to increase the sparsity, since weights are determined during training. Pruning is a common method during training phase to force more weights to become zero. Thereby, layers are over-parameterized, which allows to prune some of their weights to zero. An additional observation is that small dense layer perform worse than big sparse layers.

Reduced Precision (Sze et al. 2020) When mapping data into smaller domains than their initial domain, the new representation has a lower number of bits. This causes an irreversible loss of precision. Nevertheless, the reduced precision method has several advantages. With low bit-width more data can be treated in the same amount of time assuming the same HW-width. This raises the throughput, since multiple datasets can be processed simultaneously. Therefore, the unused/freed capacities of the available computation HW can be assigned to handle the next dataset. Additionally, shorter critical path in the computation units, caused by less complex/dependent operations, allow to increase clock frequency. In extend, energy per mac and memory read/write operation related to a single dataset is decreased. Furthermore, the required on-chip memory storage is reduced, which decreases the overall chip-cost.

But reduced precision also introduces some disadvantages. Because of the lower number of represented values the accuracy of the network is affected negatively. Therefore, trade off analysis needs to be done carefully. Typically, the reduced precision is realized via quantization, see section 1.3.1. It maps a wider encoding space into a smaller one. In order to get good approximation the quantization error needs to be minimized, which is the integral of the difference between the original values and their quantized representations. It allows to balance the impact of the reduced bitwidth and the accuracy of the network.



(a) Integer formats

(b) Floating Point formats

Figure 1.11.: Common data formats, (Kharya 2020) and (Kainz et al. 2009)

It's important to consider the distribution of the non-quantized data. Uniform quantization minimizes the error for uniformly distributed data. For non-uniformly distributed data, a non-uniform quantization delivers more accurate results. Processing of non-uniformly quantized data requires dedicated HW and a three-step approach: (1) dequantization (LUT lookup), (2) execution of the actual operation and (3) quantization step (LUT lookup), depending on the output distribution. Another way of increasing the accuracy is by deploying mixed-precision, which implies high precision training and low precision inference. This approach is feasible, since training is more sensitive to precision than inference. An extreme case of reduced precision is operation on binary values (one bit). This allows to map the involved operations into single gate logic operations. Complex and expensive computation units can be dropped or replaced by simple implementations (e.g. multipliers are replaced by AND-gates).

Figure 1.11 depicts different state-of-the-art integer and floating point formats, where (a) focuses on integer formats. Common fixed-point formats are "word", "halfword", "byte" and "nibble", but the optimal bitwidth for a given NN may be in between, depending on the particular application and expectations. Therefore, supporting custom formats is beneficial in real-world applications.

The formats are called fixed-point since their scale is fixed. The alternative to fixed-point are floating point formats. Common floating point formats are presented in Figure 1.11b. Hereby, the scale is variable and defined by the exponent. The advantage of using the floating point format is the representation of a wide range, due to the explicit storage of the exponent. This peculiarity causes a disadvantage with respect to precision, since the mantissa is lacking the bits spent for the exponent.

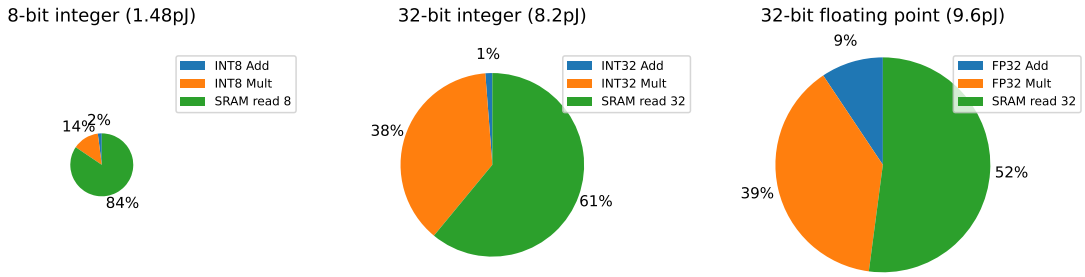


Figure 1.12.: Distribution of power consumption on involved HW-components for different data formats (Horowitz 2014)

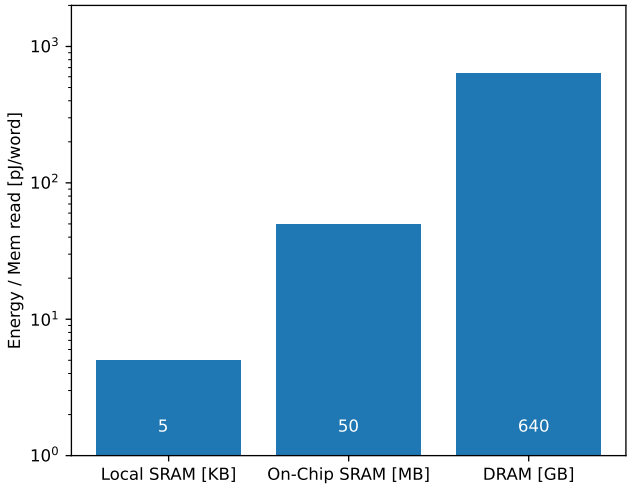
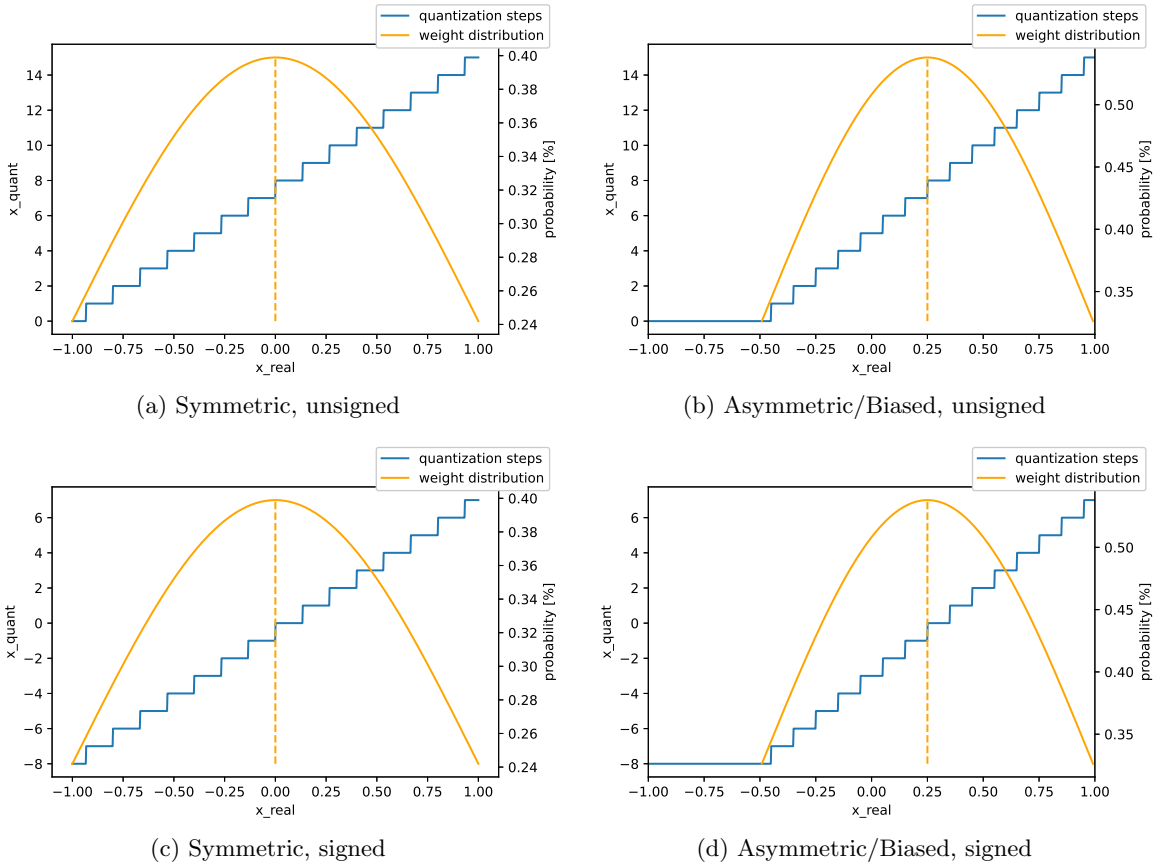


Figure 1.13.: Dependency of the power consumption on the memory distance (Horowitz 2014)

The influence of low precision formats on the energy consumption of the inference is sketched in Figure 1.12. Thereby, a 8-bit based integer inference consumes only 18% of the energy which is required for the same calculation in 32-bit integer format. Even more critical is the 32-bit floating point format. In both cases, the energy consumption is dominated by the memory read operation, followed by the multiplication.

Another aspect regarding energy consumption of the memory is depicted in Figure 1.13. It shows that the consumed energy highly depends on the type of the memory. "Local SRAM" is the cheapest from energy perspective, but rather limited, because of high manufacturing cost. "DRAM" is the most expensive from energy perspective, but manufacturing costs are low. To summarize, it's beneficial to keep the model size low to be able to fit it into the "Local SRAM".



(a) Symmetric, unsigned

(b) Asymmetric/Biased, unsigned

(c) Symmetric, signed

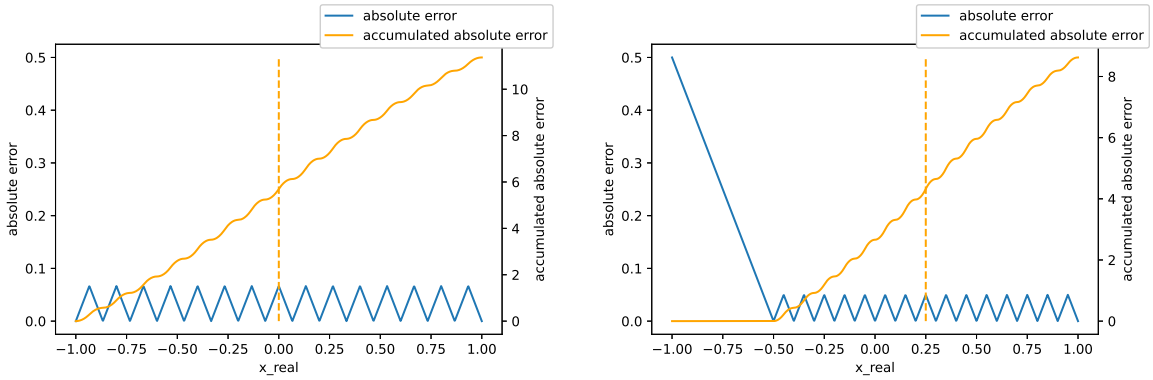
(d) Asymmetric/Biased, signed

Figure 1.14.: Quantization schemes, for a real value in the domain [-1.0,1.0] (x_{real}) into 4-bit (x_{quant})

Quantization In the following, the background of quantization and possible schemes are introduced.

In order to deploy the advantage of reduced precision formats within the execution of NN inference, the model data needs to be quantized. Quantization means to chop a domain range into pieces. Multiple similar values of any piece in the original domain are mapped to a common value, as shown in Figure 1.14. This reduces the number of possible values and discretizes the original domain at the cost of accuracy loss.

In Figure 1.14, quantization of the real domain $[-1.0, 1.0]$ is shown for a 4-bit precision example, chosen for the sake of simplicity. In (a) and (b) the quantization maps to an unsigned domain, in (c) and (d) signed integer domain is chosen. For a symmetric distribution of the real values around zero, symmetric quantization provides the best accuracy trade-off (refer to (a) and (c)). Typically, the real values have a symmetric distribution around a biased value different from zero. Therefore, a biased quantization (refer to (b) and (d)) optimizes the achievable accuracy and minimizes the absolute and accumulated absolute error (shown in Figure 1.15).



(a) Symmetric

(b) Asymmetric/Biased

Figure 1.15.: Absolute quantization errors of the schemes introduced in Figure 1.14

The transformation from real into the quantized domain and back is described by (1.3). The mean of the quantization values x_{quant} is shifted by zero-point (zero_point) to the center of the distribution. The quantization factor (s) scales the real values to match the quantized domain.

$$x_{real} = s * (x_{quant} - zero\_point) \tag{1.3}$$

where:

x_{real}	:	Floating point representation
x_{quant}	:	Quantized representation
s	:	Quantization Factor
$zero\_point$	:	Quantization zero-point

Quantized Dot-Product Computation The computation of the dot-product in real domain is represented by (1.4). The goal is to translate this computation into the quantized domain to save expensive domain translations and avoid the computation in real domain. When applying the translation function (1.3), the dot-product is transferred in the quantized domain. The quantized representation of (1.4) is (1.5). In (1.5) several offline predictable constants are involved. To reduce the number of operations required for the computation of the quantized dot-product, the equation is further reduced.

$$a_{out} = \sum_{i=0}^{N} w(i) * a_{in}(i) \tag{1.4}$$

$$s_{out}(a_{q_{out}} - zero_{a_{out}}) = \sum_{i=0}^{N} s_w(w_q(i) - zero_w) * s_{in}(a_{q_{in}}(i) - zero_{a_{in}}) \tag{1.5}$$

$$a_{q_{out}} = -zero_{a_{out}} + s * \sum_{i=0}^{N} (w_q(i) - zero_w) * (a_{q_{in}}(i) - zero_{a_{in}}) \tag{1.6}$$

$$a_{q_{out}} = s * \sum_{i=0}^{N} (a_{q_{in}}(i) * w_q(i) - a_{q_{in}}(i) * zero_w) + bias \tag{1.7}$$

$$a_{q_{out}} = s * \sum_{i=0}^{N} (a_{q_{in}}(i)(w_q(i) - zero_w)) + bias \tag{1.8}$$

where:

a_{in}	: Input activation as real value
$a_{q_{in}}$	: Quantized input activation
$zero_{a_{in}}$	: Quantization zero-point of the input activations
s_{in}	: Quantization Factor of the input activations
w	: Weight as real value
w_q	: Quantized weight
$zero_w$	: Quantization zero-point of the weights
s_w	: Quantization Factor of the weights
a_{out}	: Output activation as real value
$a_{q_{out}}$	: Quantized output activation
$zero_{a_{out}}$	: Quantization zero-point of the output activations
s_{out}	: Quantization Factor of the ouput activations
s	: Combined quantization factor
$bias$	: Offline predictable offset constant

The common quantization factor s merges s_w, s_{in} and s_{out} into a offline predictable constant (1.6), using (1.9). The zero-points and the summation of weights are offline predictable. This allows to summarize them into a common constant "bias" (according to (1.10)). Applying the bias translates (1.6) to (1.7). Finally, the activations are factorized. Equation (1.8) constitutes the reduced computation of the dot-product in the quantized domain, which can be executed more efficiently. Remaining are weight zero-point ($zero_w$), $bias$ and the combined quantization factor (s), next to the actual multiplication of weights and activations and the addition. An efficient execution of the quantized dot-product on a HW platform, requires proper treatment of them.

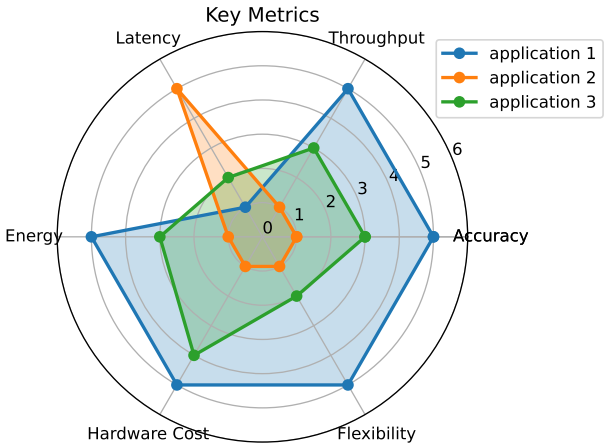


Figure 1.16.: Example distribution for key metrics with different applications

$$s = \frac{s_w * s_{in}}{s_{out}} \tag{1.9}$$

$$bias = zero_{a_{in}} * zero_w - zero_{a_{out}} - zero_{a_{in}} * \sum_{i=0}^{N} w_q(i) \tag{1.10}$$

Some peculiarities have to be considered when defining a proper HW platform for computation of the quantized dot-product, arising from (1.8). The domain resulting from the subtraction $w_q(i) - zero_w$ exceeds the original encoding domain of the weights. For the quantized weights (w_q) as well as weight zero-points ($zero_w$) all values within the corresponding domain are valid, e.g. [0,15] in 4-bit unsigned integer. Further, weight zero-points ($zero_w$) vary between layers, thus restrictive assumptions on the domain of the subtraction result cannot be taken. Therefore, the worst case scenario for the result domain (e.g. 5 bit signed) needs to be covered by the HW platform. This property generally increases the bitwidth demand for the utilized multiplier by one compared to the original quantized weights domain and makes signed support mandatory.

1.3.2. Key Metrics for Neural Network Hardware

The comparison of different DNNs on varying platforms with different applications is challenging. Therefore, a common basis for the evaluation of the efficiency is required. The meaning of efficiency is not trivial, since multiple aspects have to be considered. The most important of those aspects (shown in Figure 1.16) are discussed in the following.

Accuracy

Accuracy describes the ratio between the number of correct predictions (True Positives and True Negatives) in relation to the overall number of predictions, according to (1.11) (Powers 2020, Sze et al. 2020). It describes the quality of a certain NN for a defined application. The accuracy of the prediction done with the same NN model varies with the application. Additionally, every application comes with a different requirement with respect to accuracy. Therefore, this metric can be evaluated properly in context only. Generally spoken, a complex application requires a more complex network than a simple application, given a certain accuracy.

$$Accuracy = \frac{Number\ of\ correct\ predictions}{Total\ number\ of\ predictions} \tag{1.11}$$

Throughput and Latency

Throughput is a measure that describes the amount of operations executed in a given amount of time, see (1.12) (Sze et al. 2020). This metric heavily depends on the platform specifications and is often a critical parameter (e.g. video decompression, with a certain frame rate). The throughput of a certain platform can be modified by addressing bottlenecks and exchanging modules with more powerful solutions.

Latency describes the delay between the data arrival time at the SoC (System-on-Chip) and the completion time of the results, see (1.13). Latency and throughput are often considered closely related to each other, but estimating the latency through throughput may lead to underestimation. Especially in realtime applications such an approach causes issues, due to batching. Nevertheless, with increasing complexity of the NN the number of involved computations grows. Thereby, assuming a given throughput, the inference latency is prolonged.

$$Throughput = \frac{inferences}{second} = \frac{operations}{second} * \frac{1}{\frac{operations}{inference}} \tag{1.12}$$

$$Latency = t_{processing\ started} - t_{processing\ finished} \tag{1.13}$$

Energy

Energy is a critical design metric especially for devices located at the edge. Many of them run on battery and have a very limited energy budget. This ensures small devices (dimensions of battery), low manufacturing cost and maximizes battery life. The energy efficiency is described in $inferences/joule$ (1.14) (Sze et al. 2020). For the NN-accelerator design, the separation of the total system energy into $E_{total} = E_{mem} + E_{compute}$ is useful to identify potential inefficiencies (e.g. the chosen type of memory in Figure 1.13).

$$Energy = \frac{inferences}{joule} = \frac{operations}{joule} * \frac{1}{\frac{operations}{inference}} \tag{1.14}$$

Chip Production Cost

The revenue of a chip manufactured in industry is defined by (1.15). For high quantity chips like for edge devices, $c_{fabrication}$ becomes the major driver of cost. A strong driver of $c_{fabrication}$ is the chip-area. Especially, on-chip memory and computing HW strongly contribute to the chip-area. (Sze et al. 2020)

$$Revenue = p_{selling} * N - (C_{development} + c_{fabrication} * N) \tag{1.15}$$

where:

$p_{selling}$	: retail cost per unit
N	: Quantity of sold units
$C_{development}$	: Development cost
$c_{fabrication}$	: production cost per unit

Flexibility

The variety of DNNs and features supported by a platform describe its flexibility. With intensive research and development, new features become available and the flexibility to support them becomes increasingly important. This trend makes the design of efficient accelerators challenging, since efficiency is often coupled with specialization and concentration on deployment of special features of certain DNNs limits the acceleration capabilities for new applications in future. Instead, a more general approach needs to be followed, which generalizes the acceleration in a way that is also applicable to new and unknown features. The degree of flexibility can be roughly classified into three groups. (1) “No-flexibility” executes only a defined set of

operations, which are required for a certain DNN instance. (2) "limited flexibility" supports also not explicitly accelerated features, but a non-accelerated or slowed down processing has to be accepted. The ultimate solution comprises (3) "efficient flexibility", which allows to accelerate also unknown features. In any application a careful analysis of this metric has to be done, since the flexibility is in trade-off to cost. (Sze et al. 2020)

Scalability

If a wide range of throughputs and efficiencies need to be supported with the same platform, a well scaling acceleration approach is beneficial. For applications executed on edge devices, low throughput and high efficiency are desired. In contrast, on high performance systems the focus typically is on throughout, whereas efficiency plays a subordinate role. If the platform owns good scalability properties, the same components are reusable in both applications without major and expensive redesign efforts.

1.3.3. Memory Mapping of Sampling Data

Two dimensional images are encoded in three color channels as explained in section 1.4.5. The color channels are interpreted as third dimension (depicted in Figure 1.17). Two contradicting formats, i.e. HWC (Height Width Channel) and CHW (Channel Height Width), are common to map a multi dimensional image into a one dimensional representation. The one dimensional representation can be seen as an intermediate representation between the three dimensional image and its mapping to the memory.

The mapping followed within this thesis is the HWC mapping, also called "channel minor" format. Hereby, all color channels (C) for one pixel are indexed in increasing order first, followed by the horizontal dimension (W). The major dimension constitutes the vertical dimension (H). Algorithm (1) shows this indexing.

```
for h = 0; h < H; h = h + 1 do
    for w = 0; w < W; w = w + 1 do
        for c = 0; c < C; c = c + 1 do
            idx = h * (W * C) + w * C + c
        end
    end
end
```

Algorithm 1: "HWC"-indexing

The alternative to the HWC format is the CHW format, also called "channel major" format. Algorithm (2) describes the indexing of the CHW format. Hereby, the horizontal dimension (W) is indexed first, followed by the vertical dimension (H). The color-channel dimension (C) is the major dimension.

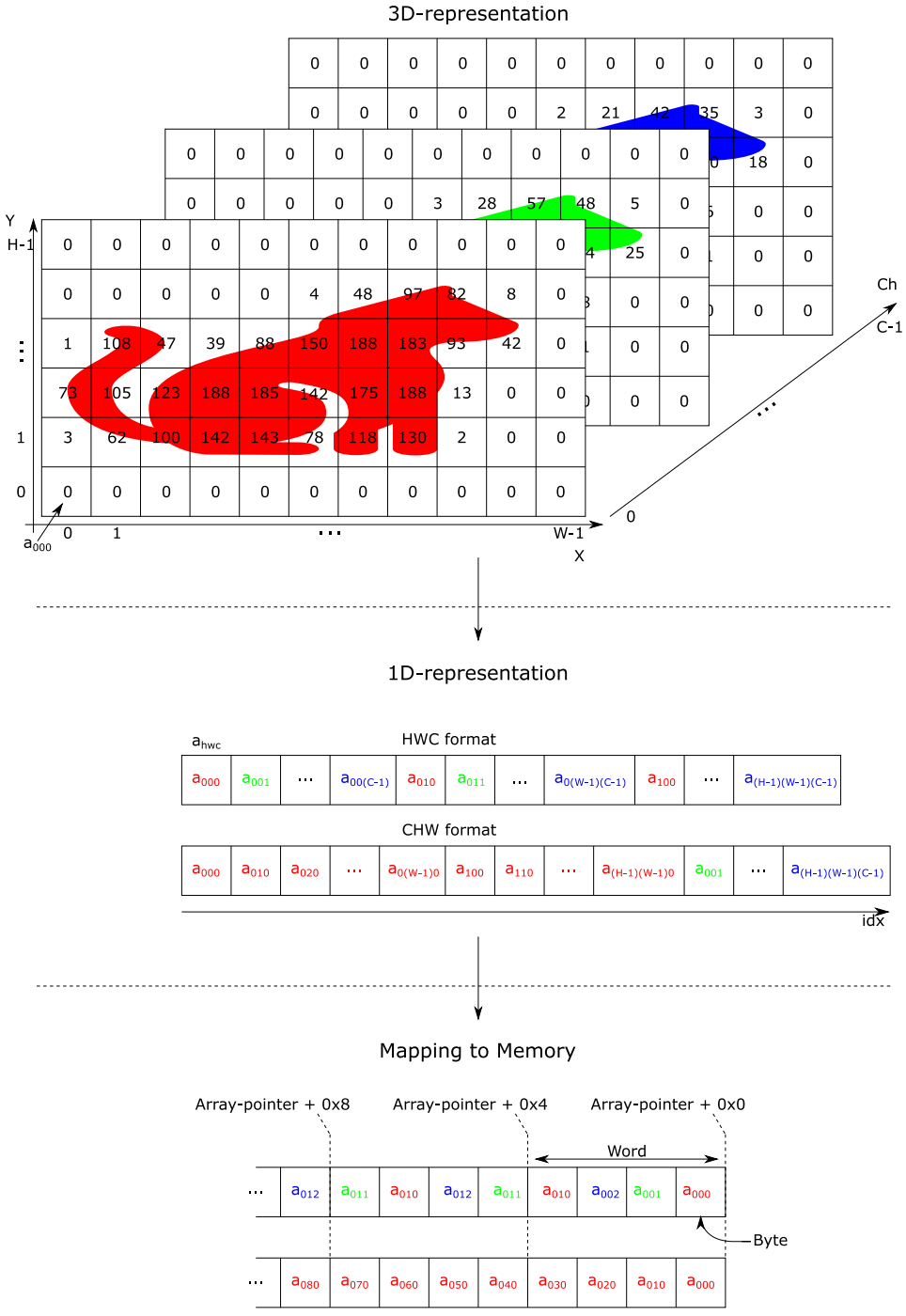


Figure 1.17.: Memory mapping of images

```
for c = 0; c < C; c = c + 1 do
    for h = 0; h < H; h = h + 1 do
        for w = 0; w < W; w = w + 1 do
            idx = c * (W * H) + h * W + w
        end
    end
end
```

Algorithm 2: "CHW"-indexing

The chosen format is platform and toolchain dependent. The supported format of the platform and the encoded image stored in memory have to be of the same format to enable efficient processing. Conversion between the formats can be done online, but increases the overhead. Instead, to avoid performance loss, offline conversion is suggested wherever possible.

Furthermore, Figure 1.17 shows the mapping between the one dimensional representation and the memory. The memory is assumed to be a byte addressable memory with accesswidth of word. The first color value in the one dimensional representation has index "0". It maps to the first entry of an array in the memory, which is addressed with an array pointer. The following array elements (bytes) hold the color values of the one dimensional representation in increasing order. Four consecutive bytes (color values) are assigned to a common word.

1.4. Objectives

In section 1.2, the motivation for enabling edge-devices with AI capabilities, especially for inference was presented. Section 1.3 introduced typical operations required during inference of DNNs and described the metrics to consider, when evaluating a NN-accelerator platform in section 1.3.2. In this section, the objectives and goals of this thesis are outlined.

1.4.1. Accelerator Performance Classes

Since AI became popular already one decade ago, mapping it to typical standard platforms have been done in the past. Figure 1.18 depicts an overview of the different categories and opposes their typical properties. On the left side of the figure, the cheapest platforms are shown, whereas the cost increases towards the right. uCs (Microcontrollers) are considered as cheap platforms. Their manufacturing costs are low, due to small on-chip area and high quantity. The small footprint is achieved by very limited compute performance and memory, which reduces their value for compute and data intensive NN-applications. Execution of NN-inferences for small models is possible, but significant response delays have to be accepted. For many applications, this trait is not acceptable.

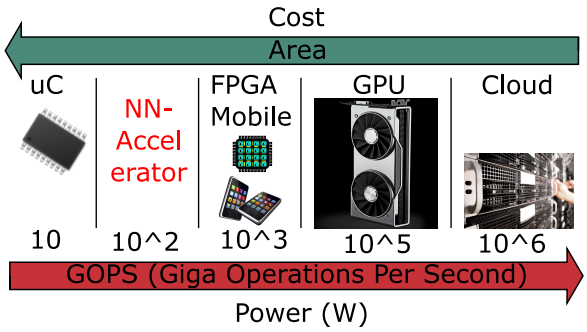


Figure 1.18.: Accelerator performance classes (adopted from(Reuther et al. 2019))

The next available performance level constitute FPGAs (Field Programmable Gate Arrays) and mobile platforms. On one hand, their cost and energy efficiency drops, since chip-area increases and more powerful compute units are used. In the case of an FPGA-based platform, they can be even redesigned and optimized for distinct NNs. On the other hand the configurability infers inefficiencies to area and energy consumption, which are avoided on ASICs (Application Specific Integrated Circuits). Mobile platforms are designed with energy efficiency in mind. Nevertheless, they only constitute a suboptimal solution for edge-AI applications. They weren't intrinsically designed for AI purposes and therefore provide a different resource balancing (e.g. memory typically is over-sized for edge optimized NN models and compute performance is too weak for model training).

AI has been also mapped onto even more powerful hardware platforms like GPUs. For inference purposes, platforms of this category are less appropriate. Due to their high power consumption, they are less portable, thus useless for many edge applications. Additionally, many applications require significantly lower computational capabilities during inference and the high cost makes them unreasonable. Instead, model training phases benefit from the increased computational capability of GPUs with respect to reduced training time. Since the model training process can be done offline, thus performed in stationary compute farms, it's sufficient to provide a limited number of GPUs in data centers for that purpose and dispatch inference execution to small, cheap and efficient high quantity platforms.

The rightmost category in Figure 1.18 comprises platforms available through cloud. Thereby, HPC platforms are shared with many edge clients simultaneously. The available compute performance is high, such that it's sufficient for model training as well as multiple parallel model inferences. In contrast, the invest and power consumption is high. Further, resource sharing may cause higher congestion and latency under certain conditions than expected.

From this discussion the gap between uC and FPGA/mobile categories for inference at the edge becomes obvious. In this thesis a platform is explored, which is equipped with higher computational capabilities and efficiency than available through common uCs. At the same time, lower cost than FPGA and mobile platforms are aimed to fulfill the desires mentioned in section 1.2. This approach addresses the execution of the inference for NNs with increased complexity locally to the edge.

1.4.2. Trade-off Triangle

In the previous discussion, three properties are considered to categorize and classify the different platforms. Those are power consumption, cost and computational capabilities. On-chip area and cost are considered in a common property, since both become similar for high quantity platforms. Thereby, manufacturing cost is the dominant term, whereas development cost are negligible. In Figure 1.19, different platforms are compared based on latency, energy consumption and area.

With increasing computational capabilities the theoretical latency is reduced. In real-world applications, latency doesn't only depend on the computation but also on the memory bandwidth. The effective latency is only loosely related with computation capability.

The main drivers for energy consumption are memory accesses and the execution of the dot-product operation. For high energy efficiency, the energy per inference needs to be minimized. Therefore, avoiding oversized system memories, unnecessary memory accesses and ensuring high dot-product payload utilization are keys to attain an energy efficient platform.

Figure 1.19 opposes cloud, GPU and uC with respect to those metrics. uCs are typically designed to be very energy efficient. Their low complexity and small on-chip footprints including memories ensure that property. Because of the low computation capabilities the according inference latency is rather high.

GPUs come with an entirely different mapping. Hereby, area and energy are less of a concern, due to their stationary application. Instead, the main focus is on performance. Fast and local memories with wide interfaces along with highly parallelized compute units enable low latency.

Earlier cloud enabled edge AI was discussed. Figure 1.19 shows this solution as suboptimal in any of the three metrics. Latency is high, even though the computational capabilities of HPCs are high, but congestion and data transmission via network constitute bottlenecks. Energy consumption exceeds even GPU based solutions, since network transmission consumes additional energy. Cloud HPCs often rely on GPU stacks, therefore the area can compete with that of GPUs.

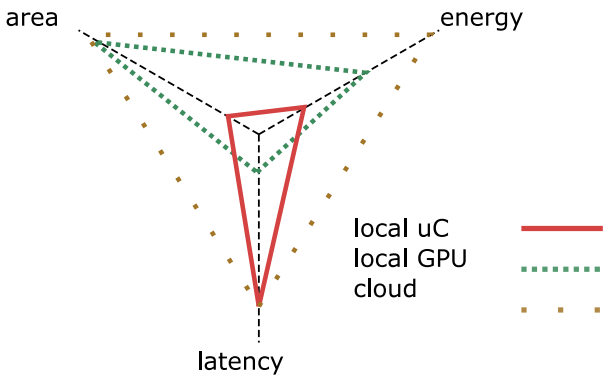


Figure 1.19.: Area, energy and latency balancing for AI-inference on different systems (Prebeck, Ashok, Vaddeboina, Devarajegowda & Ecker 2022)

All of those platforms have use cases, for which they are optimized and where they perform well. None of them fit well for the application in NN-inferences at the edge. The proposal of this thesis comprises a platform, which unifies the advantages of the presented platforms in any of those three metrics. The goal is an energy efficient platform with reasonable latency at small on-chip area optimized for the inference execution of NN applications at the edge.

1.4.3. Balancing of Resources

For every application, certain operations are performed in a distinct order, which is defined by a kernel. It loads data from memory, prepares them in a proper way and executes the compute operation. Every kernel has a unique ratio between data fetch and compute density. Some applications require a rather limited amount of data, while implying heavy computation loads. For other, the opposite holds and many are located in between.

Any execution platform provides capabilities in both domains to a certain extent. According to (Williams et al. 2009), those capabilities can be visualized in a roofline model. An example for it is shown in Figure 1.20. Herein, the horizontal line bounds the computational performance, whereas the diagonal bound limits the communication bandwidth. The minimum between both lines describes the overall platform performance. This coherence can be estimated according to (1.16). The focus is on floating point operations at computation side and off-chip memory accesses on communication side. The operational intensity is described by the kernel specific Flops/byte.

$$\begin{aligned} Attainable\ GFlops/sec = Min(&Peak\ Floating\ Point\ Perf, \\ &Peak\ Memory\ Bandwidth\ * Operational\ Intensity) \end{aligned} \tag{1.16}$$

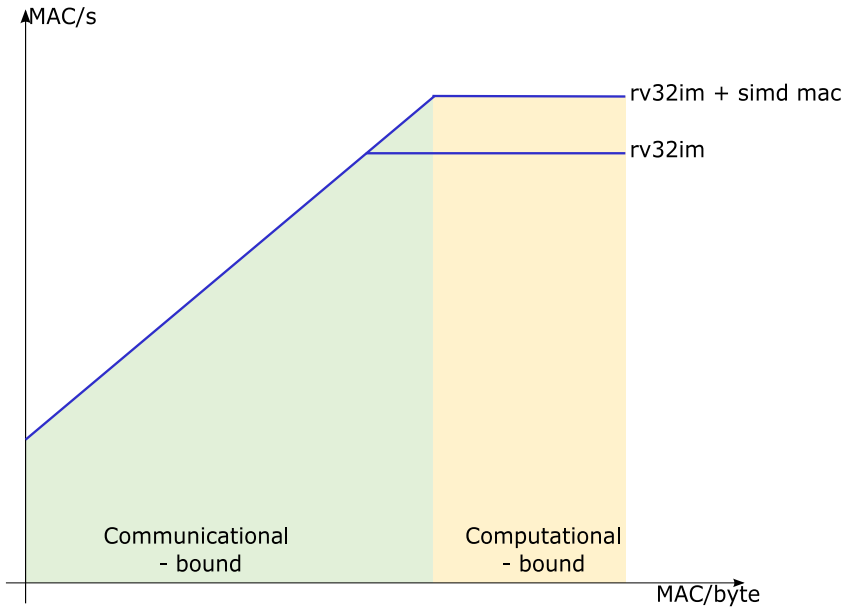


Figure 1.20.: Williams roofline model for a RISC-V rv32im vs. SIMD-MAC-extended rv32im

The roofline model defines an operation point for every kernel at any platform. A well fitting platform is indicated by a kernel operation point close to the intersection of communication and computation bound. Here, the throughput of the computation unit and the amount of delivered data from memory match. None of them acts as bottleneck. If a kernel hits the intersection, the platform is optimal for its execution. Usually kernels don't match a predefined platform well. Useful platform features need to be identified and added to raise the delimiting bound.

Specification and development of an optimized platform, which incorporates features to ensure well kernel support and good efficiency through high utilization of communication and computation components is part of the thesis.

1.4.4. Flexibility

Another objective of this thesis is the flexibility of the proposed platform. A sustainable concept, which is capable of dealing with unpredictable developments in NN-architecture is aimed for. Since the topic is growing rapidly and new research results and standards establish frequently, it's of utmost importance for the long-term success of a platform to be well-adaptable. This challenge is addressed with a tightly CPU-coupled accelerator approach, which allows fast, low-overhead interaction between accelerated and unaccelerated operations. Due to the general purpose architecture of the CPU, all conceivable cases for NN-model operations are covered and executable at the platform. This has the potential to introduce a performance disadvantage for unpredictable operations, nevertheless preserves the overall usability of the platform.

To reduce the area impact of the accelerator to the overall platform, area-driving components can be reused (e.g. multiplier). Additionally, subcomponents of the accelerator can be made available to the general purpose CPU, which enhances it's memory bandwidth and computation capabilities. This widens the platform usability to more usecases with similar properties in data- and compute-density.

The integration of the accelerator resources in a modular manner is essential. This enables low complexity in scalability and reusability of particular accelerator subcomponents and the entire accelerator block in other setups.

1.4.5. Applications

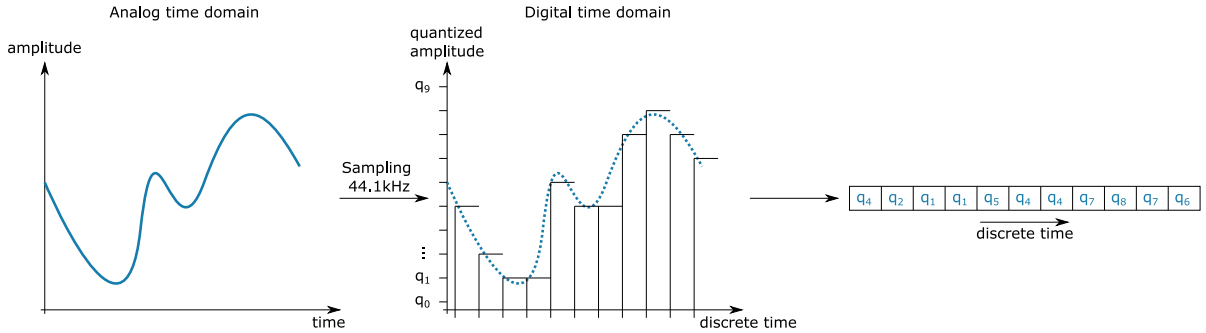
Typical NN-applications deal with audio or visual information to perform speech recognition, image classification and more. Audio information usually is recorded as one dimensional array of quantized amplitudes sampled at a distinct rate. Figure 1.21a shows the sampling process of a single channel audio signal into quantized amplitudes with a sampling rate of $44.1kHz$, which is the typical sampling rate of microphones. Sampling discretizes the analog time domain to represent it digitally.

Similarly in Figure 1.21b multi-channel audio is treated. In contrast to single-channel audio multiple tracks are recorded and sampled, which produces N-times the amount of data. Thereby, N determines the number of recorded channels.

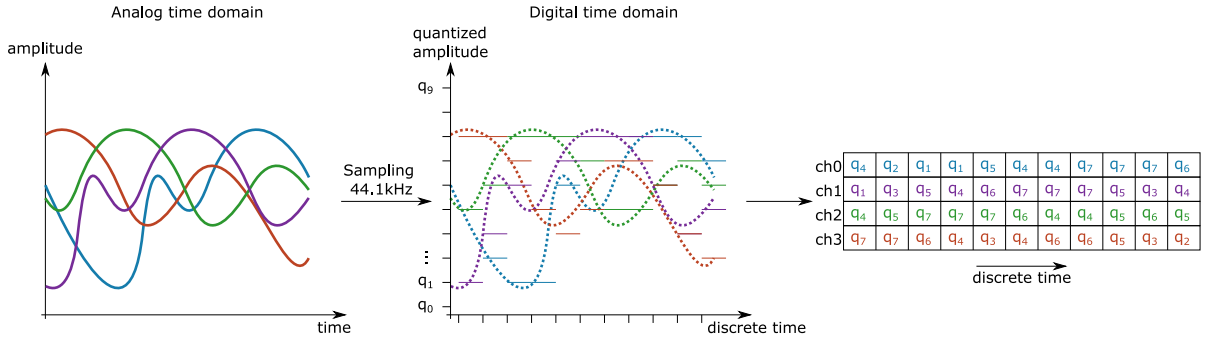
For speech recognition purposes, processing single channel audio is sufficient. This reduces the required memory size to store the input data and limits bus congestion. Additionally, the computational demand of the models are low. For audio localization applications, multiple channels are necessary. Since the amount of data grows linearly with the number of channels, the demand for memory and computational capabilities increase simultaneously.

Figure 1.22 treats the recording of two dimensional data. Hereby, an image is recorded in raw format. Therefore, the analog signal is separated into three orthogonal color channels, which allows to reconstruct the original image. The three monochrome images are sampled with a certain resolution. For simplicity reasons, in these examples the resolution is defined to 11 x 6 pixels. To each pixel a value is assigned, which defines the intensity of the according color channel for a spatial location.

An additional objective of this thesis is the definition of a flexible platform, which is capable of dealing with the mentioned 1D + T input signal types and guarantees proper acceleration for their NN inference execution.



(a) Single channel audio



(b) Multi channel audio

Figure 1.21.: 1D Sampling of audio signal in time domain

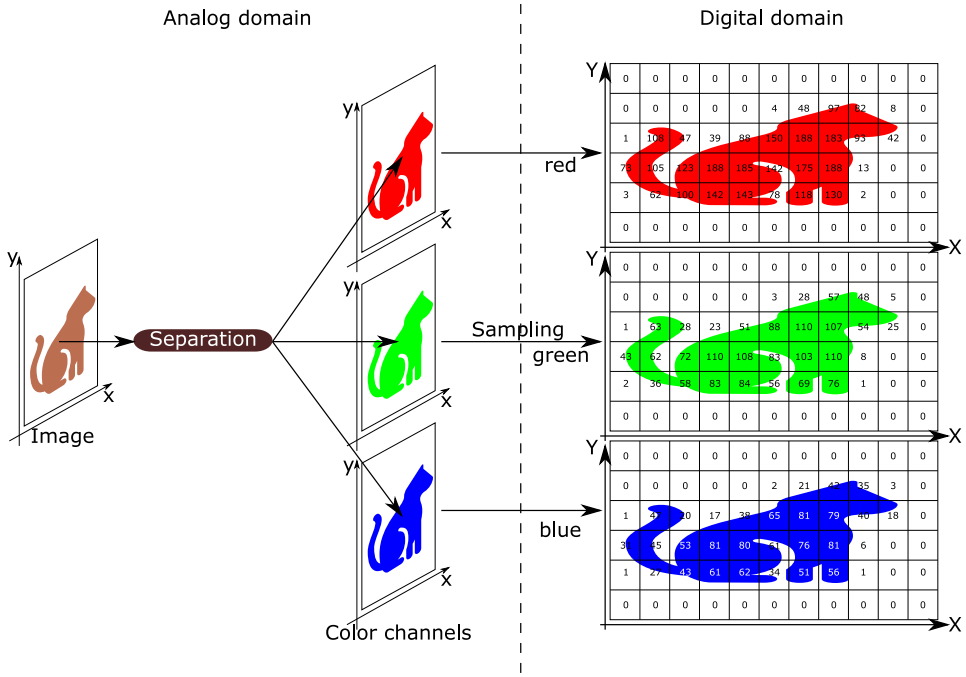


Figure 1.22.: Sampling of 2D visual signals

1.4.6. Addressing the Key Metrics

In the following, the goals targeted with the proposed accelerator architecture are discussed by means of the key metrics.

R1. Increasing the Energy Efficiency

Since one domain of this thesis are edge devices, a major concern is the energy consumption. In section 1.3.1, memory accesses and data formats are identified as strong drivers for energy consumption. Therefore, one aspect of the concept proposed in this thesis is designed to reduce the number of memory accesses and bus traffic (discussed in chapter 3). Data reuse and on-the-fly packing and unpacking of reduced precision formats avoids unnecessary memory accesses. Further, the proposed accelerator has to support fused layers, which follow convolution or fully connected layers (e.g. non-linearity). Both operations are processed back-to-back, which avoids intermediate memory accesses for storing and loading.

The number of memory accesses can be further reduced. Due to sparsity many datasets don't affect the result of the inference computation. Their fetch from memory can be avoided. Since weight sparsity is known beforehand, zero weights can be skipped in the memory. This avoids accidental fetch and energy loss. Instead, a fraction of the energy consumed for the actual weight read is invested in maintenance and evaluation of a sparsity encoding vector. For activations the same approach is applicable. Nevertheless, the activation sparsity is not know beforehand, such that a online HW sparsity encoder and the computation of a combined weight and activation sparsity code becomes mandatory. This increases the cost of the HW significantly. At the same time the sparsity ratio of the activations cannot be controlled well, such that the advantage of activation sparsity is not guaranteed. Thus, in the proposed method only weight sparsity is deployed.

The two previously introduced methods, are applicable without influencing the NN-model significantly. There is an additional aspect considered in this thesis. An application optimized HW can be designed only when additionally considering SW (Software) and training process. By doing so, the model hyper-parameter search must not only consider the model accuracy but also model footprint and compression. This allows to fit a NN-model into small memory space, which lowers the number of memory fetches as well.

The second big driver for energy consumption identified in section 1.3.1, is the multiplication and accumulation. Due to the usage of reduced precision formats, general purpose multiplier and adder units imply bad utilization. Instead, a configurable SIMD (Single Instruction Multiple Data)-MAC (Multiply Accumulate Unit) allows to increase the utilization while supporting various multi reduced precision formats simultaneously. At the same time, the throughput increases. Such an approach positively affects *energy/cycle* in an indirect fashion, since it influences the number of cycles per inference needed. Additionally, a mechanism has been established, which enables fast data pre- and post-processing.

R2. Reducing the Latency

Reduced precision in combination with SIMD does not only reduce the energy consumption, but also allows to process multiple datasets simultaneously. This increases throughput and reduces the latency with the assumption that data is always available to the computation unit. This assumption is sustained with pre- and post-processing units, which prepare data simultaneously to the computation unit. They have been constructed in a way to fulfilling the bandwidth requirements of the SIMD-computation unit.

Loading and storing of input and output activations consumes significant percentage of the overall inference execution time of a single layer. For the application of the non-linearity, direct operation on the output activations of the previous layer can be done, before storing them to the memory. This kind of layer-folding saves the load and store memory accesses for an entire activation feature map.

Further, a streaming-like architecture is seeked for, which enables a widely independent operation with low number of interruptions. Thereby, the payload cycles are maximized and latency is minimized. A logic, especially designed to generate address patterns according to a given configuration in single cycle, replaces the comparatively slow address computation of a general purpose processor.

Additionally, two types of data need to be provided for every inference computation, i.e. weights and activations. In a traditional approach, all data is handled in a common memory and is fetched via a common bus. In RISC-V, this is the data-bus. In contrast, two independent buses can be used (i.e. instruction-bus and data-bus) on either separated memories or dual-interface memories. This enables the simultaneous fetch of weights and activations and therefore doubles the memory bandwidth compared to a single common bus architecture.

Proper deployment of sparsity also reduces the latency, since zero weigts and activations don't need to be processed in the computation unit. Instead it proceeds with the next dataset immediately. This acts as a forwarding and reduces latency. Optimized or compressed models incorporate a similar behavior as the one discussed together with sparsity.

R3. Maintaining the HW-Cost

Edge devices are high quantity platforms. Thus, manufacturing cost is mainly driven by on-chip area, which constitutes the highest reduction potential. For this purpose two strategies are followed in this thesis. One limits the area spent for computation-HW, the other reduces the memory footprint through NN-model compression. The latter maintains the size of the required on-chip memory.

The area spent for computation-HW is maintained by restricting the number of PEs (Processing Elements). In best case, only one PE is configured. This lowers the maximum throughput of the overall system, but the latency benefit of an increased number of parallel PEs is lower than the involved area increase. In other words, the $performance/area$ ratio decreases with increasing number of PEs. The reason for this, is a bottleneck in data-preprocessing in such systems. This leads to significant underutilization of the available PEs. Instead, the taken approach needs to avoid bad utilization of the PE and optimizes the number of payload cycles for a single PE in favor of latency.

During NN-inference, many of the involved operations have high redundancy. For those, the provided HW experiences high utilization. Unfortunately, there are some operations left, which aren't covered by the proposed HW, but need special treatment. In an independent accelerator, additional specialized HW has to be accepted, which may be complex but with low utilization. Even though this leads to a suboptimal solution and the negative effect on energy and chip-area is known, this has to be accepted. Even worse, slightly modified requirements can exceed the accelerator capabilities. This makes an independent accelerator entirely useless. In this thesis, these special low utilization cases need to be eliminated and an alternative approach needs to be followed, which is capable of solving all of those unexpected operations while accepting lowered performance. Since their appearance is infrequent, the effect on performance is minimal. The big advantage arises from avoiding increased on-chip area and static power consumption.

R4. Incorporate Flexibility

The flexibility of the taken approach has to be high. Every conceivable DNN needs to be covered and executable on the proposed platform. For layer operations, which are not compatible to the accelerator, an appropriate fall-back solution is required. In that case, some performance loss needs to be accepted, thus the degree of flexibility is considered as "limited".

In extension to the mentioned aspect, optional support of variable bitwidth for reduced precision formats is desired. Due to the deployment of a HW-generator the specific formats can be selected for every instance of the platform. Also runtime reconfiguration between the implemented dataformats is possible, at the cost of increased HW-complexity. For a well-known application, supporting only a small set of formats is beneficial, since chip-area is optimized. For more general applications, one might accept an increased platform footprint in favor of higher flexibility.

1.4.7. Semi-automated Execution

A lower level of abstraction for the inference of the convolution layer, discussed in section 1.3.1 constitutes the pseudo-code in algorithm 3. The iteration through the different weights and activations is realized via for-loops. The loops are grouped into "inner" and "outer" loops. The "inner loops" iterate across weights of a single kernel and the overlapping activations of

the feature map. Afterwards, the according dot-product is computed, which produces one individual output activation. The "outer loops" iterate across kernels and shift the kernel frame in width and height dimensions according to the stride across the feature map.

The complexity of the NN-accelerator heavily depends on the supported loops. The remaining loops are covered via an external module (e.g. uC). A fully autonomous solution, which covers the entire convolution has to be capable of handling all of the six mentioned loops and computation operations involved in algorithm 3 without any external intervention. This extreme case of a solution gives the best performance trade off. Nevertheless, cost and efficiency are suboptimal for edge devices. Lower HW-complexity, via reduction of supported number of loops, addresses this issue at the cost of reduced performance.

Algorithm 4 shows the pseudo-code of an accelerated computation of the convolution layer for the targeted platform in Figure 1.23. The three loops previously marked as "inner loops" got folded into a single operation autonomously covered by the accelerator. Thereby the external interactions get reduced to the "outer loop".

```
for −p <= h_0 <= (H − R + p); h_0 = h_0 + s_h do                    /* outer loops */
    for −p <= w_0 <= (W − S + p); w_0 = w_0 + s_w do
        for 0 <= m < M; m = m + 1 do
            a_o h_0 w_0 m = 0;
            for 0 <= r < R; r = r + 1 do                            /* inner loops */
                h = h_0 + r;
                for 0 <= s < S; s = s + 1 do
                    w = w_0 + s;
                    for 0 <= c < C; c = c + 1 do
                        if h >= H ∨ h < 0 ∨ w >= W ∨ w < 0 then
                            a_o h_0 w_0 m += pad_val;
                        else
                            a_o h_0 w_0 m += k_mrsc * a_i hwc;
                        end
                    end
                end
            end
        end
    end
end
```

Algorithm 3: Linear computation of the convolution layer on uC, where: p: padding frame, s_h: vertical stride, s_w: horizontal stride

For FC layer computations, introduced in 1.3.1, the pseudo-code is presented in algorithm 5. The number of loops is significantly lower compared to the convolution layer. Since the kernel size is equivalent to the feature map dimensions, an "outer loop" for moving the kernel frame is not required. Additionally, fully-connected layers are flattened before computation. Therefore, kernel and activation map are one-dimensional arrays, comprising of only width or height respectively. This removes another for-loop of the "inner loops".

```
for −p <= h_0 <= (H − R + p); h_0 = h_0 + s_h do                    /* outer loops */
    for −p <= w_0 <= (W − S + p); w_0 = w_0 + s_w do
        for 0 <= m < M; m = m + 1 do
            a_o h_0 w_0 m = Σ_{r,h=0}^{R} Σ_{s,w=0}^{S} Σ_{c=0}^{C} k_mrsc * a_i hwc; /* acc inner loops */
        end
    end
end
```

Algorithm 4: Accelerated computation of the convolution layer

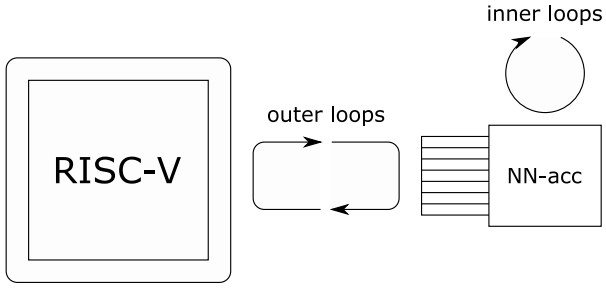


Figure 1.23.: Hybrid layer execution (accelerated SW)

Algorithm 6 presents the pseudo-code of the accelerated fully connected layer. Due to the very limited need for loops and external interaction, the entire computation of the fully-connected layer is mapped to the accelerator, which results in fast execution without relying on slow external routines.

1.4.8. Low Overhead Special Case Handling

The previous chapter explains, how to accelerate high redundancy tasks. Here, the perspective of meeting the flexibility metric and reducing HW-overhead for infrequent special handling is in the focus.

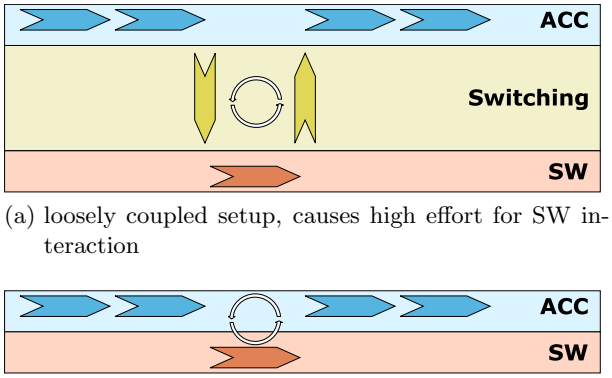
When following a high performance acceleration mechanism, the interruption process is either

```
for 0 <= m < M; m = m + 1 do
    a_o m = 0;
    for 0 <= s < S; s = s + 1 do
        h = s ;
        a_o m += k_ms * a_i h;
    end
end
```

Algorithm 5: Linear computation of the fully connected layer on a uC

$0 <= m < M$;
$a_{o\ m} = \sum_{s,h=0}^{S} k_{ms} * a_{i\ h}$;

Algorithm 6: Accelerated computation of the fully connected layer



(a) loosely coupled setup, causes high effort for SW interaction

(b) tightly coupled setup, causes low effort for SW interaction

Figure 1.24.: Effort for accelerated SW interaction

expensive or impossible. Expensive and slow switching processes need to be triggered when either leaving the accelerated mode for special handling routines or proceeding in accelerated mode after finishing special handling (see Figure 1.24a). Alternatively, the special handling is implemented inside the accelerator, which gives good performance but causes a high area footprint at low HW-utilization.

In this thesis, direct implementation of special handling is avoided, since its flexibility is limited. Especially if new operations or features are invented in future, their support isn't guaranteed. Instead, those operations are handeled externally, allowing nearly unlimited number and types of features. In order to achieve good system performance, a weaved architecture is preferred. The external controller (e.g. uC) tracks the accelerator status closely and allows immediate interruption, without significant switching overhead (shown in Figure 1.24b).

This property is also meant to be deployed for the support of operations, which cannot be efficiently mapped to HW. Possible reasons would be low utilization or high area footprint of the implementation, which is not affordable in a certain application.

1.5. Solution Approach

In order to address the objectives, which are discussed in the previous sections, this thesis proposes the following solution approach. It comprises of multiple facets.

S1 A RISC-V processor platform serves as basis for the tightly coupled accelerator. It allows rapid and low overhead integration of the accelerator modules by it's open and customizable ISA (Instruction Set Architecture). It further ensures the coverage of special/unaccelerated cases by handling in SW and weakens the requirement to support all possible cases in HW. This reduces the accelerator complexity and thereby enhances the area efficiency.

S2 The accelerator builds on top of a classical bus infrastructure. This maximizes HW-reuse since the accelerator reuses the infrastructure the RISC-V processor already requires. Hereby, the activations and weights are distributed across small and fast classical memories, which connect to the accelerator via system bus.

S3 The intention of the accelerator concept is to enhance the data transfer capabilities of the RISC-V platform and a speed up of the inference execution both at a reasonable area overhead to target the very edge.

S4 Memory is a power as well as an area critical component. Therefore, the required memory space is considered and compression methods are utilized to maintain the memory space demand. The HW provides features to cope with compressed data either in direct fashion or resolves the compression in an efficient way on-the-fly.

S5 This thesis fosters a high utilization of the MAC component, which strengthens the platform throughput. Therefore, special handling of zero values in weights is deployed and data provisioning to the MAC-unit happens in time. The latter is achieved through the aspect introduced in S3. Since a high input data bandwidth is observed, the weights and activations are read in parallel from different memories through a multilayered bus.

S6 The accelerator supports 8 and 16-bit data formats for weights and activations. This allows to evaluate the impact of reduced precision on the accuracy, memory size demand of the NN-model, and inference latency.

S7 TFlite micro is targeted as compile flow for the ML deployment. This implies the consideration of weight and activation zero-point offsets as well as the bias, which are originating from the quantized dot-product computation. Since the resulting computations exceed the domain of the in-memory data when considering the previously mentioned parameters, the accelerator datapath needs to provide additional bits i.e. 9- or 18-bit respectively.

S8 The evaluation of complex SW-mapped activation functions is cycle intensive. Therefore, the proposed solution covers the mapping of complex activation functions to HW in an efficient way.

1.6. Outline

The work is assembled to the following structure. This chapter presents the motivation, background and goals of this thesis. NN architectures as well as types of layers are discussed and deployed network properties are introduced. Further, typical key metrics for platform network evaluation are explained. Towards the end of this chapter the objectives of this thesis are presented. In chapter 2, the literature related to this topic is presented. The following chapters 3 and 4 contain the contribution of this thesis. Hereby, an in depth discussion about the proposed accelerator concept, design, building blocks and the addressing of challenges is done in the first. In the latter, the interfacing between the proposed accelerator and the SW is elaborated. Additionally, the developed SW-kernels for the supported layer types are discussed. The network compile flow for AI applications based on TF (TensorFlow) and its extension for interpreter-less execution is presented in chapter 5. A case study and the prove of concept evaluation can be found in chapter 6, whereas chapter 7 concludes the work. The overview of the author's publications follows in chapter 8.

2. Related Work

In this chapter, the literature related to the topic of this thesis is introduced. First, different NN-model compression techniques are covered, which were developed for reducing model memory footprint. Afterwards, literature related to the developed accelerator platform and its sub-components is introduced. Hereby, the main focus is on the platform processor as well as the central computation unit (dot-product unit) and its construction. Later, other NN-accelerator platforms are discussed along with their properties. The chapter ends with an overview on AI compile frameworks, which are related to the extended compile flow utilized in this thesis.

2.1. Neural Network Model Compression

Memory is crucial in embedded AI. Therefore, people propose mechanisms to reduce model footprints. Typically, two approaches are present in literature. One is sparsity, which deploys the probability of data values equivalent to zero. Hereby, the zero values are skipped from memory. For proper subsequent computation the sparsity information is encoded separately. Pruning during model training maintains the sparsity ratio of weights (Sze et al. 2020). The second compression approach exploits quantization. During model training, high precision data is required in order to achieve proper learning. In contrast, the inference process is less sensitive to model data precision. Therefore, quantization can be applied on the model data before storing into memory. Since data requires less number of bits after quantization, multiple datasets can be packed into a common memory cell. This is called packing (Sze et al. 2020). The accelerator module proposed in this thesis, provides native packing and sparsity support. The supported packing formats are 8 and 16-bits, which are packed into words. Pruning of weights and the resulting sparsity is encoded via a sparsity vector in this thesis. Additionally, the proposed HW excludes sparse weights from computation entirely and partially from memory fetching.

2.1.1. Sparsity

In (Zhang et al. 2016) the NN-accelerator Cambricon-X is proposed, which exploits the sparsity and irregularity of NN models for increased efficiency. Another proposal for a sparsity-aware accelerator is from (Parashar et al. 2017). It exploits two-sided (weight and activation) sparsity in SCNNs by computation of the Cartesian product. This method allows maintenance of sparse weigths and activations in compressed encoding. In contrast to (Parashar et al. 2017), SparTen provides native support for two sided (weight and activation) sparsity and memory storage without computation of the Cartesian product, but direct computation of the inner product

(Gondimalla et al. 2019). (Liu et al. 2022) proposed a method to exploit only structured (Density Bound Block) sparsity, which requires less complex hardware than earlier methods for unstructured sparsity. (Han et al. 2016) proposed EIE, an accelerator that natively supports compressed sparse column (CSC) format for efficient inference. In contrast to the presented literature, DeepCabac uses a video compression technique (CABAC) for NN-model compression (Wiedemann et al. 2020).

Exploiting two-sided sparsity has the potential to reduce the number of activations stored in memory and the number of dot-product computations. Nevertheless activation sparsity is network dependent and cannot be defined in advance. Thus, the potential benefit cannot be quantified for general applications. In contrast, a dedicated HW-module for handling activation sparsity online becomes mandatory, since the sparsity information cannot be predetermined offline and the handling of both weight and activation sparsity requires more expensive sparsity evaluation logic. Additionally, the sparsity information for the activations need to be stored in memory, too. For low activation sparsity this vanishes the benefit of storing a reduced number of activations.

Deploying structured sparsity reduces the HW-complexity, but at the same time underutilizes the potential of sparsity. Therefore, we propose a simple HW for one-sided unstructured sparsity, which fully deploys the potential of static and predictable weight sparsity. CSC and CABAC are fast solutions but require expensive HW to deploy sparsity, too heavy for the very edge.

2.1.2. Quantization

(Jin et al. 2019) investigated on accuracy loss, when applying quantization during training process. In this work, scale adjusted training (SAT) and parameterized clipping activation technique (PACT) are proposed to maintain accuracy degradation during quantized training. Other proposed training processes for DNNs are based on binary weights, which is the extreme case of quantization (Courbariaux et al. 2015, Hubara et al. 2017). An alternative to quantization-aware training is post training quantization. In EasyQuant, (Wu et al. 2020) deploys scale optimization for proper quantization, whereas (Shomron et al. 2021) provides an approach for sparsity-aware quantization.

In this thesis, quantization-aware training is used to reduce the bitwidth of the model parameters. In contrast to post training quantization, the model automatically learns to deal with the bit-precision loss caused by the quantization and is able to keep accuracy high. Since the applications targeted with the proposed accelerator typically need a high degree of optimization to fit on the platform, pre-trained models aren't a feasible solution anyhow. Therefore, post-training quantization which is especially useful on pre-trained models becomes obsolete. Instead the required models are trained and optimized for the specific application on demand and offline. Thus, quantization-aware training is the approach followed in this thesis.

Relying on binary weights allows a highly efficient HW implementation, since the computation of the dot-product turns from multiply accumulate into accumulate or skip. Thus, the expensive multiplier becomes obsolete and the only remaining HW is the accumulator. Nevertheless, this is a rigorous approach, which impacts the accuracy of the application more than intended in this work.

2.1.3. Entropy Encoding

The lossless data compression is motivated by Shannon's source coding theorem. It states, an information cannot be coded in less average bits per symbol than the Shannon entropy of the source defines, without loss of information (Shannon 1948). Many compression approaches attempt to reach this bound as closely as possible. All have in common that frequently used characters are coded into fewer bits, whereas scarce ones are coded into more bits. Overall the number of bits required to encode a given information is attempted to be less than the bits of the uncoded/uncompressed information. Famous compression techniques, which utilize this coherence are e.g. arithmetic coding (Witten et al. 1987) or Huffman coding (Huffman 1952). Arithmetic coding codes an entire message into a single number, whereas Huffman coding provides a mapping for a fixed set of input symbols to variable length code words. The advantage of arithmetic coding is the higher compression ratio compared to Huffman coding. In contrast, a HW capable of supporting Huffman coding is easier to implement and its performance is higher than for arithmetic coding (Shahbahrami et al. 2011). In case the approximate entropy characteristics is well known, a static code like Golomb Rice (Golomb 1966, Rice & Plaunt 1971) is applicable and provides better performance.

In this thesis entropy encoding methods are not used, since an overlap with the sparsity minimizes its benefit to the application. Sparsity exploits the property that one value has a significantly higher probability of occurrence than the others and removes it. Also entropy encoding, assigns the values with highest probability to the low bitwidth codes. Since the lower probability values remain after exploiting sparsity the compression through entropy encoding performs bad.

2.2. RISC-V Processor

RISC-V is a free and open source instruction set architecture for processors (RISC-V Specification 2021). It follows a highly modular approach of constructing processors, to cover a wide field of applications. There are many different instruction set extensions available next to the RISC-V base instruction set, which extend the processor features to fit the application requirements appropriately. It comes with a high degree of design freedom and allows custom extensions to further enhance adaptability. The license free and modular concept of RISC-V attracts many companies and leads to a fast growing community with many open and proprietary cores. Not only the number of available designs and market share increasingly grows, but also the ecosystem with software and tool-chains benefit and speed up the development.

Especially the modularity and some of the standard extensions make RISC-V appealing as fundamental platform for the accelerator proposed in this thesis. The C-extension enables the compression of program code (RISC-V Specification 2021). Hereby, the most frequently used instructions are encoded into 16-bit instead of the 32-bit wide format of the base instruction set. This allows to reduce the program memory footprint of typical binaries by $\sim 25\%$ and leads to a reduce number of instruction fetches by $25\% - 30\%$. This also reduces the spent fetch energy (Waterman 2011).

Futhermore, RISC-V provides vector (RISC-V V Extension Proposal 2021) and packed (RISC-V P Extension Proposal 2021) extensions, which allow to support reduced precision formats in a more efficient way, by simultaneous treatment of multiple datasets. Especially the vector extension requires significant amount of additional hardware to handle the introduced management overhead. This makes the extension less attractive for the purpose of a low-power, low-cost accelerator application.

The micro-architectural details of the RISC-V processor are not specified by the RISC-V foundation (RISC-V Specification 2021). Thus, many implementations of RISC-V cores are available in the literature. SiFive offers three families of RISC-V processors. The essential family starts with a 2-3 stage single issue pipeline with a minimum of 13.5 kGEs (Gate Equivalents) and ranges upto a 8 stage dual-issue superscalar pipeline. The performance family offers processors with four issue, out of order execution or vector operation support and offer high performance. The intelligence family consists out of a multi-core processor with a 512-bit wide vector extension and special AI/ML extensions especially designed for AI-applications (SiFive 2022).

Ibex is a 2-stage 32-bit open source core offered by lowRISC. It was initially developed as zeroriscy core as part of the PULP platform and it's focus is on embedded control applications. The supported extensions are Integer (I), Embedded (E), Multiply and Divide (M), Compressed (C), Bit Manipulation (B). The minimum area is $\sim$15 kGEs (lowRisc 2022).

RISCY is another representative of the PULP platform. It has a 4-stage in-order processor with single issue pipeline. It supports 32-bit Integer (I), Multiply and Divide (M), Compressed (C) and Floating point (F) extensions. Additionally, an extension for HW-loops is available, which reduces the loop overhead. The loop HW supports nested loops and maps the loop registers into the CSR space for proper restoring after handling exception routines. The HW-loop eliminates unnecessary instruction fetches, which reduces power consumption. Another feature are the post-increment load and store instructions, which increment the address pointer simultaneously to the memory access. For that, the RF (Register File) requires a second write port. This makes additional instructions for address pointer increments obsolete and thereby enables a maximum speed up of 20%. A extension for MAC operations is added, which operates on fixed point data. For packed SIMD formats other special instructions are present, which also cover dot product. The target application of that processor is energy efficient ultra low power signal processing (Gautschi et al. 2017).

The RISC-V processor used in this thesis is a 5-stage single issue processor with in-order execution. The supported extensions are Integer (I), Compressed (C), partially Multiply and

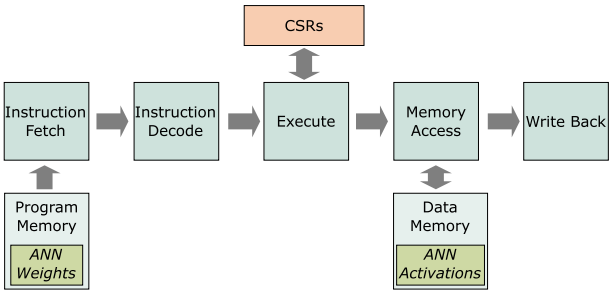


Figure 2.1.: RISC-V CPU with five stages

Divide (M) since division is not needed and custom extensions for SIMD MAC and the streaming modules, introduced in chapter 3. Post-increment load and store instructions are obsolete since the complex data fetch patterns are covered through streaming modules, which enable by far more complex patterns. HW-loops and vector extensions are not evaluated in this thesis, the implementation of the latter exceeds the on-chip area budget anyhow. Figure 2.1 shows the blockdiagram of the referred processor, which facilitates the basis of the accelerator proposed in this thesis.

2.3. Fundamental HW-building blocks for NN-Computations

2.3.1. Adder

One of the fundamental building blocks of the proposed accelerator constitute adders. The enhance dot-product unit, which is a composition of a special featured (vector modes) multiplier and final adder uses it as well as the multiplier. The architecture of the incorporated adders influences the efficiency and performance of the overall unit. Every adder architecture has its distinct properties, caused by the particular structure. The considered adder types are introduced and discussed on a conceptual level.

Carry Propagate Adder

The group of CPAs (Carry Propagate Adders) includes the RCA (Ripple Carry Adder), CLA (Carry Lookahead Adder) and Prefix Adders. They take N-bit wide inputs and an additional carry-input to produce a N-bit wide result and a carry-out bit. They are called carry-propagate, since the carry-bit of one stage (bit-wise evaluation of the sum) gets propagated from one to the next bit-wise stage (Harris & Harris 2016).

Ripple Carry Adder The simplest CPA architecture is the RCA. It chains N number of FAs (Full Adders) and connects the individual carry-out to the carry-in of the next FA, see

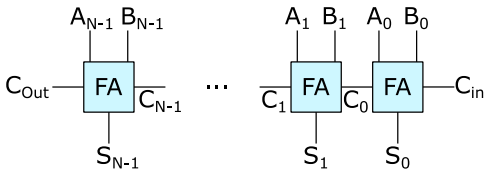


Figure 2.2.: N-bit Ripple Carry Adder

Figure 2.2. It's advantage is its high modularity and regular structure, which enables well scalability. Since the carry is propagated through all stages, the latency grows linearly with the number of FAs or bits. Due to that, this adder architecture is very slow for wider data formats, but area efficient (Harris & Harris 2016).

Carry Lookahead Adder The CLA is developed to overcome the weak performance of the RCA by separation of the carry calculation into blocks of a reduced width. These are designed in a way to do fast calculation of the carry-out, once carry-in becomes available. The carry-out calculation of a single FA can be written as (2.1). Whenever both A_i and B_i are "1" the carry-out bit is set, called "Generate". Alternatively, if at least one of them is "1" the carry-out depends on the carry-in "Propagate". When describing the carry-out of a block of multiple bits the Generate can be written as (2.2) and the Propagate is (2.3). When summarizing the carry-out equation for a block, (2.4) is attained. Figure 2.3 shows the block diagram corresponding to the equation of one of those blocks (Harris & Harris 2016).

$$C_{out} = A_i B_i + (A_i + B_i) C_{in} \tag{2.1}$$

$$G_{4:0} = G_4 + P_4(G_3 + P_3(G_2 + P_2(G_1 + P_1(G_0)))) \tag{2.2}$$

$$P_{4:0} = P_4 P_3 P_2 P_1 P_0 \tag{2.3}$$

$$C_i = G_{i:j} + P_{i:j} C_j \tag{2.4}$$

where:

A_i : i-th bit of first input
B_i : i-th bit of second input
C_{out} : Carry-out
C_{in} : Carry-in
G_i : Generate of i-th bit
$G_{i:j}$: Generate of block $[i:j]$
P_i : Propagate of i-th bit
$P_{i:j}$: Propagate of block $[i:j]$
C_i : Carry-out of i-th block

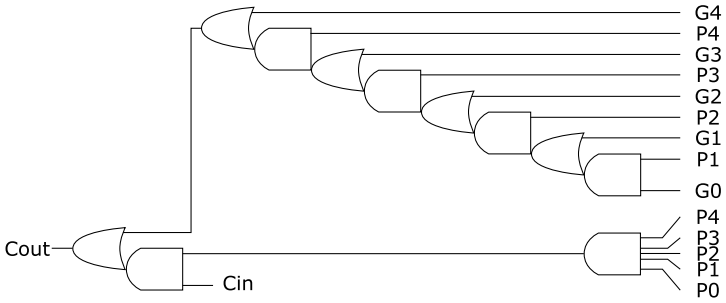


Figure 2.3.: 5-bit Carry Lookahead Block

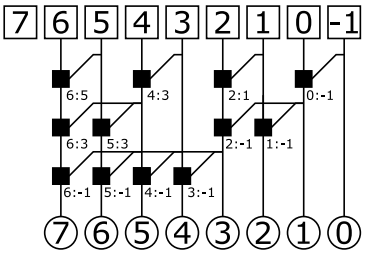


Figure 2.4.: 8-bit SK as example for prefix adders

Prefix Adders Prefix Adders extend the CLAs logic of Generate and Propagate to gain performance. Hereby, a tree structure for the prefixes (Generate and Propagate) is utilized (shown in Figure 2.4), which leads to a logarithmic complexity of the structural depth. This is especially beneficial for wide input data.

In the squared cells the columnwise Generate and Propagate prefixes are calculated according to (2.5) and (2.6). Accumulated prefixes are computed in the black cells according to (2.7) and (2.8) under utilizing of the individual prefixes. The goal of prefix adders is to compute the Generates fast. Once they are available for every column, the sum is computed in the bubbles according to (2.9).

Prefix adders allow to break the chain dependency for the carry-propagate in order to achieve better performance and logarithmic scaling, instead of linear scaling with the bit width of the inputs. At the same time the amount of used gates increases significantly.

In the following, three adders types out of the prefix adder class are introduced, which further optimize the prefix adder architecture for either area or performance.

Sklansky Adder Figure 2.4 shows the circuitry of prefix adders on the example for the SK (Sklansky Adder). It was initially proposed as the first of the prefix adders in 1960. It benefits from the parallel prefix computation in performance, but because of the high fan-out structure the performance is suboptimal (Sklansky 1960). The number of stages is defined by $\log_2(2n)$.

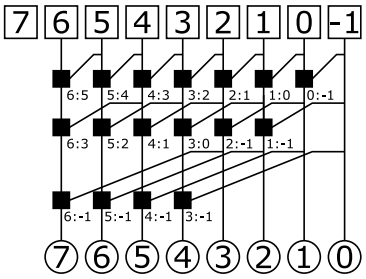


Figure 2.5.: 8-bit KS

$$P_i = A_i + B_i \tag{2.5}$$
$$G_i = A_i B_i \tag{2.6}$$
$$P_{i:j} = P_{i:k} P_{k-1:j} \tag{2.7}$$
$$G_{i:j} = G_{i:k} + P_{i:k} G_{k-1:j} \tag{2.8}$$
$$S_i = (A_i \oplus B_i) \oplus G_{i-1:-1} \tag{2.9}$$

where:

G_i : Generate of i-th bit
$G_{i:j}$: Generate of block $[i : j]$
P_i : Propagate of i-th bit
$P_{i:j}$: Propagate of block $[i : j]$
S_i : Sum of i-th bit

Kogge-Stone Adder The KS (Kogge Stone Adder) followed the SK architecture in 1973. It modifies the SK in the prefix tree structure in favor of better performance, according to Figure 2.5. The increased speed is achieved at the cost of increased area. The number of required cells is defined by $n \log_2(n) - n + 1$ (Kogge & Stone 1973).

The KS has the same amount of stages as the SK, but requires more cells. This leads to a more dense interconnect between the cells and an increased number of wires. Due to growing wire capacitance, the power consumption increases.

Brent-Kung Adder The latest development in the domain of prefix adders happened in 1982. BK (Brent Kung Adder) is designed to reduce chip area and make manufacturing easy, while achieving good performance. The cell number scales with $2n - \log_2(2n) - 2$, which outperforms SK as well as KS. The prefix tree of the BK is depicted in Figure 2.6. Hereby, the complexity is decreased at the cost of an increased number of stages (Brent & Kung 1982). Due to this structure and the low fan-out this adder architecture is faster than any other prefix adder architecture and more energy efficient at the same time.

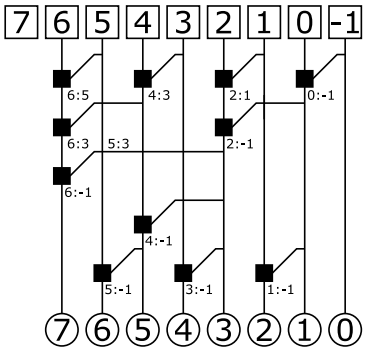


Figure 2.6.: 8-bit BK

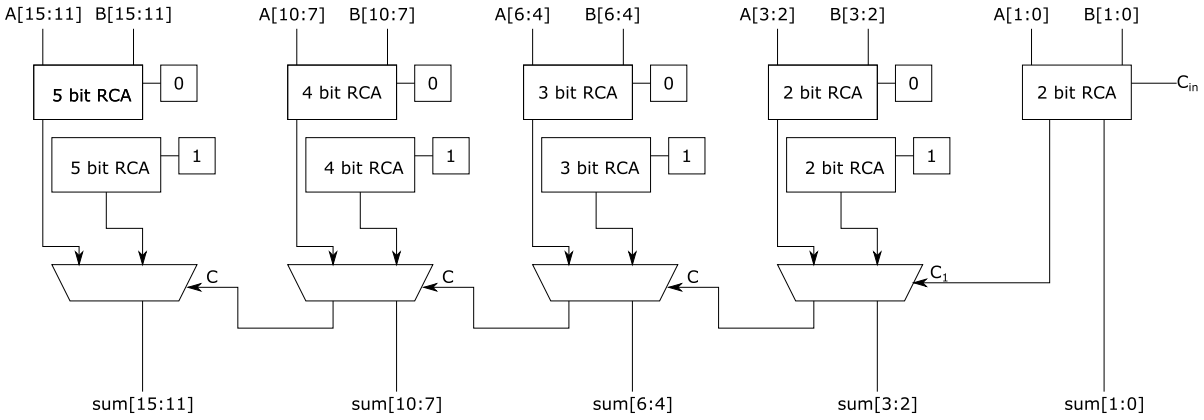
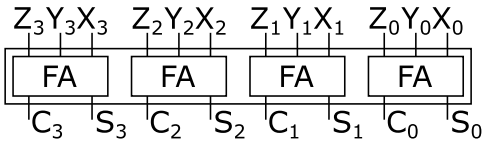


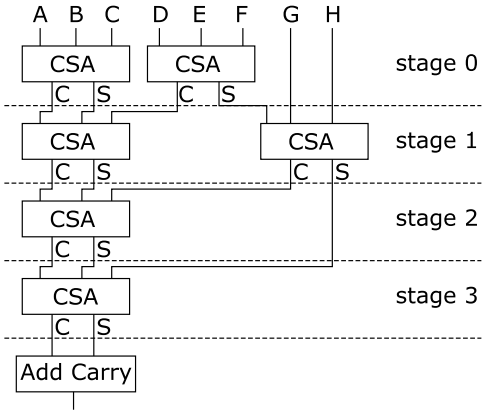
Figure 2.7.: 16-bit CSLA

Carry Select Adder

Another category of adders are CSLAs (Carry Select Adders) (Bedrij 1962), shown in Figure 2.7. They are faster, but more resource intensive than CPAs due to duplicated logic. They differentiate from CPAs in the way of treatment of the carry bit. Instead of propagating the carry bits across the stages, it calculates the sum for chunks of reduced width of the inputs for both possibilities, i.e. carry in "0" and "1", with e.g. RCA. Once the carry out of the previous chunk is determined, one of the two sums is chosen and the according carry bit is forwarded to the next selection. This structure ensures a tremendously reduced critical path, compared to CPAs. The disadvantages of this architecture are the complex and area consuming structure, which can be improved a bit by sharing of HAs (Half Adders) between the RCAs and constant propagation (Shirol et al. 2019). Later adaptions reduce the structural complexity further, e.g. (Kim & Kim 2001).



(a) Block diagram of a CSA



(b) Structure of a Wallace Tree using CSAs for 8 inputs

Figure 2.8.: Construction of the CSA

Carry Save Adder

The adders introduced so far are capable of adding two inputs. The CSA (Carry Save Adder) is entirely different in this manner. It allows to add three independent inputs simultaneously and produces two outputs with separate sum and carry information, see Figure 2.8a. For purposes where multiple operands are treated, the CSA architecture is very beneficial, e.g. for multiplications. The Wallace Tree is especially designed for that purpose (Wallace 1964). Hereby, multiple CSAs are structured in a staged tree. Carries and sums of the previous stage are feed as inputs to the next stage together with the untreated signals (shown in Figure 2.8b). Equation (2.10), describes the folding of the tree structure, where r_i is the number of addends in stage i. Such CSA-based tree structures avoid the expensive carry propagation and pushes it forward to the next stage (Lu 2005). At the final stage the carry-propagation has to be computed once. (Javali et al. 2014) proposed a fast method for the final stage addition, using a CLA. These properties enable high speed and make the architecture appealing for multi operand usecases.

$$r_{i+1} = 2[r_i/3] + r_i mod 3 \tag{2.10}$$

2.3.2. Multipliers

The application domain of multipliers is wide, e.g. in DSPs (Digital Signal Processors). The fundamental building blocks of multipliers are adders arranged in complex arrays. The multiplier and multiplicand can be represented according to (2.11) and (2.12) respectively. The formula of multiplication of A and B is shown in (2.13). Due to the intensive usage of the multiplication operation in many applications, the amount of research done in this domain is large and many algorithms are available. The main objectives during the research were on critical path, area or adder reduction and power consumption (Sadeghi et al. 2019). Because of the huge adder arrays, multipliers tend to have weak performance.

$$A = \sum_{i=0}^{m-1} a_i 2^i \tag{2.11}$$

$$B = \sum_{i=0}^{n-1} b_i 2^i \tag{2.12}$$

$$P = A(m)B(n) = \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} a_i b_j 2^{i+j} \tag{2.13}$$

In the following, common multiplier architectures are introduced.

Array Multiplier

The array multiplier is one of the lower complexity algorithms to execute a multiplication operation. It's based on a "add and shift" algorithm and has a rather regular structure (Figure 2.9). Hereby, only full-adders, half-adders and AND-gates are utilized to produce partial products and accumulate them row-wise. The structure shown in Figure 2.9 yields an RCA per row. Additionally other adder architecture can be deployed to accumulate the partial products, e.g. CPA, CSA or CLA. The individual partial products are added with the subsequent partial product in a shifted manner. This pattern is repeated for all partial products. The number of additions required is $n - 1$, where n is the bit-width of the multiplier (Liu et al. 2013).

The complex structure of the array multiplier causes some disadvantages. The multiple subsequent adder stages cause a long critical path, which limits the performance. Additionally, at least one of the operands has to be represented in unsigned (Liu et al. 2013).

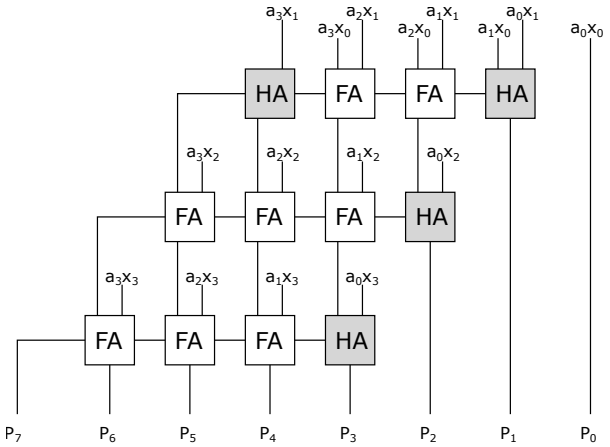


Figure 2.9.: $4x4$-bit array multiplier (Liu et al. 2013)

Baugh-Wooley Multiplier

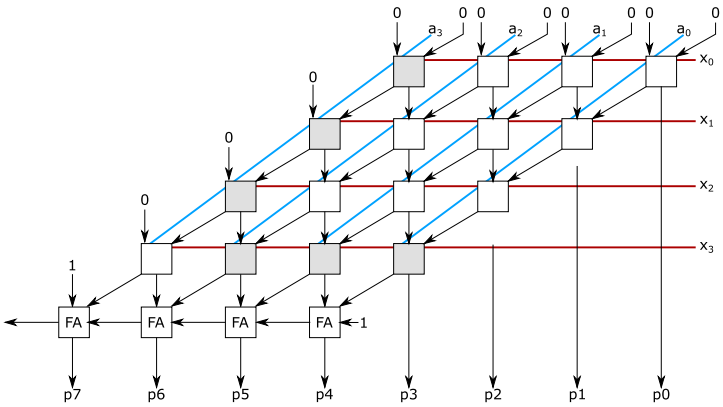
The Baugh-Wooley multiplication algorithm was invented to support 2's complement multiplication, which is not covered by the Array multiplier (Baugh & Wooley 1973). Multiplication of operands in 2's complement representation is challenging, since the sign information is embedded into the value encoding. In contrast, in sign-magnitude representation the sign is prepended. The structure of the Baugh-Wooley multiplier is uniform (shown in Figure 2.10a).

The multiplier and multiplicand in 2's complement can be represented in the form given in (2.14) and (2.15), respectively. When applying the multiplication operation on "A" and "B", the equation can be transformed into (2.16), assuming $m = n$. This results into two addition and two subtraction terms. The Baugh-Wooley algorithm is designed to avoid subtractions. Thus, those are replaced by addition of the 2's complement converted value. After refactoring, the equation is transformed into (2.17). This is the final computation pattern of the Baugh-Wooley algorithm.

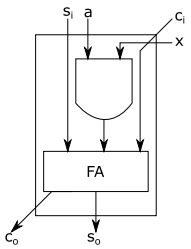
$$A = -a_{m-1}2^{m-1} + \sum_{i=0}^{m-2} a_i 2^i \tag{2.14}$$

$$B = -b_{n-1}2^{n-1} + \sum_{i=0}^{n-2} b_i 2^i \tag{2.15}$$

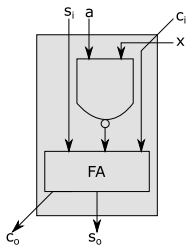
$$\begin{aligned} P = A * B = a_{n-1}b_{n-1}2^{2n-2} + \sum_{i=0}^{n-2}\sum_{j=0}^{n-2} a_i b_j 2^{i+j} \\ -2^{n-1}\sum_{i=0}^{n-2} a_i b_{n-1} 2^i - 2^{n-1}\sum_{j=0}^{n-2} a_{n-1} b_j 2^j \end{aligned} \tag{2.16}$$



(a) 4-bit Baugh-Wooley multiplier



(b) White cell



(c) Grey cell

Figure 2.10.: Construction of the Baugh-Wooley multiplier

$$
\begin{aligned}
P = a_{n-1}b_{n-1}2^{2n-2} + \sum_{i=0}^{n-2}\sum_{j=0}^{n-2} a_i b_j 2^{i+j} \\
+2^{n-1}\sum_{i=0}^{n-2}\overline{a_i b_{n-1}}2^i + 2^{n-1}\sum_{j=0}^{n-2}\overline{a_{n-1}b_j}2^j \\
-2^{2n-1} + 2^n
\end{aligned}
\tag{2.17}
$$

Figure 2.10a shows the circuitry of the discussed algorithm for 4-bit operands. The logic of the white and grey cells is according to Figure 2.10b and Figure 2.10c. The final result is calculated similar to the array multiplier by using RCAs.

Booth Multiplier

The fastest among the introduced multipliers is the Booth multiplier (Booth 1951). In it's initial version, it supports signed operands only. (Rajput & Swamy 2012) proposes an extension for handling also unsigned formats. Similar to the other multiplier architectures, partial products are generated and added in subsequent stages. The Booth multiplier differs from other multiplier architectures in the number of required partial products and the way of generating them. To reduce the number of partial product, radix encoding is deployed.

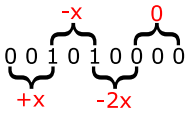


Figure 2.11.: Radix-4 encoding for an 8-bit multiplier

X_{i+1}	X_i	X_{i-1}	partial product
0	0	0	0
0	0	1	+x
0	1	0	+x
0	1	1	+2x
1	0	0	-2x
1	0	1	-x
1	1	0	-x
1	1	1	0

Table 2.1.: Radix-4 Booth encoding lookup

Radix encoding divides the multiplier into chunks of equal sizes (Figure 2.11). The chunk size is determined by $k = log_2 r + 1$, where r is the radix. For every possible value of a chunk, a rule for partial product generation is defined according to Table 2.1. Those rules add or subtract multiples of the multiplicand. The encoding can be adapted to support other formats, e.g. radix-2 or 8. The selection of the radix depends on the bit width of operands and influences the number of generated partial products. This influences the delay of the multiplier.

After obtaining the partial products, they are sign extended and added with one of the adders. To avoid a performance bottleneck, fast adders are deployed, e.g. a CSA tree. Before addition, the individual partial products are left-shifted. The shift amount is determined by the selected radix, e.g. 2-bit for radix-4. This compensates the radix encoding, depicted in Figure 2.12.

When supporting unsigned operands, additional 2-bits are added at the beginning of the operands (proposed by (Rajput & Swamy 2012)). They either sign- or zero extend the individual inputs depending on the chosen radix mode. The two additional bits cause wider partial products. Additionally, the number of partial product rows grows because of the widened multiplier.

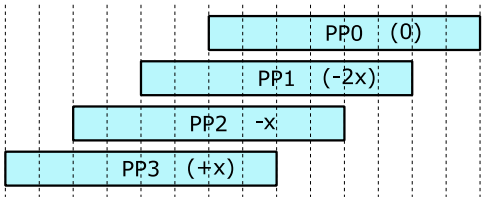


Figure 2.12.: Partial product alignment for Booth multiplication

Vedic Multiplier

The concept for this multiplier algorithm stems from Vedic mathematics. It follows an approach, which breaks down complex problems into smaller ones. Sixteen Sutras are available, which can be chosen to solve the multiplication operation. One of them is the Urdhva Tiryagbhyam Sutra, which (Chandrashekara & Rohith 2019) deploys for construction of an $8x8$-bit multiplier. The attained multiplier architecture is even faster than a Booth multiplier, which incorporates a Wallace Tree. Its disadvantage is the support of unsigned inputs only. This limits the deployment on 2's complement platforms.

The vertical and crosswise Urdhva Tiryagbhyam Sutra is used on the 2-bit operands $A = \{a_1, a_0\}$ and $B = \{b_1, b_0\}$, according to (2.18), (2.19) and (2.20). Those 2-bit building blocks are incorporated into 4-bit blocks, which further contribute to 8-bit blocks. This series can be continued to the desired width.

$$p_0 = a_0 * b_0 \tag{2.18}$$

$$p_1 = [a_1 * b_0] + [a_0 * b_1] \tag{2.19}$$

$$p_2 = [a_1 * b_1] + c_0 \tag{2.20}$$

Vector/SIMD Multiplication

Involved data formats vary across different applications. In order to efficiently support those applications with proper HW utilization, the computation unit has to support appropriate formats. When supporting multiple formats, the available resources can be shared. In the literature, different proposals for supporting various formats during multiplication are present.

(Belyaev et al. 2020) proposed a high performance multi-format SIMD multiplier, which combines multiple $8x8$ integer multiplier arrays and adder trees. For floating point format support, dedicated floating point product generators are incorporated. Deploying those resources, the proposed unit is capable of treating 8-, 16-, 32- and 64-bit fixed-point data in addition to half-, single- and double-precision floating-point data in a SIMD mode. In comparison with state-of-the-art multipliers it requires smaller area at same performance.

A reconfigurable multi-precision Radix-4 Booth multiplier is presented in (Shun et al. 2007). It's basic building block is a $8x8$-bit multiplier. Multiples of the basic building block are combined to support variable input formats. $16x16$-bit multiplications use four of the basic building blocks.

(Zhang et al. 2009)s motivation for a configurable fixed-point multiplier is driven by DSP applications. The proposed architecture consists of four pipeline stages and covers multi-precision. Additionally, signed and unsigned integer multiplication as well as 8 and 16-bit dot product operations are covered.

A power-efficient $16x16$-bit configurable Booth multiplier is presented in the work of (Kuang & Wang 2010). For an energy efficient support of multiple precision formats, a novel dynamic-range detector dynamically detects the effective dynamic ranges of both input operands and selects the operand with smaller dynamic range as multiplier. Due to Booth encoding of the multiplier the probability of partial products becoming zero is optimized in that way. Additionally, ineffective ranges in the multiplier HW are disabled for power saving. Furthermore, it supports truncation of the output by sacrificing some precision. This reduces the power consumption further. Additional HW components are required to correctly apply the discussed techniques. This increases the overall complexity, nevertheless the energy savings are significant.

(Liu et al. 2014) proposed a flexible solution based on sticky Booth coding technique. It addresses the unscalability problem, which is particular to multi-precision Booth designs. It deploys a cell array based architecture.

A sub-word parallel multiplier and a low-error reduced width multiplier is presented in (Tsai et al. 2002). The sub-word multiplier is designed to handle a single nxn, two $n/2xn/2$ or two $n/2$ complex number multiplications alternately. The low-error multiplier calculates the multiplication result with the help of a compensation vector with high accuracy.

2.3.3. Dot Product Unit

Matrix-matrix and matrix-vector multiplications are frequent operation in DSP as well as AI-applications. The fundamental mathematical operation is the dot-product (inner product), which is shown in (2.21). The applied operations can be divided into multiplication and accumulation of the products. Since both operations appear in the same context and in subsequent order, a dedicated dot-product unit is realized on many platforms. This unit is a composition of a multiplier and an adder combined with an accumulate register.

$$dp = \begin{bmatrix} a_0 & a_1 & a_2 \end{bmatrix} * \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} = a_0x_0 + a_1x_1 + a_2x_2 \tag{2.21}$$

A reduced latency and low-cost inner product processor is presented in (Smith & Torng 1985). It comprises of a tree of carry propagate or carry save adders. This tree includes three innovations. The individual leaf-multipliers are expanded into adder subtrees, which allow to keep latency low. Moreover, the partial products are realigned according to their "minimum alignments", which saves hardware. Due to truncation of the carry propagation in the final stage, the latency reduces significantly.

An energy efficient truncated inner product unit is presented in (Yan & Ercegovac 2019). It settles on a pipelined architecture to processes the individual multiplication pairs in serial order. Thereby, a huge amount of HW is reused. Additionally, the inner product is truncated and the lower part of the result is skipped to save area and energy. The reduction tree is also pipelined, which improves throughput. For non-uniform inputs, a two-level [4:2] adder is proposed to accumulate redundant vectors.

(Kazakova et al. 2001) proposes a fast and low-power two's complement fixed-point inner product processor targeted for 3D volume rendering. Speed and power optimizations were achieved at both the architecture and circuit levels. The inner product operation, consisting of multiplications and additions, is merged into a composite function in order to minimize circuit area, improve speed and minimize power dissipation. Individual functional blocks including the partial product: encoder, reduction tree and final two operand adder were built entirely using complementary pass transistor logic (CPL) with reduced swing internal nodes to minimize the energy required during switching.

2.3.4. Vector/SIMD Dot Product Unit

For low bitwidth inputs the sequential computation of the dot product can be accelerated by performing simultaneous multiplication and addition of multiple Matrix-Vector elements. In the literature, different architectures of such vector/SIMD dot product units are available.

(Prasanna & Prabhu 2019) proposed a four-term fused dot product unit, which supports single precision floating-point using Vedic multiplication. This architecture addresses low power and area efficiency. The unit comprises of a merged floating-point dot product unit, which deploys a Vedic multiplier along with other correction logic. An additional performance enhancement is driven by compound addition and rounding.

(Lin et al. 1999) proposes a parallel inner product processor architecture, which is designed for simple reconfiguration of inner product computations with variable input widths. The possible inputs are 64-bit wide with either 8-, 16-, 32- or 64-bit items in unsigned or 2's complement format. The processor is pipelined and comes with simple and reconfigurable components, while preserving compact area. The building blocks are small $8x8$ or $4x4$ multipliers combined with multiple adders arrays. This structure is highly regular, modular, symmetric and repeatable.

(Tan et al. 2003) proposes a fixed-point vector multiply-accumulator, which is capable of supporting multiple precisions by deploying a shared segmentation technique. It performs a single $64x64$, two $32x32$, four $16x16$ or eight $8x8$-bit multiply-accumulation operation in both signed and unsigned formats. The hardware of a scalar 64-bit MAC is reused at the cost of a small latency increase. The vectorization is done by format-dependent multiplexing during partial product generation and carry suppression in the reduction tree and final adder.

Quan et al. present a 32-bit multi-precision vector dot-product unit (Quan et al. 2010). It supports 32-, 16- and 8-bit SIMD formats. Additionally, Booth encoding and Wallace tree

compressing is deployed. An $8x8$ Booth unit operates as building block for a reconfigurable Booth encoding array. This enables the encoder to support wider input operands by selectively combining the blocks. It's further capable of multiplying scalar with vector operands. In (Danysh & Tan 2005), this architecture is compared against the "shared subtree" method.

In (Li et al. 2021), a multi-precision floating point many-term dot-product unit is designed for SIMD support. Multi-precision support is ensured, via configurable multiplier and alignment shifter logic. Complex terms make use of two successive clock cycles to maintain propagation delay. This also avoids idle multiplication resources. The summation is speed up with an optimized CLA.

In this thesis, a vector dot-product unit is utilized to compute the matrix-matrix/vector operations involved in the NN-inference. This unit needs to provide support for either 8 and 16-bit vectors or 9 and 18-bit vectors (needed for TF-lite deployment). It is capable of serving four multiplications and five additions at a time for 8, 9-bit input vectors and two multiplications and three additions in 16, 18-bit mode, while reusing the same hardware. Since the combinatorial path of a single cycles dot-product unit is long –data has to propagate through the multiplier and multiple additions need to be applied– fast architectures for multiplier and adders are needed, which still provide the possibility to operate on multiple vector modes. To realize a multiplier with such specifications, the Booth-Multiplier provides a proper basis. For fast addition of the products CSAs suit well, since they allow parallel operation and vectorization. The realization of those components is presented in section 3.2.

2.3.5. Linear Interpolation

Nonlinear functions are used in various applications. SW approximation of a function requires multiple cycles, depending on the function complexity. It can be meaningfully applied in applications, which are not sensitive to the execution time or have a weak dependency on this operation. Alternatively, a dedicated HW module is required, which extends the platform and solves the approximation with a reasonable delay. Definition of such a module is challenging with respect to timing and cost. In context of this work, complex functions are required to map non-linear activation functions on the proposed NN-accelerator.

In the literature, different approaches for function approximation in HW are proposed. In the following, an overview of those solutions is provided.

Lookup Table Approximation

One method utilizes LUTs (Lookup Tables). The desired function is divided into uniform segments and a fixed number of interpolation points are used to represent it. A truncation of the x-value is considered as input to the LUT and the lookup values represent the y-values of the function. Exact matching is mandatory for a successful lookup. (Piazza et al. 1993)

implements the LUT as conventional ROM, with a limited number of entries to approximate the Tanh function. The timing performance of this approach is high, due to a short critical path. Nevertheless, with increasing accuracy the complexity grows exponentially (Namin et al. 2009). To address the bad scalability, the authors of (Muscedere et al. 2005) replace the LUT with a range addressable LUT (RALUT). Instead of exact matching, range matching is applied. (Namin et al. 2009) applies the RALUT approach onto the Tanh function. The advantage of RALUT is reserved to functions, which share the saturation property, e.g. Sigmoid and Tanh shown in (2.22).

$$Sigmoid(x) = \frac{1}{1+e^{-x}} \qquad Tanh(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}} \tag{2.22}$$

In this thesis a modified LUTs approach is followed as well. In contrast to the previous work, the bad scaling with increasing function-domain width is addressed.

Piecewise Linear Approximation

Another method proposed in (Lin et al. 2013) relies on PWL (Piecewise Linear) approximation. It represents complex functions by segmentation into uniform sections and approximation of those pieces with linear functions. By reducing the section width, the individual sections become nearly linear and the approximation error decreases. H. Amin et al. apply this method to a Sigmoid function in (Amin et al. 1997), and restrict the slope of the linear functions to powers of "2". This avoids the expensive multiplier logic used for slope multiplication in favor of a cheap shift operation.

This approach allows to cover wide function-domains without having the bad scaling behavior of the LUT based approaches presented in the previous section. Nevertheless the segment width is constant across the entire domain and provides low accuracy for segments with increased fluctuation. In contrast, the approach followed in this thesis allows dynamic width segments. Thus, the user can decide to invest more interpolation points and area in the region of interest, whereas low complexity is spent on segments with low impact on the application.

Piecewise Nonlinear Approximation

A more accurate approximation is achieved, when extending the PWL approach to also use polynomials of higher order (Zhang et al. 1996). Their formalized representation is shown in (2.23). This method comes with the disadvantage of causing higher multiplication density for polynomials of high order. An increase of on-chip area or computation time follows, when assuming computation in sequential order. I. Tsmots et al. applied PWNL (Piecewise Non Linear) approximation on the Sigmoid function in (Tsmots et al. 2019). In this thesis, a decision against deployment of this approach was taken, because of the increased computation complexity.

$$f(x) = \sum_{i=0}^{j} c_i x^i \tag{2.23}$$

CORDIC

CORDIC is an iterative method to approximate complex nonlinear functions (Volder 1959). Via coordinate transformation, a function is mapped from Cartesian into Polar coordinates. This method enables approximation of many functions with high accuracy, at the cost of multi-cycle evaluation (Duren et al. 2007). For the transformation of data into the polar domain of the function, dedicated HW is utilized, e.g. shift registers, adders, and subtractors. The technique is successfully applied to approximate Sigmoid and Tanh functions in (Raut et al. 2020). (Kumar 2019) mapped trigonometric functions to HW deploying the same technique.

This method is based on an iterative approach, which creates a throughput bottleneck in the overall computation pipeline. Therefore, in this thesis CORDIC is not applied.

2.4. NN-Accelerators

In the literature, different dimensioning of NN-accelerators can be found. Common platforms deploy FPGAs, since they allow reprogramming. This is especially useful for HW-SW-AI co-design, where the accelerator is designed and optimized to execute a specific NN-kernel. Other available structures deploy systolic arrays on ASICs. They enable a specialized form of parallel computation, especially useful for compute-intensive operations, e.g. matrix multiplication and convolution. The pipelined structure of locally interconnected PEs enables high clock frequencies and low external data transfers, due to massive data reuse. PEs are not only used in systolic arrays, but can be also deployed as independent units. Hereby, additional control logic ensures data provisioning to the PEs and collection of their results. Another category of accelerators are processor-coupled ones. They are either directly integrated into a processor core or equipped with a close and fast interface.

2.4.1. FPGA Mapped

(Yin et al. 2019) proposes an efficient mechanism to map NNs onto FPGAs. Other NN mapping techniques provide solutions for convolution layers, but suffer from bad resource utilization when treating different types of layers. Yin et al. addresses the described issue with an improvement on the utilized DSP blocks. Those are extended for multiple small bit-width operation. Additionally, complementary features of the particular layers (convolution-based layers, fully connected layers and recurrent layers) are identified and are exploit for spatial mapping.

Another design flow is presented in (Wang et al. 2018). It addresses the aspects NN-model training and FPGA deployment. The goals are to support rapid design space exploration and simplification of the trade-off between efficiency and accuracy under the restrictive resource and power limitation on edge devices. Therefore, the solution deploys extremely low bit-width NNs and hybrid quantization.

A deep learning accelerator unit (DLAU) for large-scale deep learning networks is proposed by (Wang et al. 2017). It aims for performance improvements, while keeping power consumption low. The architecture ensures scalability and maintains throughput via a pipelined three staged structure. Tile techniques are utilized to explore design space for given applications.

(Wiedemann et al. 2021) presents a software-hardware optimization paradigm in combination with the execution engine FantasIC4. The approach is designed to primarily accelerate the execution of FC-layers. Compression techniques and robust quantization maintain on-chip area and power consumption. A novel training method based on entropy-constraining is proposed to achieve high compression at proper model accuracy. Additionally, the framework maintains the HW complexity by limiting the number of required multipliers to four.

A deep parallel CNN accelerator is proposed by (Duan et al. 2018). It deploys a binary weight method, which allows to perform the entire convolution without any multiplication. The data is stored into two RAM banks, which are composed out of multiple RAM blocks, according to the number of parallel processing units. The accelerator supports variable-size convolution filters.

Caffeine is a hardware/software co-designed library (Zhang et al. 2019), which facilitates an efficient acceleration of CNNs and FC networks on FPGAs. An automation flow is developed, which compiles the abstract NN-model descriptions to an FPGA accelerator HW. The derivation process of the accelerator microarchitecture deploys a revised roofline model to balance the platforms compute and bandwidth resources.

(Lian et al. 2019) proposes an accelerator, which deploys an optimized BFP (Block-Floating-Point) arithmetic. It's capable of processing 8-bit model data with optimizations for low precision in rounding and shifting operations. The feature maps are encoded in 16-bit format. This reduces the off-chip memory traffic, in favor of throughput and energy efficiency compared to the native 32-bit format. The proposed reconfigurable accelerator comprises three parallelism dimensions. Additionally, DDR3 memory accesses are done in ping-pong manner and on-chip buffers are optimized.

An accelerator for E-LSTMs (Embedded Long Short-Term Memorys) is presented in (Shi et al. 2019). Hereby, a LSTM (Long Short-Term Memory) co-processor is coupled to an embedded RISC-V processor. A special format to represent the sparse weight matrix, named eSELL is developed.

(Shuo et al. 2018) developed a structured compression technique for LSTM models, which supports elimination of irregularities in compute and data fetch patterns and reduces the model size. The weight matrices are encoded using block-circulant instead of sparse matrices.

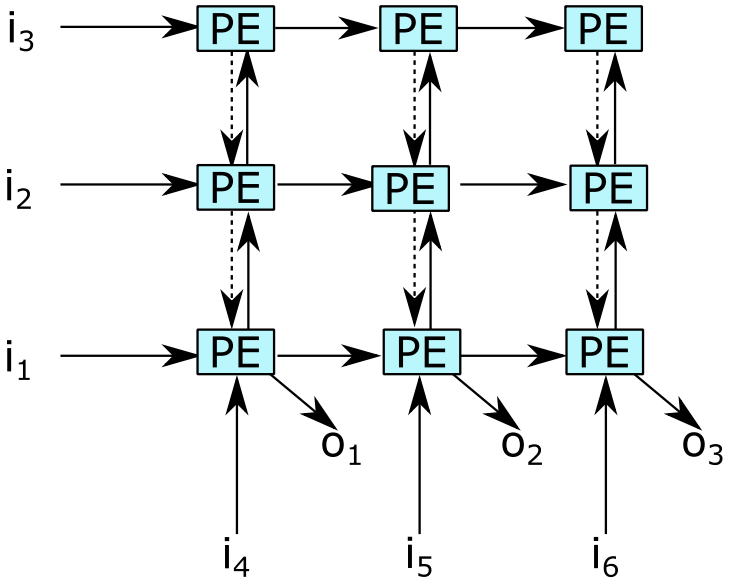


Figure 2.13.: Systolic array (Aggarwal 2010)

Additionally, the deployment of FFT (Fast Fourier Transformation) further accelerates the inference. Furthermore a framework called C-LSTM is proposed, which automatically optimizes and implements variable parameterized LSTMs on FPGAs.

2.4.2. Systolic Array

In (Xu et al. 2020), systolic arrays (Figure 2.13) implemented on ASICs are optimized for special convolutions, e.g. small-scale or depthwise. In such applications, the array utilization varies with high volatility. This issue is tackled with a configurable multi-directional systolic array (CMSA). Hereby, data transmission directions and array configuration are adapted to the individual convolution. This deploys the enhance flexibility, while keeping the original structure of the systolic array.

Googles TPU is a special purpose ASIC coprocessor designed to outperform state-of-the-art CPUs and GPUs in NN-related computations (Jouppi et al. 2017). The heart of the accelerator is a wide MAC matrix multiply systolic array comprising of $256x256$ 8-bit integer units, reusable for 16-bit formats. A software managed 24 MiB unified on-chip buffer is available to cache memory data (loaded via DMA (Direct Memory Access)) or the matrix multiply results. Due to the regularity of NN-related matrix operations, control is a minor concern. Instead, series of complex instructions, which determine the operations of the TPU are provided via PCI (Peripheral Component Interconnect) interface by the host processor and are cached in an instruction buffer. Measured by it's compute performance, the TPU is relatively small and power efficient.

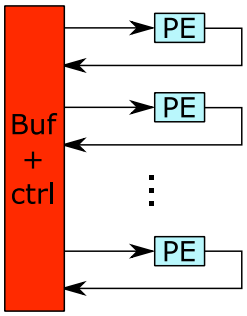


Figure 2.14.: PE-based accelerator structure

ConvAU is another accelerator, which settles on a systolic array for efficient acceleration of dense matrix multiplications in CNNs (Kiningham et al. 2016). It's inspired by Googles TPU and comprises of a pipelined architecture. The stages are data load, matrix multiplication, activation function handling and output queuing. It's capable of performing $256x256$ 8-bit integer multiplications and 32-bit integer accumulates per cycle. ConvAU improves the performance by a factor of 1.9× compared to the TPU.

(Adiono et al. 2018) proposes a method to map a neural network model as a three-layer perceptron in forward mode to a systolic array. The operations within the NN-model phases are described via a dependency graph. A recursive iterative algorithm is utilized to map it to a systolic array.

The Deep Learning Accelerator NVDLA is proposed by NVIDIA (NVIDIA 2022). It follows an highly configurable and scalable approach enabled by a modular design. The developers claim to have a simple, flexible and robust inference acceleration solution that is capable of handling a wide range of applications across multiple performance levels by sophisticated scaling capabilities. The available modules are "Convolution Core", "Single Data Processor", "Planar Data Processor", "Channel Data Processor" and "Dedicated Memory and Data Reshape Engines". For every individual accelerator, the existence and quantity of those modules are selected depending on the application. The execution of high-intensity operations can be supported in an optimized way. Additionally, NVIDIA provides an automated flow, which covers the entire deployment flow starting with training until the accelerator mapping.

2.4.3. PE-based

Figure 2.14 visualizes the PE-based approach. A configurable, versatile and flexible architecture for HW acceleration of CNNs is presented in (Christensen et al. 2019). It is designed to execute any type of CNN, while achieving high energy efficiency. Therefore, it stores and caches entire feature maps in a memory local to the accelerator.

(Sankaradas et al. 2009) presents a massively parallel coprocessor optimized for CNNs. Parallel 2D convolution primitives and programmable units are in place to perform sub-sampling and

(a) Integrate NN-accelerator

(b) Tightly coupled NN-accelerator

Figure 2.15.: Processor-coupled accelerator structures

nonlinear functions. Distributed off-chip memory banks with large bandwidth provide the necessary data to the coprocessor. Low precision data and packed word memory accesses further increase bandwidth. The CNNs mapping to the coprocessor is realized via instructions for memory data transfers.

An accelerator named Cambricon-X is presented in (Zhang et al. 2016). It aims for exploitation of sparsity and irregularity properties of NN-models to increase efficiency. The proposal for the accelerator architecture comprises multiple PEs and an Indexing Module (IM). The first enables local and asynchronous computation by storing the irregular and compressed synapses. The second ensures reduced memory congestion by independent and optimized neuron provisioning.

Eyeriss (Chen et al. 2017) optimizes the entire system for energy efficiency and provides reconfigurability to support various NN-shapes. To simultaneously reduce energy consumption and achieve high throughput, the data movements are reduced. This is ensured by the proposed row-stationary processing flow. The enhanced version of Eyeriss is Eyeriss v2 (Chen et al. 2019). It's optimized to efficiently execute compact and sparse DNNs. Thereby, the on-chip network "Hierarchical Mesh" introduces the required flexibility, which is needed to deal with varying NN-shapes. It also ensures proper adaptation to different degrees of data reuse and data format dependent fetch intensities. Furthermore, it's capable of performing the NN-inference in compressed format in both dimensions (weights and activations), which maintains speed and efficiency on sparse models.

(Bernardo et al. 2020) proposes UltraTrail, a configurable, ultralow-power TC-ResNet AI accelerator. It follows a strict hardware-model co-design approach. The derived accelerator architecture is optimized for generalized TC-ResNet topologies. It comprises of a configurable array of PEs and a distributed memory. Furthermore, conditional computing reduces the inference complexity.

2.4.4. uC-coupled

Figure 2.15 visualizes the uC-coupled approach. In the work of Zhang et.al. (Zhang et al. 2020) an open source RISC-V processor (Hummingbird E203) is expanded with a coprocessor, which is designed to execute a major portion of the CNN computation. The software ensures the completeness of the mapping for a particular kernel.

(Li et al. 2018) presents a RISC-V-based SoC for accelerating CNNs on FPGAs. At the proposed platform a dedicated convolution array logic is available, meant for acceleration of standard floating-point multiply-accumulate operations. Additionally, arrays for activation function computation are available.

A near-threshold open-source RISC-V processor core designed for tightly coupled multicore clusters is proposed in (Gautschi et al. 2017). The authors implement ISA extensions and microarchitectural optimizations for the stake of an increased computational density and relaxed memory congestion.

The work of Garofalo et.al (Garofalo et al. 2020) presents a set of RISC-V ISA extensions, to enhance the efficiency of low-bitwidth operations especially used in QNNs. The microarchitecture of the low-power microcontroller core supports the proposed extensions via energy-efficient DSP modules.

A multi-precision arithmetic unit integrated into a RISC-V processor is presented in (Garofalo et al. 2021). The integration covers micro-architecture as well as ISA extension. It's focus is on efficient acceleration of QNN inferences at the uC performance category (see section 1.4.1). Therefore, the ISA is extended with SIMD instructions, capable of handling 2 and 4-bit data formats. Additionally, a custom SIMD sum-of-dot-product paradigm is introduced, which unifies the dot product and load into a common operation. A multi core setup comprising of eight such cores shows near-linear performance improvement compared to a single core architecture.

2.5. AI Compile Flow

The mapping of trained NN-models onto individual execution platforms is challenging. General purpose CPUs and GPUs are equipped with standardized interfaces. Optimized accelerator platforms for NN-operations do not follow a common interfacing, but should be supported as well to reduce development time. Moreover, any effort in searching a definition of a common interfacing cannot cover the individual features and peculiarities of the different accelerators. A simplified automated flow for generating efficient kernels is needed, which is compatible with various NN-models and platforms. The generalization of NN-model training and inference computations are tensor operations. Multiple state-of-the-art deployment tools are available, which tackle those challenges with different mechanisms.

2.5.1. TACO

The Tensor Algebra Compiler (TACO) is proposed in (Kjolstad et al. 2017). The developers aim to support scientific computing, machine learning, data analysis and performance engineering. TACO is a library capable of generating optimized kernels, which execute user defined tensor computations. Therefore, it deploys a compiler-based transformation technique. For achieving

high performance, one of the exploited features is tensor sparsity. The kernels are generated accordingly and proper storage formats are chosen from a collection of supported formats. The definition of even complex tensor algebra computations is simplified by the tensor index notation.

2.5.2. TVM

TVM is a end-to-end optimization stack, which is designed to provide performance portability of deep learning model execution across various HW-platforms (Chen et al. 2018). Other tensor compiler frameworks provide good performance on generalized platforms (CPUs and GPUs), but require major effort when porting for specialized accelerator HW (FPGA or ASIC). TVM exposes optimizations on graph- and operator-level, to resolve the complexity of portability. TVM addresses a bunch of issues, which limit performance portability. It ensures proper operator fusion at high level, memory reuse and latency management. Further, it guarantees appropriate mapping of operators onto the available hardware primitives.

2.5.3. TensorFlow

TF (Abadi et al. 2016) is a framework, which provides support for tensor computations along with a user-friendly API. Due to it's generous, high flexible and scalable skeleton it supports a wide variety of algorithms on various heterogeneous platforms. This characteristic allows to efficiently apply TF to ML on a wide range of platform performance classes with low portability efforts. Embedded edge processing as well as HPCs are covered. With TF, both training and inference of NN-models obtain appropriate support on any platform. Individual platform-specific kernels, which map to concrete tensor operations, can be easily plugged in.

TF is a well documented and actively driven project. The community of TF is a big and experienced group of developers, which are capable of providing proper support and ensure sustainability. The code quality is good and the degree of supported platforms and features is high. Thus, the compile flow chosen within this thesis is TF.

3. Accelerator Hardware

3.1. Accelerator Architecture

This thesis strives a tightly-coupled accelerator solution, where the back-bone is constituted by a RISC-V processor. On one hand, the RISC-V with its base instruction set serves as a baseline in order to compare bandwidth and computation limitations against the extended or accelerated solution. On the other hand, it represents a fallback solution for corner cases, which aren't treated by the accelerator. Due to RISC-V's open instruction set architecture, the integration of additional custom extensions is simple and goes hand in hand with high configurability. The accelerator is targeting low power and low cost applications, which are either too heavy to fulfill the latency requirements on a micro controller or are not energy efficient enough. This especially plays a role when it comes to advanced data formats.

Sze et al. rates a read access from DRAM more than 100x more expensive than from SRAM (Sze et al. 2020). For the targeted low power accelerator, this observation infers strict avoidance of off-chip memory accesses. Reducing the model size enables the storage of model data on-chip. The discussed extensions are developed at the background of supporting reduced model footprints. In addition, reducing the model size to an extent of manageable loss in precision leads to several other desirable effects. A smaller memory footprint reduces the static memory power consumption and chip area. Since the CPU core and its extensions are supposed to be small in terms of area, memory is the major driver for chip area. In addition, a reduced model footprint leads to a decreased number of required accesses, which facilitates a lower dynamic memory power consumption.

Since different kernels constitute various operational intensities, provisioning optimized hardware support for a certain application is challenging. This issue is addressed by a highly extensible and configurable system, which offers the freedom to optimize the SoC for a particular kernel. This is achievable by interchanging extension modules and moving computation and communication bounds closer to the theoretical maximum.

3.1.1. Approach

The accelerator can be deployed in different ways, either tightly CPU-coupled, as standalone NPU (Network Processing Unit) or loosely CPU-coupled. The basic building blocks are shared between those structures. For the standalone and loosely CPU-coupled setup further controlling components are required (e.g. FSM (Finite State Machine) or synchronization), which take over setup and reconfiguration tasks. In the tightly coupled structure, this is covered by

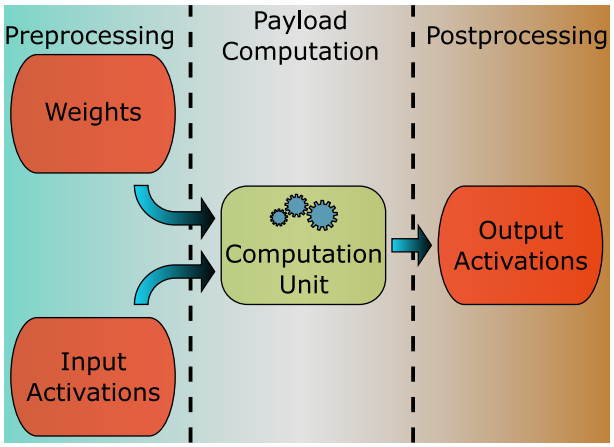


Figure 3.1.: The basic building blocks of the NN-accelerator

the CPU. Depending on the supported layer types, the complexity of reconfiguration varies. This affects the SW executed on the CPU or the FSM states of the controlling unit. In SW the routines are part of the kernel implementation. The following section provides an overview of the basic building blocks common in all three structures. The approach further elaborated in this thesis uses the tightly CPU-coupled setup, since it provides a maximum of flexibility.

3.1.2. Overview

Basic Building Blocks: Figure 3.1 shows the four main building blocks of the accelerator, which are deployed in all of the following setups. The functionality of each block stays untouched and independent of the structure to which they are integrated. The blocks in Figure 3.1 are separated into three stages. The Preprocessing stage contains the components dedicated to fetching and preparing weights and input activations. They are discussed in section 3.1.3 and 3.1.4 respectively. The Payload Computation stage comprises of a computation unit, which executes the fundamental dot-product operation. It gets further elaborated in section 3.2. The last stage is the Postprocessing. It takes the output data of the computation unit as an input and optionally does some further processing, e.g. quantization and activation function. Finally it writes the computed output activations back to the memory. This component is discussed in section 3.1.5.

Tightly CPU-coupled Accelerator: The setup of a tightly CPU-coupled accelerator is shown in Figure 3.2. It consists of the five individual stages of a RISC-V CPU introduced in section 2.2, CSRs (Control Status Registers), buses and memories. The accelerator shares the same resources with the *Instruction Fetch* and the *Memory Access* stage of the CPU. The weights of the NN are located in the *Program Memory*, whereas the *Data Memory* contains the according activations. Weights are stored in the read-only memory, since they are constants in NN inference. The distribution of the payload into two memories allows simultaneous access and maximizes throughput.

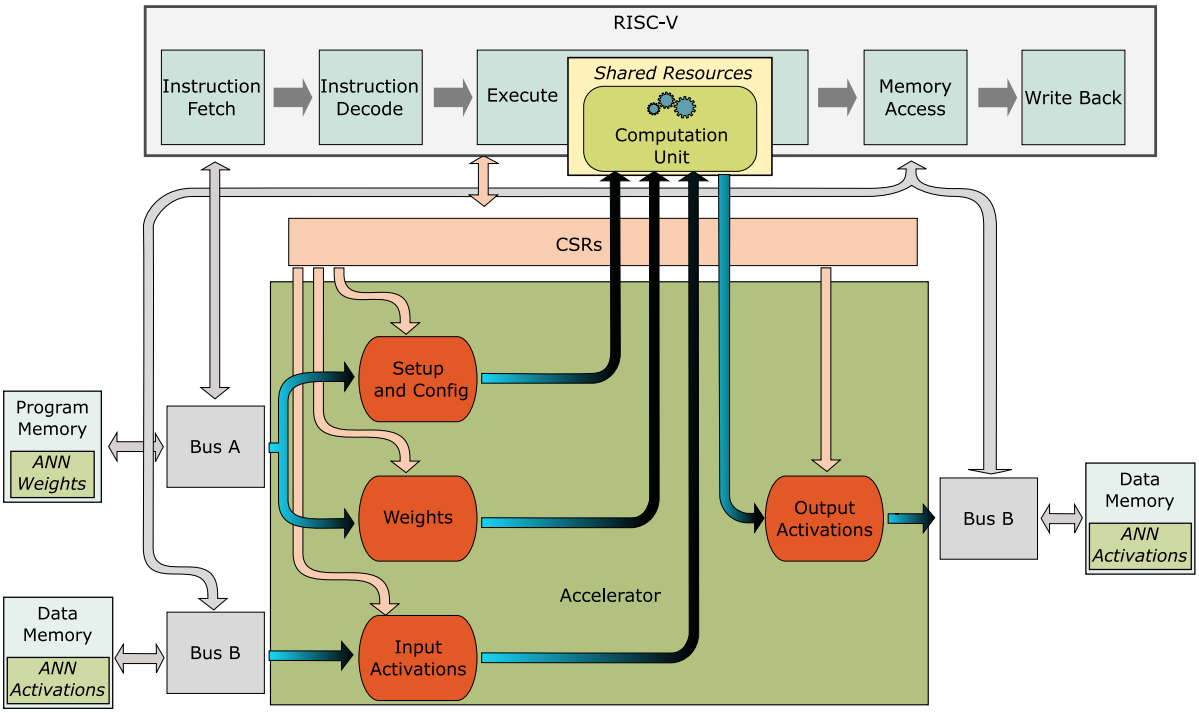


Figure 3.2.: Tightly coupled accelerator extended RISC-V CPU

The configuration of the accelerator is done via CSRs, which are accessed by the instructions of the standard zicsr-extension of RISC-V. Configuration of the HW via CSRs is superior to MMIO (Memory Mapped Input/Output) based one in terms of speed and energy, since databus utilization is lowered.

The block diagram of Figure 3.1 summarizes the pre- and postprocessing components within the accelerator component. The *Shared Resources* block attached to the *Execute* stage of the RISC-V pipeline comprises the *Computation Unit*. For every operation performed by the *Computation Unit* an instruction has to propagate through the CPU pipeline. Thus, the CPU is aware of the exact status of the inference process and is able to intervene at specific locations for special handling routines implemented in SW.

Standalone Accelerator: Figure 3.3 shows the accelerator structure in standalone mode. This structure is the most independent solution. In addition to the previously introduced common components of Figure 3.1, this setup requires further dedicated components. One of them is a *Control FSM*, which takes over the micro control of the inference loops. In the tightly CPU-coupled setup the SW executed on the CPU covers that part. The *Control FSM* enables the accelerator to execute a complete model inference without SW interaction, thus further offloads the CPU. Further, it increases the accelerators availability for parallel task processing. Additionally, an explicit synchronization of the building blocks becomes mandatory, which the CPU's pipeline resource management covered implicitly. For every operation executed on the accelerator at any cycle, it ensures that the corresponding resources deliver proper data. In

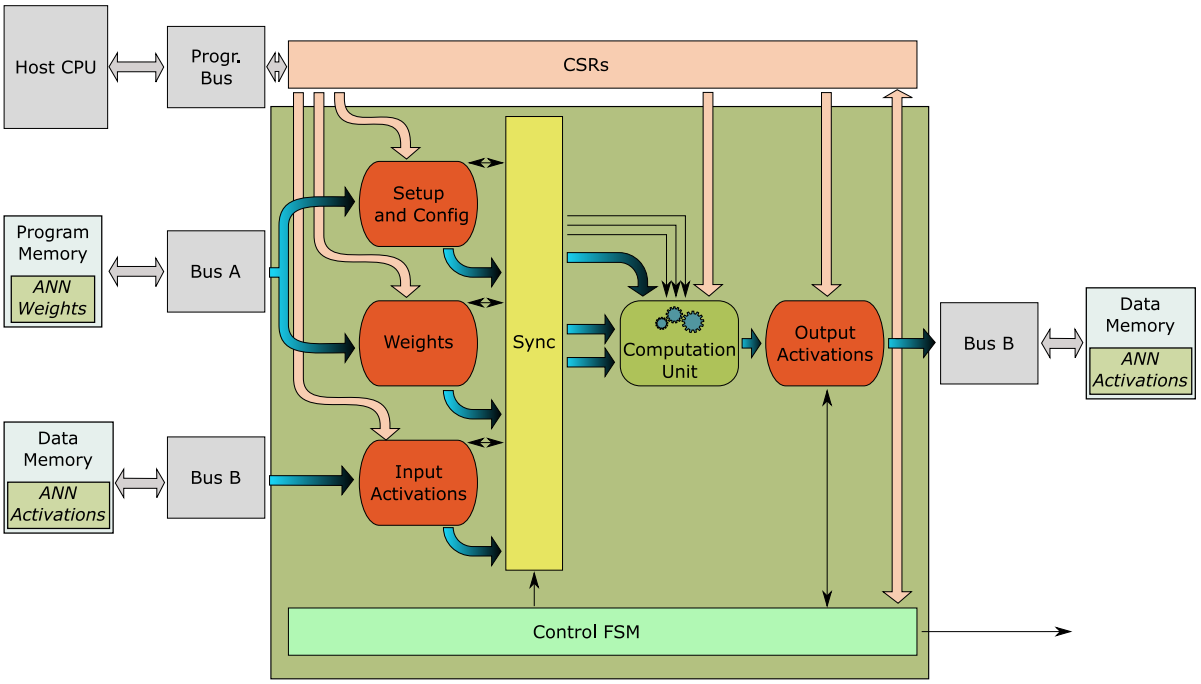


Figure 3.3.: Standalone accelerator with Host CPU

case any of the involved resources is not ready to provide data, the operation is delayed until the condition is fulfilled. The synchronization logic is seperated from the FSM to keep the number of states low. It also ensures a lean and expressive description of the layer dependent operations. For the common components any conceivable datatype is supported, except in the computation unit. The only restriction remaining is about the supported bitwidths. In order to support quantized data in integer format a bias added to the dot-product is compulsive (discussed in 1.3.1). For the tightly CPU-coupled accelerator, SW covers that task. For the standalone solution that way is prohibited. Thus, a helper streamer "Setup and Config" is included. It takes over the bias provisioning and other unspecified low activity memory data fetches.

In contrast to the tightly CPU-coupled accelerator, the CSRs aren't configured via CSR instructions. Instead, the configuration is done via MMIO. The configuration can be either written by a host CPU or DMA. A DMA is especially useful, when many configuration parameters need to be updated, e.g. for initial setup or layer switching. Programming via host CPU is beneficial for minor changes with complex access patterns or when lacking a proper DMA on the attached bus.

The advantage of this setup is an increased independence of the accelerator, which allows the host CPU to do parallel processing of other tasks during active inference phases. In order to properly evaluate the available CPU performance, the bus architecture has to be considered. Since the accelerator causes intensive bus traffic, the available access slots for the host CPU are affected.

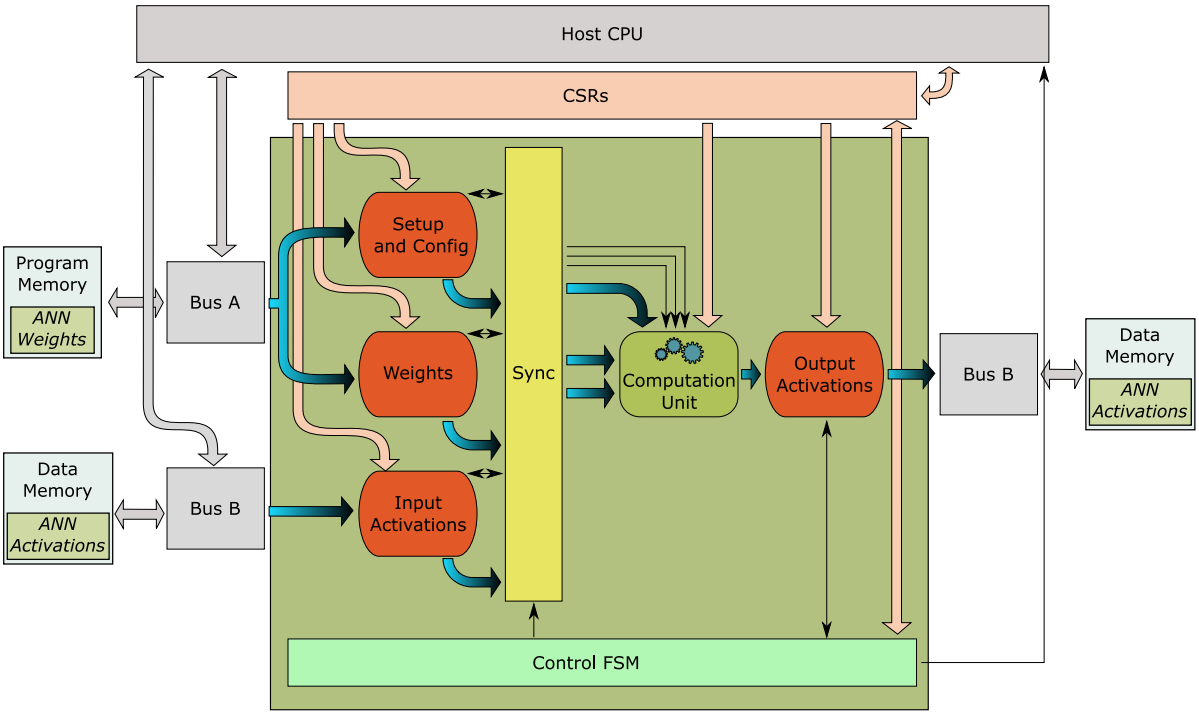


Figure 3.4.: Loosely coupled standalone accelerator with Host CPU

Loosely CPU-coupled Accelerator: The block diagram in Figure 3.4 shows the accelerator integrated in its loosely coupled form. This is an intermediate form between the tightly coupled and standalone version. Here, the accelerator is capable of an independent per-layer execution with its dedicated *Computation Unit* similar to the standalone solution. Differently, the configuration is done via direct CSR write through RISC-V's zicsr instructions. Buses and memories are shared with the CPU. This setup allows fast accelerator reconfiguration and supports per-layer execution, but is restricted due to bus congestion.

3.1.3. Weight Streamer

The component responsible for delivering kernel weights of the NN in proper ready-to-process format is the Weight Streamer. Its block diagram, with the main subcomponents is shown in Figure 3.5. The purpose of this component is an automatic parallel fetch of weight data from memory according to a desired pattern configurable via CSRs. Additionally, dropping or insertion of datasets at distinct locations for alignment and padding reasons e.g. for non multiple of byte weights is done. This module is built in a way to allow hooking of further compression methods, e.g. Golomb Rice encoded weights. To match the required SIMD processing format of the *Computation Unit*, data repacking is supported. In the following, the Weight Streamer subcomponents are described.

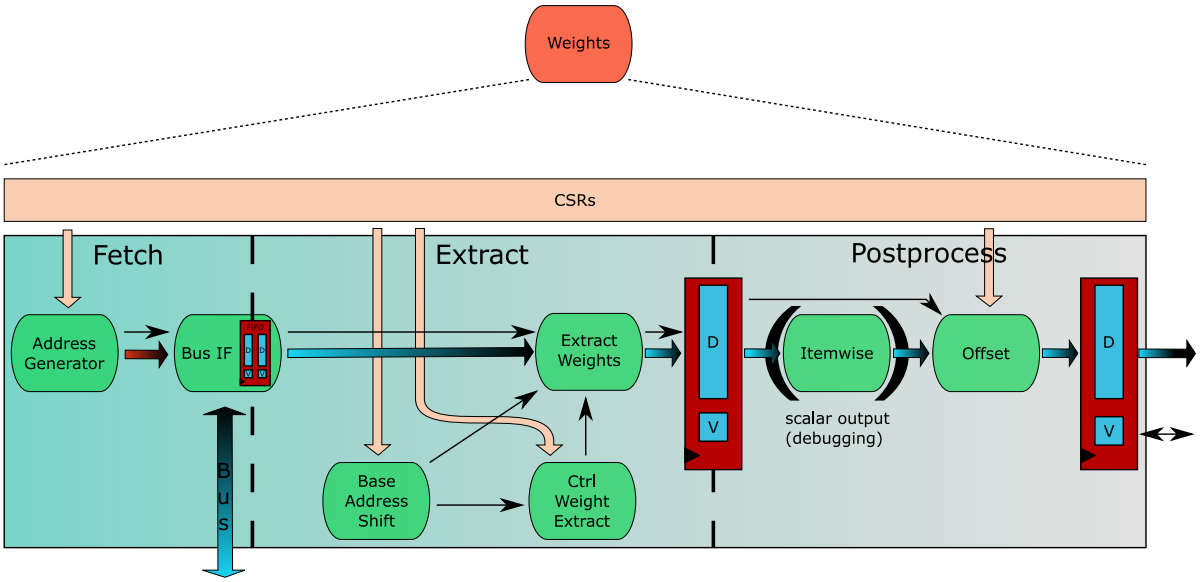


Figure 3.5.: Block diagram of the Weight Streamer

Fetch

The kernels of FC and Conv layers are multi-dimensional tensors. As presented in section 1.3.1, the tensors are flattened into 1D-arrays when storing them to memory (refer to 1.3.3). The applied mapping is according to the HWC format e.g. provided by TF FlatBuffer. Since all weights belonging to the same kernel are grouped together and the weights have to be fetched in the same order as stored in the 1D-array (see Figure 3.6), address generation is rather simple.

The address generator component is capable of handling kernel weights, where the initial weight of every kernel is aligned to byte, halfword or word boundaries. For varying bitwidth, some bits of the last word may be unused, as shown in Figure 3.7a. This is necessary for the alignment of the following kernel to byte boundaries. In case individual kernel addressing is not required, e.g. because of continuation over multiple kernel with same properties, the alignment bits can be dropped, see Figure 3.7b.

Figure 3.8 shows a detailed example of the memory mapping for 8-bit weights. The initial weight of the kernel is located at a memory cell with address 0xb, followed by additional seven weights. In order to fetch all corresponding weights for this alignment, three subsequent word addresses (i.e. 0x8, 0xc, 0x10) have to be generated by the address generator.

Equations (3.1) to (3.3) describe the computation of intermediate information produced by the address generator and the corresponding fetch addresses for certain configurations. w_{fw} describes the number of weights contained in the initial word. Its value depends on the bitwidth (bw) of the weights and the kernel alignment within the initial word, which is identified by the activation base address (a_{base}) (3.1). w_r is the number of weights remaining in the following

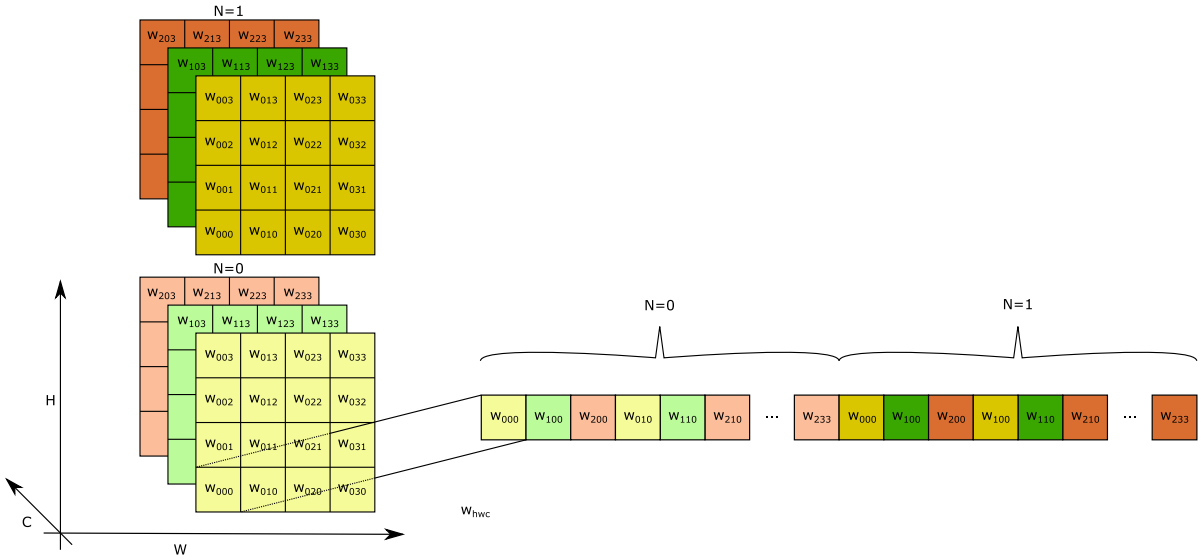


Figure 3.6.: Flatten kernel tensor into 1D array

(a) Memory-mapping/alignment of multiple kernels

(b) Unaligned Memory-mapping/alignment of multiple kernels

Figure 3.7.: Kernel alignments

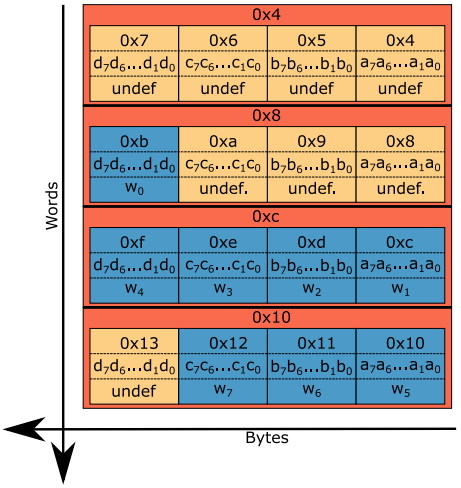


Figure 3.8.: Example for mapping of byte aligned 8-bit weights to memory

addresses. Its value is the difference between the total number of kernel weights w_f and w_{fw} (3.2). The number of words to fetch in order to receive w_f number of weights is calculated using bw (3.3).

$$w_{fw} = (4 - a_{base} \bmod 4) * \frac{8}{bw} \tag{3.1}$$

$$w_r = w_f - w_{fw} \tag{3.2}$$

$$wd_f = \lceil w_r * \frac{bw}{32} \rceil + 1 \tag{3.3}$$

where:

a_{base} : Base address (address of the first byte in memory containing referred kernel weights)
w_f : Amount of kernel weights to be fetched from memory
w_{fw} : Number of weights contained in the initial word
w_r : Number of weights contained in the words following the initial one
wd_f : Total number of words to fetch for receiving the required amount of weights
bw : Bit width of weights

Equation (3.4) describes the dependency of byte addresses and their according words. This information is necessary, since the initial kernel weights are byte addressed but the fetch is word based. With the help of (3.4), the address of the word assigned to the initial weight can be identified.

$$a_{word} = \lfloor a_{byte}/4 \rfloor * 4 \tag{3.4}$$

where:

a_{word} : Address of a word a certain a_{byte} belongs to
a_{byte} : Address of a byte

Equation (3.5) provides the relation between the i^{th} fetch element and the according address $a_f(i)$, dependent on a_{base}.

$$a_f(i) = (\lfloor a_{base}/4 \rfloor + i) * 4 \tag{3.5}$$

where:

i : $[0, wd_f]$
$a_f(i)$: Word address to be fetched, dependent on the index i of the word fetch

Extract

Kernel weights are loaded in Fetch stage as described in the previous paragraph. Word-only accesses are performed, since the generation of byte addresses increases the hardware complexity and doesn't affect throughput and performance. The initial weights of a kernel are byte instead of word aligned for better memory utilization. The *Extract Weights* component drops data, located outside of the actual kernel and accidentally fetched by aforementioned word accesses. It composes a new chunk containing only corresponding weights. Additionally, a special treatment for the end of kernel alignment is done by the *Ctrl Weight Extract* component. The *Base Address Shift* ensures proper continuation, after interruption through end of kernel.

Figure 3.9 shows the activity diagram of the weight extraction in standard notation, considering different alignments as well as continue and ending-scenarios. The number of weights present in the last chunk depends on the configured number of kernel weights. The remaining bits of the chunk have to be set to zero in order to avoid corruption of the dot-product computation. *Ctrl Weight Extract* computes a decision bit, which determines if the last element has been reached. Additionally, a mask is generated which identifies the usage of the individual weights. In continued mode, the actual base address of the kernel changes. For the stake of fast reconfiguration, the base address CSR doesn't need an update. Instead, for odd number of weights per kernel the *Extract Weights* component receives an updated/modified base address from *Base Address Shift*, which ensures proper decoding of the continued kernels.

Postprocess

The postprocessing stage consists out of two main subblocks. One is the *Itemwise* component, which is only present in scalar output mode. The other is the *Offset* component required for adding the factorized offset per weight. The reason for separating the offset and dispatching it to the Weight Streamer is discussed in the theoretical background of the dot-product computation and quantization in section 1.3.1.

Itemwise In some scenarios, where throughput and performance is regardless, e.g. Debugging, its beneficial to switch the streamer into the mode of scalar output. This allows a direct comparison of weights sent to the MAC against the ones expected. An error prone SW conversion from SIMD to scalar is obsolete. The *Itemwise* component is a PISO (Parallel In Serial Out) shift register, as shown in Figure 3.10.

Offset The Offset component can have different shapes. For the scalar Weight Streamer output the offset is added to one weight at a time. One adder is needed for the offset in that mode, see Figure 3.11a. For compatibility reasons, the output gets extended to 32-bit including consideration of the sign. For vectorized mode multiple parallel adders are required (Figure 3.11b), e.g. four offsets are added simultaneously in the 4× SIMD format. Different

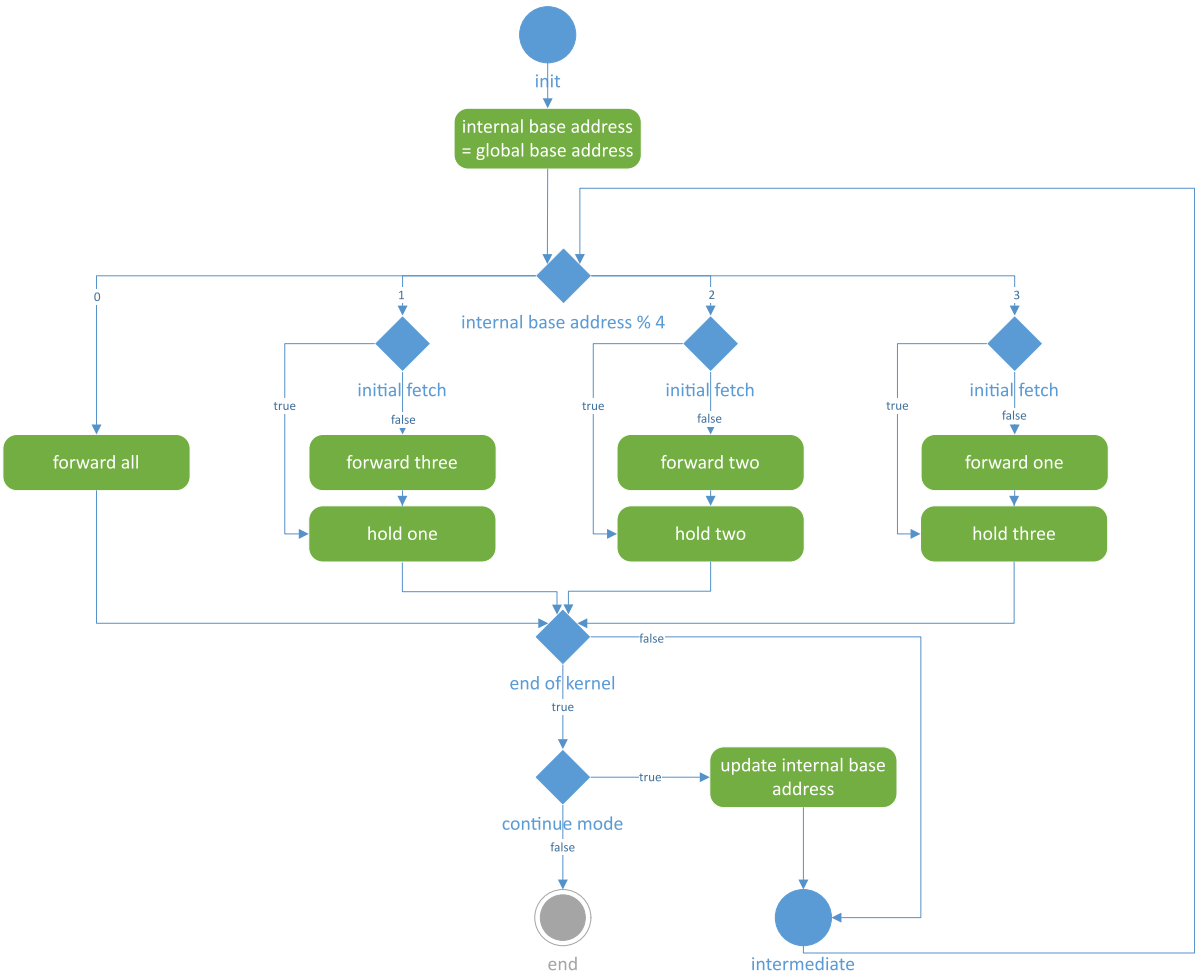


Figure 3.9.: Activity diagram for un/repacking of weights

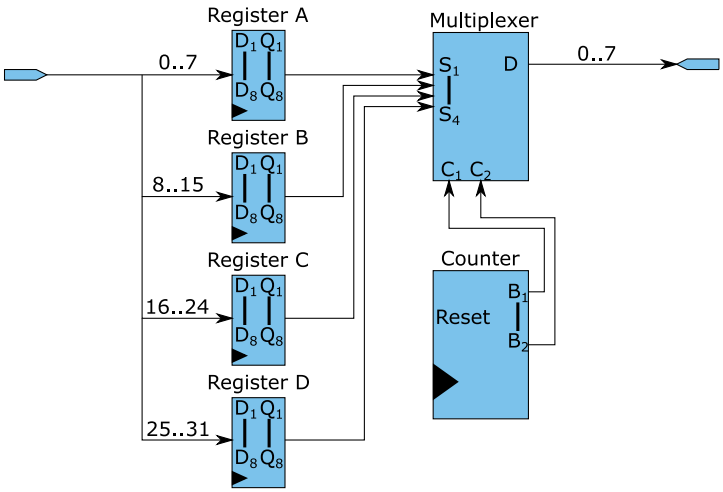
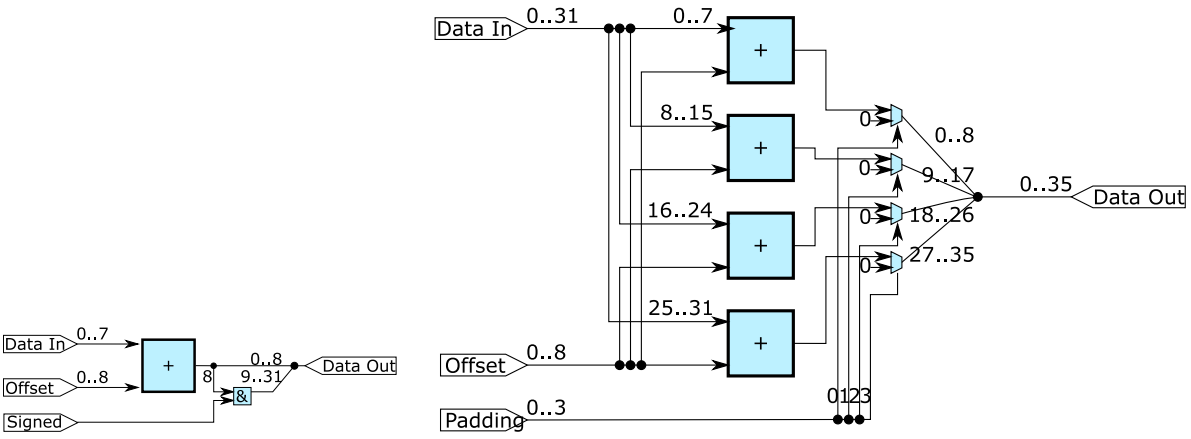


Figure 3.10.: itemwise weight output for 8-bit weights in scalar mode



(a) scalar Offset

(b) vectorized Offset 4x

Figure 3.11.: Offset weights

configurations with different bit-widths and SIMD formats are supported, since the discussed offset logic gets auto-generated from a proper specification. Analogous to the discussion of section 1.3.1, 9-bit data needs to be processed in the dot-product for a 8-bit weight format. Therefore, the input data gets extended from 8-bit to 9-bit by the addition of the offset. This overall results in a 36-bit wide streamer output.

For SIMD mode, an additional feature is needed to avoid non-zero padding weights. Weights meant for padding, which are located outside the actual feature map, are set to zero instead of propagating the original weights. This way, the dot-product doesn't get affected from padding.

3.1.4. Input Activation Streamer

In contrast to weights, the activations are stored in data memory. This enables simultaneous access of both. Provisioning of activations is more complex than weights. The linear access pattern associated with weights cannot be used for the delivery of activations. There are two reasons. First, the tensor format is HWC, as discussed in section 1.3.3. Second, inference requires a distinct compute order with multiple accesses to individual activations, which are distributed across the linearly stored tensor, as discussed in section 1.4.7. Because of that, the Input Activation Streamer applies a combined state machine, multi counter and masking based approach, which is capable of continuous N-dimensional data fetching and ready-to-compute data preparation.

The Input Activation Streamer is separated into *Meta Data*, *Sparsity Eval*, *Fetch*, *Extract* and *Postprocess* logic blocks, which are lined up in a pipeline structure. The *Meta Data* stage contains the *Address Generator*, which calculates the memory addresses of words fetched later. Those words contain multiple activations. In order to indicate the usage of the particular

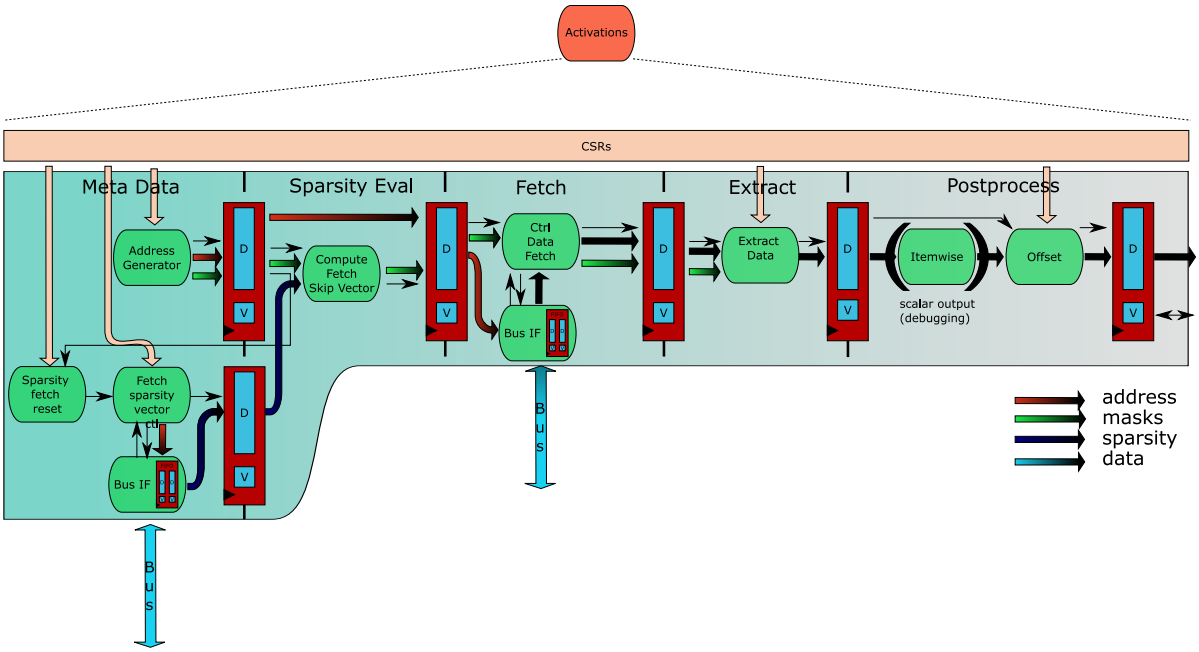


Figure 3.12.: Block diagram of the Input Activation Streamer

activations within the addressed word, the *Address Generator* provides according masks. In the remaining subblocks of the *Meta Data* stage, the sparsity information gets extracted and provided for further use.

In *Sparsity Eval* stage, the sparsity information and the masks are combined that are computed in the preceding stage. The resulting mask identifies only the used activations within a certain memory word, which are multiplied with non-zero weights later. The *Sparsity Eval* is followed by the *Fetch* stage. This block drops the fetch of words where all activations are identified as unused based on the padding and sparsity information. Only the addresses of those words initiate a memory access, which contain at least one payload activation. This reduces bus contention and power consumption at bus and memory side, which significantly contributes to the overall system power budget, according to the discussion of section 1.3.1. Once data is available to the Input Activation Streamer, it gets condensed in *Extract* stage. Therefore, the masks are applied to remove unused data and necessary paddings are inserted. This ensures feeding the *Computation Unit* with high density operands. The final *Postprocess* stage is comparable to the corresponding stage in the Weight Streamer.

The Input Activation Streamer is a pipelined component comprising of five stages. This causes some latency between the setup and the time-point, where the first dataset is ready to be processed by the *Computation Unit*. Once the run-in phase has passed, the streamer delivers with high duty load. Comparing the complexity of the address computation against a SW implementation, the pipeline is superior even though it has a depth of five stages. This holds especially for kernels of bigger dimensions, where run-in phase to deliver-phase ratio is small (Figure 3.13a). For small kernels or runs which require frequent reconfiguration (see Figure 3.13b), the *Setup* and *Run-In* phase become dominant over *Deliver*. Figure 3.13c shows

(a) big kernel, efficient streamer

(b) small kernel, inefficient streamer

(c) pareto point, software and streamer are equally efficient

Figure 3.13.: run-in vs deliver phase

an example for a kernel, where plain SW and accelerated version run equally fast. In that case, the accelerated version is still advantageous from performance perspective. Since the processor is not blocked, side band activities can be treated in parallel.

Meta Data Generation

This stage evaluates the configuration given via CSRs, which specifies the provisioning pattern and memory alignment of the activations in an abstracted fashion. A set of metadata is generated in each cycle, which consists out of an address and masks. They enable the streamer to fetch and extract data in succeeding stages, according to the abstracted configuration. Internally, the tasks are split into *address generation*, *mask generation*, *FSM* and *sparsity fetch*. All of them have strong interdependencies regarding intermediate information, except the *sparsity fetch*. Therefore, those are grouped into the same parental component to maintain data availability.

Address Generation In order to compute a fetch address, some auxiliary data is required. It is provided by the *Generate Start Column* and *Generate Column Length* subcomponents. The address for a particular cycle is determined by *Compute Memory Address*.

Generate Start Column The CSR configured startcolumn describes the alignment of the kernel on the feature map. Since the tensors are mapped into one-dimensional densely packed arrays in memory, the defined startcolumn doesn't match with the on-chip compute data. Furthermore, the startcolumn changes row dependent. To avoid frequent CPU interruptions for reconfiguration purposes, a dedicated HW addresses the required translation.

The *Divide and Conquer* approach constitutes the introduction of the translated s'_{col}, which is utilized to solve the complex process of fetching densely packed data. It allows to separate the address computation and handling of the packing style. By applying s'_{col} instead of s_{col}, densely packed activations can be processed with the logic designed for aligned packing. This is less complex and has a lower number of corner cases to handle.

Equation (3.6) formalizes the translation done by this component. The output s'_{col} is static for a particular row, since the other influencing signals are driven by the configuration CSRs, which are not changing during an activated phase. The row r changes with low frequency. Its update rate primarily depends on the configured kernel dimensions. With increasing kernel width and channel number, the update frequency of row and translated startcol s'_{col} decreases.

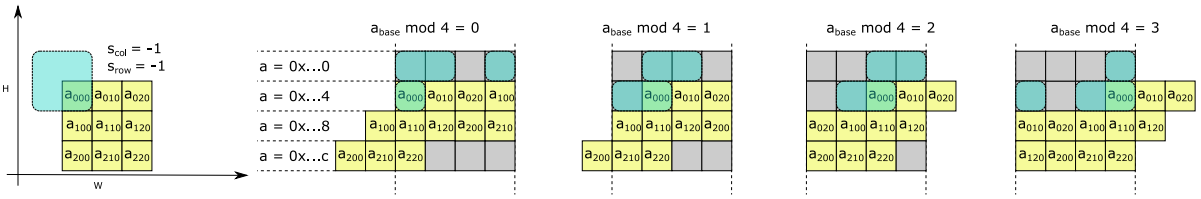
In Figure 3.14, the translation from tensor alignment to the memory domain is shown for different startcolumns, startrows and baseaddresses for a 3x3 kernel. Table 3.1 presents the evaluation of (3.6) for the example shown in Figure 3.14. s'_{col} can be interpreted as the offset between the index of the initial activation for a given kernel footprint and the LSB (least significant Byte) of the word, which contains the baseaddress. A negative s'_{col} points outside the activation feature map. In those cases, instead of using illegal data located outside the featuremap a predefined padding value pad_{val} is applied.

$$s'_{col} = (s_{col} + a_{base} \bmod 4) + ((s_{row} + r) * (l_{row} \bmod 4)) \tag{3.6}$$

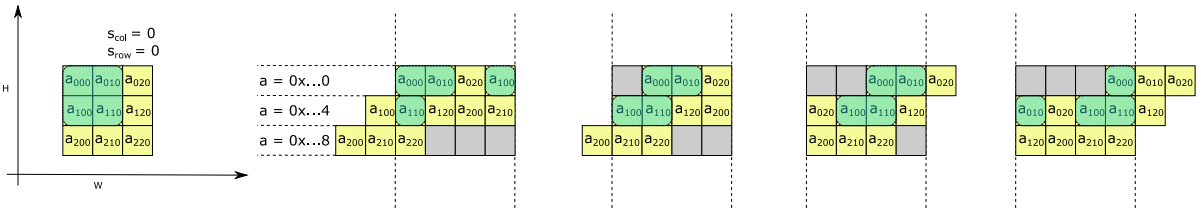
where:

s'_{col} : row dependent column index of first activation
s_{col} : column index of initial fmap activation
s_{row} : row index of initial fmap activation
l_{row} : number of activations in width (width*channels)
r : index of current row

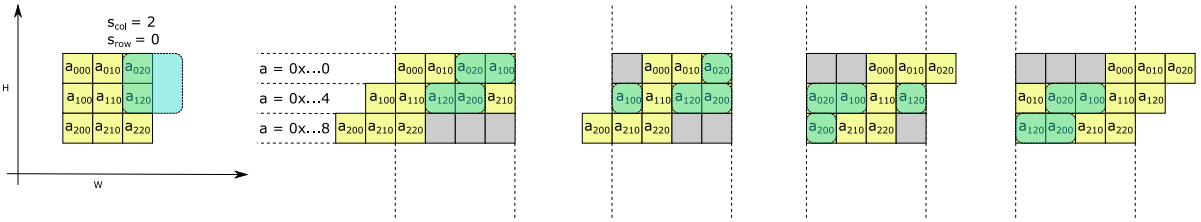
Generate Column Length Depending on the configured kernel dimensions and the startcolumn, the number of consecutive memory addresses changes. Consecutive accesses deliver data belonging to the same row of the activation tensor. Since s'_{col} changes for every row, the evaluation $l_{col\ mem}$ has to be done row-wise, too. Equation (3.7) presents the corresponding computation in a formalized way. The attained information determines the particular number of memory fetches required to obtain all referred activations.



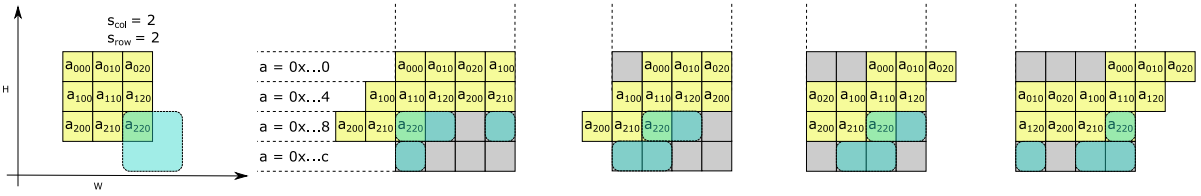
(a) $s_{col} = -1$, $s_{row} = -1$



(b) $s_{col} = 0$, $s_{row} = 0$



(c) $s_{col} = 2$, $s_{row} = 0$



(d) $s_{col} = 2$, $s_{row} = 2$

Figure 3.14.: Feature Map access pattern 2D array conv layer, 2x2 kernel, 3x3 fmap

s'_{col}	row	(a)	(b)	(c)	(d)
$a_{base} \bmod 4 = 0$	0	-4	-3	-2	-1
	1	-1	0	1	2
$a_{base} \bmod 4 = 1$	0	0	1	2	3
	1	3	4	5	6
$a_{base} \bmod 4 = 2$	0	2	3	4	5
	1	5	6	7	8
$a_{base} \bmod 4 = 3$	0	8	9	10	11
	1	11	12	13	14

Table 3.1.: Evaluation of s'_{col} for the 3x3 kernel example of Figure 3.14

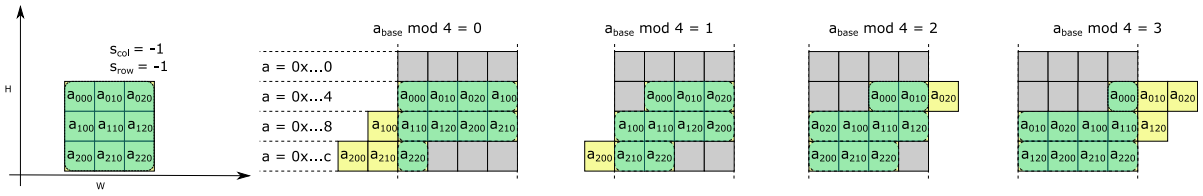


Figure 3.15.: Feature Map access pattern 2D array fully connected layer, 3x3 fmap

$$l_{col\ mem} = \lceil \frac{w_k - (4 - s'_{col} \bmod 4)}{4} \rceil + 1 \tag{3.7}$$

where:

$l_{col\ mem}$	:	memory column length
w_k	:	kernel dimension width
s'_{col}	:	row dependent column index of first activation

Compute Memory Address This subcomponent is fully combinatorial. The required state information is propagated from the FSM. It evaluates the state information, the CSR configuration and the intermediate data. Equation (3.8) formalizes the relation to compute the according memory addresses, which the component implements. Due to the usage of the translated s'_{col} instead of s_{col}, the formula is rather straight.

$$a_{frowcol} = (\lfloor \frac{a_{base}}{4} \rfloor + (r + s_{row}) * \lfloor \frac{l_{row}}{4} \rfloor + \lfloor \frac{s'_{col}}{4} \rfloor + col) * 4 \tag{3.8}$$

where:

$a_{f\ row\ col}$	:	word address to be fetched, dependent on row and col of the referred activations
s'_{col}	:	column index of initial fmap activation
s_{row}	:	row index of initial fmap activation
l_{row}	:	number of activations in width (width*channels)
r	:	index of current row
col	:	memory column to fetch within row (word addresses) within $[0, l_{col\ mem}]$

Masks Masks indicate the usage of the intrinsic activations for the corresponding word. There are three usage types. Either the according activation has to be dropped, which is the case for activations that are located outside the kernel footprint. These are not allowed to be used for inference computation, since they don't have a meaning in this and would corrupt the actual

computation. Second are datasets that are located inside the kernel footprint but outside the feature map, e.g. for negative startcolumn or startrow. There are no corresponding values stored within the memory, but predefined padding values have to be inserted. Skipping them isn't sufficient, since padding values different from zero are desired in some scenarios and impact the inference result. Third is the usual case, where the data stored in memory is required. This has to result in a data forward.

In order to cover the three described usage cases, two masks are introduced. Those need to be computed for every fetched word. One is called *kernel mask*, the other *data frame mask*. Their computation and purpose is explained in the following.

Kernel Mask The kernel mask indicates all activations located inside the kernel footprint with a *1* and all others with a *0*, no matter whether the datasets are located inside or outside the feature map. In order to compute the kernel mask two additional intermediate information have to be computed. The computation of the *index of first item* it_{1st} and *current number of items* it_{cur} deliver the necessary information for preparing the kernel mask.

it_{1st} defines the index of the first activation within a word, which is within the kernel footprint. It's calculation can be found in (3.9). it_{1st} is consistently set to one in case the *col* of a word is greater than zero. The same behavior holds for a word, which is aligned next to the word overlapping with the kernel's footprint boundary.

$$it_{1st} = \begin{cases} s'_{col} \bmod 4 + 1 & col = 0 \\ 1 & col > 0 \end{cases} \tag{3.9}$$

where:

it_{1st} : index of first item
s'_{col} : row dependent column index of first activation it_{1st}
col : memory column to fetch within row (word addresses) within $[0, mem\_col\_length]$

The *current number of items* describes the amount of activations of the according word to be used. Hereby, counting is started from *index of first item* in ever increasing fashion until the last element is reached, which is still in the kernel footprint. The corresponding formalization is presented in (3.10)

$$it_{cur} = \begin{cases} (w_k - it_{cons}) \bmod 4 & (4 - it_{1st}) + 1 > w_k - it_{cons} \\ (4 - it_{1st}) + 1 & \text{otherwise} \end{cases} \tag{3.10}$$

where:

it_{cur} : current number of items
w_k : width kernel
it_{cons} : consumed items
it_{1st} : index of first item

The kernel mask can be computed deploying those intermediate information, as shown in (3.11). Every row in the kernel gets its own kernel mask. An example for a kernel mask of a 2x2 kernel, which is interleaved with the featuremap, is shown in Figure 3.16. It also presents the row specific kernel mask for different words. For the corner case of an overlapping kernel footprint for different rows in a common word, two masks are computed for the same word. This keeps the logic lean, but adds an additional clock cycle.

$$m_k = (((\texttt{0x10} >> it_{cur})[3:0])\&0 >>> it_{1st})[3:0] \tag{3.11}$$

where:

m_k : kernel mask
w_k : width kernel
it_{cur} : current number of items
it_{1st} : index of first item

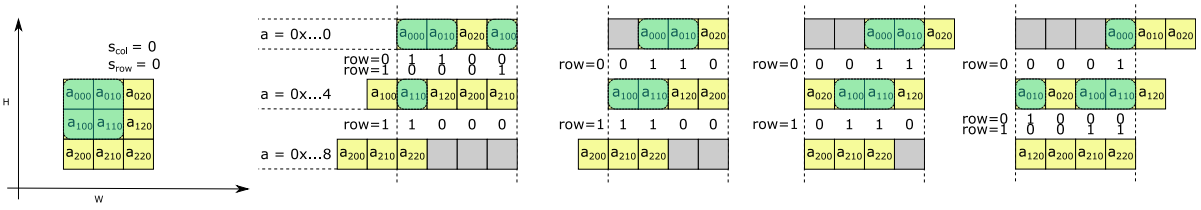


Figure 3.16.: kernel mask

Data Frame Mask The *Data Frame Mask* indicates the activations inside the featuremap, regardless of the kernel footprint locality. Again, the *Divide and Conquer* approach is deployed for mask preparation. The task is separated into four independent operations. Each of them is responsible for one of the four boundaries. The individual masks indicate datasets located outside the assigned feature map boundary with *zeros*. The other boundaries are ignored and the according activations are considered inside the featuremap, indicted by *ones*.

The pseudo-code for the mask computation of the left-side boundary is presented in Algorithm 7. Figure 3.17 shows the computed leftside mask for a 6x6 featuremap. The resulting mask shows leading *zeros* for activations either outside the left-side boundary of the featuremap or belonging to preceding rows. Contrarily, the right-side boundary is ignored.

$pot\_shift\_amount = (s'_{col} \bmod 4 - s_{col}) - col * 4;$
if $pot\_shift\_amount < 0$ **then**
  $shift\_value \leftarrow 0;$
else if $pot\_shift\_amount > 4$ **then**
  $shift\_value \leftarrow 4;$
else $0 \leq pot\_shift\_amount \leq 4$
  $shift\_value \leftarrow pot\_shift\_amount;$
end
`0b1111` $>>> shift\_value;$

Algorithm 7: Calculate m_{left}

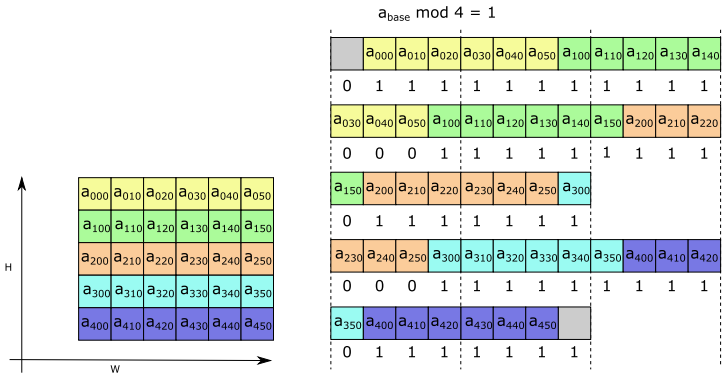


Figure 3.17.: frame mask leftside

Similar to left-side masking, the pseudocode to determine the right-side mask is shown in Algorithm 8. Figure 3.18 illustrates this for the 6x6 featuremap. The mask indicates all activations on the left side of the right boundary as *ones*. Where the right boundary is hit, *zeros* follow.

Algorithm 9 presents the pseudocode for determining the upper mask. Figure 3.19 illustrates this for the 6x6 featuremap. All words that are located above the feature map and those which don't contain at least one activation residing inside, obtain a mask of zeros only. In the opposite case, where at least one activation belongs to the featuremap or previous datasets were identified to belong to the feature map, all mask bits are *ones*. This signals that the according word fulfills the condition for the upper bound.

$a \leftarrow row\_length - s_{col} + (s'_{col} \bmod 4) - 4 * (col + 1)$;
$b \leftarrow row\_length - s_{col} + (s'_{col} \bmod 4) - 4 * col$;
$overlap\_right \leftarrow (a < 0) \& (b \geq 0)$;
$in\_activa\_right \leftarrow a > 0$;
if $overlap\_right = 1$ **then**
  if $b \geq 4$ **then**
    $m'_{right} \leftarrow$ `0b11111`;
  else $b < 4$
    $m'_{right} \leftarrow$ `0b10000` $>> b$;
  end
  $m_{right} \leftarrow m'_{right}[3:0]$;
else $overlap\_right \neq 1$
  if $in\_activa\_right = 1$ **then**
    $m_{right} \leftarrow$ `0b1111`;
  else $in\_activa\_right \neq 1$
    $m_{right} \leftarrow$ `0b0000`;
  end
end

Algorithm 8: Calculate m_{right}

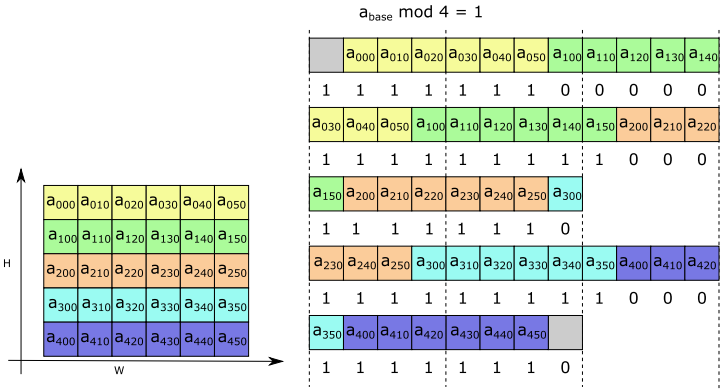


Figure 3.18.: frame mask rightside

$c \leftarrow (row + s_{row}) \geq 0$;
$m_{upper} \leftarrow$ `0b`$cccc$;

Algorithm 9: Calculate m_{upper}

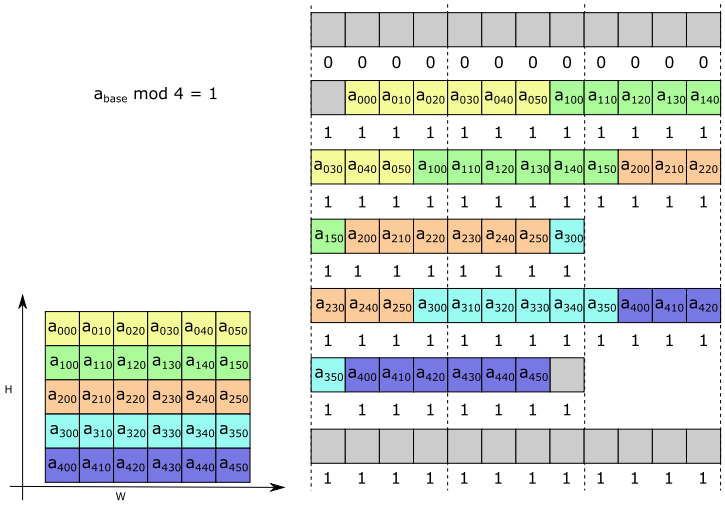


Figure 3.19.: frame mask upper

Analogous to the upper mask also the lower mask is computed. Algorithm 10 describes the according computation in pseudocode. The evaluation results in the masks as presented in Figure 3.20. It can be seen that the mask holds either all *zeros* or all *ones*. Beginning with the upper side, the mask consists of *ones* only. The *ones* are present until the first word is reached that is aligned entirely outside of the feature map. Onwards, only *zeros* succeed.

$d \leftarrow (row + s_{row}) < col\_height$;
$m_{lower} \leftarrow$ 0b*ddddd*;

Algorithm 10: Calculate m_{lower}

In order to attain a common mask, which describes the entire alignment of the featuremap within the memory, the four separately computed masks are combined. Hereby, the intersection of all of them is taken. Algorithm 11 presents the implementation of the intersection operation in pseudocode. An accurate mask is obtained (shown in Figure 3.21), which indicates all datasets within the boundaries of the featuremap by *ones*. Those datasets, which are located outside the boundaries, are marked with *zeros*.

$m_{edge} \leftarrow m_{left} \& m_{right} \& m_{upper} \& m_{lower}$;

Algorithm 11: Calculate m_{edge}

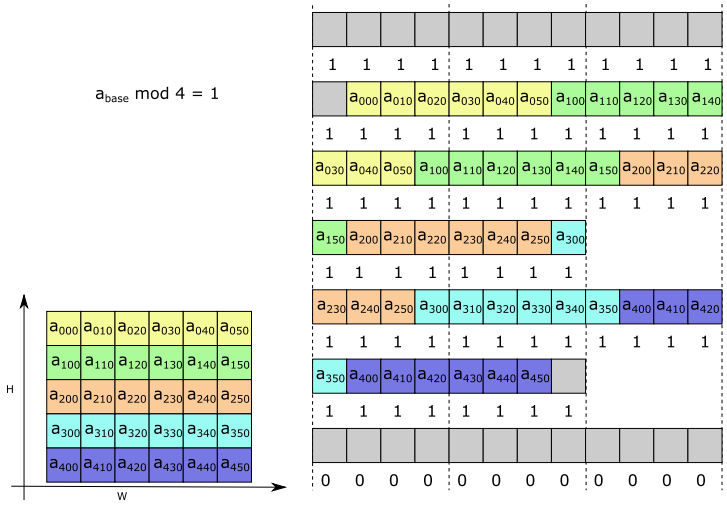


Figure 3.20.: frame mask lower

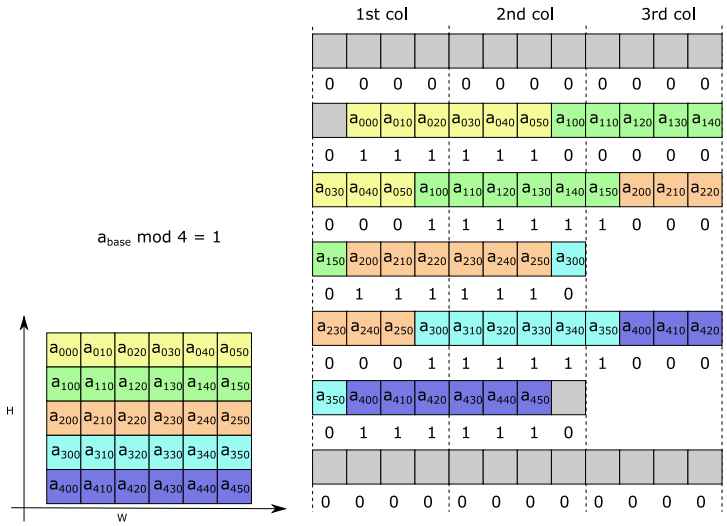


Figure 3.21.: frame mask

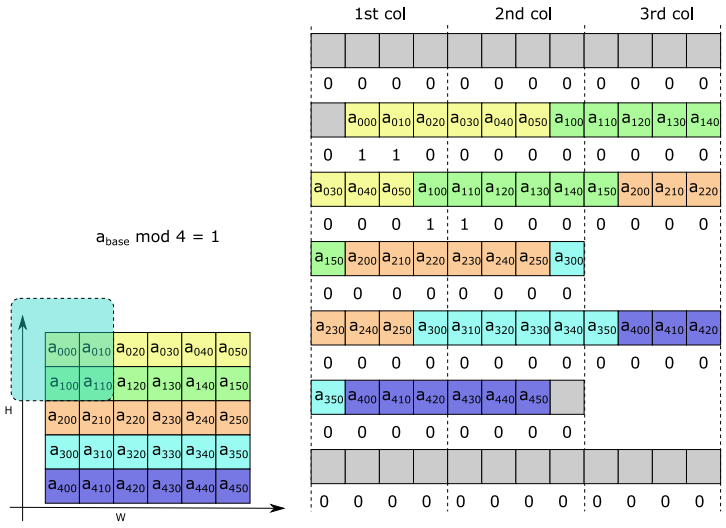


Figure 3.22.: pad mask1

Pad Mask The combination of the kernel mask and data frame mask encapsulate all necessary information to determine fetch, drop or pad for each activation. To access the information about fetch and drop the *padmask* is extracted out of the two other masks by getting the intersection of *edge mask* and *kernel mask*, as presented in Algorithm 12. Hereby, activations located inside the featuremap and kernel footprint at the same time are marked with *ones* (shown in Figure 3.22).

$$m_{padding} \leftarrow m_k \& m_{edge};$$

Algorithm 12: Calculate $m_{padding}1$

A second mask can be computed out of the *kernel mask* (m_k) and *edge mask* (m_{edge}), which indicates the elements to be replaced by the predefined padding value, (see Figure 3.23). The corresponding Algorithm 13 presents this computation.

$$m_{padding} \leftarrow (m_k \oplus m_{edge}) \& m_k;$$

Algorithm 13: Calculate $m_{padding}2$

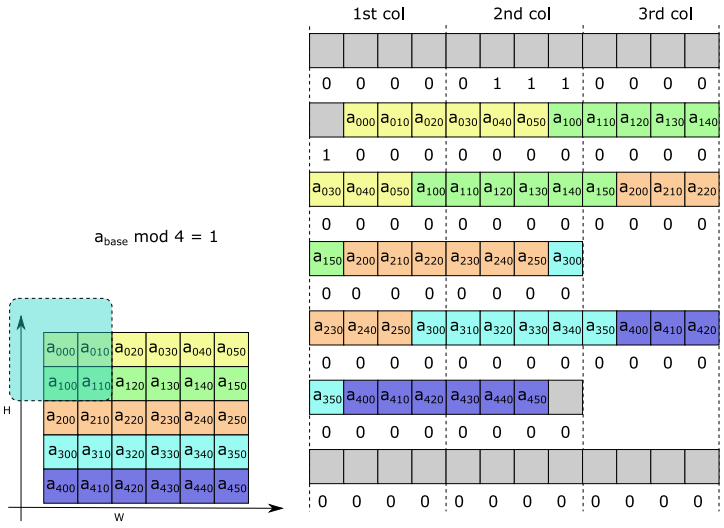


Figure 3.23.: pad mask2

FSM Previously presented address generation and mask computation logic are lacking control, since they are fully combinatorial. This control comes into play by the FSM, which is introduced in this paragraph. It's supported by counters and comparator logic as shown in Figure 3.24. The FSM mainly determines increment and reset triggers for *col count, row count* and *repeat count* based on it's internal state and the inputs driven by the comparators. These assess the current counter values and determine threshold overflows. The thresholds either originate from the CSR configuration or are computed as intermediate values, by above introduced *meta data generation* logic.

The FSM is implemented as a mealy machine. The counter control signals generated by the FSM have a direct dependency on the current counter values. An additional FSM state register in between of input and output would introduce a delay of one cycle and as a consequence to a delay of the intermediate signals. Such a behavior is not acceptable, since counter increments and resets need a direct control. Otherwise, data processing is corrupted. The transition diagram of the FSM is illustrated in Figure 3.25. It consist out of the following nine states:

init Dependent on the current inputs, either a column, row or repetition increment follows.

cols Column was incremented in the last cycle. The current inputs determine if a column increment follows. Alternatively, the row or repetition counter can be triggered. Hereby, column and row count gets reset respectively.

rows Row was incremented in the last cycle. The current inputs determines if the next row increment follows. Alternatively, the column, or repetition counter can be triggered. Hereby, the row count gets reset.

repeats Repetition was incremented in the last cycle. The current inputs determines if the next repetition increment can follow. Alternatively the column, or row counter can be triggered.

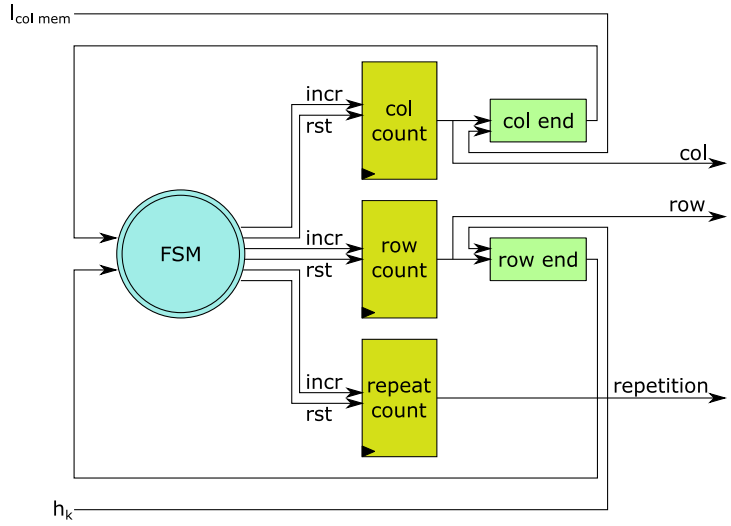


Figure 3.24.: FSM/Counter interaction

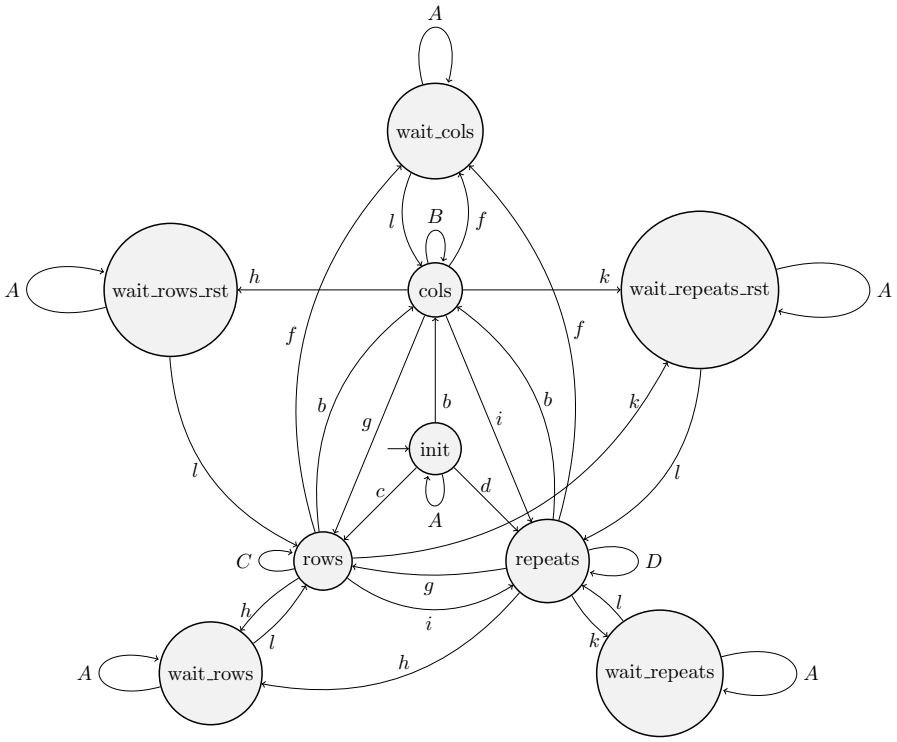


Figure 3.25.: Address Generator FSM

wait_cols The condition to increment the column was met in one of the preceding cycles. Because of a pipeline stall, the FSM could not proceed. The increment of the column is memorized and will be executed, once the stall gets released.

wait_rows The condition to increment the row was met in one of the preceding cycles. Because of a pipeline stall, the FSM could not proceed. The increment of the row is memorized and will be executed, once the stall gets released.

wait_repeats The condition to increment the repetition count was met in one of the preceding cycles. Because of a pipeline stall, the FSM could not proceed. The increment of the repetition is memorized and will be executed, once the stall gets released.

wait_rows_rst The condition to increment the row and reset the column was met in one of the preceding cycles. Because of a pipeline stall, the FSM could not proceed. The according increment and reset are memorized and will be executed, once the stall gets released.

wait_repeats_rst The condition to increment repetition count and reset column and row was met in one of the preceding cycles. Because of a pipeline stall, the FSM could not proceed. The according increment and resets are memorized and will be executed, once the stall gets released.

The inputs to the FSM are:

busy indicates a pipeline stall. The FSM cannot proceed, since the produced output data is not accepted.

en indicates the global enable of the streamer. In case the streamer is disabled, also the FSM needs to be disabled. The state is preserved in order to guarantee proper continuation.

col_end is triggered by the *col end* comparator. It signals, if the column counter exceeds the number of memory columns $l_{col\ mem}$, which are provided as intermediate data.

row_end is triggered by the *row end* comparator. It signals, if the row counter exceeds the configured number of memory rows h_k.

Due to the limited space in the transition diagram in Figure 3.25, the according transition conditions are encoded in Table 3.2. Hereby, capital letters indicate combinations of conditions, which are expressed via lower case letters.

The FSM drives the following outputs, which are used to control the counters:

incr_col triggers the increment of the column counter.

incr_row triggers the increment of the row counter.

variable	condition
a	$busy \wedge en$
b	$!busy \wedge en \wedge !col\_end \wedge !row\_end$
c	$!busy \wedge en \wedge col\_end \wedge !row\_end$
d	$!busy \wedge en \wedge col\_end \wedge row\_end$
e	$!en$
f	$busy \wedge en \wedge !col\_end$
g	$!busy \wedge en \wedge col\_end \wedge !row\_end$
h	$busy \wedge en \wedge col\_end \wedge !row\_end$
i	$!busy \wedge en \wedge col\_end \wedge row\_end$
k	$busy \wedge en \wedge col\_end \wedge row\_end$
l	$!busy \wedge en$
A	$a \vee e$
B	$b \vee e$
C	$g \vee e$
D	$i \vee e$

Table 3.2.: Address Generator FSM transition conditions

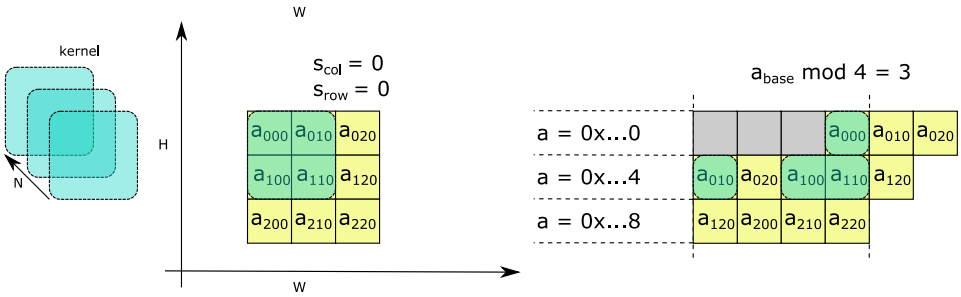
incr_repeat triggers the increment of the repeat counter.

rst_col resets the column counter to zero. It is triggered when the iteration across all columns per row has been done and a switch to next row is required.

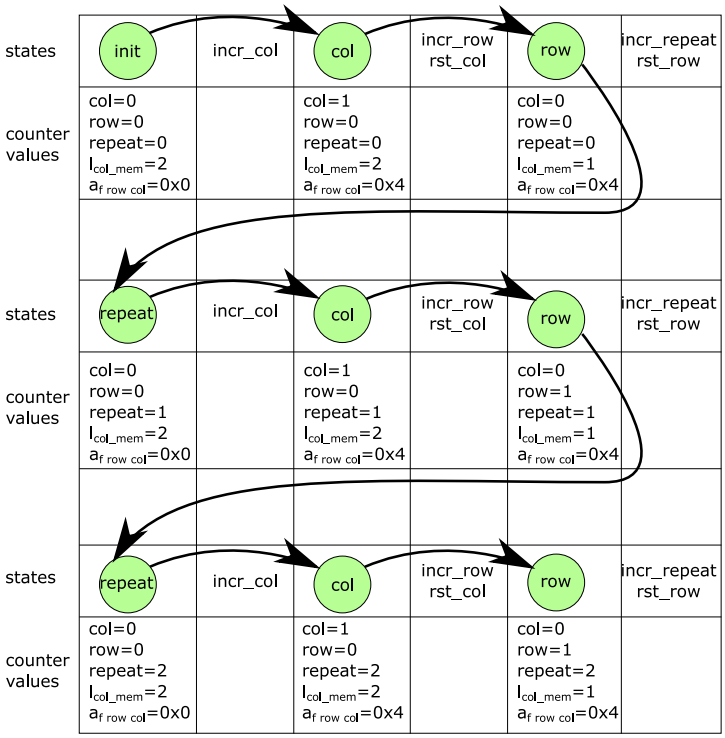
rst_row resets the row counter to zero. It is triggered when the iteration across all rows and columns of the last row has been done and a switch to next repetition is required.

Since a mealy machine is deployed, the output signals do not only depend on the current state, but are also influenced by the current inputs. Due to the massive amount of possible state and condition combinations, the outputs cannot be shown in the transition diagram in Figure 3.25. Instead, Table A.1 shows the output signal dependencies in detail.

Figure 3.26b shows an example of the FSM-counter trace for the configuration of Figure 3.26a. Because of the feature map alignment given by a_{base}, two columns are required for the first row. In contrast, for the second row the fetch of one column is sufficient. Three repetitions are done, since three kernels are applied to the activation feature map.



(a) FSM example



(b) FSM exmaple2

Figure 3.26.: FSM example

sparsity fetch The accelerator developed in this thesis deploys weight sparsity information. In contrast to sparsity in activation data, the sparsity of weights can be pruned during training phase. This facilitates a high number of zero weights, thus a high utilization of the feature can be expected. Additionally, weight sparsity is available during offline precomputation, whereas activation sparsity is dynamic and only run-time available. Due to this property and the fact that weights are static, the weight sparsity can be stored in ROM (Read Only Memory).

The sparsity information is encoded in a vector structure. More complex encodings can be deployed to further reduce the memory footprint of the sparsity encoding information. The modular building concept of the accelerator allows to slot in a decoder of any complexity. For the purpose of feasibility, a restriction to a plain encoding into a 1-to-1 vector mapping is done. Hereby, each entry of the vector is assigned to one distinct weight in the kernel, as shown in Figure 3.27. The weights marked in orange are not stored in the memory, instead the array jumps to the next non-zero weight.

The sparsity vector is stored in a back-to-back fashion in memory. This allows to encode the sparsity information of up to 32 weights within one word. In the sparsity vector, zero weights are indicated with a 0 and non-zero weights with a 1. In order to fetch the sparsity vector from memory, an address increment by 4 is sufficient. Byte aligned sparsity vectors are supported. Hereby, the word which contains the initial byte is fetched first. The parts of that word, which precede the sparsity vector, are identified based on the sparsity baseaddress $a_{sparsity}$ and are dropped accordingly. The sparsity items of the current word are combined with the following word to form a complete aligned 32-bit sparsity vector.

The achievable compression ratio of using sparsity can be calculated according to (3.12) (adapted from (Bringmann et al. 2021)).

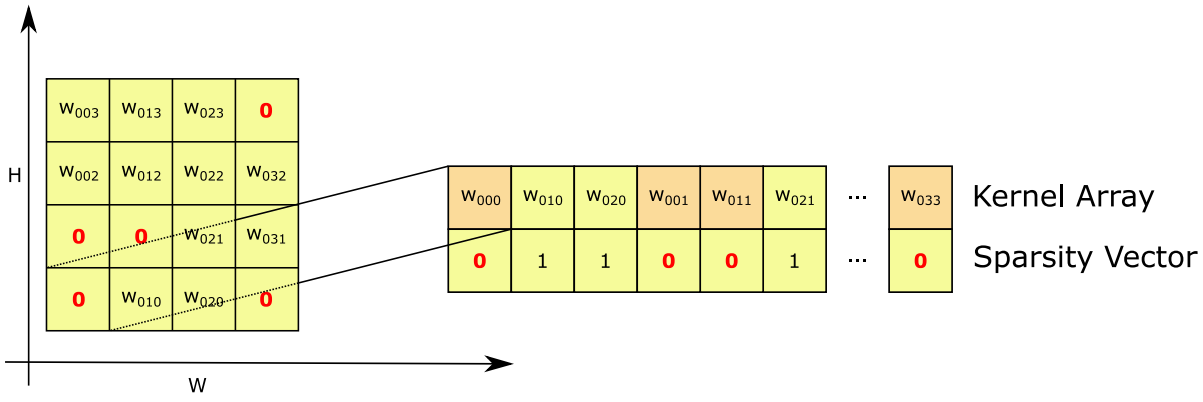


Figure 3.27.: sparsity encoding vector

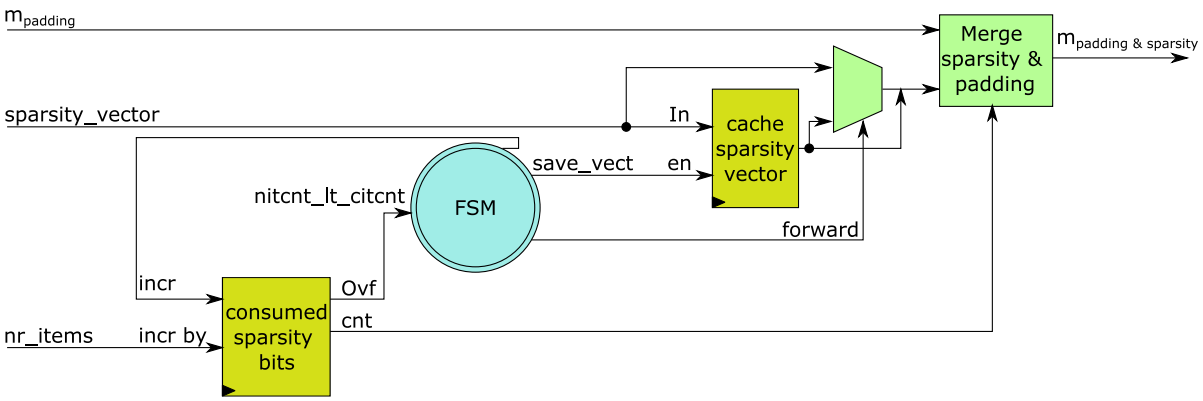


Figure 3.28.: Block diagram of the *Compute_fetch_skip_vector* unit

$$r_c = (1 - r_s) + r_p * c_{es} \tag{3.12}$$

where:

r_c : compression ratio
r_p : number of packed datasets per byte
r_s : number of zero datasets per overall number of datasets
c_{se} : sparsity encoding cost in byte per dataset

Fetch/Skip Vector Computation

The *Compute fetch skip vector* component requests data from two subblocks of the *Meta Data* stage, as depicted in Figure 3.12. It uses $m_{padding}$, which is computed in the *Address Generator* and the sparsity information delivered by the *Sparsity fetch*. Based on the sparsity information, $m_{padding}$ gets modified. Figure 3.28 shows a block diagram of the component. It consists of an FSM, two registers and some combinatorial logic for performing the modification operation. The *consumed sparsity bits* register is 5 bit wide with a maximum value of 31. It stores the number of consumed bits. This information is needed to identify, when to switch to the next word that contains the sparsity information corresponding to the succeeding data. Additionally, it constitutes a pointer to the bits inside the sparsity vector, which allows to identify the location of the sparsity bit assigned to the according weight. The second register caches the sparsity vector. It can be bypassed with the forwarding multiplexer to save the update clock cycle. The *Merge sparsity & padding* component is explained for an example later in this section. It combines the sparsity information together with $m_{padding}$ and creates a new mask ($m_{padding\ \&\ sparsity}$). This mask combines both information. The control of the introduced components is up to another FSM, discussed in the following.

The state transition diagram of the FSM is illustrated in Figure 3.29. It consists out of four states:

init The sparsity vector isn't loaded to cache, yet.

no_addr_fifo_busy $m_{padding}$ is missing or the output register is not ready to take the next output mask. The sparsity vector is available.

output_vect Both information ($m_{padding}$ and sparsity vector) are present, thus one output mask is computed.

no_data The sparsity information is missing.

The inputs to the FSM are:

busy indicates a pipeline stall. The FSM cannot proceed, since generated output data is not accepted.

en indicates a global enable of the streamer. In case the streamer is disabled, also the FSM is disabled. The state has to be preserved to guarantee proper continuation.

valid_vect signals to the FSM that a valid sparsity vector is available to the input of the *Sparsity Eval* stage.

valid_addr signals to the FSM that a valid $m_{padding}$ is available to the input of the *Sparsity Eval* stage.

nitcnt_lt_citcnt indicates an overflow in the *Consumed sparsity bits* counter. This is equivalent with an expiring cached sparsity vector, which is stored in *cache sparsity vector*.

Due to the limited space in the transition diagram in Figure 3.29, the transition conditions are listed in Table 3.3. Hereby, capital letters indicate combinations of conditions expressed via lower case letters.

The FSM drives the following outputs, which are used to control the accelerator components:

ready_vect signals to the input register of the sparsity vector that the data at its output ports got accepted.

ready_addr signals to the input register of the mask that the data at its output ports got accepted.

valid The data at the output of the *Compute_fetch_skip_vector* component is valid and can be taken into the *Fetch* stage

save_vect Update the *cache sparsity vector* with the next sparsity vector.

incr_itm Increment the *Consumed sparsity bits* counter by *nr_items*.

variable	condition
a	$\neg valid\_vect \wedge en$
b	$valid\_vect \wedge busy \wedge en$
c	$valid\_vect \wedge \neg valid\_addr \wedge en$
d	$valid\_vect \wedge valid\_addr \wedge \neg busy \wedge en$
e	$\neg en$
f	$busy$
g	$\neg valid\_addr \wedge en$
h	$valid\_vect \wedge valid\_addr \wedge nitcnt\_lt\_citcnt \wedge \neg busy \wedge en$
i	$valid\_addr \wedge \neg nitcnt\_lt\_citcnt \wedge \neg busy \wedge en$
k	$\neg valid\_vect \wedge valid\_addr \wedge nitcnt\_lt\_citcnt \wedge \neg busy \wedge en$
l	$busy \wedge en$
A	$a \vee e$
B	$b \vee c$
C	$e \vee h \vee i$
D	$g \vee l$
E	$e \vee f \vee g$
F	$h \vee i$

Table 3.3.: compute fetch skip vector FSM transition conditions

forward Directly forward the sparsity vector from the input stage, instead of utilization of the *cache sparsity vector* data.

In the following, a behavioral description of the transitions and accordingly triggered output signals is given. A formal description of the output triggering mechanism is given in A.2.

init There are three possible target states for transitions. In case neither sparsity vector nor mask are available, the FSM loops back to the init state and waits another cycle. When the sparsity vector becomes available, but either the mask or output register is not available a transition to *no_addr_fifo_busy* state is performed while storing the sparsity vector into the cache register and clearing the according input register. Alternatively, when mask and sparsity vector become available simultaneously and the output register is ready to take the next modified mask, the state transitions to *output_vect*. Both source registers update their content while the sparsity vector gets cached. The *Consumed sparsity bits* counter gets incremented and the sparsity vector is forwarded.

no_addr_fifo_busy In this state, three transitions are possible. In case the mask is unavailable or the output register is busy, the transition loops back into the same state and waits. When the mask becomes available, but the *Consumed sparsity bits* overflows and the next sparsity vector is not available, a transition to *no_data* state is executed. The third case treats the transition to *output_vect* state. When the mask is available and the sparsity vector is not expired (or the next one is available), the output mask can be computed and is propagated into the next stage. Additionally, the *Consumed sparsity bits* counter

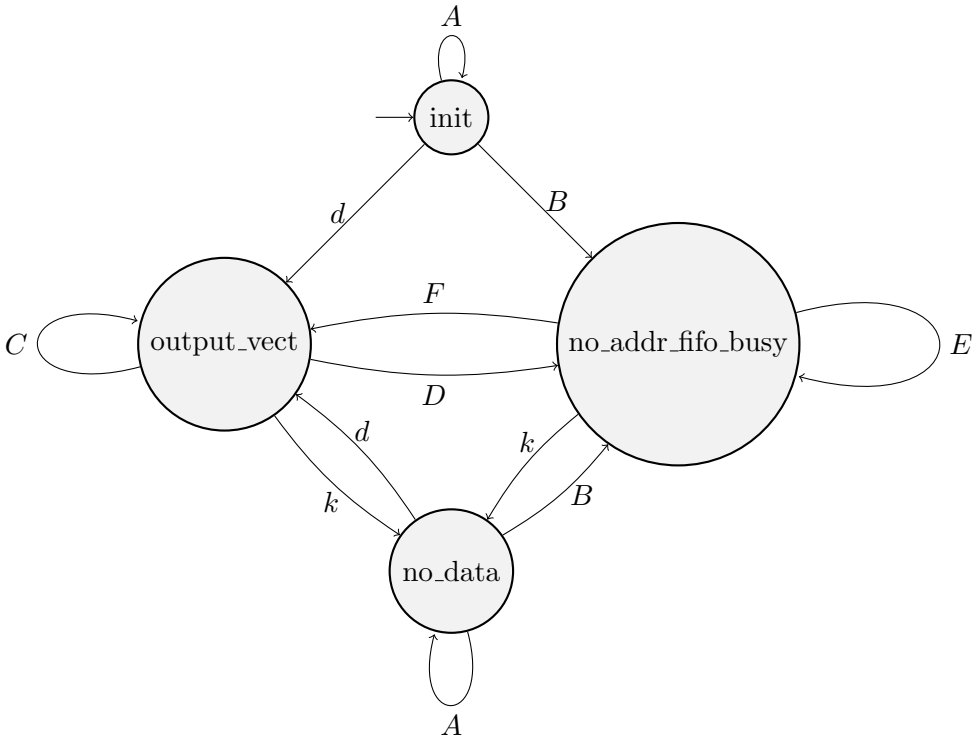


Figure 3.29.: compute fetch skip vector FSM

gets incremented by the number of consumed sparsity bits and a new sparsity vector gets stored. Further, the corresponding input register is cleared in the case of an expired sparsity vector.

output_vect Three transitions are possible. If the mask is available and the sparsity vector is cached or is available in the corresponding input register, the next output mask is computed and the FSM loops back to the same state. In the case of an expired sparsity vector, *cache sparsity vector* register gets updated and the corresponding input register is cleared. In case the next sparsity vector is missing and the cached one is expired but a mask is available, a transition to *no_data* stage is executed. Alternatively, if there is no valid mask or an occupied output register, the state transitions to *no_addr_fifo_busy*

no_data As long as the FSM is lacking the sparsity vector, it loops within this state. When the sparsity information becomes available, it transitions back into the *output_vect* state. Additionally, the sparsity vector is stored into the *cache sparsity vector* register and the corresponding input register is cleared. Simultaneously, an output mask is computed and the input register with the original mask gets cleared. The forwarding allows to proceed with the new sparsity vector immediately, without waiting for the updated *cache sparsity vector* register. The *Consumed sparsity bits* counter is incremented accordingly.

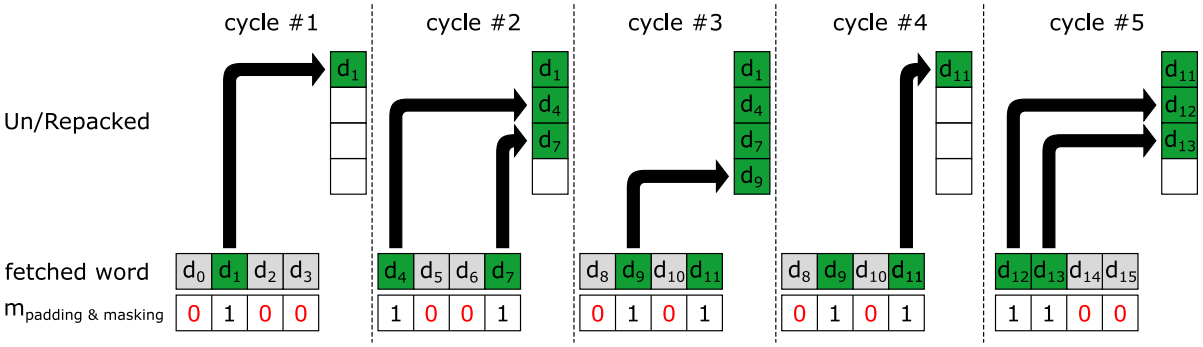


Figure 3.30.: Un- and Repacking of fetched word, into ready to process format

Un/Repacking

The feature map stored in memory contains all relevant activations. Padding, kernel footprint and sparsity information are stored in separate locations in memory. This coherence causes that the fetched data contains particularly unused data. In order to avoid corruption of the computation, un- or repacking is done

The behavior of this particular unit is explained by means of Figure 3.30. The $m_{padding\&sparsity}$ mask is aligned with the fetched word. For every entry within the fetched word, one bit of this mask is assigned. The entries of the fetched word with a corresponding "1" in the mask are loaded into the intermediate cache for un- and repacking purposes. In case of an empty cache, the data is inserted to the upmost location (e.g. "d1" in "cycle #1"). Whenever multiple datasets of a fetched word are extracted, unused datasets are skipped. The effective ones are remaining and aligned back to back. They are appended to the cache, like in "cycle #2". A cache overflow may occur, similar to "cycle #3". In that case, a two cyclic extraction is done. It completes the current cache first, before storing the remaining data into the cache in "cycle #4".

The packing component is assembled as shown in Figure 3.31. The main building blocks are an FSM, a cache and additional combinatorial logic, which is grouped into three major blocks. Detailed information about the FSM are available in Figure A.1, Table A.3 and A.4. The cache comprises of three byte-wide registers. The FSM drives the "Separation" logic, which divides the fetched word into its individual activations, drops unused bytes and appropriately aligns the data for the cache. Further the "Concatenation" logic is driven by the FSM. It composes the final un/repacked word out of cached and forwarded activations. The "Mask Evaluation" summarizes all of the precomputations, which are needed to make the control data available to the FSM.

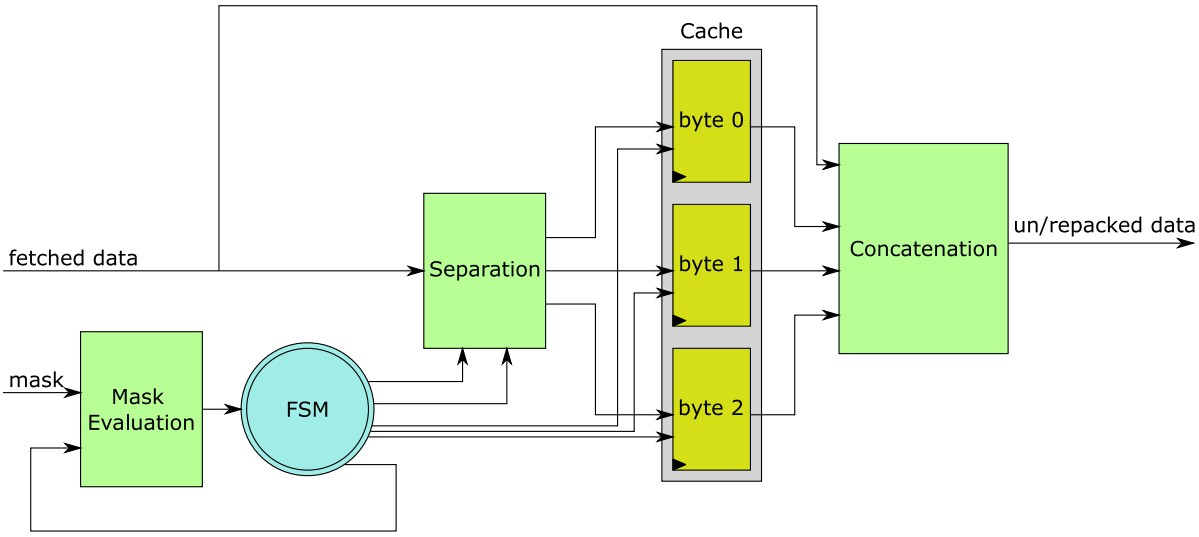


Figure 3.31.: Block diagram of Un- and Repacking unit

Repeat Mode

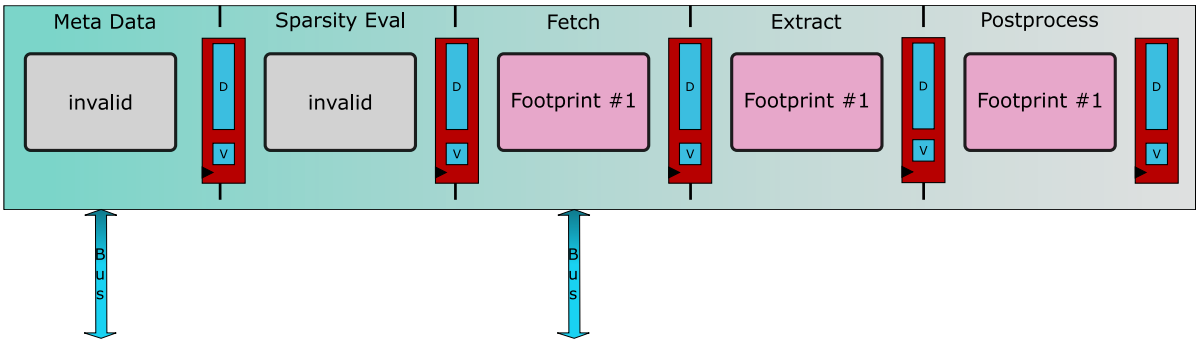
The streamer is featured with a repetition mode, as introduced earlier in this section. With enabled repetition, the same footprint is delivered iteratively for a configurable number of times. The benefit of supporting repetition is relevant for CNNs, especially.

During the computations for convolution layers, multiple kernels are applied to the same activations. Those iterations on the same footprint with different kernels produce the depth of the output activations. Since the footprint doesn't change, there is no need to update the streamer configuration in between the individual iterations, except resetting the internal states. The repetition mode avoids streamer interruptions, which would cause additional run-in phases and data drop, as long as the configuration doesn't need to be changed.

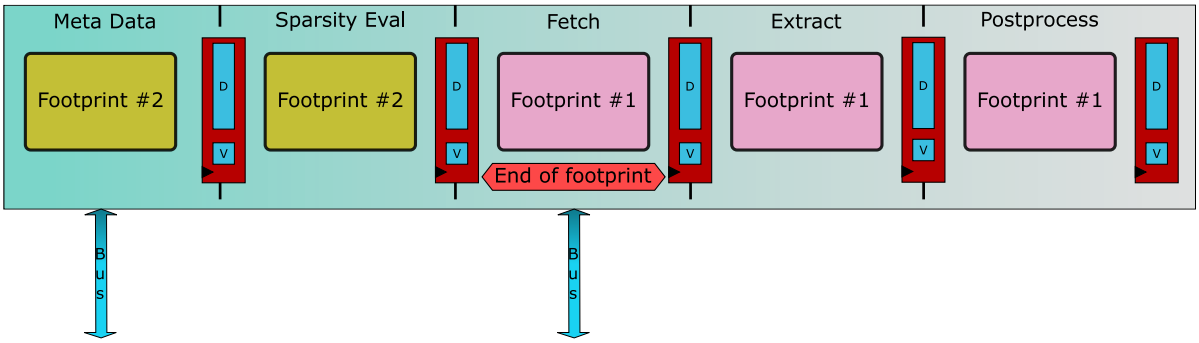
In FC layers, the kernel and featuremap dimensions match. Hereby, multiple kernels are applied to the entire feature map. The usage of the repetition mode allows to iterate through all kernels directly. A reconfiguration of the Input Activation Streamer is not needed, since the entire feature map has to be fetched for every kernel.

End of Footprint

For a particular configuration of a footprint located in the featuremap, the sparsity varies for different kernels. Thus, the numbers of cycles required to pop all data from the streamer for a particular kernel varies accordingly. Since the sparsity vector is available offline, the number of computation cycles can be precomputed and handled by the SW routines. This infers a significant drawback, since the information has to be stored into memory for every kernel and needs to be read every time the kernel changes. Depending on the kernel dimensions, this



(a) without end of footprint detection



(b) with end of footprint detection

Figure 3.32.: Pipeline utilization with and without end of footprint detection

can cause severe overhead. In order to address this issue, the streamer comes with a feature, which detects the last element for a certain footprint and sets a flag accordingly. This flag can be either forwarded through CSRs or used as interrupt signal. Handling via Interrupts is beneficial, since polling can be avoided which reduces the overhead of SW routines.

The Address Generator is responsible for detection of the end-of-footprint condition. The condition to trigger the "end of footprint" signal is fulfilled, when "col_end" and "row_end" are reached simultaneously. This condition is used as an equivalent to identify the footprint of an individual kernel. The result of this condition evaluation is propagated together with the corresponding address and mask, such that in every stage the units can treat the data in an appropriate way.

The "Sparsity fetch reset" uses this signal to identify, when to reset the sparsity pointer to the base address in repetition mode. This is done in such way that for the next repetition the same sparsity information gets fetched immediately (see Figure 3.32b). When switching from one repetition to the next, the sparsity information is kept in the pipeline, which preserves good performance and direct provisioning of proper data for the next repetition. Figure 3.32a shows the utilization of the streamer pipeline without the implemented "end of footprint" detection.

The data extraction, done in the Un/Repacking module needs to be modified when considering the "end of footprint" signal in order to deliver the relevant data in the correct way. The condition for marking a repacked word as valid changes at the end of the footprint. In usual mode, data is forwarded whenever four bytes are utilized. At the end of the footprint, data has to be forwarded early. It cannot be guaranteed that the usual condition for propagating a prepared word ($\#data\_sets \bmod 4! = 0$) is fulfilled, since one or multiple bytes in the last iteration will stay unutilized. Additionally, there is the chance that a certain dataset is identified as the last one of this footprint after several cycles have passed. This is because the sparsity information delivery and address generator work independent from each other and a pre-evaluation for future datasets isn't applicable. Due to the delayed determination of the "end of footprint" condition, the according repacked words are marked with the flag for "end of footprint" in a succeeding cycle. This means, the last dataset stays inside the pipeline until, either the next byte was fetched, or the "end of footprint" was detected.

3.1.5. Output Activation Streamer

So far the accelerator is featured with an efficient way of preparation and delivery of computation data (i.e. weights and input activations). In a tightly CPU-coupled setup, the treatment of output activations can be handeled via SW. This involves address calculations, which is needed to write the output activations back to memory in proper locations, the write executing itself, as well as quantization and packing of low precision dataformats. Furthermore the application of nonlinearities (i.e. activation functions) and pooling operations needs to be done. In order to further speedup the accelerator, those operations on output activations are addressed by the "Output Activation Streamer". Figure 3.33 shows the blockdiagram of this module. It comprises five of the six discussed operations. At time being, the pooling operation is not supported with the "Output Activation Streamer". The reason are mismatching processing orders in pooling and memory layout. A more detailed picture about pooling is given in section 1.3.1.

The "Output Activation Streamer" comprises of a "Scaling Unit" and an "Offset Adder", used for quantization. Further, an additional stage for the application of the activation function is introduced, which contains a "Linear Interpolation" and a "Clipping & Saturation" unit. Next, the "Packing" stage follows, which includes the "Address Generation" and the "Packing Unit" along with the "Packing Management". Finally, the postprocessed and packed output activations are written to the memory in the "Write" stage. An insight to the main building blocks of the "Output Activation Streamer" is given in the following.

Scaling Unit

The "Scaling Unit" is a hybrid multiplier, which treats one factor in integer format and the scaling factor in floating point *IEEE754* conform representation. The operation is separated in an integer multiplication, which treats the mantissa and the integer value, and an exponent

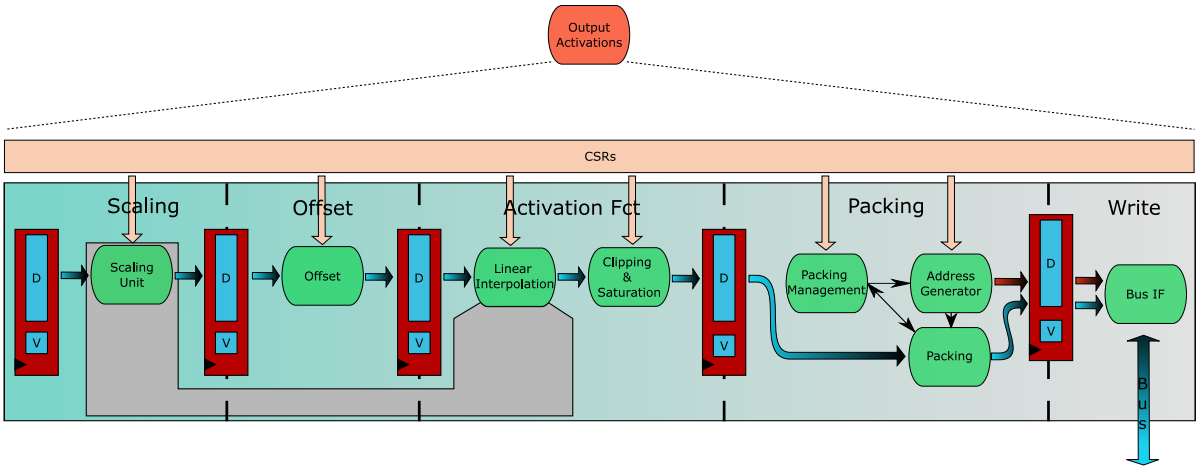


Figure 3.33.: Block diagram of the Output Activation Streamer

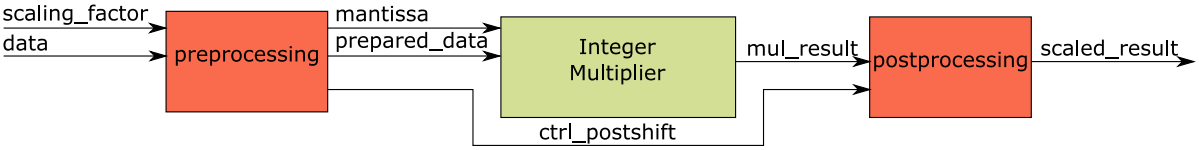


Figure 3.34.: Block diagram Scaling Unit

resolving part. The number of bits considered in the multiplication is configurable. This enables the balancing between the accuracy of the multiplication and the area consumption to fit the application best. In contrast to a conventional integer multiplier followed by a shift operation (how it is usually done in SW solutions) the "Scaling Unit" has distinct advantages. It supports high order magnitudes, while providing low mantissa accuracy to save area. The "Scaling Unit" comprises out of three main building blocks. They are preprocessing, low precision multiplication, and postprocessing (depicted in Figure 3.34). These are discussed in more detail in the following.

Multiplier Block The multiplier executes the multiplication on the mantissa of the scaling factor in unsigned integer mode and the dataset with reduced bitwidth (see the pseudocode in Algorithm 14). The exact bitwidth is configurable. To utilize the multiplier in an optimized way, which achieves a suitable accuracy, the considered bits of the dataset have to be selected carefully. This task is done by the "preprocessing block". That strategy causes a distortion of the magnitude. The "postprocessing block" restores the appropriate result.

$$mul\_res = mul\_factor1 \times mul\_factor2;$$

Algorithm 14: Calculate mul_res, inputs: $mul\_factor1, mul\_factor2$

Preprocessing Block This block separates the mantissa and exponent of the scaling factor (shown in Algorithm 15). Further, it detects if the dataset is a negative value in case of an signed operation mode. Signed negative datasets are treated via 2's complement conversion in unsigned mode. The scaling factor is considered as positive value only. The support for negative values is not implemented.

The limited accuracy during multiplication, caused by the reduced bitwidth in favor of saving area, makes the realignment of the dataset mandatory. Thereby the non-zero bit with highest significance is identified. The bits of the dataset are shifted to the left until the first bit with value "1" at the MSB. For the mantissa no realignment is needed, since *IEEE754* specifies a implicit leading "1" for the mantissa generally. Thus, the original mantissa extended with a leading "1" is directly used as factor for the multiplication. Those modifications ensure the highest possible accuracy during multiplication. The factor bits used during the multiplication are utilized from direction of the MSB downwards. This preserves consistent data.

$e = a[30:23];$
$m = a[22:0];$
$s = a[31];$
if $op_{signed} \wedge s$ **then**
  $x_{abs} = 2's\_complement\_conversion(x);$
else
  $x_{abs} = x;$
end
$amnt_{preshift} = 31 - find\_index\_of\_highest\_bit\_set\_to\_one(x_{abs});$
$x_{preprocessed} = x_{abs} << amnt_{preshift};$
$mul\_factor2 = x_{preprocessed}[31 : 31 - (accuracy - 1)];$
$mul\_factor1 = (1\&m)[23 : max(0, 23 - (accuracy - 1))];$

Algorithm 15: Calculate $mul\_factor2, mul\_factor1$, inputs: x(int), a(float), op_{signed}, accuracy

Postprocessing Block The pseudocode of the "postprocessing block" is shown in Algorithm 16. It takes the multiplication result and applies another realignment, which depends on the value of the MSB. In case the MSB of the multiplication result is set to "1" no realignment is needed. The MSB value depends on the exact values of the multiplication input factors. Regardless of the operation mode (i.e. signed, unsigned), the multiplication is executed on unsigned data. In order to convert the unsigned multiplication result back to signed data, one additional bit on the MSB position is needed. Therefore, a "0" bit is prepended.

Because of required realignments in signed mode, the original exponent of the scaling factor needs to be modified in order to reconstruct the proper magnitude. The realigned multiplication result gets shifted with the postshift amount. A positive value represents a rightshift, a negative one a leftshift. Applying an additional leftshift causes data-overflow, and therefore is considered is an illegal operation. This is signaled via a dedicated overflow signal. In that case, the data is saturated. Furthermore, for signed negative results the 2's complement conversion is applied, and the extension to the output bitwidth is done accordingly to the mode.

Activation Function

The "Activation Fct" stage comprises of two components. The complexity of the chosen activation function determines, whether "Clipping & Saturation" is sufficient, or "Linear Interpolation" is required. Both components support bypassing, which allows the programmer to select the way of function representation. The "Clipping & Saturation" is rather small and fast in compare to "Linear Interpolation". In contrast, it is only applicable to a very limited number of functions. Conceptually, the "Linear Interpolation" component can support all of the known activation functions. The more complex computation and setup routines are its disadvantages. In the following, both approaches are discussed.

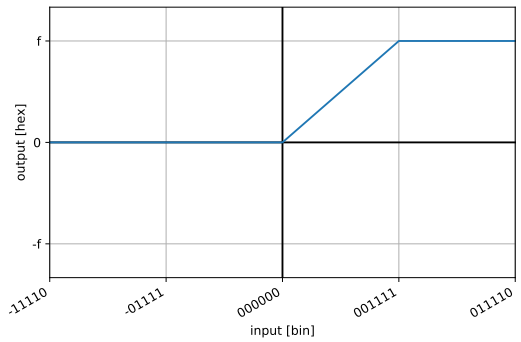
Clipping and Saturation Figure 3.35 provides samples of functions, which are covered by the "Clipping & Saturation" unit, while exploiting the provided features. The component expects signed input data and returns data either in signed or unsigned domain. This option is selectable via the configuration. Figure 3.35a and 3.35c depict the unsigned mode, whereas 3.35b and 3.35d cover the signed mode. Furthermore, two different clipping masks m_{clip} are applied, 3.35a and 3.35b use $\texttt{0x}000f$ as mask, which clips the value to the least significant four bits, while 3.35c and 3.35d utilize a wider clipping mask ($\texttt{0x}0fff$), which returns the least 12 bits.

In case the input value exceeds the boundaries defined via the clipping mask, the output saturates to its corresponding boundary. For unsigned mode the lower bound is zero, whereas the mask constitutes the upper bound. In signed mode, the boundaries are derived from the clipping mask instead of direct usage. Hereby, the clipping mask defines the active interval. The positive and negative domain are with the same magnitude, considering 2's complement paradigm.

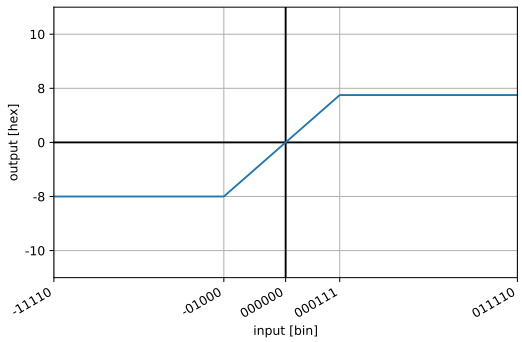
The Equation of the "ReLU" function is given in (3.13). "Clipping" fulfills all requirements to cover the "ReLU" function. The lower bound is considered to be zero. The saturation to the upperbound of the "Clipping & Saturation" unit is not represented with the "ReLU" function. Instead, this feature of the "Clipping & Saturation" unit ensures the fit of the data into a certain dataformat (e.g. "unit8" holds only the eight least significant bits). If more bits are required to represent the value, data corruption can happen. In this case, saturation is applied. It allows the software to handle those cases consistently.

$$ReLU(x) = \begin{cases} 0 & x \leq 0 \\ x & x > 0 \end{cases} \tag{3.13}$$

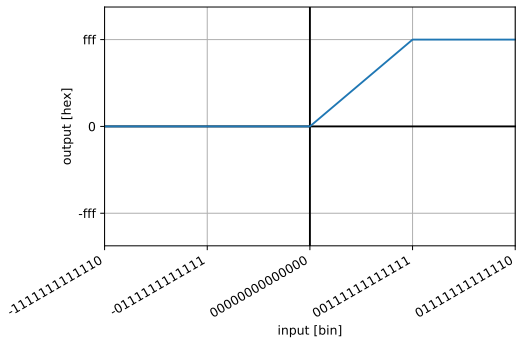
Linear Interpolation To extend the supported activation functions to nonlinear functions, a hybrid method using linear interpolation is developed. It's inspired by variations of the RALUT and PWL methods, which are introduced in section 2.3.5. Thereby, the emulated function is



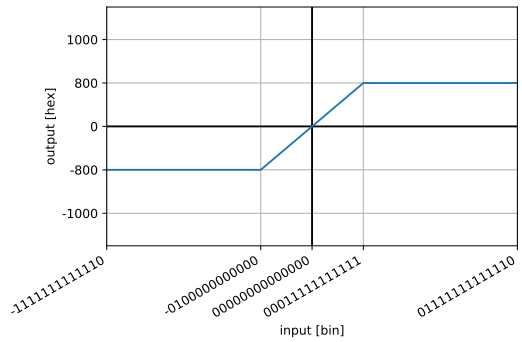
(a) Unsigned mode with clipping mask `0x`$000f$



(b) Signed mode with clipping mask `0x`$000f$



(c) Unsigned mode with clipping mask `0x`$0fff$



(d) Signed mode with clipping mask `0x`$0fff$

Figure 3.35.: Clipping and Saturation for different configurations

partitioned into linear segments of nonuniform ranges. This approach allows to increase the density of interpolation points at nonlinear ranges. This increases the accuracy and supports nearly linear ranges with a low number of interpolation points. That fosters cost efficiency. The proposed technique is designed to enhance proper resource allocation.

In order to maintain on-chip area and flexibility, additional function properties can be optionally deployed, such as symmetry, offset and reprogramming. The component is capable of supporting both, origin symmetry and y-axis symmetry, defined by (3.14) and (3.15) respectively. It allows to account HW only for half of the interpolation points for symmetric functions. Additionally, the offset feature can be used for functions that are symmetric to a point or axis different from the origin and origin axes. It allows to shift the function into the origin, which enables the deployment of the symmetry feature. Additionally, for functions displaced far off the origin the offset enables the storage of reduced bitwidth interpolation points, at the cost of an additional subtractor. Since the activation function may change from layer to layer, multiple different functions need to be covered for NN-model inference computation. Therefore, this approach optionally provides reconfigurable interpolation slopes, y-intercepts and lookup-values.

$$Origin\ Symmetry : f(-x) = -f(x) \tag{3.14}$$

$$Y - Axis\ Symmetry : f(-x) = f(x) \tag{3.15}$$

The interpolation process is separated into five independent steps to cover the selected features. The pre- and postprocessing steps enframe the remaining steps. Hereby, the optionally deployed symmetry and offset features are resolved. Furthermore, the input is translated into the domain of the encoded interpolation and backwards after application of the interpolation. Range-Decoding is the first step of the actual interpolation operation. It determines the range to which every input x belongs, by using comparators. For every range, one individual slope and y-intercept value is stored in an optionally reprogrammable lookup table. Those values together with the input x are used in the next step to compute the function value for the given input x. Therefore the slope is multiplied with the input x in the shared scaling unit, which was introduced earlier in this chapter. Afterwards the result is added together with the y-intercept.

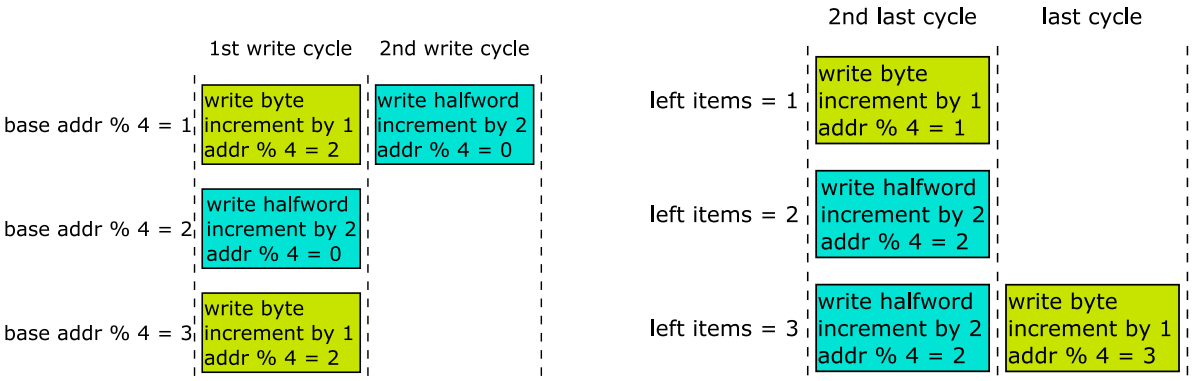
Address Generation

The Output Activation Streamer supports the generation of linear address patterns, since the order of computed output activations matches their placement in memory. In the common case, the activations are cached until a complete word is ready to be written to memory. After the execution of the memory write, the address gets incremented by four.

There are some corner cases during the run-in and run-out phases, where the increment deviates from that rule. The baseaddress of the output activation array may be a byte aligned address. This creates a need for additional increment patterns, treating byte and halfword accesses. These accesses types are utilized, until a word address got hit the first time, as depicted in Figure 3.36a. Without such mechanism, data already stored in memory but located at one of the preceding bytes to the baseaddress would be overwritten accidentally by writing the entire word. Similarly, in case of the run-out phase, byte or halfword accesses need to be applied in case the remaining datasets don't complete an entire word. To prevent unexpected over-writings of the memory content, which is located in the succeeding bytes of the assigned output activation array (as depicted in Figure 3.36b), the address generation patterns deviate from word-wise increments and do byte or halfword accesses instead.

if $MSB(mul\_res)$ **then**
  $aligned\_result = HEAD(mul\_res, accuracy)$;
else
  $aligned\_result = mul\_res[accuracy + min(accuracy, 24) - 2 :$
   $min(accuracy, 24) - 1)$;
end
if op_{signed} **then**
  $res_{signed\_aligned} = (0 \& aligned\_result)[accuracy : accuracy - 1]$;
else
  $res_{signed\_aligned} = aligned\_result$;
end
$amnt_{postshift} =$
 $((amnt_{preshift} - (e - (127 - 1 - (32 - accuracy)))) + !MSB(mul\_res)) - op_{signed}$;
$res_{postshift} = res_{signed\_aligned} >> amnt_{postshift}$;
if $op_{signed} \wedge s$ **then**
  $res\_sel = 2's\_complement\_conversion(res_{postshift})$;
else
  $res\_sel = res_{postshift}$;
end
$ovf = amnt_{postshift} < 0$;
if ovf **then**
  $saturation\_value = compile\_saturation(op_{signed}, s, accuracy)$;
  $res = saturation\_value$;
else $!ovf$
  $res = res\_sel$;
end
if op_{signed} **then**
  $res_{extended} = sign\_extend(res, 32bit)$;
else $!op_{signed}$
  $res_{extended} = zero\_extend(res, 32bit)$;
end

Algorithm 16: Calculate $data_{scaled}$, inputs: $mul\_res, accuracy, amnt_{preshift}$



(a) run-in-phase (b) runout-phase

Figure 3.36.: Address-increment and accesswidth of the Output Activation Streamer

Packing

The "Packing Unit" collects activations in a cache register, and forwards them as soon the write condition is fulfilled. During run-in and run-out phase the write condition may change since byte or halfword accesses are needed, depending on the featuremap alignment. The according condition is driven by the "Packing Management". It resolves the run-in and run-out routines.

Termination

When reconfiguring the Output Activation Streamer for the next output feature map, it is important to ensure that any cached output activation gets written to the memory before moving on to the next configuration. Otherwise, inconsistent data may be loaded in the computations for the next layer. Therefore, a terminate flag is introduced, which is realized via CSR. This flag enables the streamer to identify when to reject further activations. Unless the transaction for the last dataset of the previous configuration has finished, the streamer resides in this mode (terminate signal high). When all streamer contents are cleared, the terminate CSR gets reset by the HW automatically. After that, the streamer is in a valid state to do reconfiguration or continuation.

3.2. Computation Unit

The basic operation of the inference is the matrix-vector multiplication or dot-product (inner-product), which is introduced in section 1.3. For an efficient acceleration, there exists a strong demand for an appropriate unit, which is able to deal with the mentioned operation in an

optimized way. This is key to achieve good overall system performance. Otherwise, the matrix-vector multiplication operation constitutes a bottleneck in the design and subsequently limits the overall system throughput. The matrix-vector multiplication (given in (3.17)) boils down to a dot-product operation, which is expressed by consecutive multiplication and addition operations. A MAC unit realizes the combination of both operations in a common unit. In the following, the multiplier architecture and the corresponding adder trees as well as their combination into the MAC unit are discussed.

$$AW = A\begin{bmatrix}\bar{w}_1 & \bar{w}_2 & \cdots & \bar{w}_l\end{bmatrix} \tag{3.16}$$

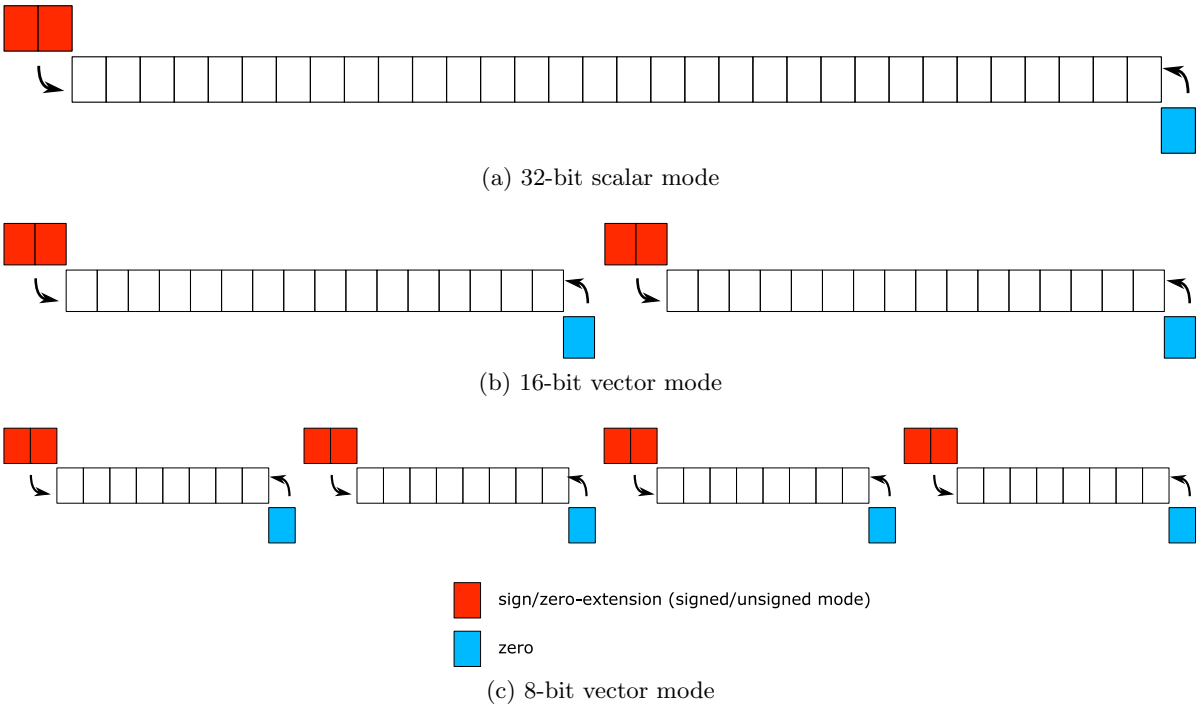
$$A\bar{w} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{1n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \end{bmatrix} = \begin{bmatrix} a_{11}w_1 + a_{12}w_2 + \cdots + a_{1n}w_n \\ a_{21}w_1 + a_{22}w_2 + \cdots + a_{2n}w_n \\ \vdots \\ a_{m1}w_1 + a_{m2}w_2 + \cdots + a_{mn}w_n \end{bmatrix} \tag{3.17}$$

3.2.1. Vector Multiplication

The multiplier integrated into the accelerator needs to be with short critical path and low area overhead, due to the strict resource limitations and performance requirements of edge platforms. The goal is to simultaneously support various fixed-point data formats, with reduced precision according to NN model parameterizations, and full-precision data, to handle side-band activities on the processor. The proposed computation unit is designed to be shared between the host-processor and the accelerator in tightly coupled systems to avoid duplication of modules in favor of a reduced on-chip area and optimized utilization of available computation resources.

According to literature (section 2.3.2), the fastest available multipliers utilize Booth encoding while achieving competitive on-chip area footprints. The original Booth encoded multiplication is limited to treat signed data only. In a multiplier, which is used for NN-acceleration applications, also support of unsigned data is necessary. Therefore, the method proposed in (Rajput & Swamy 2012) is applied to extend the functionality accordingly. In order to enable its operation in vector mode (e.g. two times 16-bit inputs to a 32-bit multiplier), this work modifies the Booth encoding and the treatment of partial products.

Figure 3.37 shows the modification of the original 32-bit multiplier input for the Radix-4 Booth encoding to guarantee the correct extraction of Booth codes depending on the chosen vector mode. For the scalar mode, the two sign/zero-extension bits are inserted to the leftmost bits. In signed mode the sign extension is selected, whereas in unsigned mode the zero extension is chosen. One additional bit with value “0” is appended to the least significant position, see



(a) 32-bit scalar mode

(b) 16-bit vector mode

(c) 8-bit vector mode

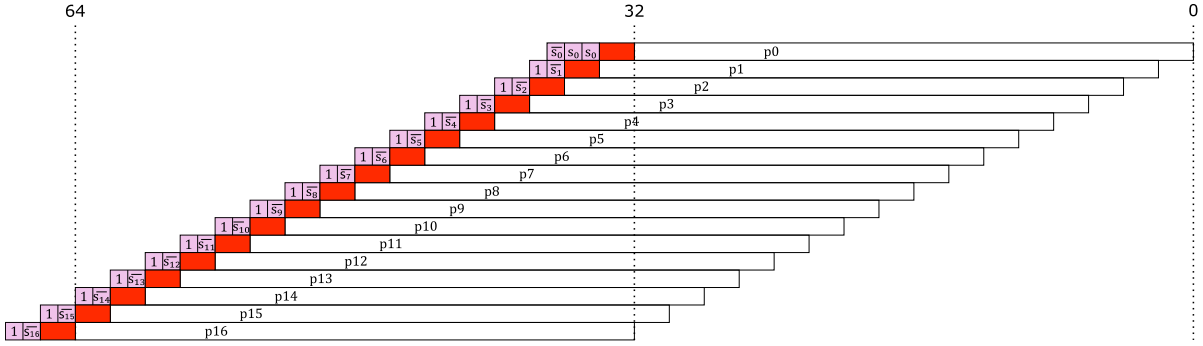
Figure 3.37.: Multiplier modification for Radix-4 Booth encoding of different vector/scalar formats

Figure 3.37a. For the 16-bit vector mode the same modifications are applied twice at each halfword boundary (shown in Figure 3.37b). Similarly, Figure 3.37c shows the modifications for 8-bit vector, applied to every byte boundary.

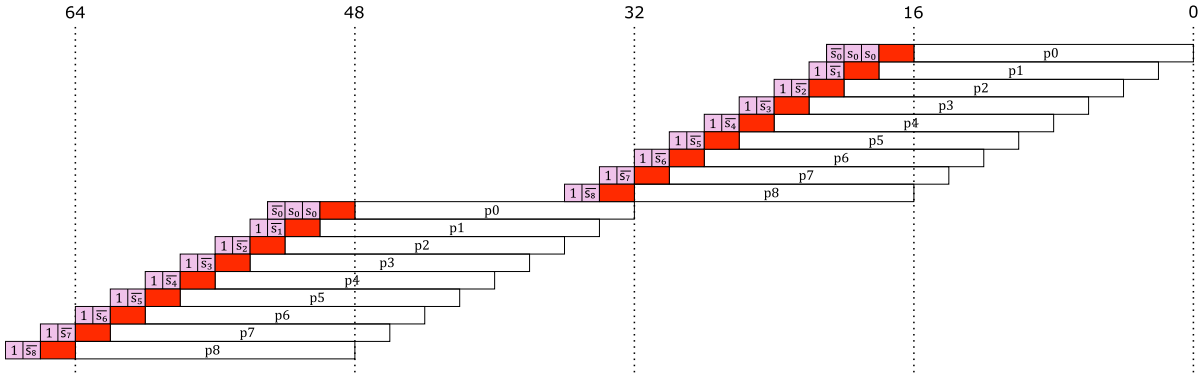
Since the vector modes increase the input bitwidth to the Booth encoder, additional encoder logic is required to support those formats properly. Furthermore, the application of smaller vector formats produces more Booth codes, which creates additional partial products, shown in Figure 3.38. Hereby, the scalar 32-bit format requires one additional partial product (see Figure 3.38a). The 16-bit vector format requires two additional partial products (see Figure 3.38b), and the 8-bit vector format produces four additional partial products (see Figure 3.38c).

When creating a Booth multiplier, which is capable of handling multiple vector formats, the propagated bits for the partial products are selected for a certain vector mode via multiplexers. The multiplexed bits for an example of an 16 and 8-bit vector multiplier are shown in Figure 3.39.

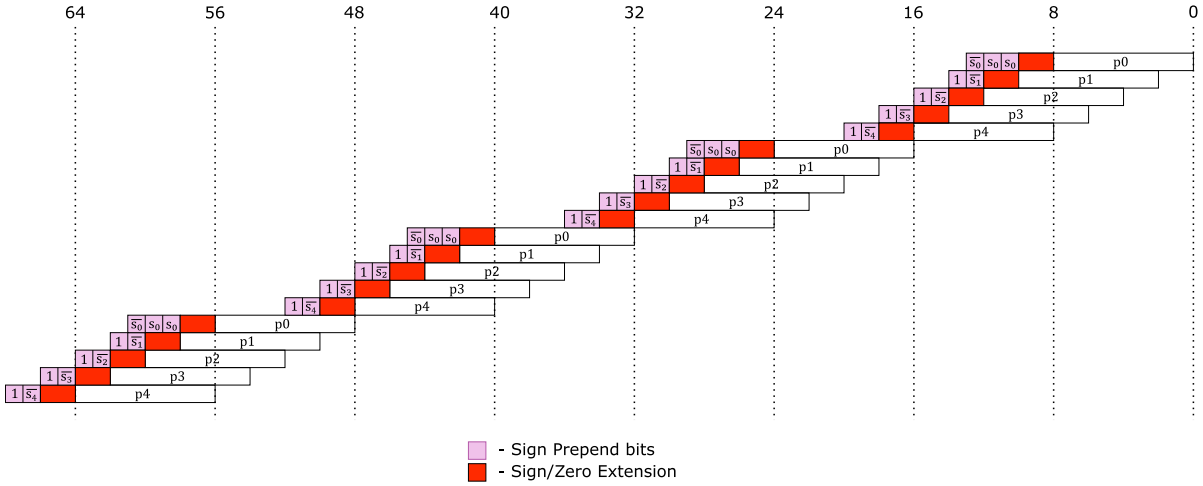
To obtain the final multiplication result, the individual partial products are added using a modified vector tree adder, which is explained in detail in the following section.



(a) 32-bit scalar mode



(b) 16-bit vector mode



(c) 8-bit vector mode

Figure 3.38.: Partial product combination for different vector formats in Radix-4

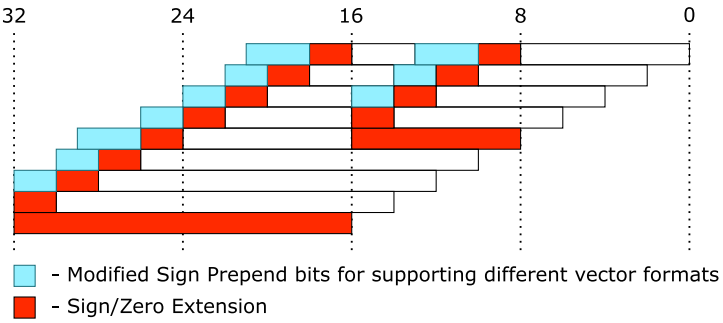


Figure 3.39.: Merged partial product generation for 16/8-bit Booth vector modes

3.2.2. Vector Wallace Tree Adder

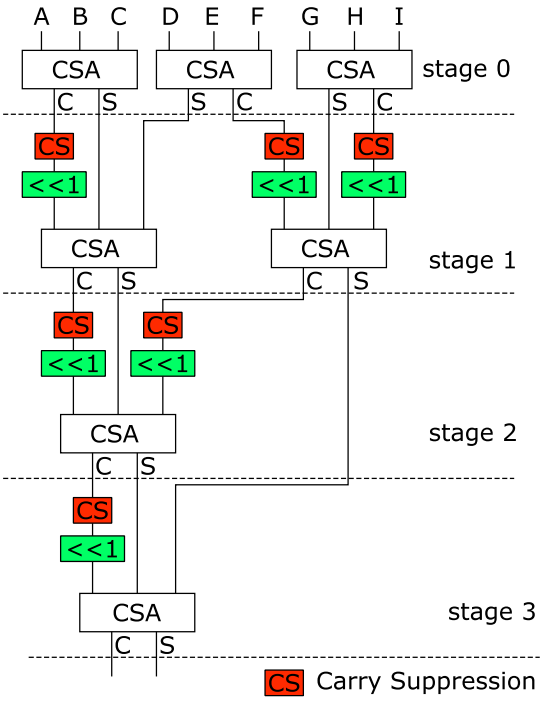
To the Wallace Tree (introduced in chapter 2) modifications are applied, which extend its functionality to execute the addition operation also on vectors. Thereby, the available logic is entirely reused and the increase of complexity is minor.

The block diagram in Figure 3.40a shows the structure of the modified Wallace tree. Since the Wallace tree is based on CSAs, the independent sum and carry outputs are generated for each addition. The carry is responsible for propagation. A suppression of the carry propagate at the vector bounds avoids interference between the individual vectors, see Figure 3.40b. In order to switch dynamically between the supported vector modes, the suppressed carry-bits have to be selected according to the chosen format. The dynamic masking is realized via logic AND-gates.

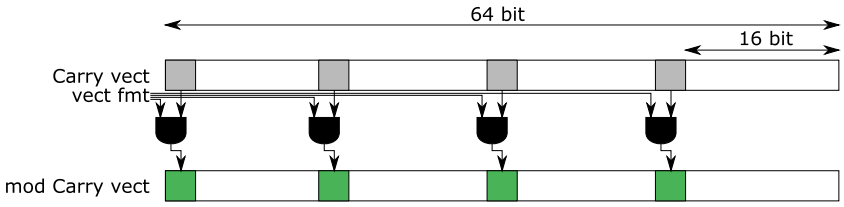
The proposed modified Wallace Tree is used in the Booth-based vector multiplier to enable the parallel addition of the partial products for the individual vectors. Therefore the vector width of the Wallace Tree is chosen to be of double the width of the multiplier input vector format. This is due to the coherence of multiplication input and output domains. The result of a multiplication may take up to double of the bitwidth of the individual inputs to entirely represent the resulting domain (see also Figure 3.38).

3.2.3. Multiply Accumulator

The Multiply Accumulator is a combination of the Booth-based vector multiplier and another vector Wallace Tree, combined according to Figure 3.41. Thereby, the multiplier calculates the vector-wise multiplication results between the individual vectors of the input multiplier and multiplicand. In order to compute the dot-product, multiple of those vector-wise multiplication results need to be accumulated. This is done by another Vector Wallace Tree, which is capable of treating proper vector formats matching the vector multiplier output formats. The number of vectors, which can be treated simultaneously by the Multiply Accumulator, typically is less than the number of factors of a dot-product in NN inference. To address this issue, an iterative approach is chosen, where a local register caches the intermediate accumulation value of the



(a) Modified Wallace tree with carry suppression for vector support



(b) Carry suppression

Figure 3.40.: Vector Wallace tree

ongoing dot-product computation. In the subsequent cycle the current accumulation value is provided as bias. This pattern is repeated until the iteration across the dot-product inputs is completed and the final accumulate value is determined.

The event of an occurring overflow of the accumulate register or the vector Wallace Tree might happen, since the number of accumulate iterations isn't known apriori and the final accumulate value cannot be predicted properly. The dimension of the executed NN-layer and the exact weight and activation distribution influences the required accumulator domain. Since the width of the corresponding HW has to be chosen in advance, a dynamically occurring overflow cannot be avoided. To avoid hidden data corruption, the Multiply Accumulator is implemented in a transparent way and signals such overflows to the host. Saturation of the accumulate value to the maximum represented value is applied in such cases.

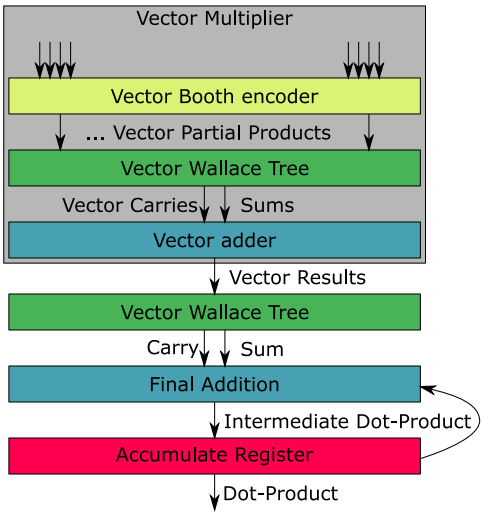


Figure 3.41.: Structure of the vector Multiply Accumulator

4. Hardware/Software Interaction

In section 1.4.8 a low barrier HW-SW interplay is suggested. In this chapter, the thesis describes this concept in detail and demonstrates its realization on the proposed NN-acceleration platform. Thereby, the interconnect and interplay of the accelerator module with the program execution on the processor is discussed, also known as the programmers view. It summarizes the corresponding communication and configuration registers as well as the definition of appropriate custom instructions used to control the accelerator module. Furthermore, the SW-routines are revealed, which supplement the accelerator-HW to complete the NN-execution.

4.1. Integration of the Streamers in the data pipeline

According to chapter 3, the data interfaces of the streamers are realized as input[1](Output Activation Streamer) or output[1] (Weight and Input Activation Streamers) registers using a ready/valid handshaking protocol. This synchronizes the program flow on the processor with the independently acting streamer and allows to elastically access streamer data from processor perspective, as shown in Figure 4.1. During design, every of the instantiated streamer gets assigned with a unique address. It is used to identify the different streamers in SW later. For that address space 32 bits are reserved. That allows to handle up to 2^{32} independent streamers. The input and output registers of the streamers can be understood to play a similar role as the general purpose RF to the CPU. They either provide the operators used in the execution stage of the CPU or are used as destination register for the computation results. The instruction encoding defines, whether to interpret a particular address as RF or streamer in-/output register address.

[1]From the streamer's point of view

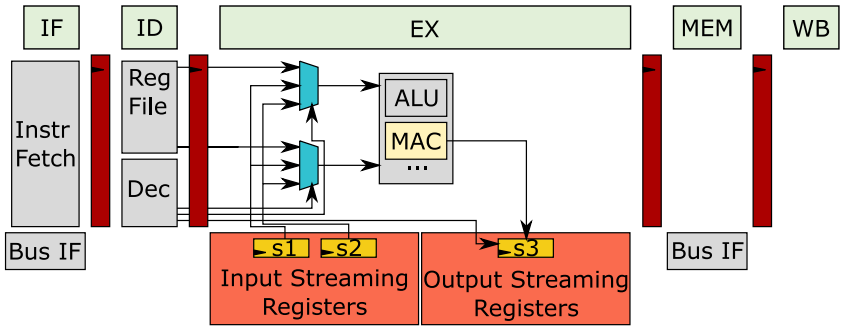


Figure 4.1.: Streamers in programmers view/model

Standard instructions don't allow to access the streamer in-/output registers, instead custom instructions defined later in this section are provided for their access. Further, the streamer in-/output registers are either read-only (Weight and Activation Streamers) or write-only (Output Activation Streamer). The read-only registers comprise an auto-update functionality. Since data-hazards cannot appear on streaming registers, due to read-only/write-only property, the data is inserted to the CPU-pipeline directly to the execution unit. This maximizes the number of cycles that the streamer may use to prepare data. Further, the input registers of the Output Activation Streamers are linked to the result ports of the computation units directly. This grants time to the Output Activation Streamer to perform additional postprocessing.

4.2. Streamer Configuration and Status Access via CSRs

The configuration and status of the streamers is set and tracked via additional registers. Those can be conceptually either MMIO mapped for standalone setups or integrated as custom CSRs to the CPU. In the later case, they are accessible via standard instructions of the RISC-V "zicsr" extension. Configuring the streamers via CSRs has a significant advantage over MMIO programming, since less cycles are required. Additionally changing particular bitfields inside a CSR register is supported at instruction level, whereas the realization in MMIO deploying systems requires the resolution of bit-manipulations via slower SW routines. For frequent reconfiguration MMIO constitutes a bottleneck.

Every streamer type requires a certain set of CSRs for configuration. In the following, the meaning of each individual CSRs with respect to the corresponding streamer is explained.

4.2.1. Common CSRs

Some of the CSRs are existing in any type of streamer. In the following, the functionality of those is described.

EnRst has two bitfields, one for "enable" and another for "reset". Setting both bitfields simultaneously starts the streamer from a cleared state. Re-enabling the streamer after a preceding "disable" by setting only the "enable" bitfield without setting the "reset" bitfield causes the streamer to continue from the last state before disabling. Setting the "reset" bitfield as well, resets the streamer immediately regardless of the "enable" bitfield. If enabled during reset, the streamer starts with the initial state, according to the configuration data.

BaseAddr Indicates the address of the first weight/activation/output-activation in memory.

4.2.2. Weight Streamer

In the following list, those CSRs are described, which are unique to the Weight Streamers.

Amnt Defines the number of weights to fetch from memory. When the corresponding weights got delivered, the streamer resets the "enable" bitfield of the "EnRst" CSR.

Offset Determines the value, which gets applied onto the weights stored in memory. It's used to map the encoded values in optimized domain range, back to their original domain for proper computation.

Modes Sets the interpretation of the weights to unsigned or signed.

4.2.3. Input Activation Streamer

In the following list, those CSRs are described, which are unique to the Input Activation Streamers.

FrameHeight Defines the height of the frame in elements, which slides across the feature map. This value has to match the corresponding kernel properties, which is specified in the corresponding Weight Streamer. For FC layers, the parameter matches the "FmapHeight" CSR, since kernel and featuremap are of the same size by definition.

FrameWidth Defines the width of the frame in elements, which slides across the feature map. This value has to match the corresponding kernel properties, which is specified in the corresponding Weight Streamer. For FC layers, the parameter matches the "FmapWidth" CSR, since kernel and featuremap are of same size.

Repeats This value defines the number of deliveries done for a certain frame alignment. This is especially useful, when applying multiple kernels to the featuremap (e.g. in Conv).

IdxStartCol Identifies the alignment of the frame on the featuremap in width dimension. Hereby, the first frame element is used as a reference.

IdxStartRow Identifies the alignment of the frame on the featuremap in height dimension. Hereby, the first frame element is used as a reference.

FmapWidth Defines the width of the featuremap. For a multi channel featuremaps the channels and width are folded into the same dimension according to $W * C$

FmapHeight Defines the height of the featuremap.

Offset Similar to the Weight Streamer, the value gets applied to every activation value.

PadValue For size equivalent convolutions, where the dimensions of input and output featuremaps match exactly, padding is applied. Padding is supported with constants only. Note: When padding featuremaps with zero, the padding value has to be equivalent to "-Offset", since the offset is applied to any activation (also padded ones) later on.

SparsityBaseAddr Defines the memory pointer to the sparsity vector.

Modes The "Mode" CSR of the Input Activation Streamer has two additional bitfields, namely "SparseEn" and "SparseRepeats", compared to the Weight Streamer (which has only the "Signed" bitfield). Those additional bitfields allow to configure the streamer for use without sparsity support and defines if the sparsity vector needs repetitions. This is especially useful for applications, where the same kernel gets applied to the same frame in the feature map.

Done Signals the event to the CPU that all activations for the given configuration have been delivered. It can be polled by SW to track the execution status.

4.2.4. Output Activation Streamer

In the following list, those CSRs are described, which are unique to the Output Activation Streamers.

EnRst Additionally to the bitfields introduced in the subsection 4.2.1, a "terminate" bitfield is available. When set, it forces the Output Activation Streamer to push all its cached output activations to the memory, before accepting new input data. Before switching to the next layer, the programmer has to make sure that all input activations are available to avoid glitches, by using that bitfield.

ScalingFactor The incoming inputs to the Output Activation Streamer are multiplied with a floating point value. It's meant to be used for treating quantization folded with normalization.

Config Comprises the two bitfields "Signed" and "ActivationFctEn". The second determines, if a function (activation function) is applied to the scaled activation before writing it back to memory.

Offset Similar to the Input Activation Streamer, the offset gets applied to each activation and transfers the activation into an efficient domain. This reverses the offset applied in Input Activation Streamer.

ClippingMask A bitpattern is expected that defines the saturation value. All activations, which exceed the range are saturated to the maximum or minimum of the chosen domain. The programmer can use it for implementation of a "ReLU" activation function, or ensuring the bitwidth of the data, which gets written to memory.

f7	src2 (reg/stream)	src1 (stream)	f3	dest (thread)	opcode	instruction
31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0	
0000000	rs2	rs1	000	mt	0001011	mac
0000000	rs2	rs1	100	mt	0001011	macr
0000001	rs2	rs1	000	mt	0001011	macu
0000001	rs2	rs1	100	mt	0001011	macru
0000010	rs2	rs1	000	mt	0001011	macsu
0000010	rs2	rs1	100	mt	0001011	macrsu

Table 4.1.: non-streaming MAC instructions

InterpolationLUTwdata Used to programm the RF for the interpolation-HW available for the activation function.

InterpolationLUTaddr Controls the address of the RF write of the interpolation-HW for the reconfigurable interpolation slopes, y-intercepts and lookup-values

XOffset For activation functions, which are shifted in order to deploy symmetry features, the input data needs to be shifted in the domain of the activation function to ensure proper evaluation. This CSR defines the shifting distance used to map the data into the activation function domain.

4.3. ISA Extensions

In this section, the custom instruction set extensions are presented, which are developed to drive the accelerator through SW. Additionally, instructions of the standard C-extension are useful, since those allow to reduce the program footprint and to have other beneficial advantages (see subsection 4.3.3).

4.3.1. MAC and SIMD Multiply Custom Instruction Set Extensions

The MAC-unit is a shared component, which is visible and accessible by both CPU and accelerator. To access the MAC unit via CPU, appropriate instructions have been defined, see Table 4.1. As explained in chapter 3 the MAC unit comprises of multiplier and adder. From CPU perspective, supporting the MAC unit with special instructions is mandatory to deploy its benefits, but sharing the multiplier for the standard multiplication operation (see Table 4.2) is also desirable. This generates synergies, since another dedicated multiplier unit is saved.

f7	src2 (reg/stream)	src1 (stream)	f3	dest (thread)	opcode	instruction
31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0	
1010100	rs2	rs1	000	rd	1110111	smul8
1010100	rs2	rs1	001	rd	1110111	smulh8
1010100	rs2	rs1	010	rd	1110111	sumul8
1010100	rs2	rs1	011	rd	1110111	sumulh8
1010100	rs2	rs1	100	rd	1110111	umul8
1010100	rs2	rs1	101	rd	1110111	umulh8
1010000	rs2	rs1	000	rd	1110111	smul16
1010000	rs2	rs1	001	rd	1110111	smulh16
1010000	rs2	rs1	010	rd	1110111	sumul16
1010000	rs2	rs1	011	rd	1110111	sumulh16
1010000	rs2	rs1	100	rd	1110111	umul16
1010000	rs2	rs1	101	rd	1110111	umulh16

Table 4.2.: SIMD-multiplication instructions, inspired by RISC-V P-extension

Every MAC unit contains a dedicated register, to store the intermediate accumulate value produced by the preceding sequence of mac instructions. This implements the dot-product between the referenced data vectors. Since an instantiation of multiple MAC units may be useful for an increased throughput or supporting various formats, each of them is addressable via a unique thread identifier *mt*.

The custom extension for the MAC instructions are placed in the opcode domain "custom-0", which is a RISC-V defined prefix, reserved for custom extensions. The instructions use two input arguments from the RF as sources, which are addressed via "src1" and "src2". The value of the according RF registers are multiplied together and added with the MAC thread register (accumulate register) value. The thread register is identified via the *mt* parameter in the "dest" bitfields. The result of the addition gets stored into the same *mt* register for the next iteration. Those instructions are also available for unsigned and signed/unsigned data (e.g. "smacu" and "smacsu" respectively) analogous to the RISC-V M-extension terminology. The symbol "r" in the instructions mnemonic (e.g. macr, macru,... see Table 4.1) identifies instructions with a similar operation as assigned to the instruction without the "r" in the mnemonic (e.g. mac, macu,...). The difference between them is the additional reset of the MAC thread register before accumulation for those with a present "r". This is needed, when a new sequence of accumulations is started for another dot-product computation.

The (non-ratified) RISC-V P-extension provides instructions for packed data formats, similar to SIMD. In this work, the SIMD multiplication instructions slightly deviate, due to more restricted HW specifications. Nevertheless, they follow the proposal of the P-extension wherever reasonable. The RISC-V implementation, which is used as basis for the accelerator, is of the 32-bit version. It allows to write one register of the RF per cycle only. Therefore, the instructions of the P-extension are extended analogous to the standard in the M-extension, where the instruction selects the write of the upper or lower half of the multiplication result. Since the

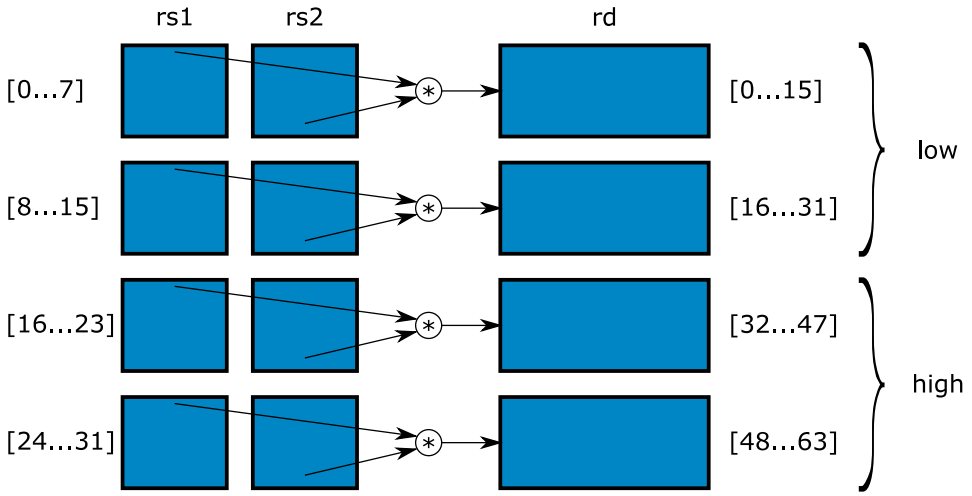


Figure 4.2.: SIMD-multiplication of mul8 and mulh8 (Ashok 2021)

processed input data is in SIMD format, also the output data is in SIMD format, according to Figure 4.2. As an example, the 8-bit SIMD multiplication is separated into a consecutive "smul8" and "smulh8" sequence. This also allows to reduce the delay of the multiplier, since only two of the SIMD results need to be computed in parallel, instead of four.

4.3.2. Streaming Instructions

The streamer data is accessible via custom instructions as well. The input streamers provide data via their output registers. The SW can pop that data by using one of the custom instructions defined for that purpose. All of those instructions are grouped into a common custom extension. Similar to the input streamers, the output streamers are accessible via particular instructions of that extension. The mentioned extension is called "Streaming"-extension.

The instructions of that custom extension follow the RISC-V "R-type" 32-bit format. The register address bitfields of the instructions refer to the RF registers, which may hold the corresponding streamer addresses. The MAC-thread is encoded directly in the instruction bitfield reserved for "dest". The function fields "f7" and "f3" allow the selection between using a referenced register value for computation directly or interpretation as streamer address. The deployed base opcode maps into RISC-V "custom-0" field.

The streaming instructions can be categorized into three groups. All of them access at least one streamer module. The first group of instructions, shown in Table 4.3, uses one or two inputs from an input streamer, i.e. "ss1", "ss2". Alternatively, the RF can serve as second source instead of the second input streamer "ss2". Between the inputs and the intermediate MAC accumulate value, a MAC operation is applied. It combines the multiplication of the input data and an addition of the multiplication result with the selected MAC-thread "mt" value.

f7	src2 (reg/stream)	src1 (stream)	f3	dest (thread)	opcode	instruction
31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0	
0000000	rs2	ss1	001	mt	0001011	smac
0000000	ss2	ss1	011	mt	0001011	ssmac
0000000	rs2	ss1	101	mt	0001011	smacr
0000000	ss2	ss1	111	mt	0001011	ssmacr
0000001	rs2	ss1	001	mt	0001011	smacu
0000001	ss2	ss1	011	mt	0001011	ssmacu
0000001	rs2	ss1	101	mt	0001011	smacru
0000001	ss2	ss1	111	mt	0001011	ssmacru
0000010	rs2	ss1	001	mt	0001011	smacsu
0000010	ss2	ss1	011	mt	0001011	ssmacsu
0000010	rs2	ss1	101	mt	0001011	smacrsu
0000010	ss2	ss1	111	mt	0001011	ssmacrsu

Table 4.3.: Streaming instructions with MAC support

f7	src2 (reg/stream)	src1 (stream)	f3	dest (thread)	opcode	instruction
31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0	
0000011	mt	rs1	001	sd	0001011	sdacadd
0000011	mt	ss1	010	rd	0001011	sacadd
0000011	mt	ss1	011	sd	0001011	ssdacadd
0000011	mt	rs1	101	sd	0001011	sdacaddh
0000011	mt	ss1	101	rd	0001011	sacaddh
0000011	mt	ss1	111	sd	0001011	ssdacaddh

Table 4.4.: Streaming instruction with accumulate support on MAC-thread registers

After that, the addressed MAC-thread register gets updated with the result of the addition. This group of instructions doesn't trigger an update of the MAC-unit's output ports, but only update the internal state (accumulate register) of the selected MAC-unit.

The second category of instructions is presented in Table 4.4. It includes all instructions, which take the MAC-thread register value and add either an input streamer value or RF entry. The MAC-thread register is updated with the addition result subsequently. Differently from the previous category, the MAC-unit outputs the addition result, which allows to either store it into the RF or push it to one of the Output Activation Streamers. The target selection for the "dest" bitfield is encoded in the instruction opcode via the "f3" bitfield. For the MAC thread register inside the MAC-unit a width greater than 32-bit may be needed. In order to treat such wide data, instructions with "h" postfix ("high", similar to RISC-V M-extension) are introduced. They allow to fetch the bits exceeding those lower 32-bits, which are generally forwarded.

f7	src2 (reg/stream)	src1 (stream)	f3	dest (thread)	opcode	instruction
31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0	
0000101	n.d.	ss1	000	rd	0001011	sload
0000101	n.d.	ss1	001	sd	0001011	smove
0000101	n.d.	rs1	010	sd	0001011	sstore

Table 4.5.: Plain streaming access instructions

The last category is depicted in Table 4.5. It ensures maximum flexibility and allows to access the streamer data independently. Hereby no computation is involved and data movement between streamers and RF is performed exclusively. Those instructions require one source and one destination, only. There are instructions available, where a streamer serves as source and the RF as destination and vice versa. Additionally, the "smove" instruction is available, which involves only the streamer input and output registers. This group of instructions allows to quickly load or store data from memory to RF and backwards by using the streamers. Further, fast memory relocation is supported through "smove".

As discussed in chapter 3, one important accelerator feature to increase utilization and throughput of the computation HW is SIMD processing. Nevertheless, the particular SIMD format is not explicitly encoded in the instructions above. Instead, the individual SIMD format is specified for streamers and computation HW through CSRs. It's up to the programmer to select proper addresses and threads for matching combinations of streamer and MAC unit with respect to SIMD configuration. This procedure reduces the utilized opcode encoding space significantly, without reducing the supported number of features. Since the SIMD formats usually stay untouched within a layer, the configuration effort is low.

4.3.3. Compressed Instructions

The C-extension is a standard RISC-V extension, and is not part of the contribution of this thesis. Nevertheless, the reason for the usage of compressed instructions in the accelerator setup is discussed in this subsection.

The instructions, belonging to this extension are encoded in a compressed format. Instead of 32-bits, it requires only 16-bits to encode them. That's why they are more memory efficient and also reduce the bandwidth on the instruction bus. Since both modules access the same bus and memory, execution of compressed instructions leaves more cycles to the weight fetching streamers. That is because one memory fetch contains two instructions, which relaxes the demand for instruction fetching.

Because of the limited number of bits reserved for the C-extension, the encoding space and the size of the operand fields is rather small and assembled in complex patterns. Therefore, the decoding process is more expensive, than for common 32 bit instructions. Additionally, the

number of supported instructions is comparatively small and only frequently used instructions can be included. Nevertheless, the typical code size reduction when using compressed instructions was found to be approximately 25% (Waterman 2011). Since the application is on the edge, this is a great opportunity to tweak the accelerator platform for better efficiency and speed.

4.4. Accelerator-specific Kernels

In this section, an abstracted program flow is presented, which enables FC and Conv layer execution on the accelerator. The particular steps can be either triggered through SW (e.g. a program executed on the tightly coupled RISC-V processor) or a state-machine. The overall picture of the program flow and the particular steps are discussed in the following.

4.4.1. Fully Connected Layer

Figure 4.3 depicts the program flow for the FC layer evaluation. The overall flow is separated into thirteen steps, each of them is indicated with an index. The colors of the individual steps are chosen according to the involved components (refer to the legend in the image). Between particular steps backward paths are available, which visualize loops. The following list elaborates on the individual steps:

Step 0 It extracts the relevant layer parameters like tensor shapes, offsets, memory pointers to kernel and input/output data, activation function details and further facility parameters required to properly setup the accelerator components.

Step 1 It ensures that the streamers are in a clean state in case previous accesses didn't finish properly or missed to reset the streamers. This step is not contributing to the actual setup required for execution of the next sequence of inference operations, but belongs to housekeeping.

Step 2 The streamers receive their initial parameter configuration derived from the layer parameters extracted in step "0". In this step, the Weight Streamer is configured with kernel size, offset and sign mode of the corresponding weights. Additionally the Input Activation Streamer gets configured with width and height dimensions of the feature map, the number of repetitions a single footprint should be delivered, the footprint alignment, replacement value of padded datasets, offset, sign mode and the pointer to the sparsity map for the corresponding kernel. The Output Activation Streamer receives information about the scaling factor (fuses quantization and normalization), the offset, a clipping and saturation mask, the sign mode and activation function related configurations.

Step 3 The Output Activation Streamer receives the memory pointer to the location, where the processed output activations need to be transferred to.

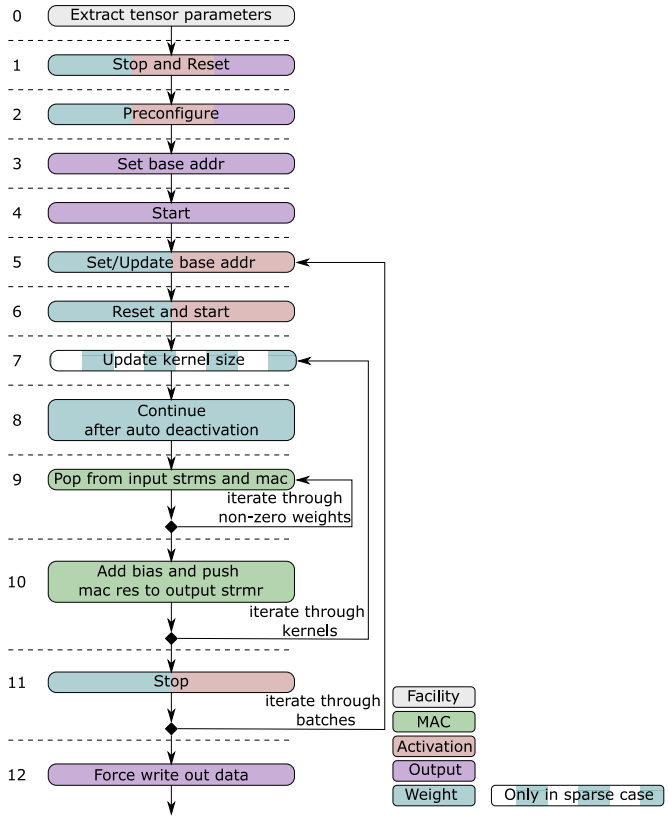


Figure 4.3.: Program execution of the fully connected kernel

Step 4 The Output Activation Streamer is enabled. From now onward, it processes every received dataset and writes it into the assigned memory array in consecutive order.

Step 5 Also the input streamers get linked to the corresponding weight and activation locations in memory. Due to more frequent updates of that parameters during program execution, the configuration of the memory pointers is separated into this step instead of including it into step “2”. The parameters configured in step “2” stay untouched, while performing updates needed during looping, where step “5” acts as target.

Step 6 After completion of the configuration, also the state of the input streamers are reset and re-enabled.

Step 7 An optional step, which is only required when executing layers with weight sparsity. Due to weight sparsity the amount of weights per kernel batch with given dimensions is not a constant anymore. Instead it changes on per kernel basis. Therefore, this step updates the configuration for the amount of kernel weights, which are non-zero and therefore placed in memory. When the desired amount of weights is reached, the Weight Streamer takes care of the correct alignment and disables itself. In case of non-sparse kernels, this parameter doesn’t change across the kernels inside a batch and therefore don’t need to be updated. Thus, this step can be omitted.

Step 8 The Weight Streamer deactivates itself, when the configured amount of weights got delivered. When continuing with the next kernel, it only needs to be re-enabled. Continuation of an enabled streamer doesn't effect the streamer. During the first loop iterations of kernel and batch such a scenario occurs.

Step 9 The actual payload computations of the dot-product are performed here. Thereby, the weights and activations are popped from the streamers, inserted to the MAC unit and accumulated to the MAC thread register. This operation is performed iteratively until all non-zero weights and corresponding elements of the featuremap got processed.

Step 10 Finally, a bias (compare to equation (1.8)) is applied to the accumulate register and its result is pushed to the Output Activation Streamer. This result is equivalent to a particular output activation of a FC-layer. In order to compute the complete set of output activations, iteration through all kernels is done.

Step 11 The computation for one batch of the FC layer is done and the Input Activation Streamer is stopped, either for reconfiguration or program termination. In case other batches are remaining, step "5" follows and the input streamers are going to be reconfigured.

Step 12 The program ends with forcing the Output Activation Streamer to write all it's remaining data to memory. This is needed to finalize the processing of the current layer and reaching a clean state to proceed.

4.4.2. Convolution Layer

The terminology used in Figure 4.4 is equivalent to the one used in the previous section 4.4.1. Analogous to the program flow explained in section 4.4.1, steps "0"-"4" are executed to preinitialize and start the accelerator. The following list elaborates on the remaining steps:

Step 5 It is utilized to configure or update the memory pointer of the Input Activation Streamer with every batch.

Step 6 The program significantly differs from the kernel of the FC layer. Hereby, two loops are entered, which slide the footprint of the kernel across the activation featuremap. First the kernel slides across the x-dimension, followed by a shift in the y-dimension in a repetitive pattern. Every time the footprint is moved the Weight Streamer state is reset, since the same weights have to be delivered repetitively across all footprints.

Step 7 Updates the Input Activation Streamer configuration, which identifies the footprint location, according to the stride for the next kernel placement.

Step 8 The Input Activation Streamer is restarted.

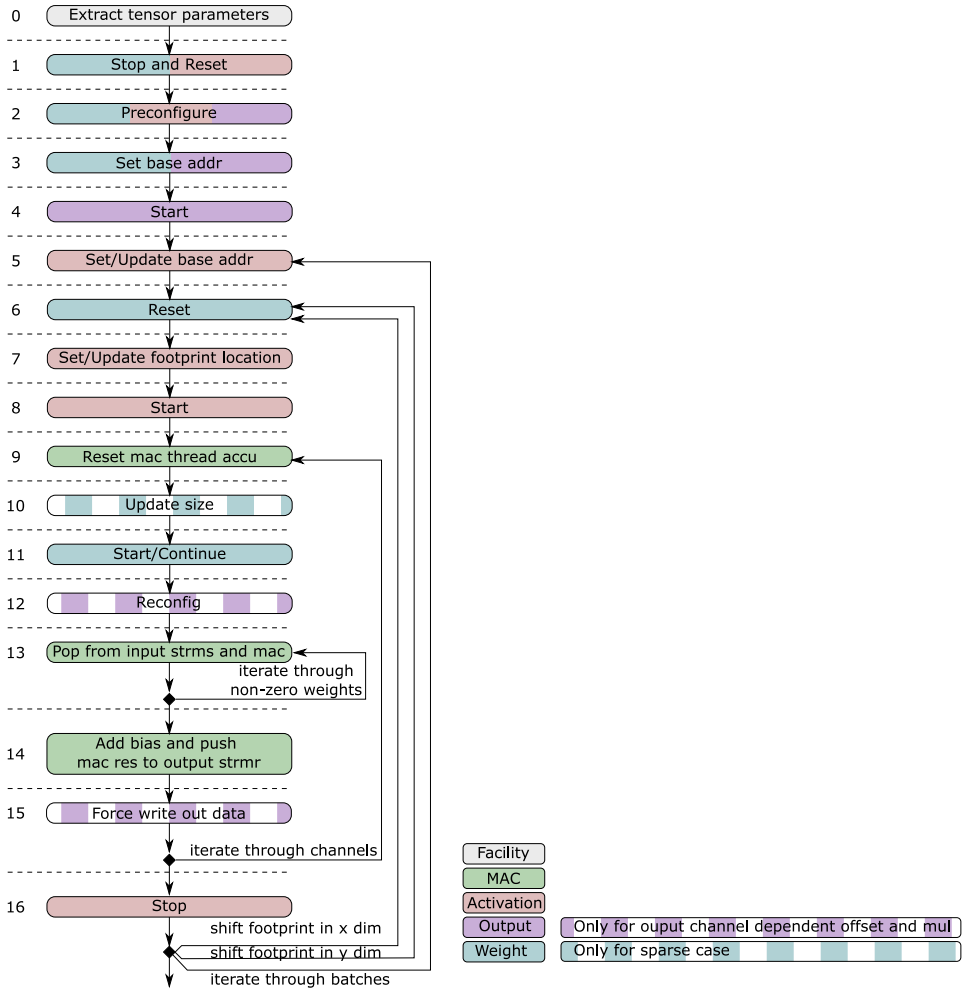


Figure 4.4.: Program execution of the convolution kernel

Step 9 The next loop is entered, which iterates through kernels. Therefore, the MAC-thread accumulate register is reset to capture the next sequence of accumulations for the first dot-product (output activation) of the next output channel.

Step 10 An optional step, which is required for sparse kernels only, analogous to step "7" of the FC-layer kernel.

Step 11 The Weight Streamer is continued after auto-deactivation, once the configured amount of weights is reached.

Step 12 The reconfiguration of the Output Activation Streamer with channel-wise offset and multiplication value is needed in case both change across the channels of the output activation feature map. Since those parameters are constant across the channels in many cases, this step is considered as optional.

Step 13/14 They are equivalent to steps “9” and “10” of the FC-layer kernel and constitute the accelerated unfolded loop.

Step 15 In case the output activation’s offset and multiplication factor aren’t channel dependent, this step can be omitted. Then, no updates to the Output Activation Streamer configuration are required and pending memory writes don’t need to be forced. Once step ”14”/“15” is finished, the output activation of a single channel is ready and the accelerator is switched to compute the next channel.

Step 16 In case the iteration through all channels has been done, the Input Activation Streamer is stopped for reconfiguration of the footprint location and batch. This is the last execution step of the SW-kernel for the Conv-layer program. The program finishes, when all batches where applied to all configured kernel placements.

5. AI Compile Flow

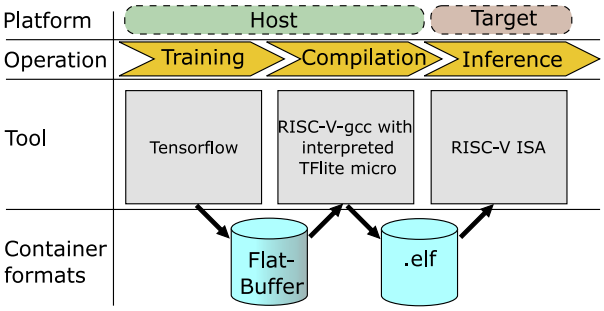
In this chapter, the deployed flow for mapping AI-SW to the developed HW-accelerator platform is presented. Hereby, the entire flow starting with raw training data towards a platform optimized ready-to-execute model is sketched and the particular steps are explained. Common mapping flows deploy the native TF compile flow, which executes an online interpreter during inference (shown in Figure 5.1a). Live interpretation makes the deployment simple and flexible, but comes with a significant overhead in memory footprint and performance. Instead, a modified flow with pre-interpretation and interpreter-less TFlite micro execution is developed to reduce the runtime overhead (e.g. housekeeping). The according flow is shown in Figure 5.1b. It depicts the involved steps, with corresponding operations, execution platforms, tools and container formats.

5.1. Tensorflow

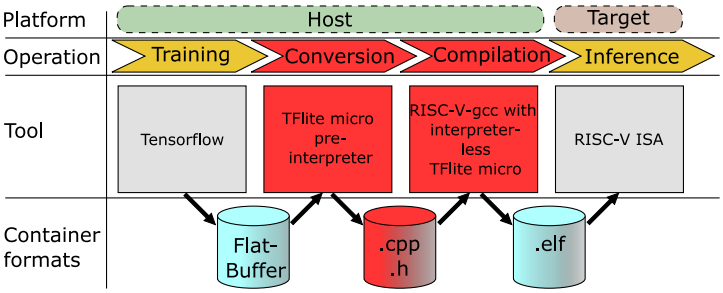
For model training, the flow deploys TF, a graph-based environment. Together with Keras, which adds advanced API features for model description to the default TF API, it constitutes a convenient, powerful and highly flexible training framework. TF is an open source platform, especially developed for the needs of ML and Deep Learning. It is accompanied by various libraries, which enhance extensibility.

The API exposes two major construction blocks (i.e. "Operations" *op* and "Tensors" t_x) to the user, which allow to describe the neural network as graph. According to Figure 5.2, "Operations" can be seen as the combination of the nodes t_0 and t_1 connected to a third node t_3 via edges. Typical "Operations" are matrix multiplications, additions, application of activation functions (e.g. ReLU) and more. Following the terminology of TF, "Tensors" are equivalent to nodes. A "Tensor" is a multidimensional container for values of constant or static type, which store handles to parameters and data arrays assigned to certain datasets. Imagine flowing data from the input tensors to the output tensor while transforming it in the operation. Due to such a visualization, the framework is called "Tensorflow".

The approach followed by TF enables major flexibility, since the integration of new operations is intuitive and can be implemented with low effort. In TF, platform specific "kernels" are defined for every operation individually. This allows to optimize the performance for any device. Such a degree of modularity provides the freedom to distribute the training process across diverse devices by plugging in new kernels (e.g. CPU, GPU, Googles TPU or at time-being even unknown platforms). Additionally, even the training process for a single NN can be separated on per operation basis and distributed across multiple devices. Such a solution



(a) Overview of the native TF compile flow



(b) Overview of the optimized AI compile flow

Figure 5.1.: Comparison of native TF and optimized compile flow

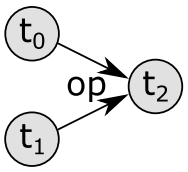


Figure 5.2.: Graphical view of TF in “Operations” and “Tensors”

has the potential to optimize the overall network training performance. In order to determine the best fit of mapping for a certain operation, every platform specific kernel is assigned with a performance number. The goal is to minimize the accumulated performance number for the entire network, to figure out the best device mapping. Also device transfers are modeled and are considered in the accumulated performance number. In case a certain device is not capable of executing a certain operation, this is modeled as well. In such cases the assigned performance number is infinite. This procedure may also exclude a certain device from possible mappings entirely, if too weak. (Abadi et al. 2016)

In the proposed AI-deployment flow (see Figure 5.1b), the training process is dispatched to a powerful host device, which is only available offline and allows to speed up the training process. Training is significantly more complex than inference. Once the model is trained, it’s exported into a “FlatBuffer”. This is an efficient portable format to store the model. In contrast to the native TF protocol buffer, the advantage of this format is it’s reduced footprint and simple accessibility. For the “FlatBuffer” no extra parsing is needed, which ensures efficient execution on devices with limited resources.

5.2. Tflite Micro Pre-interpreter

The native TF compile flow (shown in Figure 5.1a) includes the interpreter into the compiled target code. This allows to update the flatbuffer models during runtime, without recompilation of the TF lite micro framework. For inference platforms without strict resource limitations, this approach eases the deployment flow and ensures high flexibility. Since in this thesis the focus is on edge devices with strictly limited resources in computation capabilities as well as memory budget, separation of offline predictable routines allows to shrink the binary size and online computation efforts. These offline predictable routines are executed on a powerful host platform in a precomputation step, whereas only relatively static model-specific inference code remains to the actual target binary.

5.2.1. Concept

To achieve a reduction of the binary size, a pre-interpretation step is introduced. Its purpose is the offloading of offline predicable routines, which are involved in the TF lite micro framework, to the host system. Furthermore, TF lite micro comes with a bunch of libraries, which support a wide variety of operations. Since for a given network only a minor subset of operations is utilized and those are well known during compile time, the code base can be easily leaned by removing them. From computation perspective all relevant functions are still available, whereas a positive effect on the memory footprint is expected. In contrast, when deploying the interpreter based TF lite micro framework, the compiler is rather limited in removing those operations or even libraries since these are needed for providing the flexibility and network adaptability. The proposed pre-interpretation solves this issue, since the compiler analyzes the model and the required operations are selected appropriately.

Additionally, an online interpreter requires setup, initialization and preparation routines for every NN-model before inference execution. First, this requires some additional compute time, but the second effect is more crucial. Those SW-routines increase binary size, thus memory footprint, which is a very critical variable in the domain of edge devices. Removing those routines from online computation and executing them offline during pre-interpretation reduces the binary code size or creates space, which can be used for more advanced models.

The pre-interpretation is not time critical nor restricted by the limited performance budget of the target platform. Due to the offline pre-execution at the host, the limitations of the host device need to be considered instead, which typically are magnitudes stronger. The additional freedom can be invested to reach a more optimized model packing, in favor of a further reduced memory footprint. Especially the footprint of kernel structs can be optimized, since those are pre-computed and can be seen as static at the target platform.

Interpreted model execution requires runtime scheduling and memory allocation. During interpreter-less execution, scheduling and memory allocation is handled during compile time at the host platform, since all tensors are well-known in advance. This saves further computation time at the target platform, as well as memory footprint, which doesn't need to be spent for the storing scheduler and allocation routines.

As another side effect of the pre-interpretation, the model data are converted from flatbuffer format into C-arrays. This transfers the model from the encrypted TF format into a human readable format, which eases the validation process.

5.2.2. Pre-interpretation Flow

As discussed in the previous section, the pre-interpreter is an essential tool to make NN-models of advanced applications fit onto edge devices. In this thesis, the target platform is the accelerator presented in chapter 3. The steps comprised in the pre-interpreter are discussed by means of Figure 5.3. It ensures the coverage of the corresponding demands.

The input to the pre-interpreter is the NN-model stored in the TF flatbuffer format. Since the pre-interpretation is model and especially operation dependent, the first step is to load the model. Afterwards, the native TF lite micro interpreter executes the loaded flatbuffer model on the host platform. Hereby, the tensor dimensions are extracted. A subsequent interpreter run is executed, while persistent and scratch buffers are recorded. Both informations are required for memory planning, which is integrated in a plugin manner. This ensures a simple interchanging of the planner. The last three steps generate C++ files, which contains the pre-interpreted model data in a human readable format. First, the tensors get written in form of C-arrays into the new format. Afterwards, the model specific operations, which define the layer types and transform the tensors, are extracted under utilization of the memory planner and are added to the C++ source files. In the last step of the code generation, the setup routines used to initialize the model inference in C++ and the command to run the inference are added. As a result, the model is converted from the flatbuffer format into C++ source files, which contain all required information needed for compilation of the model inference program (Stevens et al. 2020).

5.3. Interpreterless Compilation

In the native TF lite micro compilation flow, the NN model in flatbuffer format is compiled together with the TF lite micro source for the interpreter. In this approach, interpretation is already executed in the previous pre-interpretation step and the interpreted model is available in a generated C++ file along with it's invocation code, see Figure 5.4. The compilation makes still use of the TF lite micro, but in a different way. Now, the TF lite micro is seen as library. Instead of including the entire pre-compiled framework, only the referenced code is included into the binary. This optimizes the binary size for a given NN model, since unused TF lite micro framework content (e.g. interpreter and unused operations) is identified by the compiler and excluded from the binary.

In order to ensure efficient mapping to the accelerator HW, it's mandatory to provide proper accelerator kernels to the TF lite micro framework, which the compiler can reference (compare section 4.4). In case proper kernels are missing for distinct operations, the fallback to the

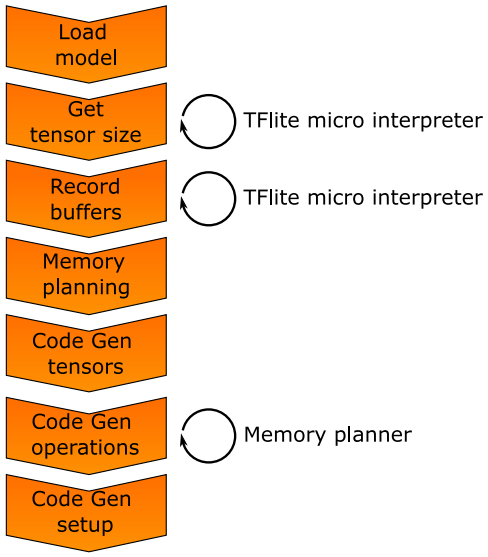


Figure 5.3.: Workflow of the TFlite micro preinterpreter

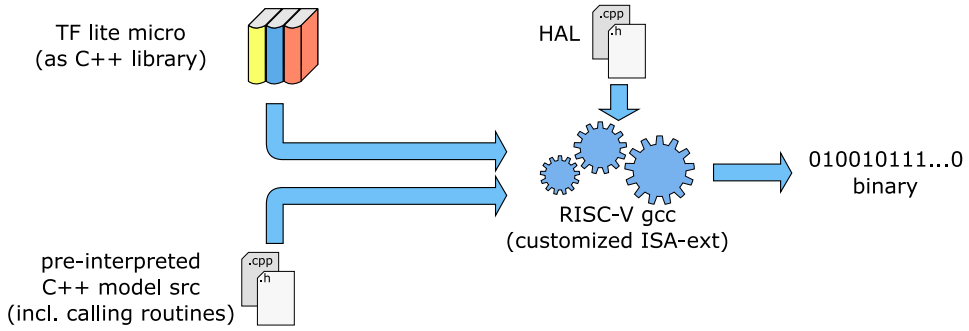


Figure 5.4.: Compilation of the C++ model together with TFlite micro as library

default CPU mapping is chosen. This ensures maximum model coverage with the drawback of bad performance for unsupported operations, but high performance for a range of widely used networks.

The accelerator developed in this thesis, is coupled to a RISC-V CPU. The chosen compiler is the according RISC-V gcc. In order to familiarize the compiler with the custom instructions and the accelerator interfaces, which are introduced in sections 4.1, 4.2 and 4.3, the compiler is patched accordingly. This ensures that the compiler is able to map the routines of the custom kernels properly onto the available accelerator HW.

Since the described compilation process is performed at the host platform, resource intensive optimization mechanisms of the compiler toolchain can be enabled. This further reduces the memory footprint of a particular model and the according kernels in contrast to an online interpreted execution.

6. Case Study and Evaluation

In this chapter, the developed accelerator platform is evaluated. In the first section, the on-chip area of the accelerator is measured for various configurations. These are compared against each other, to showcase the impact of the individual features. Afterwards, an exemplary application is introduced, which constitutes a vehicle for later performance and resource comparison between a conventional and the ML approach. Additionally, the impact of the pre-interpreter on the memory footprint is evaluated. The last section of this chapter focuses on the achieved acceleration factors for different layer types on various accelerator configurations, assuming variants of model parameters.

The following evaluation shows that the proposed NN-accelerator allows to execute inference layers upto 30x faster than on a 5-stage single core RISC-V processor with "rv32imc" configuration. The introduced area increase of the effective HW (excluding memory) is between 2x to 3x depending on the exact feature set. The full-featured accelerator including the RISC-V processor results in 61.9kGE.

The solution proposed in this thesis, increases the on-chip footprint of the effective HW but allows to drastically reduce the memory footprint. The memory saving attained through pre-interpretation, sparsity and properly featured HW results in 87.5% and by far exceeds the area increase caused by the extension of the effective HW.

The accelerator allows to execute the inference on a speed of upto $\sim 190MOPS$ referred to a clock speed of 60Mhz, where with the proposed HW much higher frequencies are possible at the same 40nm technology.

6.1. Area Analysis of the Accelerator sub-modules

In this section, the impact of the platform configuration on the on-chip area demands is discussed. Useful standard RISC-V processor extensions are monitored, as well as the proposed custom dot-product unit and different levels of acceleration, including HW-support for activation functions. The area results are extracted via Synopsis DC compiler for a 40nm ASIC technology at 60Mhz clock frequency.

Table 6.1 shows an excerpt of the synthesized design configurations. With increasing number and complexity of the enabled platform features, the area increases by a factor of ~ 3 from less than 20kGE to more than 60kGE. Hereby, the area is expressed in GEs.

The baseline constitutes the plain 5-stage RISC-V with Base ISA, indicated as "I". It has the merest compute performance, compared to any other configuration. It doesn't allow to dispatch operations of advanced complexity onto dedicated HW-modules (e.g. HW-multiplier), but supports an absolute minimum of operations. Instead, complex tasks are handled via SW-routines (e.g. SoftMul).

The next evaluated configuration is featured with the standard "C"-extension for support of compressed instructions. This requires more complex instruction decoding, due to the 16-bit format and refactored parameter encoding. Additionally, the instruction fetch becomes more complex, since instruction alignment restrictions on word-bounds are relaxed to half-word boundaries. According to Table 6.1, the additional on-chip area demand is within 3% compared to "I".

A major part of NN-inference related computations are multiplications and SW-emulated multiplications are time consuming. To address this, the standard RISC-V "M"-extension is enabled for the next performance level. The scalar 32-bit HW-multiplication requires a complex logic, which consumes 6.4kGE of on-chip area.

The multiplier introduced via "M"-extension is reused in the scalar custom "MAC"-extension. Hereby, the multiplier is shared between both extensions, thus doesn't contribute to the raised area demand. Instead, an accumulate register and an adder needed for performing MAC operations mainly contribute to the minor area increase of less than 1kGE.

The biggest step in performance but also area can be observed when enabling the streamers. The Weight and Input Activation Streamers together cause an increase of 19.2kGE. This is approximately the size of a 5-stage RISC-V Base ISA processor. Their performance potential can only be utilized when also enabling the SIMD feature of the dot-product unit, which requires another 4.4kGE. That is mainly driven by the complexity of the SIMD product addition and the increased multiplier complexity.

The Input Activation and Weight Streamers (A+W in Table 6.1) are capable of accelerating the input data delivery. The SIMD dot-product unit is meant to increase computation throughput. The missing piece for an entirely accelerated data path is the Output Activation Streamer. It increases the area demand by 10.4kGE, similar to each Weight and Input Activation Streamer, when supporting "ReLU" as hardwired implementation. For covering a wide spectrum of activation functions, the interpolation module proposed in chapter 3 is enabled. It requires additional 1.8kGE when configured for a reprogrammable leaky-ReLU.

More evaluations on the interpolation module are discussed by means of Figure 6.1. The main driver of complexity are the number of interpolation points and the degree of reprogrammability. The domain range (number of bits) are identified to be no significant contributor. The smallest footprint of the interpolation module is achieved, when dealing with static ranges and lookup values. With increasing reconfigurability in input ranges and lookup values the area demand raises rapidly. When dealing with four interpolation points, a fully static interpolation module requires ~ 0.7kGE. For dynamic ranges additional 50% area need to be allocated. Allowing reprogrammable lookup instead, raises the demand by more than 100%. Enabling full

I	C	M	MAC	Accelerator	ActFct	Gate Cnt 40nm technology [kGE]
X						18.3
X	X					18.9
X	X	X				25.3
X	X	X	scalar			26.1
X	X	X	scalar	A+W		45.3
X	X	X	simd	A+W		49.7
X	X	X	simd	A+W+O	ReLU	60.1
X	X	X	simd	A+W+O	Interpol	61.9

Table 6.1.: On-chip area for multiple platform configuration synthesized on the 40nm ASIC-technology for 60Mhz

reprogrammability in both domains creates additional $\sim$ 2.2kGE. The scaling unit is shown separately in Figure 6.1, since it's designed to be shared with the Output Activation Streamer. It contributes to $4.5 - 5.5$kGE of the Output Activation streamers area demand.

According to table 6.1, the dot-product unit contributes significantly to the overall on-chip area demand. To address this issue, a dot-product unit based on Booth encoding and Wallace Tree is proposed in section 3.2.3. Figure 6.2 shows a comparison of the proposed solution against an Array and Primitive multiplier based solution in scalar and SIMD-mode with different product adder methods. The Primitive multiplier doesn't identify a distinct multiplier architecture, instead it defines the operation as an abstract operator and leaves the selection of the best fitting architecture to the synthesis tool."mWT" indicates the modified Wallace Tree, which is proposed in section 3.2.2. "AC" is a chain of RCAs, which accumulate the individual vector results together with the intermediate register accumulate value.

Figure 6.2 points out the superiority of the proposed Booth based mWT architecture across all other evaluated solutions in terms of area demand, except for the scalar mode. Here, the synthesis tool is able to find a more area efficient solution for the Primitive multiplier. Across all multiplier architectures, the mWT shows the best area efficiency. The total area demand of a programmable SIMD dot-product unit, which simultaneously supports 32-bit scalar, two times 16x16-bit and four times 8x8-bit vector modes, is at $\sim$ 14kGE.

6.2. Knock Localization Application

In this section an application is introduced, which is suitable to showcase the validity of the proposed accelerator approach and to conduct a performance analysis. The chosen application predicts the origin of noise source in two dimensions, produced by a knock on a table surface.

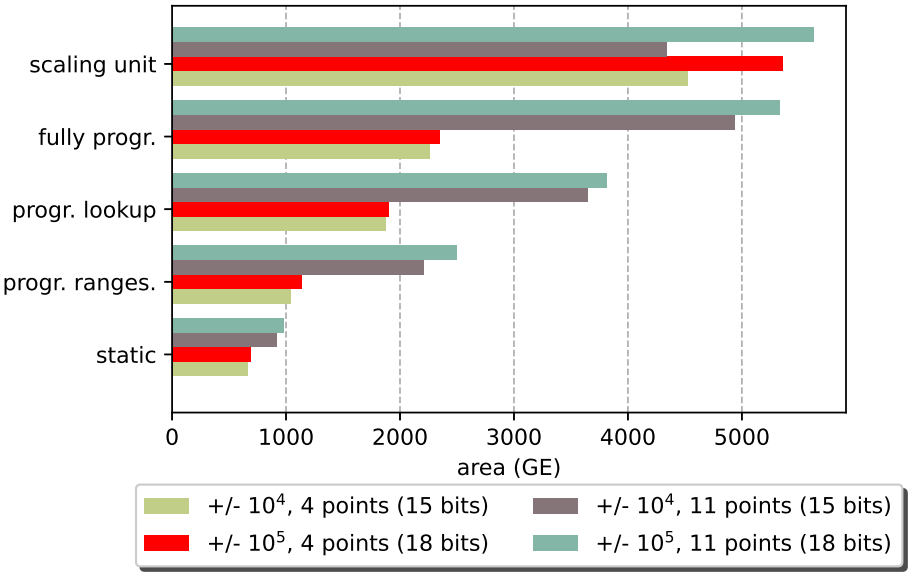


Figure 6.1.: Area demand breakdown for interpolated Tanh with 4 and 11 points for different x-domain ranges (Prebeck, Lawand, Vaddeboina & Ecker 2022)

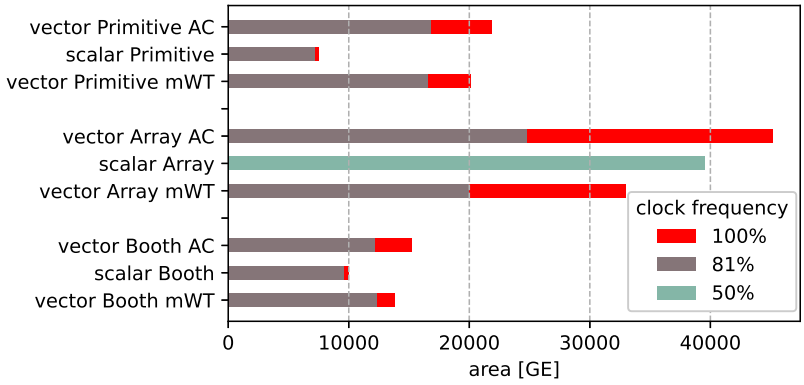


Figure 6.2.: Area comparison for various dot product units based on different multiplier and reduction tree architectures for 32-bit scalar and 32x32-bit, two times 16x16-bit, four times 8x8-bit vector modes, where 100% clock frequency maps to 150Mhz (Prebeck, Ashok, Vaddeboina, Devarajegowda & Ecker 2022)

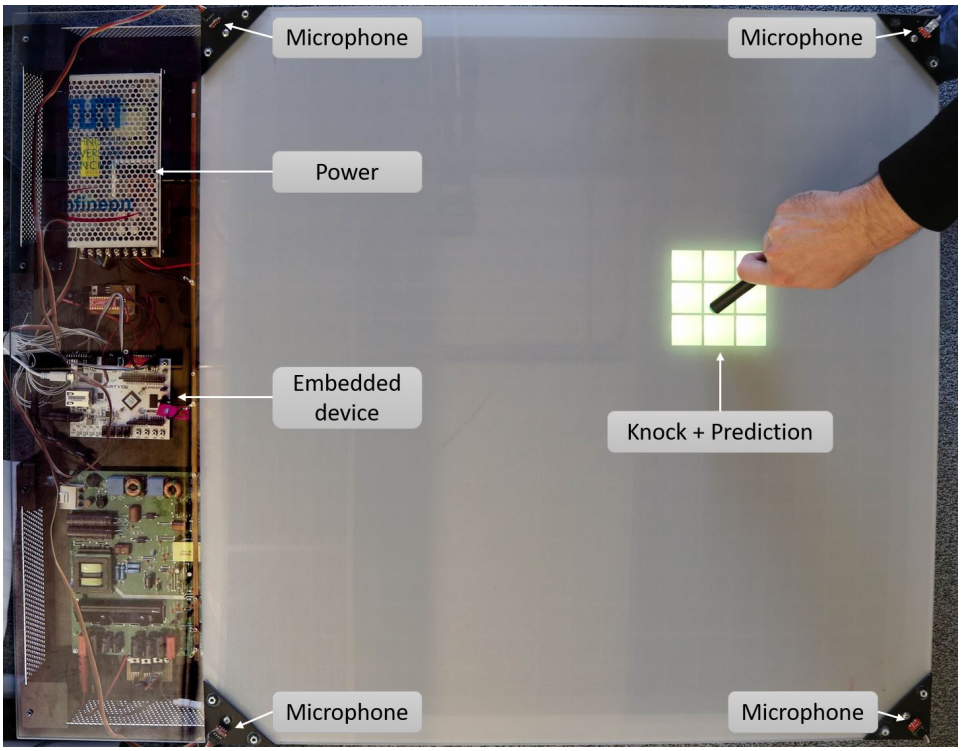


Figure 6.3.: Knocking table setup (Neumeier 2020)

6.2.1. Experimental Setup

In this experiment the noise source is chosen to be a knock onto a surface of a table. The table is of 80cm length in both x- and y-dimension. A microphone is mounted at every corner of the table (see Figure 6.3). The raw audio samples of the microphones are delivered to a stereo converter (IM69D130), which packs the data into I^2S protocol (Philips 1986). I^2S is a serial protocol mainly used in audio transmission. The I^2S signal is transmitted to an FPGA prototyping board (Xilinx Arty A7-100 Board, XC7A100T/35T), with an I^2S receiver and RISC-V processor. The processor is implemented on an FPGA (see Figure 6.4) and is featured with the tightly coupled NN-accelerator. The utilized FPGA allows to run the proposed platform with 40Mhz clock frequency. The available on-board memory is of 64KB ROM and 64KB RAM. For training and visualization purposes the table acts as low-resolution display with 20x20 pixels.

6.2.2. Conventional Localization Method

In order to evaluate the achieved accuracy, when using NNs for the localization, a conventional method of localizing a knock on the table is applied. This alternative approach relies on TDoA (Time Difference of Arrival) measurement (Singh & Agrawal 2013). Thereby, the cross correlation between the recorded audio samples of every microphone pair is calculated. The

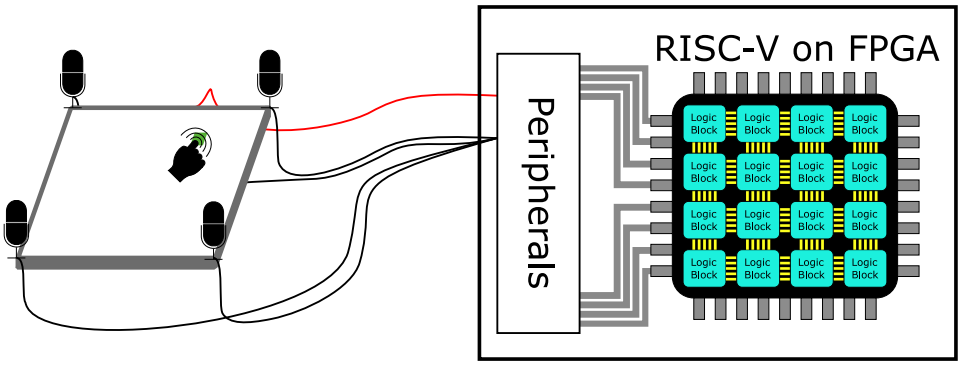


Figure 6.4.: Diagram of the knocking setup

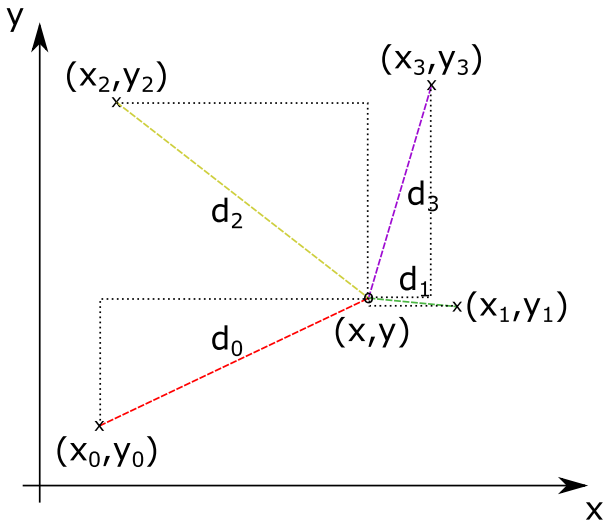


Figure 6.5.: Microphone source geometry

peak of the cross correlation determines the TDoA (Kulaib et al. 2011), which describes the time difference between arrival of the knocking sound between two microphones. Considering the traveling speed of noise in air, TDoA is converted into a spatial distance.

With the TDoA information of one microphone pair, the position of the noise source can be successfully determined in one dimensional applications. For two dimensional applications like this, at least a third microphone is required. This creates two further possible microphone pairs. The more microphones are available, the more accurate the position can be determined. An example of such a geometry, which comprises of four microphones, is shown in Figure 6.5.

The spatial distance d_i between the microphone position (x_i, y_i) and the origin of the noise (x, y) is described by (6.1), according to Pythagoras.

$$d_i = \sqrt{(x_i - x)^2 + (y_i - y)^2} \tag{6.1}$$

The spatial difference $\delta_{i,j}$ between the distance of the two microphones to the noise origin d_i and d_j can be formulated as (6.2).

$$\delta_{i,j} = d_i - d_j = \sqrt{(x_i - x)^2 + (y_i - y)^2} - \sqrt{(x_j - x)^2 + (y_j - y)^2} \tag{6.2}$$

TDoA can be converted from time domain into spatial distance, as explained earlier. Therefore, $TDoA_{i,j}$ has to meet $\delta_{i,j}$ (6.3) for every combination of microphone pairs as close as possible.

$$TDoA_{i,j} \approx \delta_{i,j} \tag{6.3}$$

Since measurements infer small deviations to the actual value, $TDoA_{i,j}$ and $\delta_{i,j}$ never match exactly. To account for this issue and to find the best coordinate match, cost function minimization (6.4) is applied.

$$c_{j,i} = TDoA_{i,j} - \delta_{i,j} = TDoA_{i,j} - (\sqrt{(x_i - x)^2 + (y_i - y)^2} - \sqrt{(x_j - x)^2 + (y_j - y)^2}) \tag{6.4}$$

For every microphone pair, such a cost function is available. The unambiguous origin can be found by calculating the optimum of the global cost with a minimum search algorithm. Therefore, the individual cost functions are combined (6.5).

$$c(x, y) = \sum_{c_{j,i} \in C} |c_{j,i}| \tag{6.5}$$

6.2.3. Collecting Training Data

For providing a ML-solution, off-the-shelf model training datasets are not available. Therefore, proper training data needs to be collected first. A small program is implemented to accelerate and simplify the data collection process. It randomly lights up any of the table's pixels and samples the audio within a three second time window. The program expects the knocking signal within this window. Afterwards, the snippet is analyzed for sanity and trimmed to a 42 ms long window of 1000 samples at 24KS/s, which contains only the knocking sound. This segment size is found to be sufficient for acoustic knock localization, while maintaining storage. The program assigns a label to each valid sample, which are chosen according to the light up pixel coordinates. The audio samples are stored as raw 16-bit pressure values in wav format.

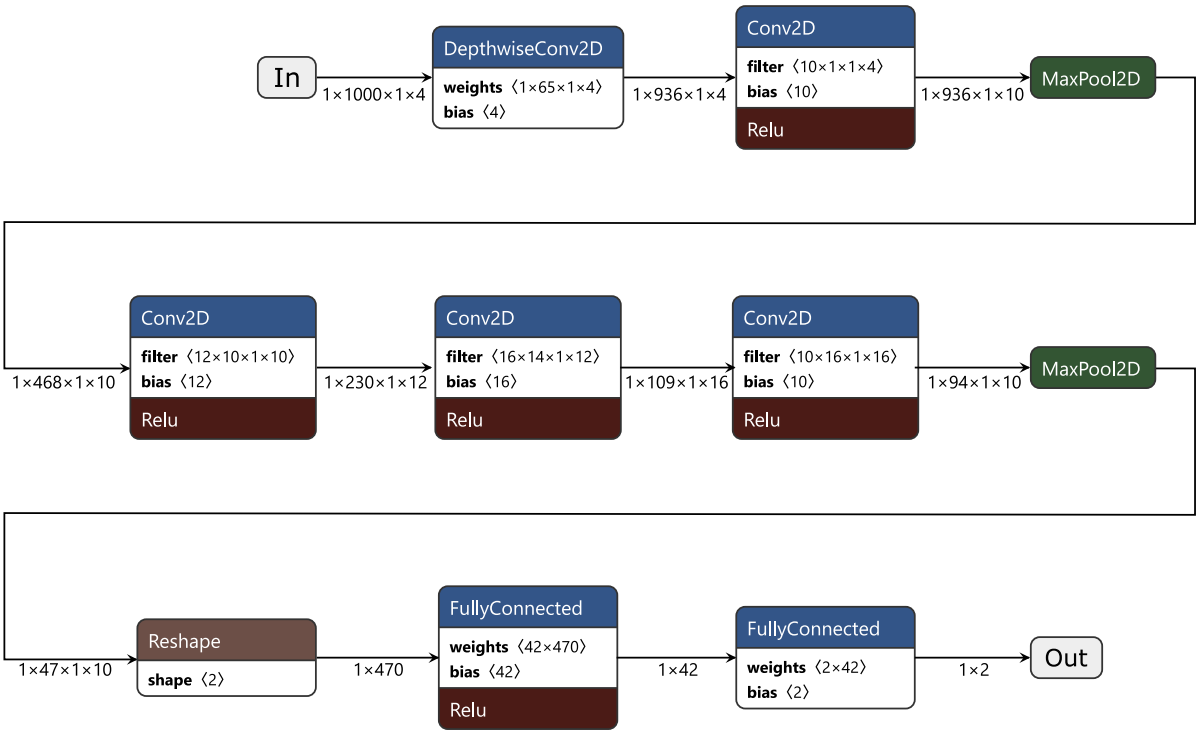


Figure 6.6.: NN-model for the knock localization application

6.2.4. NN-model architecture

The NN-model developed for the described application is depicted in Figure 6.6. It consists of a 2d-depth-wise convolution layer, which performs well for a huge amount of input values, since it reduces the number of required multiplications for the first layer. Afterwards, four 2d-convolution layers are placed, which are intersected with two intermediate MaxPooling layers. The last two layers are FC layers, which are utilized for the purpose of classification. As activation function, "ReLU" is utilized. Overall the model comprises 26797 parameters.

6.3. Conventional versus ML Approach

Both the conventional and the ML driven approach are executed on the RISC-V platform. The ML approach is separated into two steps, i.e. model training and inference. The model training is done offline, on a more powerful platform to reduce training time, whereas the inference is executed on the embedded platform. The following comparison accounts for the inference exclusively, which is a suitable assumption in edge computing applications. The deployed NN-model utilizes 8-bit quantization, to fit the very limited memory capacities of the edge platform, while maintaining accuracy.

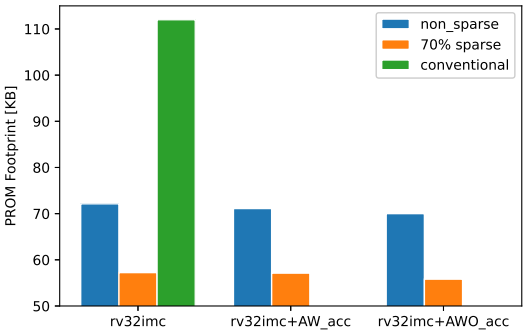


Figure 6.7.: Program Code Footprint of sparse/nonsparse model on different architectural configurations

	Computation Time
NN-model on rv32imc	620ms
conventional	53ms

Table 6.2.: Comparison of the computation time

Figure 6.7 compares the memory requirements of the conventional against the ML approach with different configurations. While the conventional approach exceeds the available platform ROM memory by far, the NN-model fits into it when deploying model weight sparsity. Without sparsity the model ROM footprint exceeds the 64KB target of the platform slightly. The ML approach gives room to optimize the footprint for the available platform memory at the cost of reduced accuracy, due to construction. The conventional approach relies on mathematical transformations. This doesn't allow to address the issue in a direct fashion. Furthermore, the involved operations in NN-inferences are rather regular but data intensive. Therefore, the control tasks are minor and the corresponding ROM footprint is low. The conventional approach is more control intensive instead.

A big disadvantage of the ML approach, especially on platforms that don't incorporate a specialized accelerator (e.g. the plain RISC-V), are the involvement of high computation efforts and data intensive operations. Both characteristics raise the computation time. In Table 6.2, the computation time of both methods are shown for execution on an unaccelerated RISC-V with "IMC" extensions. Hereby, the NN-inference consumes approximately 11.7x of computation time compared to the conventional approach. The measured time describes the actual computation and excludes data transfers to peripherals and more application specific overhead.

A significant advantage for the ML-approach can be observed, when comparing it's accuracy against the conventional method. According to Table 6.3, more than 90% of the localization predictions are located within a cell-distance of 3 pixels or less, when relying on NN-inference. The same percentage cannot even be reached with the conventional method within a cell-

Cell-Distance	0	1	2	3	4	5
NN-model	17%	73%	96%	99%	99%	99%
conventional	1%	10%	20%	34%	64%	81%

Table 6.3.: Knock localization accuracy for AI vs. conventional TDoA approach

distance of 5 pixels. A cell-distance of 3 pixels or less is hit in 34% of the cases. One reason of the bad performance of the conventional method is identified to be the noise transmission via the table itself. Because of different traveling speeds of noise in solid materials and air, interference disturbs proper audio sampling and its analysis. This application shows the superiority of ML, which is capable of self-adapting or even deploying such effects.

6.4. NN-model Acceleration Analysis

In this section, the acceleration factor for different architectural configurations are analyzed for different types and parameterizations of NN-layers. Additionally, the impact of model weight sparsity and platform type is evaluated with respect to performance and program code footprint. The examined architectures comprise a plain RISC-V processor extended with standard extensions for multiplication and compressed instructions (rv32imc). Furthermore, the analysis uses a configuration, which extends the rv32imc with a SIMD dot-product unit and Weight/Input Activation Streamers (rv32imc+AW_acc). This approach requires SW-based postprocessing of the output activation, i.e. quantization, activation function, address resolution and memory writes. Those tasks are covered in the Output Activation Streamer, which is exclusive to the "rv32imc+AWO_acc" platform that has the highest expansion stage.

6.4.1. Model Footprints

An overview of the required ROM footprints is given in Figure 6.7. Two major trends are visible herein. An obvious trend is the reduction of footprint, when introducing sparsity. In these evaluations, the sparse models are pruned with 70% sparsity in weights. This reduces the amount of stored model weights to 30% compared against the non-sparse model. Those savings are affected by the introduction of an additional sparsity encoding vector, which requires either SW routines or dedicated HW to be decoded. In "rv32imc", the sparsity decoding is entirely SW-based, which negatively impacts the ROM footprint again.

The second trend is observed, when comparing the different architectures against each other. With an increasing number of accelerator features, the ROM footprint continuously drops. The reason of this observation can't be changing model footprints, since the identical model is used in each of the cases. Instead, the divergence is explained by higher coverage through the accelerator. Due to that, a lower number of SW routines need to be supported, which positively affects the ROM footprint.

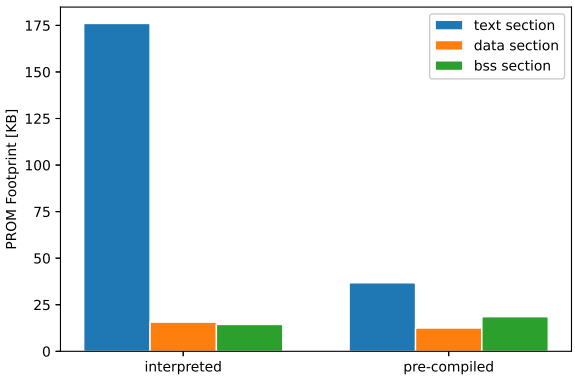


Figure 6.8.: Interpreted and Pre-interpreted Program Code Footprint

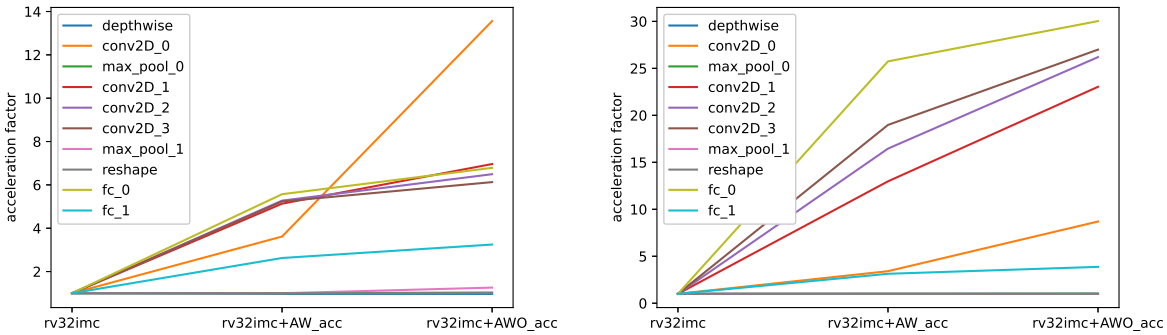
The lowest ROM footprint is observed for a sparse model on the platform with the highest expansion stage (rv32imc+AWO_acc). In contrast, the execution of a non-sparse model on a non-accelerated hardware is most memory intensive (1.6x of the best case), excluding the conventional approach (2x of the best case).

The impact of the pre-interpreter on the program code footprint shows Figure 6.8. Herein, the code size of an online interpreted sample model (marked with "interpreted") and its pre-compiled pendant after applying the pre-interpreter are compared. The text section of the program using the online interpreter is 80% smaller for the pre-compiled model. The text section of the compiled binary maps to actual instruction code. Thus, the pre-interpretation identifies 80% of instruction code as unused and prevents them from being compiled to the binary, saving lots of memory space and lowering the memory requirement of the execution platform. In contrast, the size of the data and bss section, which are relevant for storing constants and static variables are nearly untouched, since the actual model data cannot be removed from the binary.

The overall memory saving enabled by pre-interpretation, sparsity deployment and appropriate platform features reaches 87.5% when compared against the original NN-model inference.

6.4.2. Layerwise Acceleration

In this section, the acceleration of the NN is broken down into its individual layers (shown in Figure 6.6). Section 6.4.3 compares the acceleration of the individual layers for a sparse against non-sparse model on the different architectural configurations. In section 6.4.3, the efficiency of the acceleration is discussed for different layer parameterizations, which are supported by the accelerator.



(a) No kernel sparsity

(b) 70% kernel sparsity in supported layers

Figure 6.9.: Acceleration factor broken down to the involved layer operations

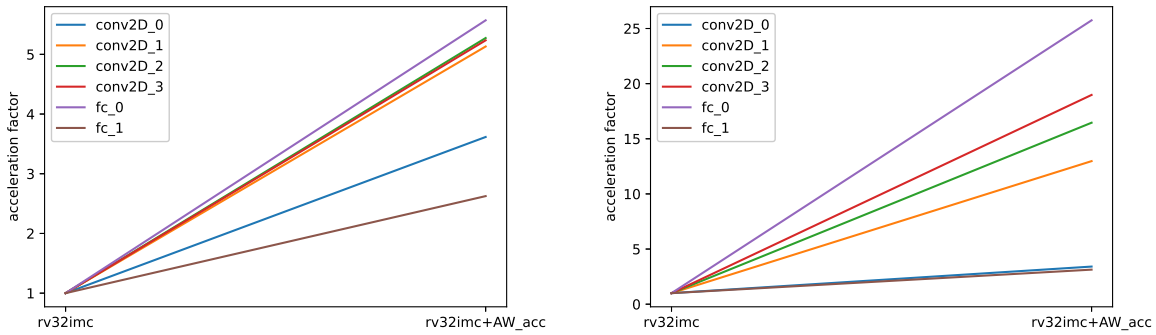
6.4.3. Acceleration Factors

For every individual layer of the NN, the acceleration factors are shown across the different platforms in Figure 6.9. Herein, (a) focuses on non-sparse weights, whereas (b) does so for sparse weights.

In both figures, "depthwise", "max_pooling" and "reshape" layers are not accelerated on any of the three expansion stages. That is because of a lack of appropriate accelerator features and TF kernels, which are needed to utilize the accelerator for those layer types. Future extensions of the accelerator allow to also cover these layers, since the data access patterns are inherent for various layer types (e.g "depthwise" and "max_pooling"). At time being, these layers are executed in plain SW in a non-accelerated fashion on the processor.

To summarize Figure 6.9, the acceleration of most of the layers supported by the accelerator is between 6× to 7× in non-sparse case and between 23× to 30× in sparse case on the "rv32imc+AWO_acc" when referred to the "rv32imc" platform. Sparsity enormously influences the achievable acceleration factor of the proposed accelerator HW. Contrarily to the previous observation, the "conv2D_0" and "fc_1" layers deviate significantly and reach a lower acceleration factor. The reason behind such a behavior is their tiny kernel dimensions, which impairs the payload/setup ratio. The "rv32imc+AWO_acc" platform addresses this bottleneck for the "conv2D_0". For non-sparse kernels, the achieved acceleration factor outperforms any other layer. For sparse kernels the acceleration is raised to 9×.

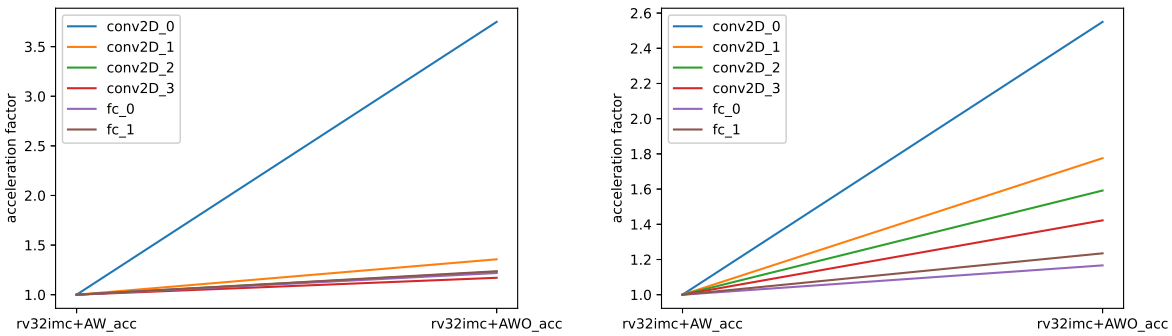
Generally, FC layers of sufficient dimensions experience a better acceleration than Conv layers, since less frequent reconfigurations are needed, which maximizes the throughput. This condition is fulfilled for the "fc_0" layer.



(a) No kernel sparsity

(b) 70% kernel sparsity

Figure 6.10.: Acceleration factor of supported layers with Weight/Activation Streaming and SIMD dot product unit



(a) No kernel sparsity

(b) 70% kernel sparsity

Figure 6.11.: Acceleration gain of supported layers with additionally enabled Output Activation Streaming

Figure 6.10 compares the acceleration factors of the first accelerator expansion stage (rv32imc+AW_acc") against "rv32imc" on supported layers. Figure 6.11 compares the same layers on the second accelerator expansion stage (rv32imc+AWO_acc) against "rv32imc+AW_acc". For both non-sparse and sparse cases, the first expansion stage unleashes significantly stronger acceleration gains, than the 2nd expansion stage (addition of Output Activation Streamer), where typical acceleration gains are below 1.8×.

Acceleration Efficiency

To showcase the effectiveness of the acceleration for different kernels, Figure 6.12 proportions the number of MAC operations per layer and the corresponding execution time for Conv and FC layers. In both of the sub-figures, the median for the sparse and non-sparse case is plotted. Kernels located above the median posses good efficiency (i.e. a high computation unit utilization).

Figure 6.12a, shows low efficiency for the small "10x1x1x4" Conv kernel in both sparse and non-sparse cases. The reason for this behavior is the frequent reconfiguration of the accelerator, which is required when dealing with kernels of small dimensions. In the sparse case, the execution time of that layer is even higher than for the non-sparse case. Since sparse kernels have less weights, the reconfiguration effort grows. In contrast, with increasing kernel size, the efficiency grows and reaches $\sim 150 MOPS$.

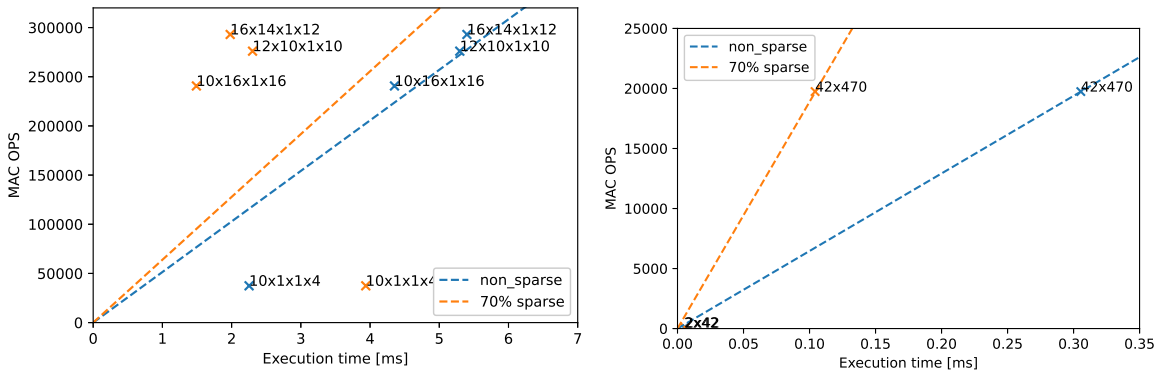
For the FC-layer with sparsity, an increasing efficiency of $\sim 190 MOPS$ can be observed (Figure 6.12b). The impact of the kernel dimension is negligible in the FC case, since reconfiguration within an entire layer execution is not needed. Thus, any possible FC kernel configuration is located close to the mean diagonal in Figure 6.12b.

6.4.4. Network Acceleration

Figure 6.13 illustrates the acceleration for the knock localization application achieved with the proposed methods. In (a), the acceleration of the overall network is displayed. (b) summarizes the acceleration of the layers, which are supported by the proposed accelerator methods.

From that Figure, two major conclusions can be drawn. First, sparsity and its efficient deployment in the acceleration modules significantly contribute to the achieved acceleration factor. Second, even though the acceleration of the supported layers raises up to 25× (refer to Figure 6.13b), the overall network acceleration doesn't exceed an acceleration factor of 3.5×.

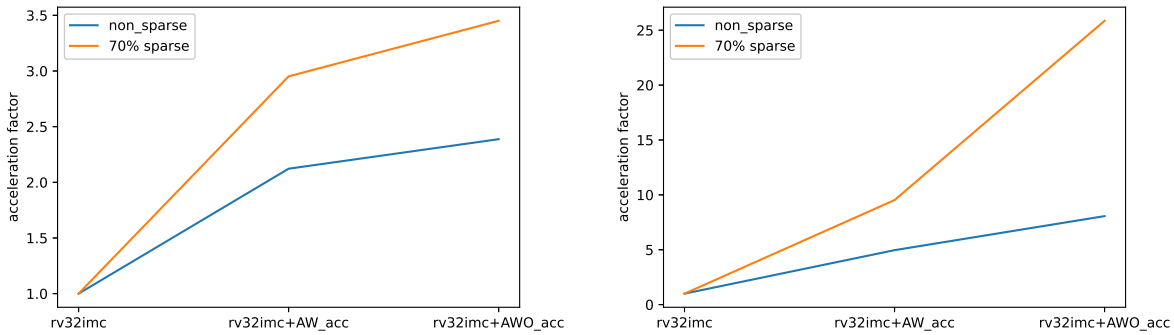
The reason for that can be identified, when breaking down the overall NN-model execution time into its individual layer execution times. Those are displayed in Figure 6.14. It can be observed that the share of the depthwise layer execution time raised from 30% in baseline to 80% in the accelerated setup. The execution time of the other layers become negligible in contrast. The bottleneck of the overall network execution for the knock localization application with the proposed accelerator feature set is the depthwise layer.



(a) Convolution layers

(b) Fully connected layers

Figure 6.12.: Effectiveness of the acceleration for different layer parameterization (kernel dims: Kernels x Height x Width x Channels) in OPS



(a) Overall network

(b) Considering supported layers of network only

Figure 6.13.: Acceleration factor of the network inference execution

For deeper NN-models with an increased number of Conv layers and a low share of depthwise layers, the impairment of the latter layer on the overall model inference time decreases. That raises the overall network execution acceleration factor. Other networks cope without depthwise layers, such that no such degradation of the acceleration factor is expected.

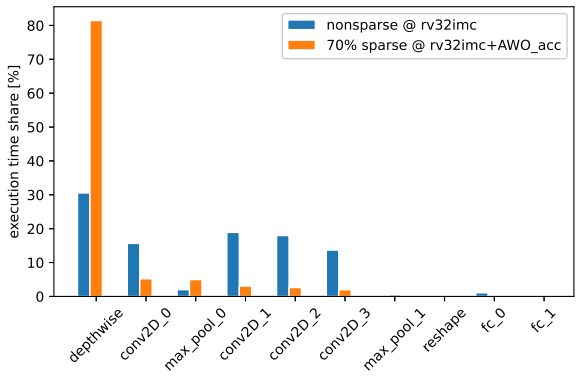


Figure 6.14.: Execution time share for each layer of the evaluated NN-model, displayed for the baseline setup (nonsparse layers @ “rv32imc”) compared to the accelerated setup (sparse layers @ ”rv32imc+AWO_acc”)

6.4.5. Sparsity Impact

The impact of the sparsity of individual layers on the acceleration factor is illustrated by Figure 6.15. In the subfigures (a), (b) and (c) the evaluation is displayed for the three platforms, "rv32imc", "rv32imc+AW_acc"and "rv32imc+AWO_acc" respectively.

Figure 6.15a shows a decreasing execution speed across all analyzed layers on the baseline platform when using sparse kernels. In worst case (i.e."fc_0") more than 30% performance degradation has to be accepted. This reveals a conceptional conflict at the "rv32imc" platform, since sparsity is deployed to reduce the model memory footprints, which fosters low on-chip area.

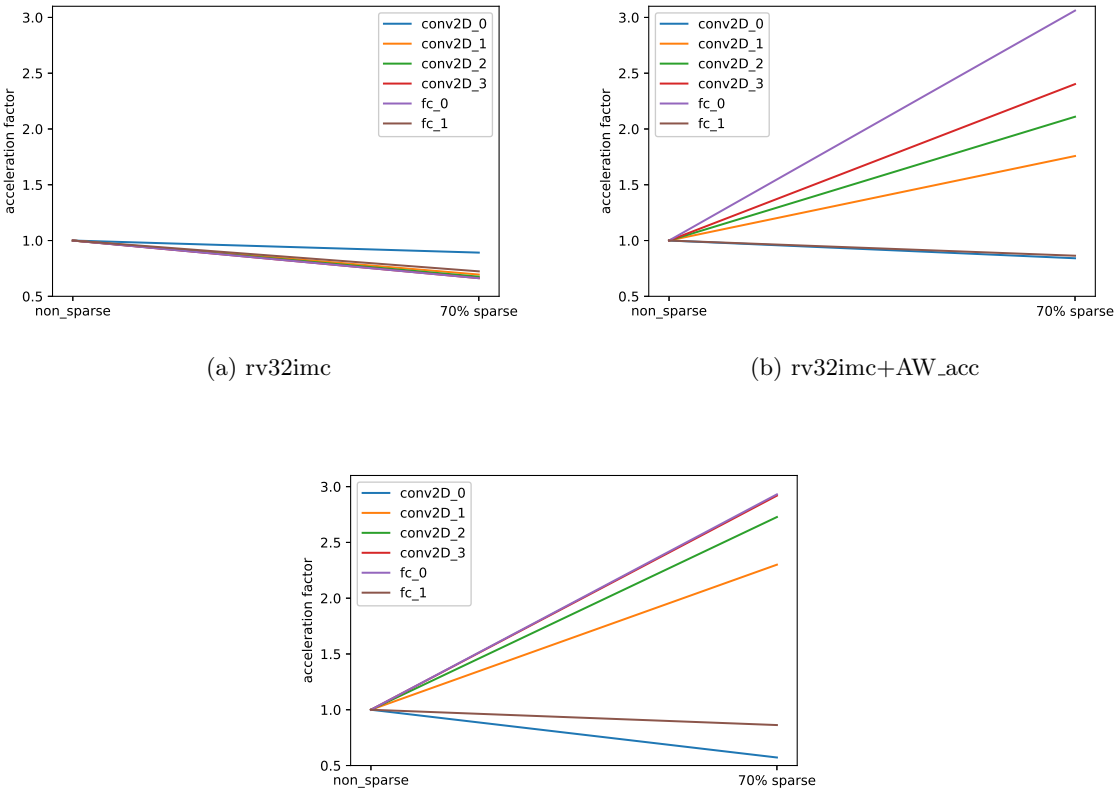
The replacement of the baseline platform with "rv32imc+AW_acc" commutes the trend of a decreasing execution performance into an acceleration of 1.5× to 3× when using sparsity (shown in Figure 6.15b). Hereby, FC layers perform better than Conv layers. Two exceptions are visible. Tiny Conv and FC layers, i.e. "conv2D_0" and "fc_1" cannot generally benefit from sparsity, instead their particular performance slightly suffers.

A similar trend is visible in Figure 6.15c. As an highlight, the Conv layer execution could have been further accelerated and reaches acceleration factors close to the FC layer. With Output Activation Streamer deployment, a reduced number of tasks need handling via SW routines. In contrast, FC layers cannot gain more performance by deploying sparsity on the "rv32imc+AWO_acc" platform than on "rv32imc+AW_acc", since that layer type requires similarly minor SW interaction in both cases. Furthermore, applying sparsity to small kernels doesn't have any positive effect on the "rv32imc+AWO_acc" platform either. Instead, the performance loss is amplified to even 40%, which is caused by more expensive and frequent reconfigurations.

6.4.6. Impact of the Activation Function Type

As discussed in chapter 3.1.5, the Output Activation Streamer provides a feature to cover complex fused activation functions via interpolation. In the analysis, done for the model of the knock localization application (see Figure 6.6), "ReLU" was the only deployed activation function. "ReLU" is frequently used in many applications, but of low complexity. An appropriate HW mapping of that function is small or the alternative SW routine tiny, which covers the corresponding operation. In this section, the impact of a more complex activation function on the performance and the acceleration factor is evaluated to complete the analysis for models with complex activation functions. As an example the "TanH" function is deployed, see (2.22).

In the following, the involved "ReLU" functions of the NN-model in Figure 6.6 are exchanged with "TanH" functions. The execution time measurement of the modified model execution is done for the same architectural variants as in the previous sections. Since the "rv32imc" and



(a) rv32imc

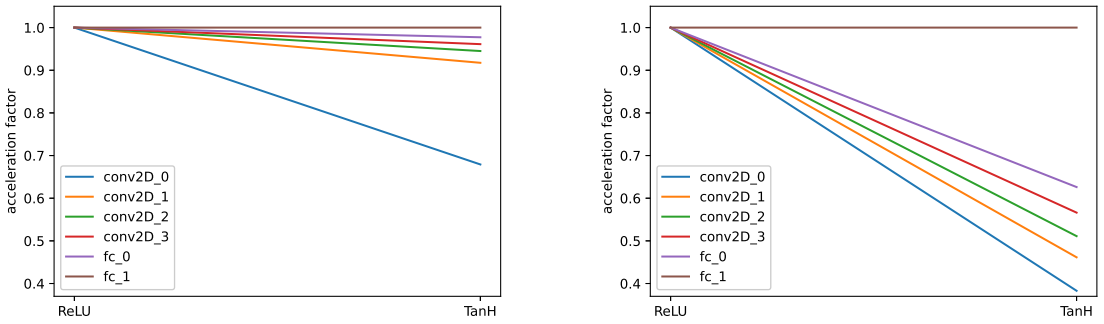
(b) rv32imc+AW_acc

(c) rv32imc+AWO_acc

Figure 6.15.: Comparison of the acceleration factor between nonsparse and 70% sparse kernels for different architectural configurations

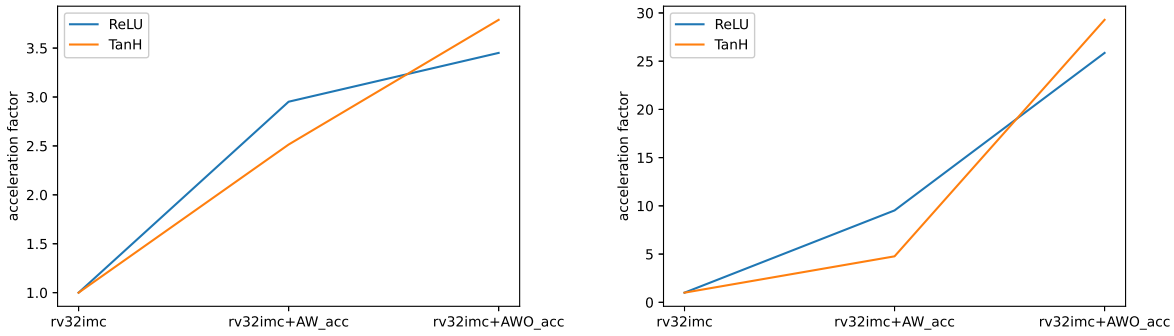
"rv32imc+AW_acc" platform don't provide dedicated HW support for activation functions, the "TanH" function is SW emulated thereby. The measurements show that the application of the "TanH" function onto a single signed 8-bit fixed-point integer value in SW takes approximately 173 clock cycles. Since every layer produces an output of high order, the impact on the individual layers becomes significant. For every individual output activation, the estimated amount of cycles need to be accounted.

Figure 6.16 shows a comparison of the acceleration factors when using fused "TanH" instead of "ReLU" for the different architectural platforms. For "rv32imc" a performance degradation of the fused Conv and FC layers of less than 10% can be observed (Figure 6.16a). The small "conv2D_0" layer constitutes an exception, which produces even worse results. In comparison, the degradation on the "rv32imc+AW_acc" platform is more significant with up to 50% (Figure 6.16b). This is due to the fact that the dot-product computation of these layers is accelerated and therefore the proportion of the actual activation function computation increases. On the fully accelerated "rv32imc+AWO_acc" platform, no further impact can be observed, since the "TanH" is handled via the dedicated interpolation HW in a single pipeline stage of the Output Activation Streamer, similar to the "ReLU".



(a) rv32imc (b) rv32imc+AW_acc

Figure 6.16.: Comparison of the acceleration factor between "ReLU" and "TanH" as activation function for different architectural configurations, considering 70% sparse weights



(a) Overall network (b) Considering accelerated layers of network only

Figure 6.17.: Acceleration factor of the 70% sparse network inference execution, with "ReLU" or "TanH" as fused activation function

The impact of the choice for the activation function type on the acceleration factor is shown in Figure 6.17 in an abstract view. Hereby, (a) plots the acceleration factor using "ReLU" and "TanH" function on every platform for the entire network (70% weight sparsity). Alternatively, (b) refers to accelerated layers only. When switching from "ReLU" to "TanH" on the "rv32imc+AW_acc" platform, the acceleration factor decreases by 0.5× and 4.8× for (a) and (b) respectively. In contrast, on the "rv32imc+AWO_acc" platform the acceleration factor increases by 0.3× and 3.4× for (a) and (b) respectively (related to "rv32imc"). Though, the maximum acceleration factor reaches approximately 30×.

7. Conclusion and Future Work

In this thesis an accelerator for NNs is proposed that uses a classical platform infrastructure. A multilayer bus connects the SRAM memories as slaves to the accelerator and processor master interfaces. This is one of the elements, which are deployed to maximize the utilization of the available platform features, since the uC platform relies on the same bus and memory infrastructure. That design decision fosters area efficiency, by intensive sharing of the available HW resources and cautious spending of additional area.

A tightly RISC-V processor coupled concept is proposed. It reduces the NN-inference latency by a low-overhead accelerator-to-processor interface. Additionally, the accelerator shares a significant portion of it's computation HW, e.g. MAC-unit, scaling-unit and bus interfaces, with the processor. This increases the utilization of those modules. Additionally, the tightly-coupled approach significantly contributes to the flexibility of the platform, where non-accelerated layers can be executed still. For this the assumption is taken that some performance penalties are accepted. Further, the proposed acceleration concept is also applicable to a loosely and tightly coupled acceleration approach, which was not part of this thesis.

The proposed NN-accelerator comprises of three main modules, i.e. weight, input-activation and output-activation streamers. These modules handle the fast and efficient data transfers and are configurable via CSRs. The first two modules independently load data from memory with complex fetch patterns, extract and prepare the data in ready-to-compute order and deliver it to the MAC-unit. The last module takes the output data, packs it and writes it back to the memory applying complex write patterns. The streamers resolve the complex identification and preparation of weights and activations in a performant way in HW. That replaces slow many cycle SW-routines. Thereby, energy efficiency is maintained, since the portion of payload cycles rises in the same way the overall inference latency reduces.

The data bandwidth limitations are addressed by the streamers. In this thesis, the bottleneck of the dot-product computation is addressed by a MAC-unit and its extension with the support for vector modes. Thereby, multiple dot-products are computed and accumulated in parallel. For supporting the vector modes a major portion of the HW is reused, which is already in place for the scalar mode.

To provide an efficient support of complex activation functions, a unit for piecewise linear interpolation for non-uniformly distributed interpolation points is developed. It allows to reduce the number of interpolation points, while providing low approximation errors. At the same time HW-cost is maintained by selecting a reduced but optimized number of interpolation points. The proposed interpolation unit is configurable and reprogrammable, which makes it very flexible.

The preinterpreter plugin module proposed for the TF lite micro significantly influences the binary size of the compiled model. Up to 70% binary size reduction could have been proven. This makes models with increased complexity fitting the limited memory space available at edge devices. Additionally, the preinterpreter allows the compiler to get rid of the on-the-fly interpretation overhead and thereby speeds up the accelerator boot up.

Further, weight sparsity is deployed, which reduces the required memory space (compressed model data) and an on-the-fly sparsity decompressor is incorporated to resolve the skipping of the dot-product computation for sparse weights. This contributes to the energy efficiency and reduces the inference latency, since a lower number of weights and activations need to be fetched from memory and computed in the MAC-unit.

The streamers and MAC-unit of the proposed accelerator HW share the same submodules for the support of multiple data types with various precisions. By this sharing the on-chip area benefits and the flexibility is enhanced.

The overall area increase starting with a RISC-V rv32imc processor to the full fletched tightly-coupled accelerator is quantified to $\sim 2.4\times$. In contrast, the speedup of the inference computation for an NN-model comprising only of accelerated layers turned out to be $\sim 30\times$. At the same time the utilization of the MAC-unit got raised to $\sim 80\%$. These results show that the proposed solution increases performance, maintains the on-chip cost and saves energy.

For future explorations the proposed approach can be applied in a loosely coupled setup. Hereby, the RISC-V processor is not directly involved in the NN computations. Instead, it acts as a controlling instance for the mainly FSM driven circuit. Additionally, it allows to treat special cases (not covered by the FSM) with better performance than unaccelerated SW routines.

Other research can be applied in the direction of intensified parameter (weight and activation) compression. In this thesis, weight sparsity and quantization are evaluated, but more advance compression techniques like sub-byte parameter types or dynamic length encoding may decrease the model footprints further. This comes at the cost of a more complex decompression HW or a need for compressed execution mechanisms. A trade-off analysis considering benefits and disadvantages is needed to identify potentially beneficial advanced compression techniques with respect to performance, on-chip footprint and power.

The supported parallelism of the proposed approach is limited to maximum four simultaneous dot-product computations. This is caused by the limited operand number of the MAC-unit and the streaming modules. In order to reach even higher acceleration factors the parallelism can be enhanced by increasing the operand number in all of the involved modules. Thereby, an acceleration factor of $\sim 100\times$ seems feasible.

8. Publications

This chapter summarizes the authors publications and brings them into context of this thesis. Hereby, related peer-reviewed conference papers are described in the first section 8.1, whereas the related patent goes into the second section 8.2 of this chapter. For each individual publication the reference, the abstract and the context with respect to this thesis is presented.

8.1. Published Papers

8.1.1. A Smart HW-Accelerator for Non-Uniform Linear Interpolation of ML-Activation Functions

Reference: (Prebeck, Lawand, Vaddeboina & Ecker 2022)

Abstract The compulsive nonlinearity in neural networks (NN) is introduced by well-known nonlinear functions called activation functions. Performing AI-inferences on edge devices calls for efficient approximation of those complex functions on highly restrictive hardware (HW) platforms. When designing such systems, balancing area, footprint and power consumption at an application appropriate latency is key. To address those challenges, we propose a HW-based interpolation component capable of approximating arbitrary mathematical functions. A combinatorial search-based optimization algorithm is employed to find the optimal set of interpolation points for a set of functions while also considering non-uniform distributions. The proposed solution is accompanied by a Python-based HW generator, which facilitates the process of deploying software-computed search results on HW and provides room for generating different flavors of application-optimized HW. In an effort to reduce area footprint and delay, the proposed approach exploits symmetry and biased symmetry properties of functions and applies bit width optimizations to reduce the size of the utilized computational units. Additionally, property-aware reprogrammable solutions for multifunctional use cases are incorporated into our design. Experimental analyses show that our proposed method permits achieving better area utilization and deviation error results than state-of-the-art implementations.

Context The concept for the linear interpolation published in this paper is used in the proposed accelerator to map the activation function onto the output activation streamer.

8.1.2. A Scalable, Configurable and Programmable Vector Dot-Product Unit for Edge AI

Reference: (Prebeck, Ashok, Vaddeboina, Devarajegowda & Ecker 2022)

Abstract Second generation artificial intelligence (AI) migrates inference related computations from cloud towards edge devices. Due to increasingly sophisticated neural network (NN) architecture search, even more complex applications come into range for execution on the edge. This enables a significant drop in latency, power consumption and bandwidth, since data transmission to cloud becomes obsolete. We address challenges to enable edge platforms with computation hardware capable of dealing with more complex applications locally. For NN inferences in general, dot product operation is the most commonly and intensively used. Thus, defining a proper unit supporting the mentioned operation in an efficient way has huge impact. Between different applications the network hyperparameters may change, including the data formats of kernels and activations. Therefore, supporting a wide variety of data formats with the dot product unit, while keeping the area increase low, seems appealing. Additionally, the computational load varies dependent on the particular application, thus a scalable solution is desireable. Next to the configurability and programmability, area as well as power efficiency plays an important role. We propose a scalable, configurable and programmable vector dot product unit, targeting an optimized footprint for low power applications to overcome the challenges of second generation AI on edge devices. The proposed solution is supported by a Python-based HW generator, which enables the derivation of featured dot product units optimized for certain applications. It is developed with the assumption to be utilized as a standalone component as well as loosely or tightly coupled component associated with a CPU instruction set extension.

Context The vector dot-product unit published in this paper is used as computation unit, which enables the support of variable width weights and activations, while simultaneously operating on multiple vectors.

8.1.3. Automated HW/SW Co-design for Edge AI: State, Challenges and Steps Ahead: Special Session Paper

Reference: (Bringmann et al. 2021)

Abstract Gigantic rates of data production in the era of Big Data, Internet of Thing (IoT), and Smart Cyber Physical Systems (CPS) pose incessantly escalating demands for massive data processing, storage, and transmission while continuously interacting with the physical world using edge sensors and actuators. For IoT systems, there is now a strong trend to move the intelligence from the cloud to the edge or the extreme edge (known as TinyML). Yet, this shift to edge AI systems requires to design powerful machine learning systems under very strict

resource constraints. This poses a difficult design task that needs to take the complete system stack from machine learning algorithm, to model optimization and compression, to software implementation, to hardware platform and ML accelerator design into account. This paper discusses the open research challenges to achieve such a holistic Design Space Exploration for a HW/SW Co-design for Edge AI Systems and discusses the current state with three currently developed flows: one design flow for systems with tightly-coupled accelerator architectures based on RISC-V, one approach using loosely-coupled, application-specific accelerators as well as one framework that integrates software and hardware optimization techniques to built efficient Deep Neural Network (DNN) systems.

Context The presented concerns and approaches, which are involved in the automation of the construction of an NN-accelerator are considered during the HW design process.

8.1.4. Optimized HW/FW Generation from an Abstract Register Interface Model

Reference: (Werner et al. 2020)

Abstract The HW/SW interface is a common and crucial component in System-on-Chips, enabling the interaction between software and hardware. Generating architecture and firmware code of the interface from extended IP-XACT, SystemRDL, or proprietary formalism is an established technology. This paper describes a new area and performance optimization step in the HW/SW interface generation process that reduces the silicon area and hardware access time through firmware. Three improvements of the underlying formalism are applied to achieve the optimization: First, a decoupling of bit fields from registers, which allows the rearrangement of the memory layout easily. Second, the specification of hardware accesses, which constraints the bit field arrangement. Third, different implementations of bit field accesses, such as memory-mapped or via CPU special registers. The used generation framework follows the approach of model-driven architecture, which includes optimization. Initially, abstract models specify the requirements of the IP or the HW/SW interface. Transformations turn these models into platform-independent models of hardware and firmware. These models are further transformed into implementation-specific models of a target language, such as hardware description languages or C. The proposed optimization has been successfully applied to peripheral variants of a CPU subsystem used in an industrial demonstrator. An area reduction of 19% and a performance gain of 11% has been achieved by optimizing the interfaces.

Context The optimized HW/FW (Firware) generation ensures efficient and optimized drivers for the platform peripherals. The HAL (Hardware Abstraction Layer) is generated using the same spec as used for the generation of the MMIO peripherals. This ensures consistency between HW and FW and reduces the manual and error-prone effort to keep both synchronous. This work strengthens an efficient and reliable accelerator platform.

8.1.5. Towards Fault Simulation at Mixed Register-Transfer/Gate-Level Models

Reference: (Kaja et al. 2021)

Abstract Safety-critical designs used in automotive applications need to ensure reliable operations even under hostile operating conditions. As these designs grow in size and complexity, they are facing an increased risk of failure. Consequently, the methods applied to validate the reliability of designs require increasingly more compute resources (e.g., fault simulation time) and manual efforts. Rigorous and highly automated safety analysis methods are needed to cope with this rising complexity. In this paper, we propose a model-based safety analysis flow to enable fault injection at different abstraction levels of a design. The fault simulation is performed at register transfer level (RTL) of a design, in which parts of the design targeted for fault simulation are represented with gate-level granularity. This mixed representation of a design provides a significant rise in fault simulation performance while maintaining the same accuracy as a gate-level fault simulation. To demonstrate the applicability of the proposed approach, various RISC- V based CPU subsystems that are part of automotive SoCs are considered for fault simulation. The experimental results show an increase of 3.5x - 8.4x in the fault simulation performance with substantially less manual effort as all the design activities are automated utilizing a model-driven RTL generation flow.

Context For safety critical applications an efficient fault simulation method is required for the RISC-V processor and the NN-accelerator to evaluate the impact of Faults on the overall application. Additionally, applied safety methods can be evaluated fast on their coverage.

8.1.6. ISA Modeling with Trace Notation for Context Free Property Generation

Reference: (Devarajegowda et al. 2021)

Abstract The scalable and extendable RISC-V ISA introduced a new level of flexibility in designing highly customizable processors. This flexibility in processor designs adds to the complexity of already complex functional verification process. Although formal methods are increasingly used to exhaustively verify the processors, the required manual effort and verification expertise become the major hurdles in industrial flows. Furthermore, efficient ISA modeling techniques are required that are scalable to multiple ISA extensions and to different architectural variants of a processor. This paper proposes a trace notation for ISA definition to capture the implicit execution behavior of a processor and specific characteristics. The proposed trace notation can be annotated with timing and hierarchy information to adapt the trace to any kind of processor architecture. From this trace notation, a complete set of properties are generated to detect all functional bugs in a processor implementation. The approach requires significantly less manual effort compared to the contemporary techniques.

Its industry strength has been demonstrated by formally verifying a wide variety of RISC-V processor implementations with one or more ISA extensions (RV32I, C, Zicsr, M and custom extensions supporting AI acceleration and safety features).

Context The functional correctness of the RISC-V processor used in the tightly-coupled accelerator is verified by applying the method proposed in this paper.

8.1.7. RTL Delay Prediction Using Neural Networks

Reference: (Lopera et al. 2021)

Abstract Nowadays, the digital chip design flow starts with formal specifications, which are mapped to Register Transfer Level (RTL) models using different underlying implementation variants and (micro-) architectures. By doing so, a hardware designer predicts and resolves time-critical parts to achieve an RTL-design that intentionally meets all constraints after synthesis. However, wrong predictions can be detected only later in the design flow, thus leading to long design iterations. Classical methods estimating delay in early design stages are constrained to the type of components or are computationally expensive for larger designs. In this paper, we propose a Machine Learning-based approach to estimate pin-to-pin delays for RTL combinational circuits. To gain accuracy, we combine slew and delay estimation. To that end, a training set is built using features of components generated by a model-driven hardware generator framework. Ground truth labels for delays, slews, and their interdependencies are extracted using open-source tools for logic synthesis and static timing analysis. Evaluations in unseen designs show that the delay estimation has on average an accuracy of 87% and it is 13x faster compared with results of synthesis and timing analysis tools. Based on the estimation, critical areas of the design can be detected and proper microarchitecture decisions can be taken earlier in the design flow.

Context The balancing of the critical path between the pipeline stages in either the RISC-V processor or the accelerator modules is a challenging and time consuming task. Relying on the available synthesis tools is slow. Therefore, deploying ML to predict the delays on RTL (Register Transfer Level) speeds up the design time significantly and enables an advanced delay balancing.

8.1.8. Aspect-Oriented Design Automation with Model Transformation

Reference: (Han et al. 2021)

Abstract Despite the high configurability of IPs and hardware generators, code modifications are still required to introduce aspect-oriented instrumentation to satisfy emerging design requirements such as on-chip debug and functional safety. These code modifications lead to escalated development, verification efforts and deteriorate the code reuse. This paper proposes a highly efficient aspect-oriented design automation approach that leverages graph-grammar-based model transformations. With the proposed approach, main design functionalities and aspect-oriented instrumentation are separately developed, automatically integrated and verified. To demonstrate the applicability, industrial SoCs were transformed to support on-chip debug. Experimental results confirm the efficiency of the approach. Further, reduced code is needed with the proposed automation approach, which also replaces the error-prone manual RTL coding. Finally, the transformation scripts are applicable to different SoCs, which promotes the overall code reuse.

Context Debugging of NN-accelerators is a challenging task, because of the heterogeneity of the available designs. No common debugging interfaces are defined yet and a simple and appropriate method is needed to enable debugging without redesigning the accelerator HW design. Model transformation has the capability to achieve that and allows post design time modifications to insert debug functionalities.

8.1.9. Early RTL delay prediction using neural networks

Reference: (Sánchez Lopera et al. 2022)

Abstract Nowadays, the digital chip design flow starts with formal specifications, which are mapped to Register Transfer Level (RTL) models using different underlying (micro-) architectures. By doing so, a hardware designer predicts and resolves time-critical parts to achieve an RTL-design that meets all constraints after synthesis. However, wrong predictions can be detected only later in the design flow, thus leading to long design iterations. Classical methods estimating delay in early design stages are constrained to the type of components or are computationally expensive for larger designs. This paper proposes a machine learning-based approach to estimate pin-to-pin delays for RTL combinational circuits. To improve the quality of the predictions we combine slew and delay estimation. To that end, training data are built using features of components generated by a model-driven hardware generator framework. Ground truth labels for delays, slews, and their interdependencies are extracted using open-source tools for logic synthesis and static timing analysis. Two different datasets are built: one targeting logic gates and multiplexers and an enlarged one, which generalizes to more RTL primitives A

model trained using the former dataset achieves, on average, a coefficient of determination R^2 of 87% when evaluating over 4-bit prefix adders. Using the enlarged dataset, the best model reaches an R^2 of 77%. On average, our models are 8.4× faster w.r.t the time required to run synthesis and timing analysis. Results show that generalizing to more primitives decreases the models' performance, but the runtime benefit is maintained. Based on the delay estimation, critical areas of the design can be detected and proper microarchitecture decisions can be taken earlier and faster in the design flow.

Context During the inference computation of an NN on HW the parameters flow through multiple stages of the proposed accelerator including the input streamers, the CPU and the output streamer. Each of the involved stages covers a different task and comes along with a particular complexity. Definition and optimization for the modules of every stage are often based on gut feeling and it takes many iterations to get a well timing balanced circuit, which makes the design a challenge. The timings of the individual module signals are hard to estimate often and therefore performance bottlenecks are found late in the design flow. Especially, when using the streamers in standalone and loosely coupled setups the timing behaviors change compared to the one in the tightly coupled approach. Thus, further iterations of balancing need to be applied. The proposal of this paper addresses the speed up of the timing balancing process.

8.2. Published Patent

8.2.1. Accelerating processor based artificial neural network computation

Reference: (Stevens et al. 2022)

Abstract An apparatus employed in a processing device comprises a processor configured to process data of a predefined data structure. A memory fetch device is coupled to the processor and is configured to determine addresses of the packed data for the processor. The packed data is stored on a memory device that is coupled to the processor. The memory fetch device is further configured to provide output data based on the addresses of the packed data to the processor, where the output data is configured according to the predefine data structure.

Context This patents the concepts beneath the tightly processor-coupled accelerator approach.

Bibliography

Abadi, Martín, Agarwal, Ashish, Barham, Paul, Brevdo, Eugene, Chen, Zhifeng, Citro, Craig, Corrado, Greg S., Davis, Andy, Dean, Jeffrey, Devin, Matthieu, Ghemawat, Sanjay, Goodfellow, Ian, Harp, Andrew, Irving, Geoffrey, Isard, Michael, Jia, Yangqing, Jozefowicz, Rafal, Kaiser, Lukasz, Kudlur, Manjunath, Levenberg, Josh, Mane, Dan, Monga, Rajat, Moore, Sherry, Murray, Derek, Olah, Chris, Schuster, Mike, Shlens, Jonathon, Steiner, Benoit, Sutskever, Ilya, Talwar, Kunal, Tucker, Paul, Vanhoucke, Vincent, Vasudevan, Vijay, Viegas, Fernanda, Vinyals, Oriol, Warden, Pete, Wattenberg, Martin, Wicke, Martin, Yu, Yuan & Zheng, Xiaoqiang (2016): Tensorflow: Large-scale machine learning on heterogeneous distributed systems, ArXiv Preprint ArXiv:1603.04467 .

Adiono, Trio, Meliolla, Grasia, Setiawan, Erwin & Harimurti, Suksmandhira (2018): Design of neural network architecture using systolic array implemented in verilog code, 2018 International Symposium on Electronics and Smart Devices (ISESD) pp. 1–4.

Aggarwal, Khushboo (2010): Simulation of artificial neural networks on parallel computer architectures, 2010 International Conference on Educational and Information Technology (ICEIT) **2**: V2–255–V2–258.

Amin, H., Curtis, K.M. & Hayes-Gill, B.R. (1997): Piecewise linear approximation applied to nonlinear function of a neural network, IEE Proceedings - Circuits, Devices and Systems **144**(6): 313.

Ashok, Sathya (2021): Hardware generator of a configurable and pipelineable multiplier unit, Master's thesis, Technical University of Munich.

Baugh, C.R. & Wooley, B.A. (1973): A two's complement parallel array multiplication algorithm, IEEE Transactions on Computers **C-22**: 1045–1047.

Bedrij, O. J. (1962): Carry-select adder, IRE Transactions on Electronic Computers **EC-11**: 340–346.

Belyaev, Ivan A., Belyaev, Andrey A., Solokhina, Tatiana V. & Petrichkovich, Yaroslav Y. (2020): A high-perfomance multi-format simd multiplier for digital signal processors, 2020 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (EIConRus) pp. 1780–1783.

Bernardo, Paul Palomero, Gerum, Christoph, Frischknecht, Adrian, Lübeck, Konstantin & Bringmann, Oliver (2020): Ultratrail: A configurable ultralow-power tc-resnet ai accel-

erator for efficient keyword spotting, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **39**: 4240–4251.

Booth, Andrew D (1951): A signed binary multiplication technique, The Quarterly Journal of Mechanics and Applied Mathematics **4**(2): 236–240.

Brent & Kung (1982): A regular layout for parallel adders, IEEE Transactions on Computers **C-31**: 260–264.

Bringmann, Oliver, Ecker, Wolfgang, Feldner, Ingo, Frischknecht, Adrian, Gerum, Christoph, Hämäläinen, Timo, Hanif, Muhammad Abdullah, Klaiber, Michael J., Mueller-Gritschneder, Daniel, Bernardo, Paul Palomero, Prebeck, Sebastian & Shafique, Muhammad (2021): Automated hw/sw co-design for edge ai: State, challenges and steps ahead: Special session paper, 2021 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS), pp. 11–20.

Chandrashekara, M N & Rohith, S (2019): Design of 8 bit vedic multiplier using urdhva tiryagbhyam sutra with modified carry save adder, 2019 4th International Conference on Recent Trends on Electronics, Information, Communication & Technology (RTEICT), pp. 116–120.

Chen, Tianqi, Moreau, Thierry, Jiang, Ziheng, Zheng, Lianmin, Yan, Eddie, Shen, Haichen, Cowan, Meghan, Wang, Leyuan, Hu, Yuwei, Ceze, Luis, Guestrin, Carlos & Krishnamurthy, Arvind (2018): Tvm: An automated end-to-end optimizing compiler for deep learning, 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18), USENIX Association, Carlsbad, CA, pp. 578–594.

Chen, Yu-Hsin, Krishna, Tushar, Emer, Joel S. & Sze, Vivienne (2017): Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks, IEEE Journal of Solid-State Circuits **52**: 127–138.

Chen, Yu-Hsin, Yang, Tien-Ju, Emer, Joel & Sze, Vivienne (2019): Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices, IEEE Journal on Emerging and Selected Topics in Circuits and Systems **9**(2): 292–308.

Christensen, Steinar Thune, Aunet, Snorre & Qadir, Omer (2019): A configurable and versatile architecture for low power, energy efficient hardware acceleration of convolutional neural networks, 2019 IEEE Nordic Circuits and Systems Conference (NORCAS): NORCHIP and International Symposium of System-on-Chip (SoC), pp. 1–6.

Courbariaux, Matthieu, Bengio, Yoshua & David, Jean-Pierre (2015): Binaryconnect: Training deep neural networks with binary weights during propagations, Advances in Neural Information Processing Systems, Bd. 28, Curran Associates, Inc.

Danysh, A. & Tan, D. (2005): Architecture and implementation of a vector/simd multiply-accumulate unit, IEEE Transactions on Computers **54**: 284–293.

Deng, Li, Li, Jinyu, Huang, Jui-Ting, Yao, Kaisheng, Yu, Dong, Seide, Frank, Seltzer, Michael, Zweig, Geoff, He, Xiaodong, Williams, Jason, Gong, Yifan & Acero, Alex (2013): Recent advances in deep learning for speech research at microsoft, 2013 IEEE International Conference on Acoustics, Speech and Signal Processing, pp. 8604–8608.

Devarajegowda, Keerthikumara, Kaja, Endri, Prebeck, Sebastian & Ecker, Wolfgang (2021): Isa modeling with trace notation for context free property generation, 2021 58th ACM/IEEE Design Automation Conference (DAC), pp. 619–624.

Duan, Yunzhi, Li, Shuai, Zhang, Ruipeng, Wang, Qi, Chen, Jienan & E.Sobelman, Gerald (2018): Energy-efficient architecture for fpga-based deep convolutional neural networks with binary weights, 2018 IEEE 23rd International Conference on Digital Signal Processing (DSP), pp. 1–5.

Duren, R.W., Marks, R.J., Reynolds, P.D. & Trumbo, M.L. (2007): Real-time neural network inversion on the SRC-6e reconfigurable computer, IEEE Transactions on Neural Networks **18**(3): 889–901.

Garofalo, Angelo, Tagliavini, Giuseppe, Conti, Francesco, Benini, Luca & Rossi, Davide (2021): Xpulpnn: Enabling energy efficient and flexible inference of quantized neural networks on risc-v based iot end nodes, IEEE Transactions on Emerging Topics in Computing **9**(3): 1489–1505.

Garofalo, Angelo, Tagliavini, Giuseppe, Conti, Francesco, Rossi, Davide & Benini, Luca (2020): Xpulpnn: Accelerating quantized neural networks on risc-v processors through isa extensions, 2020 Design, Automation & Test in Europe Conference & Exhibition (DATE), pp. 186–191.

Gautschi, Michael, Schiavone, Pasquale Davide, Traber, Andreas, Loi, Igor, Pullini, Antonio, Rossi, Davide, Flamand, Eric, Gürkaynak, Frank K. & Benini, Luca (2017): Near-threshold risc-v core with dsp extensions for scalable iot endpoint devices, IEEE Transactions on Very Large Scale Integration (VLSI) Systems **25**(10): 2700–2713.

Golomb, S. (1966): Run-length encodings (corresp.), IEEE Transactions on Information Theory **12**: 399–401.

Gondimalla, Ashish, Chesnut, Noah, Thottethodi, Mithuna & Vijaykumar, T. N. (2019): Sparten: A sparse tensor accelerator for convolutional neural networks, Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO '52, Association for Computing Machinery, New York, NY, USA, p. 151–165.
`https://doi.org/10.1145/3352460.3358291`

Han, Song, Liu, Xingyu, Mao, Huizi, Pu, Jing, Pedram, Ardavan, Horowitz, Mark A. & Dally, William J. (2016): Eie: Efficient inference engine on compressed deep neural network, 2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA), pp. 243–254.

Han, Zhao, Wang, Deyan, Rutsch, Gabriel, Li, Bowen, Prebeck, Sebastian Siegfried, Lopera, Daniela Sanchez, Devarajegowda, Keerthikumara & Ecker, Wolfgang (2021): Aspect-oriented design automation with model transformation, 2021 IFIP/IEEE 29th International Conference on Very Large Scale Integration (VLSI-SoC), pp. 1–6.

Harris, Sarah L. & Harris, David Money (2016): 5 - digital building blocks, in S. L. Harris & D. M. Harris (eds), Digital Design and Computer Architecture, Morgan Kaufmann, Boston, pp. 238–293.

He, Kaiming, Zhang, Xiangyu, Ren, Shaoqing & Sun, Jian (2016): Deep residual learning for image recognition, Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pp. 770–778.

Horowitz, Mark (2014): 1.1 computing's energy problem (and what we can do about it), 2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC), pp. 10–14.

Howard, Andrew G., Zhu, Menglong, Chen, Bo, Kalenichenko, Dmitry, Wang, Weijun, Weyand, Tobias, Andreetto, Marco & Adam, Hartwig (2017): Mobilenets: Efficient convolutional neural networks for mobile vision applications, ArXiv Preprint ArXiv:1704.04861 .

Hubara, Itay, Courbariaux, Matthieu, Soudry, Daniel, El-Yaniv, Ran & Bengio, Yoshua (2017): Quantized neural networks: Training neural networks with low precision weights and activations, The Journal of Machine Learning Research **18**(1): 6869–6898.

Huffman, David A. (1952): A method for the construction of minimum-redundancy codes, Proceedings of the IRE **40**: 1098–1101.

Ioffe, Sergey & Szegedy, Christian (2015): Batch normalization: Accelerating deep network training by reducing internal covariate shift, in F. Bach & D. Blei (eds), Proceedings of the 32nd International Conference on Machine Learning, Bd. 37 von Proceedings of Machine Learning Research, PMLR, Lille, France, pp. 448–456.

Javali, Ravikumar A, Nayak, Ramanath J, Mhetar, Ashish M & Lakkannavar, Manjunath C (2014): Design of high speed carry save adder using carry lookahead adder, International Conference on Circuits, Communication, Control and Computing, pp. 33–36.

Jin, Qing, Yang, Linjie & Liao, Zhenyu (2019): Towards efficient training for neural network quantization, ArXiv Preprint ArXiv:1912.10207 .

Jouppi, Norman P., Young, Cliff, Patil, Nishant, Patterson, David, Agrawal, Gaurav, Bajwa, Raminder, Bates, Sarah, Bhatia, Suresh, Boden, Nan, Borchers, Al, Boyle, Rick, Cantin, Pierre-luc, Chao, Clifford, Clark, Chris, Coriell, Jeremy, Daley, Mike, Dau, Matt, Dean, Jeffrey, Gelb, Ben, Ghaemmaghami, Tara Vazir, Gottipati, Rajendra, Gulland, William, Hagmann, Robert, Ho, C. Richard, Hogberg, Doug, Hu, John, Hundt, Robert, Hurt, Dan,

Ibarz, Julian, Jaffey, Aaron, Jaworski, Alek, Kaplan, Alexander, Khaitan, Harshit, Killebrew, Daniel, Koch, Andy, Kumar, Naveen, Lacy, Steve, Laudon, James, Law, James, Le, Diemthu, Leary, Chris, Liu, Zhuyuan, Lucke, Kyle, Lundin, Alan, MacKean, Gordon, Maggiore, Adriana, Mahony, Maire, Miller, Kieran, Nagarajan, Rahul, Narayanaswami, Ravi, Ni, Ray, Nix, Kathy, Norrie, Thomas, Omernick, Mark, Penukonda, Narayana, Phelps, Andy, Ross, Jonathan, Ross, Matt, Salek, Amir, Samadiani, Emad, Severn, Chris, Sizikov, Gregory, Snelham, Matthew, Souter, Jed, Steinberg, Dan, Swing, Andy, Tan, Mercedes, Thorson, Gregory, Tian, Bo, Toma, Horia, Tuttle, Erick, Vasudevan, Vijay, Walter, Richard, Wang, Walter, Wilcox, Eric & Yoon, Doe Hyun (2017): In-datacenter performance analysis of a tensor processing unit, 2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA), pp. 1–12.

Kainz, Florian, Bogart, Rod & Stanczyk, Piotr (2009): Technical introduction to openexr, Industrial light and magic **21**.

Kaja, Endri, Gerlin, Nicolas, Vaddeboina, Mounika, Rivas, Luis, Prebeck, Sebastian, Han, Zhao, Devarajegowda, Keerthikumara & Ecker, Wolfgang (2021): Towards fault simulation at mixed register-transfer/gate-level models, 2021 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT), pp. 1–6.

Kazakova, N., Sung, R., Durdle, N., Margala, M. & Lamoureux, J. (2001): Fast and low-power inner product processor, ISCAS 2001. The 2001 IEEE International Symposium on Circuits and Systems (Cat. No.01CH37196), Bd. 4, pp. 646–649 vol. 4.

Kelleher, J.D. (2019): Deep Learning, The MIT Press Essential Knowledge series, MIT Press.

Kharya, P (2020): Tensorfloat-32 in the a100 gpu accelerates ai training hpc up to 20x, NVIDIA Corporation, Tech. Rep .

Kim, Youngjoon & Kim, Lee-Sup (2001): A low power carry select adder with reduced area, ISCAS 2001. The 2001 IEEE International Symposium on Circuits and Systems (Cat. No.01CH37196), Bd. 4, pp. 218–221 vol. 4.

Kiningham, Kevin, Graczyk, Michael, Ramkumar, Athul & Stanford, SCPD (2016): Design and analysis of a hardware cnn accelerator, Small: Nano Micro **27**: 6.

Kjolstad, Fredrik, Kamil, Shoaib, Chou, Stephen, Lugato, David & Amarasinghe, Saman (2017): The tensor algebra compiler, Proc. ACM Program. Lang. **1**(OOPSLA). `https://doi.org/10.1145/3133901`

Kogge, Peter M. & Stone, Harold S. (1973): A parallel algorithm for the efficient solution of a general class of recurrence equations, IEEE Transactions on Computers **C-22**: 786–793.

Krizhevsky, Alex, Sutskever, Ilya & Hinton, Geoffrey E. (2017): Imagenet classification with deep convolutional neural networks, Commun. ACM **60**(6): 84–90. `https://doi.org/10.1145/3065386`

Kuang, Shiann-Rong & Wang, Jiun-Ping (2010): Design of power-efficient configurable booth multiplier, IEEE Transactions on Circuits and Systems I: Regular Papers **57**: 568–580.

Kulaib, A. R., Shubair, R. M., Al-Qutayri, M. A. & Ng, Jason W. P. (2011): An overview of localization techniques for wireless sensor networks, 2011 International Conference on Innovations in Information Technology, pp. 167–172.

Kumar, Puli Anil (2019): FPGA implementation of the trigonometric functions using the CORDIC algorithm, 2019 5th International Conference on Advanced Computing & Communication Systems (ICACCS), IEEE.

Lecun, Y., Bottou, L., Bengio, Y. & Haffner, P. (1998): Gradient-based learning applied to document recognition, Proceedings of the IEEE **86**: 2278–2324.

Li, Dong-Ze, Gong, Hao-Ran & Chang, Yu-Chun (2018): Implementing riscv system-on-chip for acceleration of convolution operation and activation function based on fpga, 2018 14th IEEE International Conference on Solid-State and Integrated Circuit Technology (ICSICT), pp. 1–3.

Li, Kai, Mao, Wei, Xie, Xinang, Cheng, Quan, Xie, Huan, Dong, Zhenjiang & Yu, Hao (2021): Multiple-precision floating-point dot product unit for efficient convolution computation, 2021 IEEE 3rd International Conference on Artificial Intelligence Circuits and Systems (AICAS), pp. 1–4.

Lian, Xiaocong, Liu, Zhenyu, Song, Zhourui, Dai, Jiwu, Zhou, Wei & Ji, Xiangyang (2019): High-performance fpga-based cnn accelerator with block-floating-point arithmetic, IEEE Transactions on Very Large Scale Integration (VLSI) Systems **27**: 1874–1885.

Lin, Ming-Hua, Carlsson, John Gunnar, Ge, Dongdong, Shi, Jianming & Tsai, Jung-Fa (2013): A review of piecewise linearization methods, Mathematical Problems in Engineering **2013**: 1–8.

Lin, Rong, Botha, A.S., Kerr, K.E. & Brown, G.A. (1999): An inner product processor design using novel parallel counter circuits, AP-ASIC'99. First IEEE Asia Pacific Conference on ASICs (Cat. No.99EX360), pp. 99–102.

Liu, W., Swartzlander, E. & O'Neill, M. (2013): Design of Semiconductor QCA Systems, Microtechnology/Nanotechnology, Artech House.

Liu, Yangxurui, Liu, Liang, Öwall, Viktor & Chen, Shuming (2014): Implementation of a dynamic wordlength simd multiplier, 2014 NORCHIP, pp. 1–4.

Liu, Zhi-Gang, Whatmough, Paul N., Zhu, Yuhao & Mattina, Matthew (2022): S2ta: Exploiting structured sparsity for energy-efficient mobile cnn acceleration, 2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA), pp. 573–586.

Lopera, Daniela Sánchez, Servadei, Lorenzo, Kasi, Vishwa Priyanka, Prebeck, Sebastian & Ecker, Wolfgang (2021): Rtl delay prediction using neural networks, 2021 IEEE Nordic Circuits and Systems Conference (NorCAS), pp. 1–7.

lowRisc (2022): ibex, [online], accessed on 04.10.2022.
`https://github.com/lowrisc/ibex`

Lu, Mi (2005): Arithmetic and logic in computer systems, John Wiley & Sons.

MathWorks (2022): Convolutional neural network, [online], accessed on 04.10.2022.
`https://de.mathworks.com/discovery/convolutional-neural-network-matlab.html`

McCarthy, John (2007): What is ai, [online], accessed on 04.10.2022.
`http://jmc.stanford.edu/articles/whatisai.html`

Muscedere, R., Dimitrov, V., Jullien, G.A. & Miller, W.C. (2005): Efficient techniques for binary-to-multidigit multidimensional logarithmic number system conversion using range-addressable look-up tables, IEEE Transactions on Computers **54**(3): 257–271.

Namin, Ashkan Hosseinzadeh, Leboeuf, Karl, Muscedere, Roberto, Wu, Huapeng & Ahmadi, Majid (2009): Efficient hardware implementation of the hyperbolic tangent sigmoid function, 2009 IEEE International Symposium on Circuits and Systems, IEEE.

Neumeier, Andreas (2020): Audio source localization on resource constrained devices using supervised machine learning., Master's thesis, Hochschule München University of Applied Sciences.

NVIDIA (2022): Nvdla open source project, [online], accessed on 04.10.2022.
`https://github.com/nvdla`

Parashar, Angshuman, Rhu, Minsoo, Mukkara, Anurag, Puglielli, Antonio, Venkatesan, Rangharajan, Khailany, Brucek, Emer, Joel, Keckler, Stephen W. & Dally, William J. (2017): Scnn: An accelerator for compressed-sparse convolutional neural networks, SIGARCH Comput. Archit. News **45**(2): 27–40.
`https://doi.org/10.1145/3140659.3080254`

Philips, Semiconductors (1986): I2s bus specification, [online], accessed on 31.09.2022.
`https://www.nxp.com/docs/en/user-manual/UM11732.pdf`

Piazza, F., Uncini, A. & Zenobi, M. (1993): Neural networks with digital LUT activation functions, Proceedings of 1993 International Conference on Neural Networks (IJCNN-93-Nagoya, Japan), IEEE.

Powers, David M. W. (2020): Evaluation: from precision, recall and f-measure to roc, informed-

ness, markedness and correlation, International Journal of Machine Learning Technology 2:1 (2011), pp.37-63 .

Prasanna, Dasari Lakshmi & Prabhu, E. (2019): An efficient fused floating-point dot product unit using vedic mathematics, 2019 3rd International Conference on Trends in Electronics and Informatics (ICOEI), pp. 12–15.

Prebeck, Sebastian, Ashok, Sathya, Vaddeboina, Mounika, Devarajegowda, Keerthikumara & Ecker, Wolfgang (2022): A scalable, configurable and programmable vector dot-product unit for edge ai, MBMV 2022; 25th Workshop, pp. 1–9.

Prebeck, Sebastian, Lawand, Wafic, Vaddeboina, Mounika & Ecker, Wolfgang (2022): A smart hw-accelerator for non-uniform linear interpolation of ml-activation functions, in A. Orailoglu, M. Reichenbach & M. Jung (eds), Embedded Computer Systems: Architectures, Modeling, and Simulation, Springer International Publishing, Cham, pp. 267–282.

Quan, Heng, Xiao, Ruijin, You, Kaidi, Zeng, Xiaoyang & Yu, Zhiyi (2010): A novel vector/simd multiply-accumulate unit based on reconfigurable booth array, 2010 10th IEEE International Conference on Solid-State and Integrated Circuit Technology, pp. 524–526.

Rajput, Ravindra P. & Swamy, M.N. Shanmukha (2012): High speed modified booth encoder multiplier for signed and unsigned numbers, 2012 UKSim 14th International Conference on Computer Modelling and Simulation, pp. 649–654.

Raut, Gopal, Rai, Shubham, Vishvakarma, Santosh Kumar & Kumar, Akash (2020): A CORDIC based configurable activation function for ANN applications, 2020 IEEE Computer Society Annual Symposium on VLSI (ISVLSI), IEEE.

Reuther, Albert, Michaleas, Peter, Jones, Michael, Gadepally, Vijay, Samsi, Siddharth & Kepner, Jeremy (2019): Survey and benchmarking of machine learning accelerators, 2019 IEEE High Performance Extreme Computing Conference (HPEC), pp. 1–9.

Rice, R. & Plaunt, J. (1971): Adaptive variable-length coding for efficient compression of spacecraft television data, IEEE Transactions on Communication Technology **19**: 889–897.

RISC-V P Extension Proposal (2021): [online], accessed on 04.10.2022.
`https://github.com/riscv/riscv-p-spec`

RISC-V Specification (2021): [online], accessed on 04.10.2022.
`https://riscv.org/`

RISC-V V Extension Proposal (2021): [online], accessed on 04.10.2022.
`https://github.com/riscv/riscv-v-spec`

Sadeghi, Mohsen, Zahedi, Mahya & Ali, Maaruf (2019): The cascade carry array multiplier–

a novel structure of digital unsigned multipliers for low-power consumption and ultra-fast applications, Annals of Emerging Technologies in Computing (AETiC), Print ISSN pp. 2516–0281.

Samuel, A. L. (1959): Some studies in machine learning using the game of checkers, IBM Journal of Research and Development **3**: 210–229.

Sankaradas, Murugan, Jakkula, Venkata, Cadambi, Srihari, Chakradhar, Srimat, Durdanovic, Igor, Cosatto, Eric & Graf, Hans Peter (2009): A massively parallel coprocessor for convolutional neural networks, 2009 20th IEEE International Conference on Application-specific Systems, Architectures and Processors, pp. 53–60.

Shahbahrami, Asadollah, Bahrampour, Ramin, Rostami, Mobin Sabbaghi & Mobarhan, Mostafa Ayoubi (2011): Evaluation of huffman and arithmetic algorithms for multimedia compression standards, ArXiv Preprint ArXiv:1109.0216 .

Shannon, C. E. (1948): A mathematical theory of communication, The Bell System Technical Journal **27**: 379–423.

Shi, Runbin, Liu, Junjie, Hayden So, K.-H., Wang, Shuo & Liang, Yun (2019): E-lstm: Efficient inference of sparse lstm on embedded heterogeneous system, 2019 56th ACM/IEEE Design Automation Conference (DAC), pp. 1–6.

Shirol, Suhas B., Ramakrishna, S. & Shettar, Rajashekar B. (2019): Design and implementation of adders and multiplier in fpga using chipscope: A performance improvement, Information and Communication Technology for Competitive Strategies, Springer Singapore, Singapore, pp. 11–19.

Shomron, Gil, Gabbay, Freddy, Kurzum, Samer & Weiser, Uri (2021): Post-training sparsity-aware quantization, ArXiv Preprint ArXiv:2105.11010 .

Shun, Zhou, Pfander, Oliver A., Pfleiderer, Hans-Jorg & Bermak, Amine (2007): A vlsi architecture for a run-time multi-precision reconfigurable booth multiplier, 2007 14th IEEE International Conference on Electronics, Circuits and Systems, pp. 975–978.

Shuo, Wang, Zhe, Li, Caiwen, Ding, Bo, Yuan, Qinru, Qiu, Yanzhi, Wang & Yun, Liang (2018): C-lstm: Enabling efficient lstm using structured compression techniques on fpgas, Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA '18, Association for Computing Machinery, New York, NY, USA, p. 11–20.
`https://doi.org/10.1145/3174243.3174253`

SiFive (2022): [online], accessed on 30.09.2022.
`https://www.sifive.com/risc-v-core-ip`

Simonyan, Karen & Zisserman, Andrew (2014): Very deep convolutional networks for large-scale image recognition, ArXiv Preprint ArXiv:1409.1556 .

Singh, Prince & Agrawal, Sunil (2013): Tdoa based node localization in wsn using neural networks, 2013 International Conference on Communication Systems and Network Technologies, pp. 400–404.

Sklansky, J. (1960): Conditional-sum addition logic, IRE Transactions on Electronic Computers **EC-9**: 226–231.

Smith, S. P. & Torng, H. C. (1985): Design of a fast inner product processor, 1985 IEEE 7th Symposium on Computer Arithmetic (ARITH), pp. 38–43.

Stevens, Andrew, Ecker, Wolfgang & Prebeck, Sebastian (2022): Accelerating processor based artificial neural network computation, US Patent App. 17/001,977.

Stevens, Andrew, Petig, Christof, Mueller-Gritschneder, Daniel & Rafael, Stahl (2020): Rfc interpreter-less code generation for tensorflow lite for microcontrollers, [online], accessed on 30.9.2022.
`https://github.com/cpetig/tflite_micro_compiler`

Sze, Vivienne, Chen, Yu-Hsin, Yang, Tien-Ju & Emer, Joel S. (2017): Efficient processing of deep neural networks: A tutorial and survey, Proceedings of the IEEE **105**(12): 2295–2329.

Sze, Vivienne, Chen, Yu-Hsin, Yang, Tien-Ju & Emer, Joel S (2020): Efficient Processing of Deep Neural Networks, Morgan & Claypool Publishers.

Szegedy, Christian, Liu, Wei, Jia, Yangqing, Sermanet, Pierre, Reed, Scott, Anguelov, Dragomir, Erhan, Dumitru, Vanhoucke, Vincent & Rabinovich, Andrew (2015): Going deeper with convolutions, Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR).

Sánchez Lopera, Daniela, Servadei, Lorenzo, Prebeck, Sebastian & Ecker, Wolfgang (2022): Early rtl delay prediction using neural networks, Microprocessors and Microsystems **94**: 104671.
`https://www.sciencedirect.com/science/article/pii/S0141933122002010`

Tan, D., Danysh, A. & Liebelt, M. (2003): Multiple-precision fixed-point vector multiply-accumulator using shared segmentation, Proceedings 2003 16th IEEE Symposium on Computer Arithmetic, pp. 12–19.

Tsai, Meng-Hung, Chen, Yi-Ting, Cheng, Wen-Sheng, Teng, Jun Xian & Jou, Shyh-Jye (2002): Sub-word and reduced-width booth multipliers for dsp applications, 2002 IEEE International Symposium on Circuits and Systems (ISCAS), Bd. 3, pp. III–III.

Tsmots, Ivan, Skorokhoda, Oleksa & Rabyk, Vasyl (2019): Hardware implementation of

sigmoid activation functions using FPGA, 2019 IEEE 15th International Conference on the Experience of Designing and Application of CAD Systems (CADSM), IEEE.

Volder, Jack E. (1959): The CORDIC trigonometric computing technique, IRE Transactions on Electronic Computers **EC-8**(3): 330–334.

Wallace, C. S. (1964): A suggestion for a fast multiplier, IEEE Transactions on Electronic Computers **EC-13**: 14–17.

Wang, Chao, Gong, Lei, Yu, Qi, Li, Xi, Xie, Yuan & Zhou, Xuehai (2017): Dlau: A scalable deep learning accelerator unit on fpga, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **36**: 513–517.

Wang, Junsong, Lou, Qiuwen, Zhang, Xiaofan, Zhu, Chao, Lin, Yonghua & Chen, Deming (2018): Design flow of accelerating hybrid extremely low bit-width neural network in embedded fpga, 2018 28th International Conference on Field Programmable Logic and Applications (FPL), pp. 163–1636.

Waterman, Andrew (2011): Improving energy efficiency and reducing code size with risc-v compressed, Master's thesis .

Werner, Michael, Zeraliu, Igli, Han, Zhao, Prebeck, Sebastian, Servardei, Lorenzo & Ecker, Wolfgang (2020): Optimized hw/fw generation from an abstract register interface model, 2020 23rd Euromicro Conference on Digital System Design (DSD), pp. 35–39.

Wiedemann, Simon, Kirchhoffer, Heiner, Matlage, Stefan, Haase, Paul, Marban, Arturo, Marinč, Talmaj, Neumann, David, Nguyen, Tung, Schwarz, Heiko, Wiegand, Thomas, Marpe, Detlev & Samek, Wojciech (2020): Deepcabac: A universal compression algorithm for deep neural networks, IEEE Journal of Selected Topics in Signal Processing **14**(4): 700–714.

Wiedemann, Simon, Shivapakash, Suhas, Becking, Daniel, Wiedemann, Pablo, Samek, Wojciech, Gerfers, Friedel & Wiegand, Thomas (2021): Fantastic4: A hardware-software co-design approach for efficiently running 4bit-compact multilayer perceptrons, IEEE Open Journal of Circuits and Systems **2**: 407–419.

Williams, Samuel, Waterman, Andrew & Patterson, David (2009): Roofline: An insightful visual performance model for multicore architectures, Commun. ACM **52**(4): 65–76.
`https://doi.org/10.1145/1498765.1498785`

Witten, Ian H., Neal, Radford M. & Cleary, John G. (1987): Arithmetic coding for data compression, Commun. ACM **30**(6): 520–540.
`https://doi.org/10.1145/214762.214771`

Wofk, Diana, Ma, Fangchang, Yang, Tien-Ju, Karaman, Sertac & Sze, Vivienne (2019): Fast-

depth: Fast monocular depth estimation on embedded systems, 2019 International Conference on Robotics and Automation (ICRA), pp. 6101–6108.

Wu, Di, Tang, Qingming, le Zhao, Yong, Zhang, Ming, Fu, Yingnan & Zhang, Debing (2020): Easyquant: Post-training quantization via scale optimization, ArXiv Preprint ArXiv:2006.16669 .

Xu, Rui, Ma, Sheng, Wang, Yaohua & Guo, Yang (2020): Cmsa: Configurable multi-directional systolic array for convolutional neural networks, 2020 IEEE 38th International Conference on Computer Design (ICCD), pp. 494–497.

Yan, Wen & Ercegovac, Miloš D. (2019): An energy efficient carry-free inner product unit, 2019 IEEE International Conference on Signal, Information and Data Processing (ICSIDP), pp. 1–6.

Yin, Shouyi, Tang, Shibin, Lin, Xinhan, Ouyang, Peng, Tu, Fengbin, Liu, Leibo & Wei, Shaojun (2019): A high throughput acceleration for hybrid neural networks with efficient resource management on fpga, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **38**: 678–691.

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

Zhang, Ming, Vassiliadis, S. & Delgado-Frias, J.G. (1996): Sigmoid generators for neural computing using piecewise approximations, IEEE Transactions on Computers **45**(9): 1045–1049.

Zhang, Shijin, Du, Zidong, Zhang, Lei, Lan, Huiying, Liu, Shaoli, Li, Ling, Guo, Qi, Chen, Tianshi & Chen, Yunji (2016): Cambricon-x: An accelerator for sparse neural networks, The 49th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO-49, IEEE Press.

Zhang, Shuhua, Tong, Jie, Zhang, Jun, Lei, Yuqing, Zhang, Minghao, Li, Dang & Wang, Lanruo (2020): A RISC-v based coprocessor accelerator technology research for convolution neural networks, Journal of Physics: Conference Series **1631**: 012002. `https://doi.org/10.1088/1742-6596/1631/1/012002`

Zhang, Xinyue, Li, Zhaolin & Zheng, Qingwei (2009): Design of a configurable fixed-point multiplier for digital signal processor, 2009 Asia Pacific Conference on Postgraduate Research in Microelectronics & Electronics (PrimeAsia), pp. 217–220.

A. Appendix

start_state	condition	incr_col	rst_col	incr_row	rst_row	incr_repeat
init	A	0	0	0	0	0
	b	1	0	0	0	0
	c	0	0	1	0	0
	d	0	0	0	0	1
cols	b	1	0	0	0	0
	e	0	0	0	0	0
	f	0	0	0	0	0
	g	0	1	1	0	0
	h	0	0	0	0	0
	i	0	1	0	1	1
	k	0	0	0	0	0
rows	b	1	0	0	0	0
	e	0	0	0	0	0
	f	0	0	0	0	0
	g	0	0	1	0	0
	h	0	0	0	0	0
	i	0	0	0	1	1
	k	0	0	0	0	0
repeats	b	1	0	0	0	0
	e	0	0	0	0	0
	f	0	0	0	0	0
	g	0	0	1	0	0
	h	0	0	0	0	0
	i	0	0	0	0	1
	k	0	0	0	0	0
wait_cols	A	0	0	0	0	0
	l	1	0	0	0	0
wait_rows	A	0	0	0	0	0
	l	0	0	1	0	0
wait_repeats	A	0	0	0	0	0
	l	0	0	0	0	1
wait_rows_rst	A	0	0	0	0	0
	l	0	1	1	0	0
wait_repeats_rst	A	0	0	0	0	0
	l	0	1	0	1	1

Table A.1.: Address Generator FSM transition outputs

start_state	condition	ready_vect	ready_addr	valid	save_vect	incr_itm	forward
init	A	0	0	0	0	0	1
	B	1	0	0	1	0	1
	d	1	1	1	1	1	1
no_data	A	0	0	0	0	0	0
	B	1	0	0	1	0	0
	d	1	1	1	1	1	0
output_vect	D	0	0	0	0	0	0
	e	0	0	0	0	0	0
	h	1	1	1	1	1	0
	i	0	1	1	0	1	0
	k	0	0	0	0	0	0
no_addr_fifo_busy	E	0	0	0	0	0	0
	h	1	1	1	1	1	0
	i	0	1	1	0	1	0
	k	0	0	0	0	0	0

Table A.2.: compute fetch skip vector FSM transition outputs

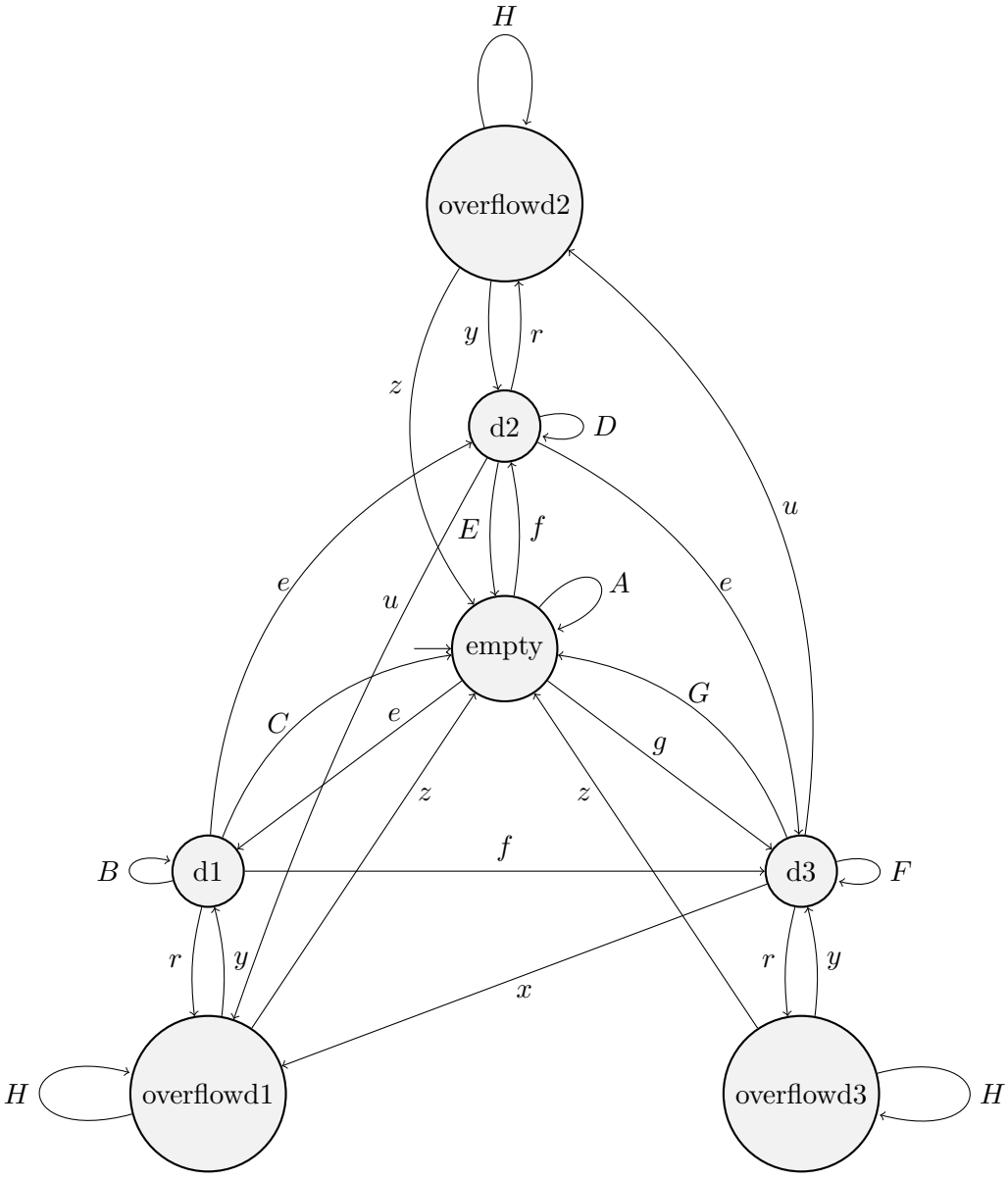


Figure A.1.: extract data FSM

variable	condition
a	$\neg valid \wedge en$
b	$valid \wedge \neg items1 \wedge \neg items2 \wedge \neg items3 \wedge \neg items4 \wedge \neg footprint\_done \wedge en$
c	$valid \wedge items4 \wedge \neg footprint\_done \wedge busy \wedge en$
d	$valid \wedge items4 \wedge \neg footprint\_done \wedge \neg busy \wedge en$
e	$valid \wedge items1 \wedge \neg footprint\_done \wedge en$
f	$valid \wedge items2 \wedge \neg footprint\_done \wedge en$
g	$valid \wedge items3 \wedge \neg footprint\_done \wedge en$
h	$\neg en$
i	$valid \wedge footprint\_done \wedge busy \wedge en$
k	$valid \wedge \neg items1 \wedge \neg items2 \wedge \neg items3 \wedge \neg items4 \wedge footprint\_done \wedge \neg busy \wedge en$
l	$valid \wedge items1 \wedge footprint\_done \wedge \neg busy \wedge en$
m	$valid \wedge items2 \wedge footprint\_done \wedge \neg busy \wedge en$
n	$valid \wedge items3 \wedge footprint\_done \wedge \neg busy \wedge en$
o	$valid \wedge items4 \wedge footprint\_done \wedge \neg busy \wedge en$
p	$valid \wedge items3 \wedge \neg footprint\_done \wedge busy \wedge en$
q	$valid \wedge items3 \wedge \neg footprint\_done \wedge \neg busy \wedge en$
r	$valid \wedge items4 \wedge \neg busy \wedge en$
s	$valid \wedge items2 \wedge \neg footprint\_done \wedge busy \wedge en$
t	$valid \wedge items2 \wedge \neg footprint\_done \wedge \neg busy \wedge en$
u	$valid \wedge items3 \wedge \neg busy \wedge en$
v	$busy \wedge en$
w	$valid \wedge items1 \wedge \neg footprint\_done \wedge \neg busy \wedge en$
x	$valid \wedge items2 \wedge \neg busy \wedge en$
y	$\neg footprint\_done \wedge \neg busy \wedge en$
z	$footprint\_done \wedge \neg busy \wedge en$
A	$a \vee b \vee c \vee d \vee h \vee i \vee k \vee l \vee m \vee n \vee o$
B	$a \vee b \vee c \vee h \vee i \vee p$
C	$k \vee l \vee m \vee n \vee q$
D	$a \vee b \vee c \vee h \vee i \vee p \vee s$
E	$k \vee l \vee m \vee t$
F	$a \vee b \vee h \vee v$
G	$k \vee l \vee w$
H	$h \vee v$

Table A.3.: extract data FSM transition conditions

start_state	condition	valid	reg0	reg1	reg2	reg3	ready	ovf1	ovf2	ovf3	footprint_done	skip	items1	items2	items3	items4
empty	a	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	b	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
	c	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	d	1	1	1	1	1	1	0	0	0	0	0	0	0	0	1
	e	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0
	f	0	1	1	0	0	1	0	0	0	0	0	0	0	0	0
	g	0	1	1	0	0	1	0	0	0	0	0	0	0	0	0
	h	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	i	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	k	1	1	1	1	1	1	0	0	0	1	1	0	0	0	0
	l	1	1	1	1	1	1	0	0	0	1	0	1	0	0	0
	m	1	1	1	1	1	1	0	0	0	1	0	0	1	0	0
	n	1	1	1	1	1	1	0	0	0	1	0	0	0	1	0
	o	1	1	1	1	1	1	0	0	0	1	0	0	0	0	1
d1	a	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	b	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
	c	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	e	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0
	f	0	0	1	1	0	1	0	0	0	0	0	0	0	0	0
	h	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	i	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	k	1	0	1	1	1	1	0	0	0	1	0	0	0	0	0
	l	1	0	1	1	1	1	0	0	0	1	0	0	1	0	0
	m	1	0	1	1	1	1	0	0	0	1	0	0	0	1	0
	n	1	0	1	1	1	1	0	0	0	1	0	0	0	0	1
	p	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	q	1	0	1	1	1	1	0	0	0	0	0	0	0	0	1
	r	1	0	1	1	1	0	0	0	0	0	0	0	0	0	1
d2	a	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	b	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
	c	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	e	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0
	h	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	i	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	k	1	0	1	1	1	1	0	0	0	1	0	1	0	0	0

continued ...

. . . continued

start_state	condition	valid	reg0	reg1	reg2	reg3	ready	ovf1	ovf2	ovf3	footprint_done	skip	items1	items2	items3	items4
	l	1	0	0	1	1	1	0	0	0	1	0	0	0	1	0
	m	1	0	0	1	1	1	0	0	0	1	0	0	0	0	1
	p	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	r	1	0	0	1	1	0	0	0	0	0	0	0	0	0	1
	s	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	t	1	0	0	1	1	1	0	0	0	0	0	0	0	0	1
	u	1	0	0	1	1	0	0	0	0	0	0	0	0	0	1
d3	a	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	b	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
	v	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	w	1	0	0	0	1	1	0	0	0	0	0	0	0	0	1
	x	1	0	0	0	1	0	0	0	0	0	0	0	0	0	1
	u	1	0	0	0	1	0	0	0	0	0	0	0	0	0	1
	r	1	0	0	0	1	0	0	0	0	0	0	0	0	0	1
	h	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	l	1	0	0	0	1	1	0	0	0	1	0	0	0	0	1
	k	1	0	0	0	1	1	0	0	0	1	0	0	0	1	0
overflowd1	H	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	y	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
	z	1	1	1	1	1	1	1	0	0	1	0	1	0	0	0
overflowd2	H	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	y	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
	z	1	1	1	1	1	1	1	0	0	1	0	0	1	0	0
overflowd3	H	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	y	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
	z	1	1	1	1	1	1	1	0	0	1	0	0	0	1	0

Table A.4.: extract data FSM transition outputs

List of Acronyms

A

AI - Artificial Intelligence 17–21, 39–41, 53, 55, 58, 70, 74, 78, 145, 146
ANN - Artificial Neural Network 22
ASIC - Application Specific Integrated Circuit 40, 74, 76, 80, 151

B

BFP - Block-Floating-Point 75
BK - Brent Kung Adder 62

C

CHW - Channel Height Width 37
CLA - Carry Lookahead Adder 59–61, 64, 65, 72
CNN - Convolutional Neural Network 22, 23, 26, 75, 77–79, 115
Conv - Convolutional 23, 26, 86, 133, 140, 144, 162, 164, 165, 167, 168
CPA - Carry Propagate Adder 59, 63, 65
CPU - Central Processing Unit 19, 43, 44, 76, 79–85, 94, 117, 131, 132, 134, 135, 145, 149, 179
CSA - Carry Save Adder 64, 65, 68, 72, 128
CSLA - Carry Select Adder 63
CSR - Control Status Register 82–85, 89, 93, 94, 96, 104, 124, 132–135, 139, 171

D

DL - Deep Learning 21
DMA - Direct Memory Access 76, 84
DNN - Deep Neural Network 17–19, 21–23, 34, 36, 37, 39, 48, 56, 78
DSP - Digital Signal Processor 65, 69, 70, 74, 79

E

E-LSTM - Embedded Long Short-Term Memory 75

F

FA - Full Adder 59, 60
FC - Fully Connected 23, 26, 49, 75, 86, 115, 133, 140, 142–144, 158, 162, 164, 167, 168
FFNN - Fast Forward Neural Network 22, 23
FFT - Fast Fourier Transformation 76
FPGA - Field Programmable Gate Array 40, 41, 74–76, 79, 80, 155
FSM - Finite State Machine 81–84, 96, 104, 106, 107, 110–114, 172
FW - Firware 175

G

H

I

K

L

M

N

O

P

R

RCA - Ripple Carry Adder 59, 60, 63, 65, 67, 153
RF - Register File 58, 131, 135–139
RNN - Recurrent Neural Network 22, 23
ROM - Read Only Memory 109, 159–161
RTL - Register Transfer Level 177

S

SIMD - Single Instruction Multiple Data ... 46, 47, 58, 59, 69, 71, 72, 79, 85, 89, 91, 136, 137, 139, 152, 153, 160
SK - Sklansky Adder 61, 62
SoC - System-on-Chip 35, 79
SW - Software 46, 51–53, 72, 74, 82–84, 89, 92, 93, 115, 117, 118, 131, 132, 134, 135, 137, 140, 144, 145, 147, 152, 160, 162, 167, 168, 171, 172

T

TDoA - Time Difference of Arrival 155–157
TF - TensorFlow 53, 72, 80, 86, 145–148, 162, 172
TPU - Tensor Processing Unit 19, 76, 77, 145

U

uC - Microcontroller 39, 41, 49, 51, 79, 171