POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master's Degree Thesis

PULP-TCN: Optimizing temporal convolutional networks on ultra-low power multi-core RISC-V based edge nodes

Supervisors

Prof. Daniele Jahier PAGLIARI

Dott. Alessio BURRELLO

Candidate

Alberto DEQUINO

Academic Year 2019-2020

Summary

The rapidly expanding IoT market is pushing Neural Network (NN) inference directly directly on the edge sensors rather than depend on cloud computing. This way, latency and energy consumption for data transfer is greatly reduced. However, the scarcity of memory and energy resources of deeply embedded devices poses a challenge for the development of smart applications. Optimization strategies, such as quantization, have to be implemented to reduce computational cost and memory footprint .

One area of interest in Machine Learning is Sequence Modelling, the ability to automatically generate, interpret or re-model sequential data, such as audio or text. Recurrent Neural Networks (RNNs) have been canonically considered the ideal starting point for this kind of models. However, recent results prove that Temporal Convolutional Networks (TCNs) can provide better performances with a simpler and easier to understand architecture.

We present PULP-TCN, a set of 1-dimensional convolution kernels aimed for TCN inference on IoT edge devices. These kernels have been developed on GAP-8, an open source RISC-V and PULP (Parallel Ultra-Low Power) processing platform. We used PULP-NN, a library for Quantized Neural Network (QNN) inference, as base.

Experimental results show that PULP-TCN kernels provide better performance and lower memory footprint than PULP-NN on convolution over 1 dimensional data sequences. PULP-TCN also supports deployment of sequence dilation, which allows the convolution layer to explore a wider history size without having to increase the filter size, further lowering its memory footprint.

Acknowledgements

First in order to be acknowledge are my parents, Milva Tolosa and Mario Dequino, and my sister, Federica Dequino. They have always been extremely supporting during my course of study, both economically and emotionally. I'm very happy to know that they have always believed in my capabilities, and never stopped cheering me during these years.

I want to thank my study colleagues, in particular Francesco Campanaro, Marco Lentini, Cosmin Daniel Solomon and Alberto Solaro, who helped me by working in group projects, sharing notes, or simply by keeping company during these years, making attending classes not such a pain.

I also want to give my thanks to my supervisors, Daniele Jahier Pagliari and Alessio Burrello, for helping me out so much during all the phases of the thesis. They have always been ready for ask any question and very patient every time I didn't understand, and I'm very grateful for that.

Finally, a big THANK YOU to all my relatives and friends, near and far, who have supported me during these years. I'd like to write down the name of you all, but I'd never end, because you are so many.

Table of Contents

List of Figures

Chapter 1

Introduction

Machine learning systems have been taking a more relevant role in the past few years in computer science. Thanks to Deep Learning (DL), a subset of Machine Learning, it is possible to create models that can automatically learn complex representations from data[1]. A wide range of applications can be performed with these models, such as computer vision, speech recognition, machine translation, audio synthesis, health or structural monitoring[2].

The connected devices -also known as Internet of Things (IoT)- market has been steadily growing and has been projected to reach 1 trillion devices by 2035[3]. Large networks of wirelessly connected IoT devices are increasingly deployed. The edge nodes act as sensors, collecting data of every kind (like images, audio, temperature, humidity) to be processed and communicated to other nodes. Due to the sensors technical limitations and the computational complexity of DL models, the majority of neural network operations is usually executed on cloud, over high-performance, specialized server Graphical Processing Units (GPUs).

The ever increasing number of connected devices causes an explosion in the amount of data to be elaborated, saturating the bandwidth and increasing latency on the application. Dependency on cloud computation can also be a problem in areas with unreliable network connectivity. Therefore, new solutions are being researched in order to move DL models on the edge nodes[4], where the data is being directly collected. The training phase of the model should still be done on cloud, while moving the inference phase on edge devices. Training is, in fact, a very resource-hungry one-time task. NN inference is instead a lighter task, continuously done on the incoming data stream. This solution removes the necessity to send data to the cloud for processing, decreasing the overall system latency. Due to the fact that wireless transmission is usually less efficient than computation, energy consumption is reduced too[5].

Due to the edge sensors' nature of being deeply embedded systems, the design process for DL models must take in account their scarcity of of resources (such as

memory) and energy issues derived by the fact that they are typically powered by battery or energy harvesters. On the bright side, Micro-Controller Units (MCUs) are highly flexible, very cheap and low-power. Optimization strategies for deep learning on edge devices include trading off small accuracy losses for reduced computational costs and memory footprints, by compressing the models' weight or activation parameters to 8 (or lower) bit data types, so called quatization[6], or eliminating completely redundant or low-significant weights and activations, so-called pruning [7].

One general application of Deep Learning is Sequence Modelling, which is the ability to automatically understand, predict, re-elaborate or generate any kind of sequential data[8]. Sequential data is any form of data where the temporal order matters, such as audio, text, or electrograms.

In the field of Sequence Modelling, deep learning practitioners usually use Recurrent Neural Networks (RNNs) as a starting point, as stated in canonical textbooks too[9]. However Temporal Convolutional Networks (TCNs), a type of Convolutional Neural Network (CNN) specifically designed to deal with temporal sequence, have been recently revalued. Researchers have shown them to be better suited for domains where a long history is required, thanks to their substantially longer memory, outperforming RNNs on tasks such as music and voice modeling or word/character level language modeling[10]. TCNs can be the base for a wide array of applications working over sequential data, such as the analysis of bio-signals gathered from wearable devices[11], voice recognition and synthesis for commercial devices[12], accelerometer signal analysis[13] etc.

The TCNs are based on a 1D fully-convolutional network architecture, where the input data sequence length remains constant in each layer, using causal convolutions to guarantee that no information is "leaked" from future to the past. In causal convolutions, an output at time t is convolved only with elements from time t and earlier in the previous layer. The history size explored by a simple causal convolution is linear in the depth of the network. This may pose a problem on sequence tasks requiring long history. Introducing so-called dilation, i.e. spacing between the input values in the kernel, every output value represent a wider range of inputs, effectively expanding the receptive field of the net, without having to increase the convolution's filter size.

This work focuses on the study and development of TCN inference on multi-core fully programmable edge devices, by presenting a set of kernels focused on implementing convolution over 1-dimensional 8-bit data arrays. these kernels were built upon GAP8, an open-source platform based on the Parallel Ultra-Low-Power (PULP) RISC-V architecture[14].

There are currently no solutions proposed that can properly implement such operations on MCUs. The closest one is PULP-NN[15], a library designed and optimized to implement standard Convolutional Neural Networks that process

2-dimensional data on multi-core architectures based on PULP platforms[16], like GAP8. PULP-NN was used as starting point for the development of the new convolutional kernels.

PULP-NN is in turn inspired by CMSIS-NN, a set of kernels for efficient implementation of neural network applications on Arm Cortex-M processors targeted for intelligent IoT edge devices [17].

While the PULP-NN convolution kernel could, in fact, work on 1D inputs without any modification, the performance were not at a desirable level. Moreover, it did not support any dilation rate higher than 1 without ... zero padding, with a corresponding waste of operations and memory.

Four different convolution kernels have been created, each one implementing a different optimization strategy:

- The first kernel got entirely rid of the buffering step that was necessary in the original PULP-NN by exploiting the 1-dimensional input data pattern. While testing reveals very good performance results, this kernel still doesn't support dilation.
- The second kernel implements dilation by reintroducing the buffering step, exploiting MCHAN, a Direct Memory Access (DMA) engine specifically developed for integration in the PULP platform[18].
- The third kernel is an attempt to exploit the DMA's parallel memory transfers by doubling the buffering space.
- Finally, the fourth kernel is an implementation of the indirect convolution algorithm[19], which replaces the data buffer with an "indirection" buffer of pointers to data. This drastically decreases the memory usage, but adds an extra execution loop in the multiply-and-accumulate phase due to the fact that data cannot be accessed linearly this way.

Each one of these kernels is optimal under different conditions. By modelling the testing results it's possible to automatically choose the best convolution kernel for each layer based on its parameters (input/output channels, filter and input size) to be implemented.

The rest of the thesis is structured as follows.

Chapter 2 introduces the required background on deep learning, and the description of the PULP+GAP8 architecture and SDK.

Chapter 3 summarizes various works related to the Convolutional Neural Network inference over IoT devices.

Chapter 4 is a description of the developed kernels, and chapter 5 contains information about their performance and experimental results.

Chapter 2

Background

2.1 Neural Networks

2.1.1 Overview

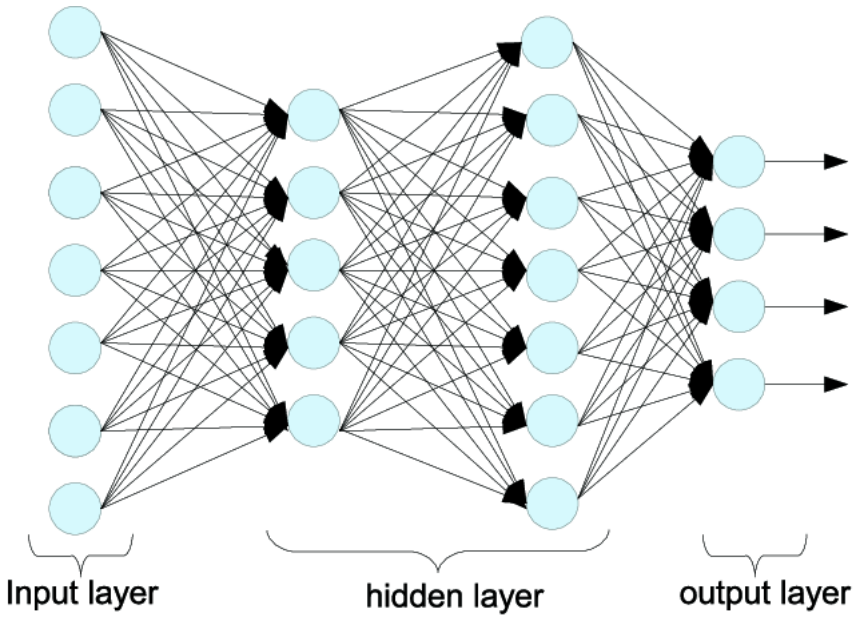

Figure 2.1: Neural Network example [20]

The basic technical ideas behind deep learning aren't recent at all, however only recently they have started to take off. One of the main reasons of this sudden raise in popularity is the explosion on availability of data for training, caused by the radical changes in our society in the last decades. Traditional learning algorithms

simply cannot profit from this amount of data. Combined with recent algorithmic and technological innovations, and easy access to open-source frameworks, deep learning managed to become one of the main focuses in the machine learning field [1]. Neural Networks are artificial networks of Neurons, generic computational units roughly inspired by the human brain. Their role is to elaborate multiple input data to produce some output information.

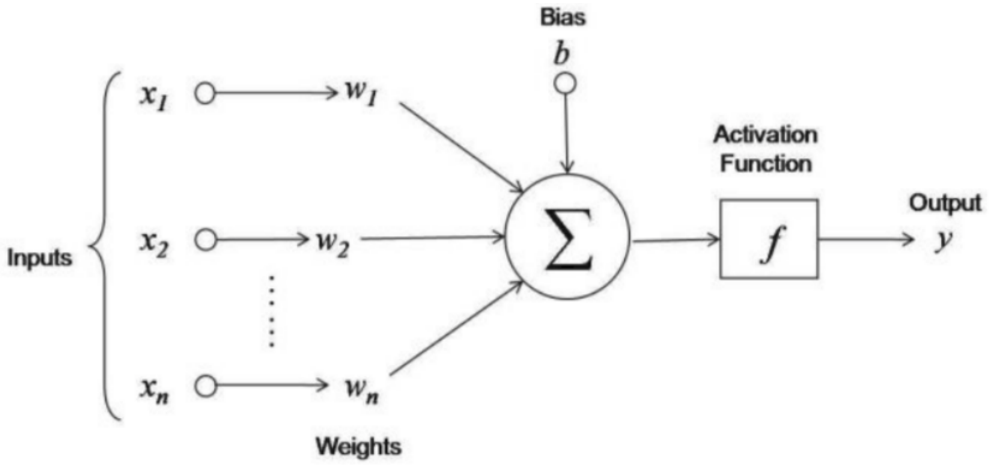

Figure 2.2: Neuron structure [21]

A neuron is a weighted sum of multiple input features (with an optional bias). The result is then passed in a non-linear Activation Function, in order to remove the linearity of the algebraic addition. Figure 2.2 is a typical neuron structure that can be represented by the formula:

$$y_i = f(\sum_{i=1}^{n} w_i x_i + b) \tag{2.1}$$

Where w_i are the weights assigned to each input x_i, b is the bias and h the non-linear activation function. Biases are used to shift the input activation on the x plane, increasing the overall network flexibility. Weights and biases are updated during the training phase of the model, which is based on gradient descent.

In particular, based on an error (or loss) function computed at the output of the network, the weights and biases gradients are computed via a technique called back-propagation. Also, if linearity is not removed, any benefit from stacking multiple layers is lost, since they could all be replaced by a mathematically equivalent single layer.

Some popular activation functions are:

- **Sigmoid function**

$$h(x) = \frac{1}{1 + exp(-x)} \tag{2.2}$$

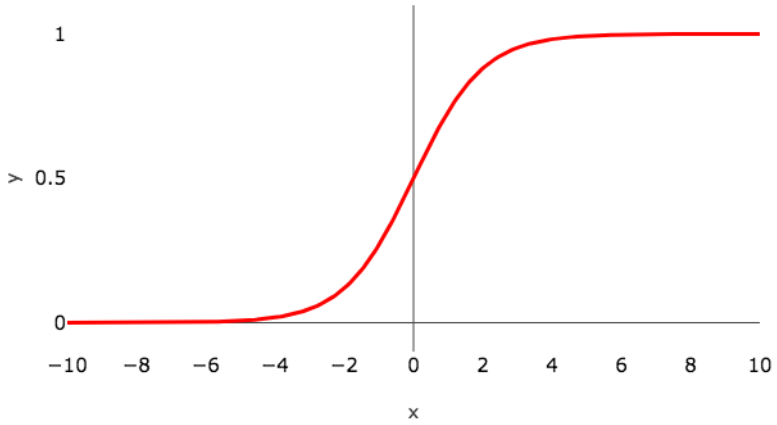

Figure 2.3: Sigmoid function [22]

The Sigmoid or Logistic function is one of the most classic and simple activation functions in NN literature, forcing the output value to be in the (0,1) range. Used in larger neural networks it could make the error gradient vanish or explode, therefore it's not the most optimal choice. However, this function is still useful in the output layer of a binary classification network (where output is either 0 or 1), because it allows to interpret the output as a probability.

- **Hyperbolic tangent function**

$$h(x) = tanh(x) = \frac{exp(x) - exp(-x)}{exp(x) + exp(-x)} \tag{2.3}$$

The Hyperbolic tanget function (tanh)is similar to the sigmoid, but moves the output range to $(-1,1)$. This makes the output mean to be closer to zero, which helps the learning process of the next layer by saturating the gradient slower than the Sigmoid function.

- **Rectified Linear Unit**

$$h(x) = max(0, x) = \begin{cases} 0 & x \leq 0 \\ x & x > 0 \end{cases}$$

The Rectified Linear Unit, also known as ReLU, is a function with a derivative equal to 1 as long as the input is positive, and 0 when it's not. It is steadily becoming the standard activation function in most NN projects because it doesn't saturate when the input value is too large. It exists a different version (called "Leaky ReLU) with slightly higher performances, which has a very

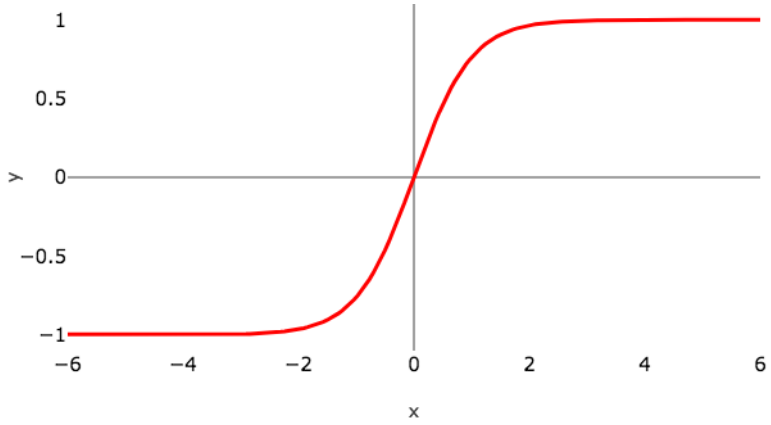

Figure 2.4: Hyperbolic tanget function [22]

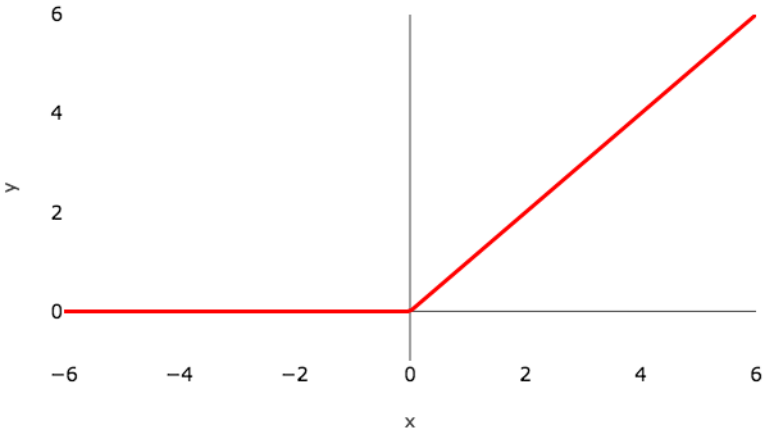

Figure 2.5: ReLU function [22]

small slope on the negative values instead of zero. This forces the learning process on all nodes, regardless of the fact their sum is constantly negative or zero (a rare phenomenon called the "dying ReLU").

Deep Learning refers to the training of the Neural Network's weight parameters. By stacking multiple layers of neurons like in figure 2.1, it's possible to achieve high levels of representation of complex transformation functions. These representations have to be automatically "learned" by a general learning procedure, as it is unfeasible for human engineers to do it by hand on non-basic problems.

The most common procedure used to train deep neural networks is *supervised learning* [23]. It consists in feeding already labelled data to the network to be elaborated. First, a forward pass is performer, in which the input data is fed to the network to produce an output. Then, an error or loss function is applied to measure

the difference between the output produced by the network and the expected one. An example of error function for regression problems is the Mean Squared Error (MSE), which uses the following equation:

$$L(w) = \sum_{i=1}^{n}(y_i - f_w(x_i))^2 \tag{2.4}$$

Where y_i is the correct value, and f_w is the predicted one. Weights are modified in order to minimize this cost, via the process of *gradient descent*.

Partial derivatives of the cost function in regards of the single weights are calculated backwards, following the *backward propagation* pattern, starting from the output layer. Then the weights are modified by a multiple of that derivative, called the *learning rate*(α) of the network.

$$w_{ij}^{t+1} = w_{ij}^{t} - \alpha(\frac{\delta L}{\delta w_{ij}}) \tag{2.5}$$

The process is then repeated with the new weight (w) values, until the cost is minimized. Each iteration of the training process typically uses a small subset of the available data, called a mini-batch.

The now trained model can now be used on unlabeled data. This is the process of *Inference*, where only the forward passage is done, since it's not possible to calculate errors not knowing the real output values. This means that the weight values remain unchanged.

In order to model complex functions, like those required by difficult regression and classification tasks, it is necessary to have a large enough number of layers. This is principle behind Deep Neural Networks (DNNs).

2.1.2 Convolutional Neural Network (CNN)

Convolutional Neural Networks (CNNs) are specialized networks usually used on computer vision tasks [24] such as object detection, object localization, face recognition and verification. The core of these networks is, as the name suggests, the convolution operation. Those are implemented in special hidden layers (the convolution layers) combined with the ReLU function. Here is an example of convolution over 1D data:

$$x_j^n = max(0, \sum_{k=1}^{K} x_k^{n-1} * w_{kj}^n) \tag{2.6}$$

In this operation, K values of the previous layer $n-1$ (the *input layer map*) are multiplied with a set of weights, and then summed together to produce a single output value. By spatially moving the window of K analyzed elements, more values

can be computed using the same set of weights, or *kernel*. Convolution can be seen as a form of filtering of the input map, extracting only the most valuable information from it. This information is saved in the next layers (the *output layer map*). As shown in picture 2.6, convolution is mathematically a sequence of matrix

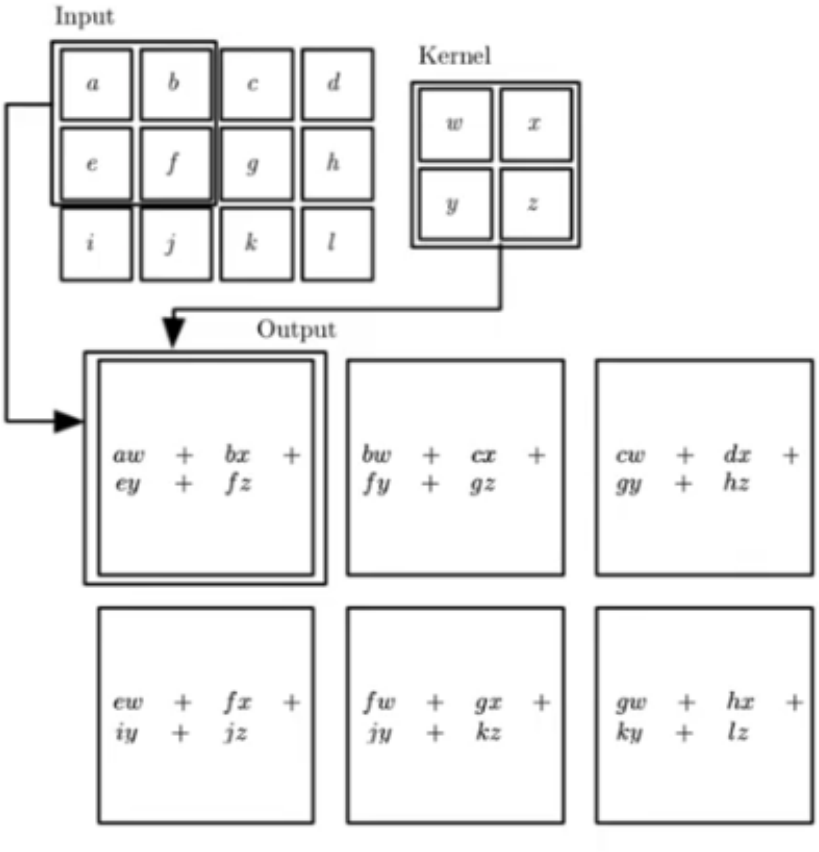

Figure 2.6: 2D Convolution example [25]

multiplications. The kernel weight values can be learned to be sensible to different input features, like vertical lines on an image.

One of the main advantages of convolution is *parameters sharing*. It is the ability of reusing the same weight parameters over different areas of the input feature map. This greatly reduces the memory required for storing the different weight parameters.

Convolution is not limited to single 2D images, but can be applied to any input map, regardless of dimensions. Each output feature map dimension can be calculated with the following formula:

$$d = \frac{n + p - f}{s} + 1 \tag{2.7}$$

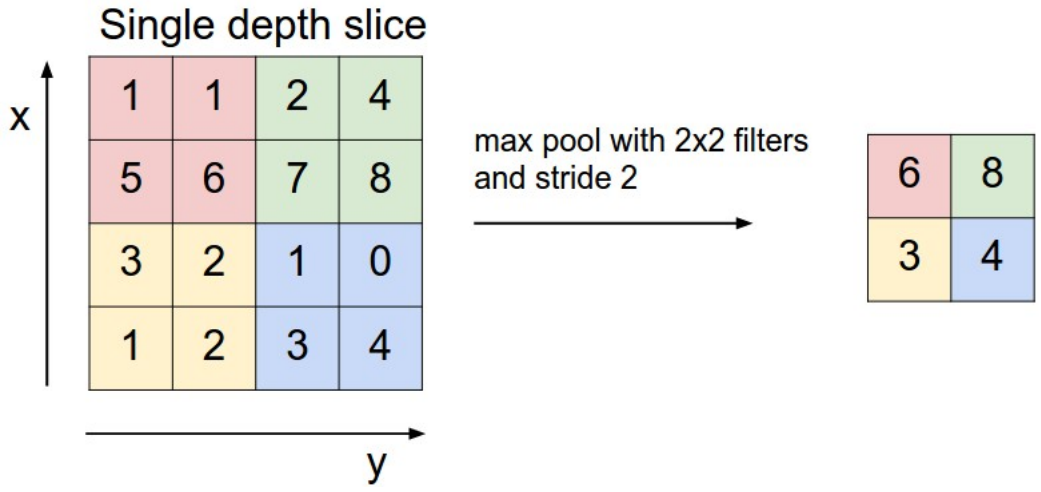

Figure 2.7: Max Pooling example [26]

Where n is the dimension value of the input map, p is the number of zero-padding elements introduced on that dimension, and s is the *stride* along it (the number of pixels the kernel shifts each iteration).

For example, if we take a 32x32x3 input (like an RBG image), without padding ($p = 0$), convolve it with a set of six 5x5x3 weight filters while using $stride = s = 1$ on every dimension, the output dimensions would be *dxdxc*:

$$d = \frac{32 + 0 - 5}{1} + 1 = 28 \tag{2.8}$$

$$c = \frac{3 + 0 - 3}{1} + 1 = 1 \tag{2.9}$$

However, since 6 different filter banks are used, the number of channels is also multiplied by 6, ending up with a 28x28x6 output feature map.

In order to have the same result on a classic fully connected layer, like the ones in figure 2.1, we'd have to train more than 14 million parameters, against the 456 of the cited example. The critically reduced usage of memory is one of the CNNs strong points.

In order to discard unimportant features, *Pooling layers* are also employed in CNNs. These layers do not have any weight parameter to be learned. In fact, they are just composed by a sliding window over the input map, calculating summary statistic of the framed values. The output map dimensions still follow the same formula at 2.7.

Some of the most common pooling techniques are *max pooling*(figure 2.7) and *average pooling*. As the names suggest, max pooling filters every value that isn't

the highest one in the selected window, while average pooling calculates a mean of the values under it. Pooling further reduces the memory required by the NN, while limiting accuracy losses.

In order to facilitate the training phase, *Batch Normalization* layers may be also included [27]. Their task is to fix means and variance of each layer input map, stabilizing the distribution of their values.

$$\hat{x}_i^{(k)} = \frac{x_i^{(k)} - \mu_B^{(k)}}{\sqrt{\sigma_B^{(k)^2} + \epsilon}} \tag{2.10}$$

Because it's impractical to normalize the entire training set in one go, it is divided in multiple batches (B) of m elements. After calculating mean (μ_B) and variance (σ_B^2), each dimension is normalized separately (in the example, the k dimension is being normalized). ϵ is a small constant for numerical stability (avoids divisions by zero). After normalizing, a transformation step is taken:

$$y_i^{(k)} = \gamma^{(k)} \hat{x}_i^{(k)} + \beta^{(k)} \tag{2.11}$$

Where γ and β are parameters to be learned.

2.1.3 Temporal Convolutional Network (TCN)

Sequence modelling has been canonically related to *Recurrent Neural Networks* (RNNs) in literature. RNNs are special NNs that conserve a "memory" vector in the hidden layers, with the objective is that at a certain timestamp t the computation of an input value x^t depends on the previous values.

Recent researches have put these notions under question [10]. Certain convolutional architectures can reach state-of-art performance in sequence modelling tasks like audio synthesys, word-level language modeling, and machine translation.

The *Temporal Convolutional Network* is a recently researched, generic and simple architecture, combining best practises out of different successful convolutional networks. This architecture has been confronted with canonical recurrent architecture (*LSTM* and *GRU*). Experimental results show that, with minimal tuning, TCNs can outperform recurrent solutions on various sequence modelling tasks used to benchmark RNNs themselves.

The key features of TCNs are:

1. Causality of convolution operations, which prevents any form of leakage of information to the past layers or iterations.

2. TCN can take a sequence of any length. The output will also be a sequence, of the same length of the input (like RNNs do).

To achieve the second point, TCNs are built upon 1-D *Fully-Convolutional Networks* (FNC). In order to have an effectively long history, a high number of layers is required, therefore TCNs are, by design, very deep network.

Because simple causal convolutions can only look back to a number of history data linearly proportional to the layer's depth. They cannot be properly applied to sequence tasks. Dilated convolution F is defined as:

$$F(s) = \sum_{i=0}^{k-1} f(i) x_{s-di} \tag{2.12}$$

Where d is the dilation rate, f is the filter function, d the filter size. $s - di$ is the direction of the past, meaning that d is basically a fixed step between two adjacent filter taps.

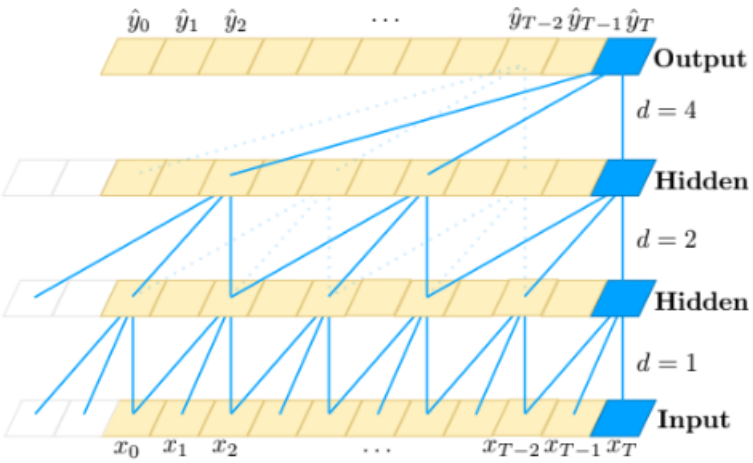

Figure 2.8: Dilation improving the reception field of the output sequence [10]

The effective history of a layer is therefore $d(k-1)$. d is tipically increased exponentially with network's depth, so that each input is effectively hit by the filter, as shown in figure 2.8.

Perceived history depends on depth, kernel size and dilation rate, all factors that can be increased by creating a very large and deep network. Big architectures like this benefit from using *residual blocks*.

A residual block (figure 2.9) is composed by two parallel branches. The left one leads to a series of transformations F, the right one is an identity. The branches results are summed at the end of the block, before the activation function. The output can be expressed with:

$$o = Activation(x + F(x)) \tag{2.13}$$

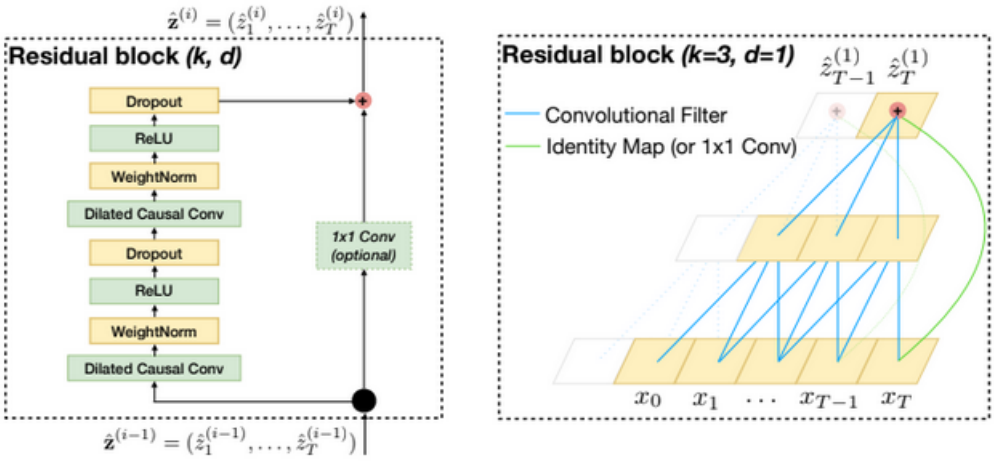

Figure 2.9: Residual block structure and example [10]

Residual blocks help the training phase in very deep neural networks. Spatial *Dropout* [28] is also introduced after each dilated convolution layer for regularization, which is a technique that randomly zeroes some features in the input sequence, during training stage, to avoid model overfitting the training set.

An optional 1x1 convolution block may have to be used on the right branch. This block is composed by n 1x1 filters with weight equal to 1. What the block effectively does, is to change the number of channels on the input, in order to match with the left branch's output dimensions.

2.2 PULP & GAP8

2.2.1 PULP

Recent technological developments in embedded computing devices have shifted the computer vision field of study to embedded applications, such as smart cameras [29] or self-driving cars [30]. present many problem such as the scarcity of memory and the low computation power. On the other hand, their power consumtion is really low, usually in the order of few mWs. Nevertheless, the power consumption becomes a bottleneck when integrating other kind of devices that cannot host a battery and must rely on energy harvester, such as micro/nano-UAVs. These devices employ MCUs to limit energy usage, but they have great difficulties when performing even the most basic convolutional algorithms, due to their very scarce memory and energy access. The PULP (*Parallel processing Ultra-Low Power platform*) project [16] was born to respond to this unmet demand for high-computational power at the edge with a low power budget. Many chips have been produced since its

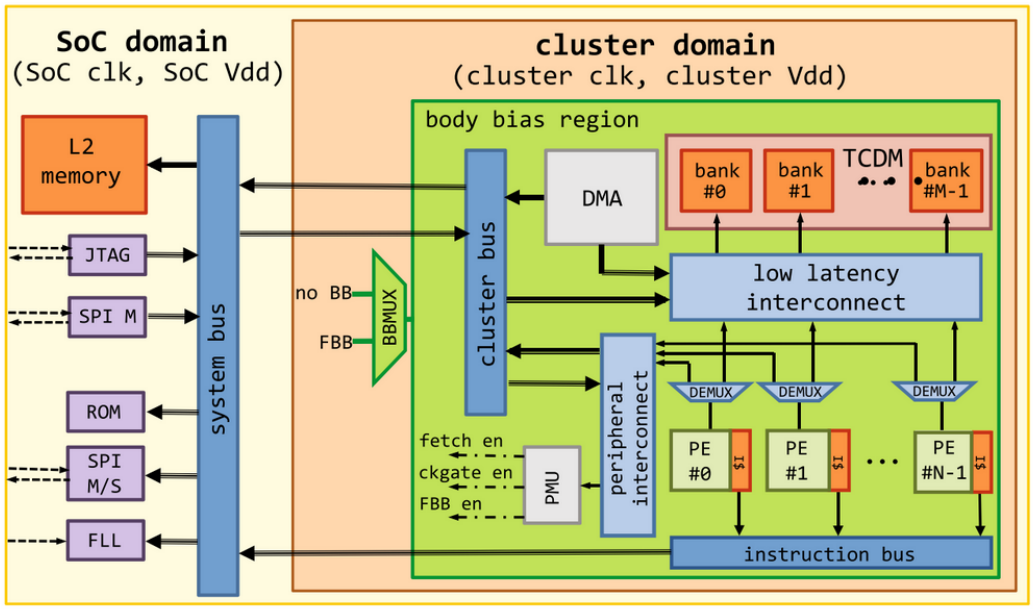

Figure 2.10: PULP architecture [16]

creation [14]. The features that all the chips share are as follow:

- Low-power platform, capable of switching to a sleep state with very low power consumption when not performing any task.
- High-performance on demand, to manage high frame-rate requirements with the lowest energy budget.
- High flexibility and programmability, to keep on track with the rapid development in the computer vision field of study.

To achieve high computation performances, PULP uses a cluster of *OpenRISC* cores, typically 4 to 8. They are called the *Processing Elements (PEs)*, and allow various degrees of data or task-level parallelism. The cores have been optimized to reach high instruction-per-cycle values over a wide range of applications, including control-intensive code. PEs share the same L1 *multi-banked tightly coupled data memory (TCDM)*. The TCDM has a number of ports equal to the number of PEs, allowing concurrent access to different memory locations. Each PE also has a private instruction cache, but no data cache. TCDM size is variable, but usually smaller than the L2 memory.

A multi-channel DMA allows for fast memory transfers between cores, L2 memory (32 to 128 KB range) and peripherals. It's connected to the TCDM with a low latency interconnect, the same used by the PEs. This eliminates any form

of internal buffering when managing L1 data transfers. The cluster domain is connected to all external resources and peripherals via a peripheral interconnect.

In order to provide energy efficiency, each core can operate on private voltage and frequency. To do so, a *Frequency-Locked Loop (FLL)* is implemented on SoC level. A set of clock dividers (one for the Soc, and one for each cluster core) can divide the FLL-generated clock frequency. For the voltage, a *Body Bias Multiplexer (BBMUX)* is used. It works synergically with the *Power Management Unit (PMU)* to quickly switch each part of the architecture between normal and "boost" mode, whenever the computed task needs it. The PMU guarantees that the different operating modes stay transparent to the software, by generating control signals for fetch enables, clock gating units, and the BBMUX.

Other peripherals integrated are a set of two *Serial Peripheral Interfaces (SPIs)*, one for master and one for slave, a bootup ROM, and a JTAG interface used for testing purposes. The SPIs can be set in single or quad mode depending on bandwith, can be linked to various off-chips components (like sensors), up to 4 slave peripherals. The peripheral architecture allows for the system to be in two different operating modes:

- Slave mode: PULP acts as a multi-core accelerator of a standard host processor. The host has to load the application on PULP L2 by using the SPI master interface, and synchronize the computation with dedicated signals.

- Standalone mode: PULP detects external flash memory on the SPI master interface. If none are linked L2 is used instead.

2.2.2 GAP8

GAP8 is an IoT application processor developed by Greenwave Technologies [14]. It is built upon the open-source PULP platform, implementing an extended version of the RISC-V instruction set. GAP8 enables cost-effective solutions for intelligent, low-memory, low-energy devices. GAP8 focuses on a wide range of algorithms, including convolutional neural network inference. This architecture can be used to integrate artificial intelligence applications on IoT devices, such as image recognition, machine health monitoring, automatic home security, smart toys etc.

As seen is figure 2.11, GAP8 architecture strongly resembles the original PULP one, but with multiple additions:

- A *Fabric Controller (FC)* subsystem is used for control, communication and security. It can be seen as a simple MCU. This core is a Risc-V core, identical to the ones which are comprised in the cluster. The same applications can be either run on the FC or on the cluster.

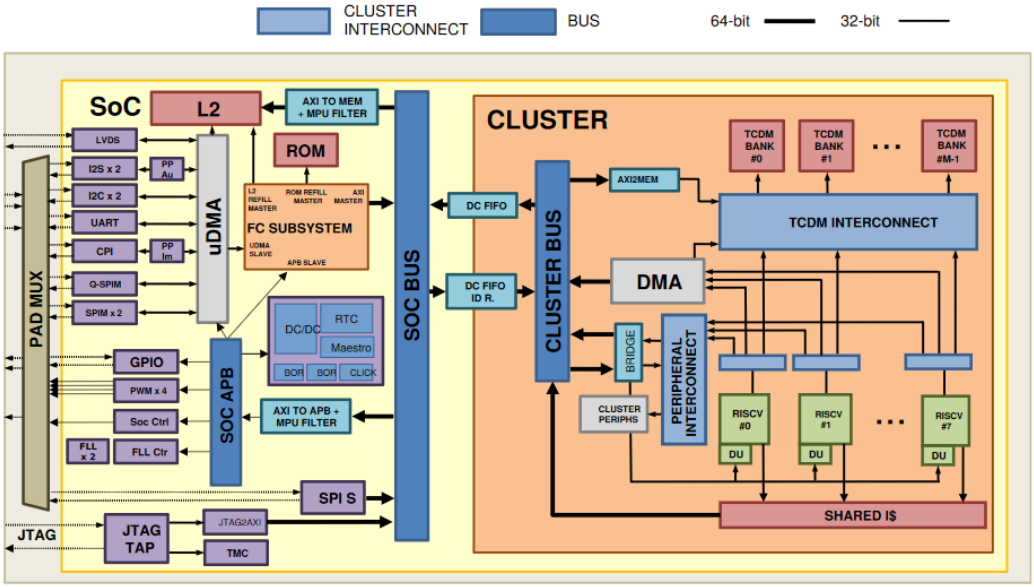

Figure 2.11: GAP8 architecture [14]

- 8 RISC-V cores are used inside the cluster, coupled with a specialized *Convolutional Neutral Network accelerator (HWCE)*. The cluster is activated from the FC and used as an hardware accelerator.
- DC/DC (*Direct Current to Direct Current*) regulators and clock generators with ultra-fast reconfiguration times, to manage power and voltage for each core and peripheral.
- A larger L2 memory (512 KB), and two TCDMs, one exclusive to the FC (16 KB), and one shared by the cluster cores and HWCE (64 KB).
- Multiple DMA units to allow fast transfers between cluster L1 and L2 memories, and between L2 memory and peripherals. Two different DMA are present, one for cluster - L2 communication and one for L2 - external peripherals communication.
- A *Memory Protection Unit (MPU)*, controlled by the FC. Works with different address filters, directly implemented on the bus system, to secure specific memory map areas. Security is based on the user/machine privilege mode configured on FC and rules set in the MPU.
- A rich set of peripheral interfaces, like CPI for cameras, LVDS for radio frequency, I2S for digital audio devices etc.

All 9 (FC+cluster) cores share the same RISC-V extended instruction set architecture. It has been extended to support the most common signal processing and machine learning algorithms. Extensions include: zero overhead hardware loops, pre/post pointer increment memory access, computation-control flow mixed instructions (max, min etc.), multiply and accumulate, vector operations, fixed-point operations, bit manipulation and dot product.

There is only a single shared instruction cache, instead of multiple private ones like in the PULP architecture. This is because the cluster cores execute the same area of code, while parallelization is done on data level. By using a single instruction cache, memory access for instructions is generally reduced.

Because there is no data cache, autonomous DMAs and good memory management is required for energy efficiency. Tools like the *GAP8 auto-tiler* can be used to facilitate the process [14].

Up to two tasks can be queued for each peripheral. The μDMA schedules active transfers by checking the peripherals' signals in a round-robin fashion. The μDMA is usually only directly used by the peripherals' drivers, not by the programmer. The cluster-DMA focuses on transferring data between L1 and L2 memory areas. It can queue up to 16 requests and has very short instructions set to minimize overhead and instruction cache pollution.

As stated before, FC is used as a micro-controller. The cluster can be seen as a "peripheral" of the FC, specialized in doing high-precision, parallelized calculations for image, audio and signal processing.

Each core is uniquely classified by two identification numbers. The first one is the *Cluster ID*, set to 0x00 if the core is part of the cluster, 0x20 otherwise. The second is the *Core ID*, always set to 0x00 for the FC, and from 0x00 to 0x07 for the cluster cores.

By default, the cluster is turned off, and must be powered up first by the FC. Once awake, the core 0 of the cluster (the one with both IDs at 0x00) is regarded as the "master" core of the cluster. The master core is the only one directly communicating with the FC, and has to manage the parallelization on the remaining cores, by dispatching the tasks and applications suitably. The cluster has to be turned off once the tasks are done to save energy, transferring memory from L1 to L2 first.

2.2.3 GVSOC

GVSOC is a lightweight virtual platform, simulating the GAP8 IoT application processor [31]. This allows to test out programs without any hardware limits. It is also possible to simulate full applications with real device drivers, by using device models. Currently, it is possible to simulate devices such as cameras, microphones and LCDs. Installation of GVSOC is integrated with the GAP8-SDK, and can be

easily called by setting the *–platform* options equal to "gvsoc" when compiling an application.

Multiple options can be set by modifying the system configuration. This can be done on command line by using the *option –property=<path>=<value>* option. This will overwrite the desired value on the JSON file. The path is hierarchical, similarly to a file system path.

In order to facilitate the debugging process for developers, the platform allows dumping architecture events. For example, it can show instructions being executed on assembly level, all DMA transfers, signals generated, memory access and so on. This feature can be enabled by using the *–trace=<path>* option on command line. The path argument is the architecture component to be dumped. More than one component can be analyzed by calling the same option multiple times.

For example, in order to trace the operations done by cluster cores 0 and 1, the following command line can be used:

```
1    pulp-run --platform=gvsoc --config=gap_rev1 --binary=test prepare
     run --trace=pe0/insn --trace=pe1/insn"
```

The trace file has usually one line for every single event. Each line is divided in three parts.

On the left part there's the timestamp in picoseconds, followed by the number of cycles. While the timestamp is linear, cycle number depends on the domain's frequency.

On the middle part there's the location where the event has been registered. Finally, in the right part, the core information about the event, including its name, the memory locations and registers involved in it, and the values they had at the time of the event.

By default, text is dumped on standard output. However it is possible to save on a specified file by using the > or : operator.

GVSOC does not provide any particular feature for profiling, except for hardware performance counters (see GAP8-SDK section). However, since the platform is virtual, it is possible to implement all performance counters at the same time, instead of just one as limited from the GAP8 hardware.

2.2.4 GAP8-SDK

GAP8's software developer kit (GAP-SDK) is an extract of elements from the *PULP-SDK* necessary for compiling and running applications over the GAP8 IoT processor [32].

The SDK contains:

- GAP8 RISCV GNU Toolchain: pre-compiled toolchain from the RISC-V project
- PLPBRIDGE: a PULP project tool allowing comunication between PC and PULP device such as GAP8. It allows to control the platform, debug it using GDB, and load programs on the GAPuino (a board developed by GreenWave Technologies for developing prototype applications [33]) flash memory.
- Four different open source operating systems/sets of API, ported by Greenwave Technologies on GAP8:
  - PULP OS
  - Arm Mbed OS
  - FreeRTOS
  - PMSIS

A series of examples are provided during installation to learn the various functions available. In order to run them once installation is over, we first set up the shell environment on the terminal by running:

```
source ~/gap_sdk/sourceme.sh
```

Then we can test the example programs, using the GVSOC platform.

Here is a quick analysis of the various examples, focusing on the most important methods necessary to correctly program an application using the GAP8-SDK. In this work we'll use the *PULP-OS* operating system.

Hello world example

The hello world example is a very simple program showing how to turn on the cluster, dispatch tasks on it, and then turn it off when it's finished.

rt/rt_api.h is the main library to be included, that contains a large number of sub-libraries, such as *rt_event.h*, *rt_cluster.h*, *rt_utils.h* etc.

The *main()* function is entirely executed by the FC, acting as a controller for the cluster. In order to dispatch the tasks and manage concurrency, asynchronous events are used.

```
if (rt_event_alloc(NULL, 4)) return -1;
```

This function allocates 4 events and puts them in the free list. Later on, the events get reserved and pushed into the scheduler appointed. If NULL is passed as

the scheduler (like in the example), the default one is used. If the allocation fails, the program stops immediately.

```
rt_event_t *p_event = rt_event_get(NULL, end_of_call, (void *) CID);
```

The next function pulls a free event from the scheduler list, and initializes it with a callback function, *end_of_call* in the example, passing the core id (CID) as an argument to it.

In order to use the cluster, we must first turn it on, because it is powered down by default to save energy.

```
rt_cluster_mount(MOUNT, CID, 0, NULL);
```

This function is also used to turn the cluster off. In the example, the MOUNT flag is equal to 1, while the UNMOUNT flag is 0. CID is the ID of the target cluster (0 in this case). It's possible to pass an event too in order to notify when the operation is done, or just NULL if we want the function to simply return, like in the example.

Once mounted, the cluster can be woken up in the following step:

```
rt_cluster_call(NULL, CID, cluster_entry, NULL, NULL, 0, 0, rt_nb_pe
    (), p_event);
```

This is a coarse-grain job delegation to the cluster side. Specifically, it will wake up the cluster core 0 and will make it do the specified *entry* function, using the specified stacks, if appointed.

The first argument should only be set if multiple calls are enqueued at the same time.

Two stack pointers (one for the master core, the other for the slave ones) are set to NULL, allowing for their automatic allocation and deallocation. Those stacks' sizes are set to 0, which gets automatically interpreted as the default size.

rt_nb_pe() is an utility method that returns the number of available cores. By using it, we can wake up all of them.

Lastly, we pass the event we reserved earlier, that will be used to notify when this function has been successfully executed.

```
while(!done)
    rt_event_execute(NULL, 1);
```

The FC locks itself up into an endless cycle, waiting for the *done* flag to be set to 1. Each cycle, it executes any events present in the scheduler (the default one if NULL is selected). If none is available, it will wait one because the second argument (the wait flag) is set to 1.

Meanwhile, the *cluster_entry* task has been now dispatched to the cluster core with CID equal to 0, also known as the "master" core. Once it returns, the *p_event* will be automatically scheduled. The FC will execute it, triggering the *end_of_call* function, which breaks the cycle by setting the *done* flag to 1.

```
static void cluster_entry(void *arg)
{
  printf("Entering cluster on core %d\n", rt_core_id());
  printf("There are %d cores available here.\n", rt_nb_pe());
  rt_team_fork(8, hello, (void *)0x0);
  printf("Leaving cluster on core %d\n", rt_core_id());
}
```

Cluster entry first prints out the core ID of the "master" core, prints the number of cores available in the cluster, then calls the *rt_team_fork* method. This is an important function that creates a team of slave cores (including himself) to execute a task, passing the third argument to it. The *hello* task, in this case, is just a single instruction that prints the id of the core calling it.

The expected output will look like this:

```
Entering main controller
Entering cluster on core 0
There are 8 cores available here.
[clusterID: 0x 0] Hello from core 5
[clusterID: 0x 0] Hello from core 0
[clusterID: 0x 0] Hello from core 1
[clusterID: 0x 0] Hello from core 4
[clusterID: 0x 0] Hello from core 2
[clusterID: 0x 0] Hello from core 3
[clusterID: 0x 0] Hello from core 6
[clusterID: 0x 0] Hello from core 7
Leaving cluster on core 0
[clusterID: 0x20] Hello from core 0
Test success: Leaving main controller
```

The described example is a solid base to be expanded for more complex programs. In fact, all the examples follow the same schema:

1. Initialize synchronization event with a callback function
2. Mount the cluster

3. Dispatch master task on the cluster core with CID = 0
4. Dispatch slave tasks on the team of cluster cores
5. Execute callback function
6. Unmount the cluster

Also more complex programs share the same programming paradigm.
In the next code analysis, we'll solely focus on useful directives for programming on GAP8-SDK that haven't been shown already.

Cluster_alloc example

This example shows API functions useful for manual allocation of memory on the FC and the cluster cores. We are going to use these for dynamically allocate the supporting buffers in the convolution kernels.

```
1 void *stacks = rt_alloc(RT_ALLOC_CL_DATA, STACK_SIZE*rt_nb_pe());
```

After mounting the cluster, it's possible to manually allocate stack to be used future dispatched tasks. *RT_ALLOC_CL_DATA* is a flag specifing that the data must be stored in the cluster's L1 memory.

```
1  rt_alloc_req_t req0;
2  rt_free_req_t  req1;
3  char * a;
4
5  rt_alloc_cluster(RT_ALLOC_L2_CL_DATA, alloc_size, &req0);
6  a = (char *) rt_alloc_cluster_wait(&req0);
7
8  if((a)==NULL){ printf("L2 Allocation Error...\n"); return; }
9
10 free_mem(a,alloc_size,&req1);
```

Cluster cores can dynamically allocate data on the L2 memory. In order to do so they have to use the *rt_alloc_req_t* and *rt_free_req_t* structures. This operation is not fast, and requires waiting for it.

Cluster_notif

This other example illustrates some synchronization mechanisms provided by the GAP-SDK, useful for synchronizing the parallel tasks on the different cores.

```
static rt_notif_t notif;
```

This structure declared at global level manages the notification mechanism at runtime.

```
rt_cluster_notif_init(&notif, 0);
```

The FC has to initialize the *notif* structure, declaring the cluster ID it has to work on.

```
rt_cluster_notif_trigger(&notif, 0x03);
rt_cluster_notif_trigger(&notif, RT_TRIGGER_ALL_CORE);
```

Once the cluster cores are working, the FC can notify one specific core by passing the core ID as argument, or all of them at the same time, using the *RT_TRIGGER_ALL_CORE* flag.

```
int event = rt_cluster_notif_event(&notif);
rt_cluster_notif_wait(event);
```

The cluster cores, in the example, extract the blocking event from the notif structure and call a wait over it. They go into a low-powered state until the FC triggers the event with the previous instructions.

```
rt_team_barrier();
```

This is a very important synchronization directive. This blocks the execution of each calling block, until every single core of the slave team reaches the barrier. A team is created every time *rt_team_fork* is called, the barrier always refers to the last team created. By using barriers, it is possible to manage task synchronization over multiple cluster cores.

Performance example

Performance profiling is a vital part of software developing. GAP-SDK provides a wide range of performance counters to profile on both the GVSOC simulator and GAP8 hardware.

```
static rt_perf_t *cluster_perf;
```

The hardware performance counters can be invoked by using this structure, declared at global level.

When performing performance tests, it is recommended to wrap the entire program in a loop. Skip the first few cycles (the "warm up" cycles), and calculate the mean of the selected performance counter over the remaining cycles. This is done to guarantee profiling accuracy.

The *cluster_perf* is allocated in L2 memory by the FC.

```
rt_perf_init(perf);
rt_perf_conf(perf, (1<<RT_PERF_CYCLES) | (1<<RT_PERF_INSTR));
rt_perf_reset(perf);

rt_perf_start(perf);

/*Code section to be profiled*/

rt_perf_stop(perf);
rt_perf_save(perf);

instr = rt_perf_get(perf, RT_PERF_INSTR);
cycles = rt_perf_get(perf, RT_PERF_CYCLES);

printf("\tPerformance of funct2monitor: cycles %d instructions %d\n",
    cycles, instr);
```

After initializing the performance structure, we configure the hardware counters we want to use for profiling. There are many counters available, useful for in-depth code analysis. They can count the number of cycles, instructions, load data hazards, jumps, instruction cache misses, memory loads/stores, branches, etc.

Usually, the most meaningful counters are the number of cycles and instructions, analyzed in the example. We'll be also focusing mostly on those two counters when profiling PULP-TCN.

SoC_Cluster_Frequency example

This example highlights how to change the frequency settings on each single system core.

```
rt_freq_get(__RT_FREQ_DOMAIN_FC);
rt_freq_get(__RT_FREQ_DOMAIN_CL);
rt_freq_set(__RT_FREQ_DOMAIN_CL,175000000);
```

Frequency in the FC and cluster cores can be accessed and modified separately at any moment by the programmer.

Cluster_mergesort example

This is an handcrafted example, showing how to implement a non-elementary algorithm, such as the parallel mergesort, on GAP8 using its SDK.

Full code can be read on appendix A.

The main function is very similar to the example ones, following the same directives for starting up the cluster core and dispatching instructions over it. Because the dispatched task can only accept a single argument, it is necessary to wrap the *int* array and its size in a single *vector* structure. This structure is the argument passed to the cluster master core in *rt_cluster_call*.

Parallelization on the 8 cores is done over data. In this example, the master cluster core splits the input vector in 8 equivalent chunks, one for each core, to be passed as argument in the *rt_team_fork* call.

First, each core has to apply the *merge_sort* algorithm on its appointed chunk. Then, every core has to merge his chunk with its neighbour's, a number of times that depends on its own ID: core 0 has to merge three times, core 1 has to do it two times, cores 2 and 3 have to do it once, while the rest of the cores don't have to do it. Synchronization is guaranteed by a series of *rt_team_barrier*s.

The mergesort algorithm allocates a support buffer on L2 memory. *Merge* takes a vector that has the left and right side (with respect to the element at the m position) already sorted, and generates a new, fully sorted vector. *Merge_sort* takes an unsorted vector and splits it multiple times, until it reaches single-element arrays, which are sorted by construction, then recursively merges them back in order.

Chapter 3

Related works

3.1 CMSIS-NN

CMSIS-NN is a set of efficient kernels [17] developed for maximizing NN applications performances on Arm Cortex-M processors for IoT edge devices, while minimizing their memory footprint.

The kernel code is divided in two parts: *NNFunctions* and *NNSupportFunctions*. The first group contains functions used to implement popular NN layer types, such as convolution, pooling, activation etc. Kernel APIs are simple, to ease implementation in any machine. There are multiple versions for the most common layers, like the fully connected, to optimize execution in specific environments.

The second group has utility functions such as data conversions and activation tables. They are used by NNFunctions, and can help when building more complex NN architectures.

While training a NN higher precision data types are preferred, but during inference it is possible to use lower-precision data type to save up memory, suffering negligible drops in accuracy [6].

CMSIS-NN adopts, as the name suggests, the same data types of the CMSIS hardware abstraction: q7_t as int8, q15_t as int16 and q31_t as int32. Quantization is done with a fixed-point format and power-of-two scaling. The values are represented in the $A \times 2^n$ format, where A is the integer value and n the location of the radix point. The scaling factors are passed as arguments in the kernels. The scaling operation is done with bitwise shifts, thanks to the power-of-two format. This design decision is taken to avoid inter-layer floating point de-quantizations, because some devices don't have a *Floating Point Unit (FPU)* to efficiently process such operations.

Most NNFunctions use 16-bit *Multiply-And-Accumulate (MAC)* instructions, requiring to convert 8-bit data to 16. CMSIS has an utility function just for that,

arm_q7_to_q15. In the first step, it extends the data by using the sign extension instruction. In the second step, data is reordered to follow the original pattern. In CMSIS-NN, the second step is omitted, assuming the operands always follow the same ordering.

The Matrix Multiplication (MatMul) operation is the most important in NNs. Similar to the original CMSIS implementation, CMSIS-NN's MatMul kernel is implemented using 2×2 kernels for preventing stalls.

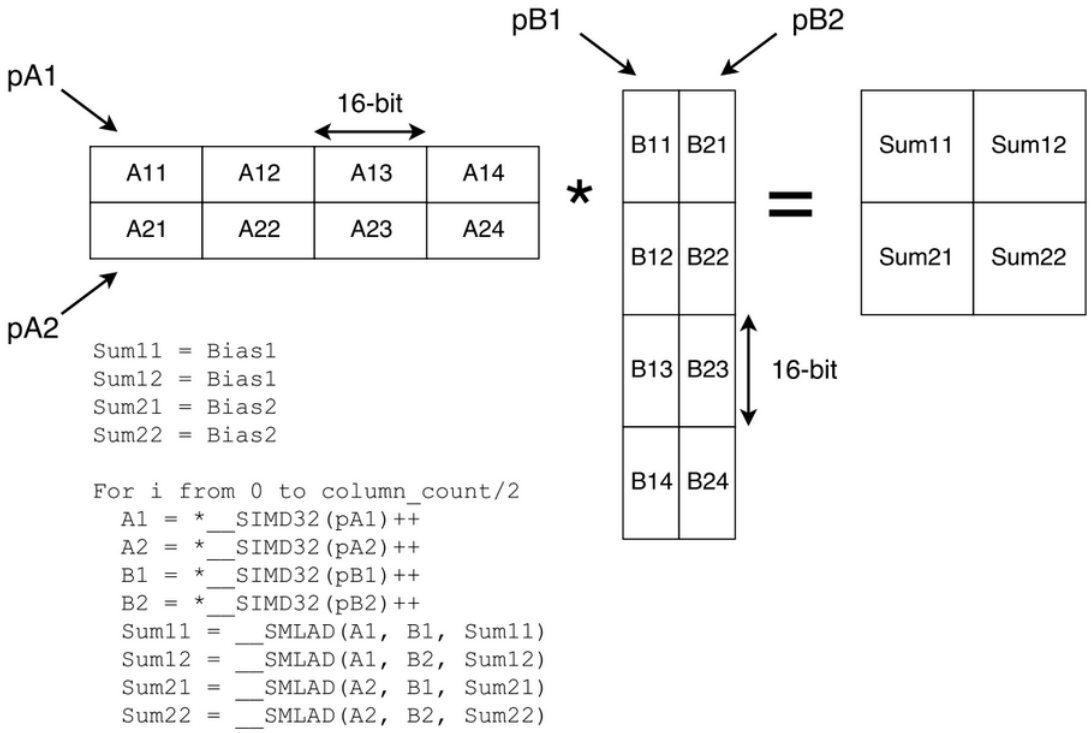

Figure 3.1: 2×2 MatMul kernel inner loop [17]

Both operands (input map and weights) must be 16-bits precision, while the accumulators are 32-bits. Each loops computes the dot product between 2 columns and 2 rows, resulting in a total of 4 outputs.

In fully-connected layers with a batch size of one the Matrix-Matrix operation simplifies to a Matrix-Vector one. A 1×2 kernel could be used to optimize the operation. But by reordering the weight values it is possible to implement a 1×4 kernel, which obtains superior performance. In this reordering process, weights are interleaved every 4 rows and shuffled every 4 entries. Additional rows do not get reshuffled. Refer to figure 3.2 for an example.

For the Convolution kernel, input decomposition and reordering is necessary. The *Image to Column (im2col)* process transforms a image-like input into columns

Figure 3.2: Weights reshuffling to be compatible with the 1x4 kernel [17]

containing the data required for each convolution filter. This allows to implement convolution as a standard matrix-matrix multiplication, taking advantage of the optimized kernels for this operation. However, this causes a memory overhead, given that multiple pixels are repeated in different columns.

CMSIS-NN implements *partial im2col*, only expanding 2 columns at a time, sufficient for the MatMul kernel structure mentioned above (2×2).

When 3D data is involved, the two most common data layouts found in software libraries are *Channel-Width-Height (CHW)*, where channels are the outermost dimension, and *Height-Width-Channel (HWC)*, where channels are innermost. In HWC, data along the channel dimension is stored with a stride of 1, data along width is stored with stride equal to the number of channels, and data along the height is stored with $stride = n.channels * imagewidth$.

HWC is more efficient due to the fact that data regarding the same pixel is stored contiguously, allowing efficient memory transfers for SIMD instructions.

CMSIS-NN also supports depth-wise separable convolutions, which are often found in compact network architectures.

Pooling kernels in CMSIS-NN are implemented following the *split x-y* approach, instead of the classic window-based pooling. This involves splitting the pooling operation first by width, then by height. Pooling is done *in situ*, making it a destructive task on the input.

For the activation functions, Sigmoid and Tanh are both implemented with

look-up tables, using the MSB and LSB to find the corresponding value. ReLu, instead, uses the MSBit of the 8-bit number as sign bit and extends it as a mask using a byte-level subtraction instruction.

With CMSIS-NN, Neural Network inference achieves 4.6× improvement in runtime/throughput, and 4.9× in energy efficiency for a CNN targeting the $CIFAR-10$ [34] image dataset, compared to previous existing solutions [17].

3.2 PULP-NN

PULP-NN is an open-source library optimized for PULP-like architectures [15]. The key innovation is a set of kernels for Quantized Neural Network (QNN) [6] inference parallelized on the 8 cluster-cores. The data types being targeted are 8-bit long or lower, down to single bit precision, following the trend of aggressive quantization in DNNs.

The main focus of this library is multi-core execution on fully programmable edge devices, with low bit-width operations. PULP-NN is based on CMSIS-NN [17]. It optimizes its convolution kernel by improving data reuse.

In PULP-NN, quantization of real-valued weight parameter w to a Q-bit signed fixed-point values is implemented with the following formula:

$$q(w) = clip_{[-1,1)}(2^{-(Q-1)} \cdot round(w \cdot 2^{(Q-1)})) \tag{3.1}$$

Where *clip* is:

$$clip_{[a,b)}(x) = max(a, min(x, b)) \tag{3.2}$$

Then, the Q-bit representation of w is defined as:

$$W = q(w) \cdot 2^{(Q-1)} \tag{3.3}$$

This quantization method also works for activation values. As long as both weights and activation values are represented by the same Q-bit data type, the convolution is mathematically a sum of products in the integer domain:

$$\phi(w, x) = 2^{-2(Q-1)} \sum_{i \epsilon C} W_i X_i \doteq 2^{-2(Q-1)} \Phi(W, X) \tag{3.4}$$

Where C is the number of input channels, ϕ is the convolution operation and Φ is the high-precision accumulator, 32 bits if $Q = 8$, 16 bits if Q is lower precision. The accumulator is compressed back to Q bits before producing the output activation value for the next layer, using scaling and clamp operations in case Q is 8 bits, threshold if Q is 4 or 2 bits. For a Q-bit output, it is necessary to store $2^Q - 1$ threshold values per channel to compare, on each single layer.

In case $Q = 1$, the quantization operation simplifies to:

$$\Phi_{bin}(X) = popcount(WxnorX) \tag{3.5}$$

Where *popcount* is the bit counting instruction, returning the number of bits set to 1 in the *xnor* output.

Like in CMSIS-NN, PULP-NN implements the convolution in two steps. The first step being the *im2col* buffer construction, the second step being a dot product between the loaded buffer and the filter weight banks.

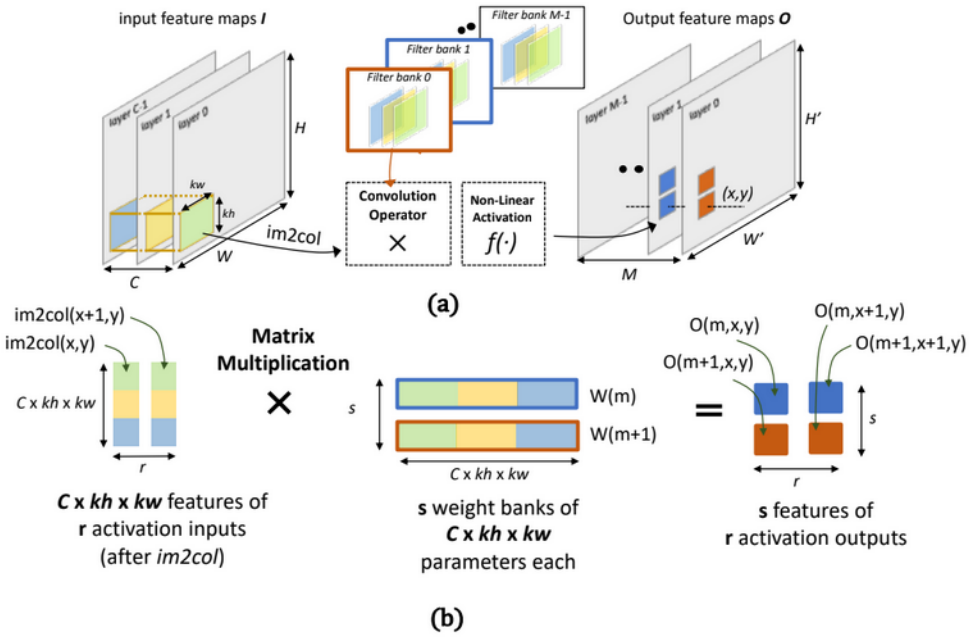

Figure 3.3: (a): Dataflow of spatial convolution kernel (b):Convolution inner loop as matrix multiplication [15]

The inner loop of the convolution dot product is realized with a matrix multiplication kernel. As stated previously, CMSIS-NN works on two spatially adjacent output pixels of two consecutive channels each iteration. We called this the 2×2 *MatMul kernel*.

PULP-NN Implementation

PULP-NN accelerates the cycle computation by using *hardware loops*. Data access on the *im2col* buffer is extremely regular by construction. This allows usage of load and store instructions with the post-increment pattern, that further improve the *MatMul* operation.

PULP-NN also uses SIMD vectorial instructions to allow processing multiple sub-word data in parallel. Most of them require just one clock cycle. These instructions use specific vectorial data types: *v4s* is a 32 bit data operand that can store up to four INT-8 data, whereas *v2s* is a 32 bit operand that can store two INT-16 data. Some vectorial instructions include *sdotp4*, which computes the sum of the dot products on two v4s data, *max4*, that computes an element-wise max operation on two v4s in a single cycle, etc.

When working over INT-8 data, PULP-NN fills 2 *im2col* buffers to compute 2 adjacent output pixels. Then, it loads 4 consecutive elements from the buffers and filter weight banks, by casting the INT-8 pointers to *v4s* ones. With a total of 4 load instructions, it is possible to execute 4 *sdot4* operations, which are equivalent to 16 MAC operations.

When implementing a fully connected layer, data reusage is maximized by using the 2×1 *MatMul Kernel*, like proposed in CMSIS-NN. Using *sdot4* instructions, it's possible to execute 8 MACs after 3 load operations.

Ancillary operations also take advantage of the DSP extensions. The hardware loops and post-increment load/store help in the ReLu kernel, that also uses the *max4* SIMD instruction. Max Pooling also uses these features, while splitting the workload on the height and width dimensions (like in CMSIS-NN).

INT-8 is the smallest natively-supported data type. INT-2 and INT-4 have to be scaled up to INT-8 with additional support functions. To reduce overhead, the SIMD's *bextract* and *pack4* are used. The first one extract a selected number of bits from a register and sign-extends them in a single cycle, while the second stores 4 INT-8 values into a *v4s* in two clock cycles. The result will be compressed back to the original size by comparing it with a series of threshold values.

No casting or unpacking operations are needed for INT-1 data. Different support operations are already provided by the ISA for binary operations. The result will still be on a 16 bit accumulator, which gets compressed by comparing it with a single threshold value.

The *HWC* data layout is used, like in CMSIS-NN. For parallelization, it is convenient to split the workload between the various cluster cores over the height dimension of the output feature map. In order to do so, the kernel divides the output height by the number of available cores, approximating by excess. That is the "chunk" dimension, which is the number of rows on the output feature map each core has to compute. The start and stop row index of the chunk are calculated using the core's ID:

$$Start\_Row = min(Chunk\_Dimension \cdot Core\_ID, Output\_Map\_Height) \tag{3.6}$$

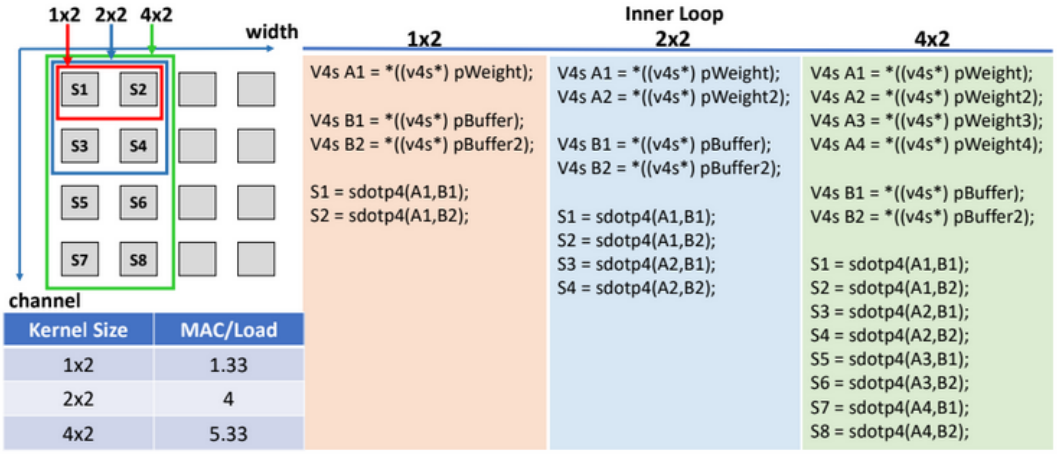

Figure 3.4: Inner loop of the matmul kernel, considering different sizes [15]

$$Stop\_Row = min(Start\_Row + Chunk\_Dimension, Output\_Map\_Height) \tag{3.7}$$

In this way, if the chunk dimension is higher than the leftover rows the core has to compute, the last output row is set as that core's *Stop_Row*. It certain cases, like when the output height is very small or not a multiple of the number of cores, the most optimal parallelization strategy may exclude one or more cluster cores from the task. In this case, both the start and stop row indexes are set on the last row, forcing the core to skip the main execution cycle.

In order to maximize data re-usage in a very memory intensive kernel such as *MatMul*, PULP-NN reuses the weight elements in the inner loop to compute a second *sdotp4* operation on the second *im2col* buffer, adding the cost of an additional load. PULP-NN upscales the *MatMul* kernel to the 4×2 size, which consists in computing 4 consecutive channels over 2 adjacent output pixels on the output feature map each loop iteration. This is the most that the register structure can support without spilling variables into the stack. A 2×4 kernel is also viable, but requires 2 extra *im2col* buffers, doubling the memory requirements.

Splitting the rows to be worked on implies the necessity for each core to have private *im2col* buffers, 2 each for the 4×2 *MatMul* kernel. Parallelization profits come with an increased memory footprint.

3.3 DORY

IoT endnodes usually couple a fast but small L1 memory with a larger but slower L2 as background memory. By not using hardware caches, the energy usage is

```
LTO: for (o=0; o<o_M; o++)                    // output chan tiles
     LTH: for (h=0; h<h_M; h++)               // spatial height tiles
     LTW: for (w=0; w<w_M; w++)               // spatial width tiles
          LTI: for (i=0; i<i_M; i++)          // input chan tiles
               dma_wait (X_L1_load); swap (X_L1_load, X_L1_exec)
               dma_async (X_L1_load ← X_L2[i,w,h])
               dma_wait (X_L1_load); swap (W_L1_load, W_L1_exec)
               dma_async (W_L2[i,o] ← W_L1_load)
               if (o + h + w + i > 0)
                    DNN_kernel (X_L1_exec, W_L1_exec, Y_L1_exec)
          // from 3° cycle - fully operating pipeline
          if (o + h + w + i > 1)
               dma_wait (Y_L1_load); dma_async(Y_L1_load → Y_L2[o,w,h])
          swap (Y_L1_load ,Y_L1_exec)
```

Figure 3.5: DORY loop nest, implementing the double buffering scheme[35]

much lower, but every memory transfer has to be explicitly managed, complicating the development of complex algorithms with high memory footprint, like DNN inference.

DORY is a lightweight software level cache dedicated to *DNN Deployment Oriented to memoRY* [35]. It uses static data tiling and DMA-based double buffering to hide the L1-L2 manual memory transfers complexity. Experimental results reach less than 4% performance overhead when using activation and weight values stored in L2 instead of L1.

DORY has two main blocks: *DORY Optimizer* and *DORY SW-cache*.

The DORY Optimizer takes as input the maximum L1 memory and layer constraints passed as a PyTorch layer (PyTorch is a popular open source machine learning library [36]), like the number of channels, dimension of the zero-padding required and spatial dimensions. The optimizer has to maximize back-end performance and energy efficiency, mathematically abstracted to a integer constraint problem. To solve it, DORY Optimizer uses the open-source OR-tools from Google-AI [37], a tool specialized in matching user-defined constraints over data. By solving the problem, OR-Tools defines the most optimal tiling dimensions, that can fit in L1, to be applied on the input feature map.

The DORY SW-Cache is used whenever a DNN layer can't fit in L1 memory. It automatically generates code to minimize overhead in the target layer, like the asynchronous DMA transfers and implementing double buffering, without any input from the programmer.

Figure 3.5 shows the scheduling scheme, looping over the output channels, width, height and input channels. Looping limits are found by the DORY Optimizer's tiling tool. Minimal imbalance is added thanks to the fact that DMA transfers are asynchronous and non-blocking.

Chapter 4

PULP-TCN

4.1 Overview

PULP-TCN is a computing library optimized for TCN inference on IoT edge devices. It uses the PULP-NN libray [15] as a starting point, optimizing its strong points in the field of convolutional operations over one dimensional sequential data.

Besides the four convolutional kernels accurately described in this chapter, PULP-TCN also includes a set of kernels for implementing Fully Connected, Add, Max/Average Pooling layers, necessary for creating a real, complete TCN. These supporting kernels are very similar to the PULP-NN's implementation, but with slight modifications to reduce overhead by exploiting the fact that data is 1-dimensional.

The PULP-NN convolution kernels can be used with 1D data sequences without editing the source code. This is done by considering width or height as the time dimension, while setting to 1 the unused dimension. This workaround, however, negatively impacts on the performance level. If the width dimension is converted to the time dimension, parallelization is not possible because the height dimension would be set to 1, bringing the number of MAC operations per cycle to less than one if the input data sequence length is non-optimal. Converting the height dimension gives better results, but with a lower performance level than the one documented in [15]. Furthermore, the PULP-NN library offers no support for dilation. The only way to implement it is by using zero-padding on the weight banks, critically increasing the memory footprint and reducing the convolution kernel's speed even more.

PULP-TCN solves these problems by adapting the existing solution to the field of one dimensional sequential data, and by exploring new optimization strategies to be integrated.

Multi-core Parallelization

TCN's main field of applications is sequence modelling tasks [10]. As explained in chapter 2.1.3, both input and output features of the convolutional layer are temporal data sequences over multiple channels. The chosen data layout to represent them is:

- Data along the channels, referring to the same time slot, is stored with a stride of 1.
- Data along time, referring to the same channel, is stored with a stride equal to the number of channels.

With this layout, parallelization can be done done on the time dimension, equally dividing the number of output time slots to be computed by each core. This is done in a similar fashion of PULP-NN, as illustrated in chapter 3.2. The starting and stopping time slots delimiting the area computed by each core are calculated using the core ID, like in PULP-NN.

MatMul kernels

The core operation for convolution is done by the MatMul kernel. Despite the fact that we operate over 1D sequential data, the PULP-NN solution designed for 2D feature maps is an excellent starting point. In fact, the *im2col* support buffer was used to "flatten" input data, while we already use linear sequences. Thanks to the chosen data layout, access to adjacent channels on the same time slot is extremely regular, allowing us to use hardware loops and load/store operations with post increment in the inner loop. Slightly different versions of the *MatMul* 4×2 *kernel* have been developed for each convolution kernel, but they all share the same design choices:

1. Get as argument the pointers to the memory areas where the necessary data to compute the 2 output sequence time slots are stored.
2. Load 4 INT-8 values from each input pointer, storing them into two *v4s* variables from the SIMD architecture.
3. Load 4 INT-8 weight values from 4 adjacent weight filter banks, saving them in four *v4s* variables.
4. Multiply each input *v4s* variable with the weights, for a total of 8 element-wise *v4s* multiplications. Each product is summed to a different INT-32 accumulator.
5. Increase the pointers to load the next values.

6. When reaching the end of the filter weight banks, accumulator values get quantized before being saved in the output sequence. The quantization method is chosen by the programmer using specific flags.

The differences between the *MatMul* kernel versions are highlighted in the convolution kernel descriptions.

Quantization

Accumulated values in the *MatMul* kernels have to be quantized from 32 to 8 bit. Three quantization methods are available, by setting the appropriate flags in the convolution kernels. Those flags depend on the optimization techniques applied to the convolution layer.

If no flags are set, *clip8* is used. This method calls *___builtin_pulp_clipu_r(x, 255)*, a simple built-in instruction that sets x to 0 if it's a negative value, or to the max value if higher than it. In this case, the maximum value is set to 255, the highest number representable by an unsigned 8-bit integer.

If only the *relu* flag is set, *pulp_nn_quant_u8* is called. this function multiplies then shifts the accumulated value by *out_mul* and *out_shift*, respectively. These two are constant values passed to the convolutional kernel. After these operations, *clip8* gets called once again.

If both the *relu* and *batch_norm* flags are set, *pulp_nn_bn_quant_u8* is invoked:

$$integer\_image\_phi = (k * phi) + lambda; \tag{4.1}$$

Where *phi* is the accumulator, while k and *lambda* are constant values that depend on the output channel. In fact, k and *lambda* are $nChannels$-sized vectors of constant INT-32 values passed to the convolutional kernel. This allows us to implement both quantization and normalization in a single function.

Convolution kernel main loop

Developed kernels share a common base structure. First, each core calculates the output time slots they have to work on (see "Multi-core Parallelization" above), then they enter the main execution loop.

The main loop is divided in two stages: *Buffering* and *MatMul*. An exception is the first described kernel (the *No Pad, no Dilation* one) where buffering is not required. In the buffering stage, the core prepares the memory array containing the input features required to compute the 2 output time slots currently being targeted. Once the buffer is completed, the core proceeds to pass a pointer to the buffer as argument on the *MatMul* 4×2 kernel, then shift the target to the next two time slots, until the *stopping* time slot is reached.

If the number of time slots to be computed is odd, the last one will be left over from the main cycle. In this case, half of the support buffer is used by the *MatMul* 4×1 kernel. This kernel is basically the same as its 4×2 counterpart, but only takes input values from a single input buffer, and produces output sequence values on a single time slot.

The following sections present more detailed information of the convolution kernels. Each one presents different advantages and disadvantages, which motivates the need to model their individual performance, then evaluate which kernel is best using considering the layer parameters.

4.2 No pad no dilation convolution kernel

The first approach taken to maximize performance was to eliminate the buffering step from the convolution kernel. This is possible when the convolution does not include padding nor dilation, due to the fact that the input sequence is extremely regular by construction, and doesn't require any "flattening" operation because it's already one-dimensional.

There is a linear relation between the input and output sequence, in fact the pointer to the first input feature required for computing the i-th output time slot can be found with:

$$pFirst = pInputSequence + (i \cdot Stride \cdot n.\_Channels\_Input) \tag{4.2}$$

Where *pInputSequence* is the pointer to the start of the input sequence.

The MatMul kernel takes *pFirst* as input. It also calculates the pointer to the first input feature required for computing the next time slot. This is simply done by adding the product of stride and the number of channels to the first pointer. From there, it extracts the necessary values for computing the output sequence directly from the input layer, without using any supporting buffer or preparation process.

This convolution kernel presents the best overall performance out of the four (see results chapter). This is due to the heavily reduced number of memory transfers required. However, complete removal of the support buffer precludes the kernel from automatic padding and implementing dilation, one of the key features of TCNs.

While padding can be introduced manually by the programmer on the input sequence, dilation requires the reintroduction of support buffers, as seen in the next convolution kernels. This kernel is still useful when dilation is not required or the history size to be explored is not very large.

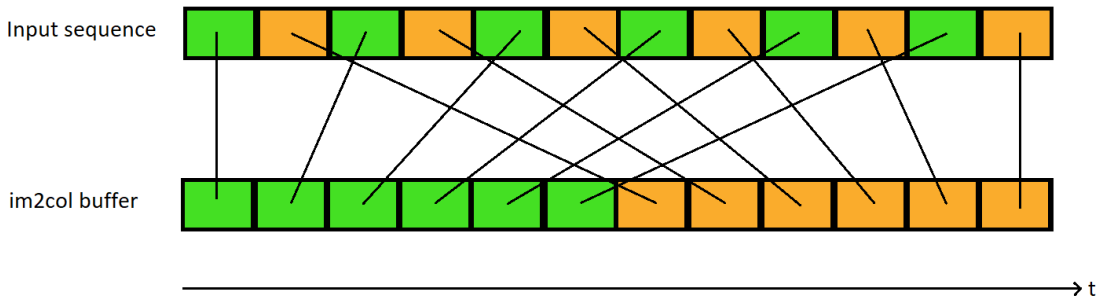

Figure 4.1: Graphical representation of *im2col* buffer building, with dilation rate 2 and a filter size of 6. Green cells are the values required to compute the first output time slot, while the orange ones are required for the next one.

4.3 Dilated convolution kernel

The second developed kernel reintroduces the buffering step in the main cycle. This process is similar to the *im2col* buffer construction already seen in PULP-NN. Appendix B shows the source code used to build the *im2col* buffer while managing automatic padding and dilation rates higher than 1.

Similarly to the original solution, the support buffer holds all the input sequence's values required for computing two sequential output time slots. The memory transfers from the input feature map to the buffer are delegated through MCHAN-DMA calls, instead of being explicitly done by the programmer.

Despite using the *im2col* name in the support buffer and methods, the input layer doesn't need to be flattened because it's a temporal sequence. The *im2col* buffer is used to artificially store non-adjacent input time slots in a sequential fashion.

Like in the previous kernel, the pointer to the first input element required for computing the i-th output time slot is found by using the equation 4.2.

```
1 void pulp_nn_im2col_uint8(uint8_t * pInput, uint8_t * pOutput,
    unsigned int blockSize)
2 {
3     mchan_transfer(blockSize, 1, 1, 0, 1, 0, 0, (unsigned int) pInput
    , (unsigned int) pOutput, 0, 0);
4 }
```

mchan_transfer accepts an input and an output pointer, transferring *blocksize* bytes from one to the other. In our case, the block size is equal to the number of input channels. This moves all the input data available regarding the selected time slot in the *im2col* buffer. While this function is usually used to move data

from L2 to L1 memory and vice-versa, in this case it's used to move data from a L1 location to another.

In order to automatically manage padding, we pass to the *mchan_transfer* a pointer to a memory area filled with 0 as input, moving it inside the support buffer.

Using *mchan_transfer* is more efficient than simply implementing software copy instructions. The DMA is a high-performance hardware component working in parallel with the cluster. The cluster cores can therefore delegate multiple memory transfers on it and then execute other tasks in the meantime. Using *mchan_barrier* guarantees that all delegated memory transfers have been successfully completed. In case they are not, the calling core is pushed on a low-energy consumption waiting routine, that is interrupted by an event register.

Dilation is implemented when moving the input sequence pointer after each memory transfer delegation. The pointer is moved by $dilation\_rate \cdot n\_channels$ elements, effectively skipping $dilation\_rate - 1$ time slots over the input sequence.

The buffering phase is repeated until the *im2col* buffer isn't filled with all the necessary data for the *MatMul* 4×2 kernel, which means that this buffer stores a number of values from the input sequence equal to:

$$n\_Channels\_Input \cdot filter\_size \cdot 2 \tag{4.3}$$

Once the buffer is complete, the main cycle can proceed to the second phase, where the output sequence is actually computed.

The *MatMul* 4×2 kernel is basically the same used by the the first convolution kernel, and described in this chapter's overview. The main difference are the two pointers used to extract data: in this version, the first pointer points to the start of the *im2col* buffer. The second one points exactly to the middle point of the buffer ($p2 = p1 + n\_Channels\_Input \cdot filter\_size$).

If the chunk dimension is not even, one output time slot will be left over. All data required to compute it is stored in the first half of the *im2col* buffer, passed as argument to the *MatMul* 4×1 kernel. This method is the same used by the first convolution kernel.

4.4 Double Buffer Convolution Kernel

As stated before, delegating memory transfers to the DMA allows cluster cores to do some other tasks in parallel. This kernel is an attempt to exploit the asynchronous and non-blocking nature of memory transfers to implement a pipeline using the double buffering pattern.

As the name suggests, two *im2col* buffers are used. After filling the first one, the *MatMul* 4×2 kernel is not immediately invoked. Instead, the core starts delegating memory transfers to the DMA to fill the second buffer. Once all

necessary *mchan_transfer*s are invoked, the convolution kernel executes *MatMul* 4×2 over the first buffer. Meanwhile, the DMA is busy transferring values on the second buffer. Once the *MatMul* operation is done, the two buffers swap their roles: the core calls *mchan_transfer*s to fill the first buffer, then calls *MatMul* 4×2*2* over the second buffer, swap the buffers' roles, and so on. This continues until the stopping time slot is reached. At that point, *MatMul* 4×2 is called one last time, using the last stored buffer. *MatMul* 4×1 may be used too, if the number of output time slots is not even.

While this solution is theoretically faster than the previous kernel, the DMA speed is so high that, in practice, real speedups are only achieved on very long sequences. In smaller sequences the pipeline overhead actually slows the solution down. Is it also worth noticing that the memory footprint of this kernel is doubled, due to the fact it uses two *im2col* buffers.

4.5 Indirect convolution Kernel

The *Indirect convolution* algorithm is a solution more focused on reducing memory footprint than improving execution performances [19].

The *im2col* buffer is replaced by an *indirection* buffer. This buffer doesn't store the input sequence values, instead it stores the pointers to the input time slots.

The kernel structure is basically the same as the previous ones. In the buffering phase, there's no need to invoke the DMA. Instead, the pointer to the first item of the targeted input time slot is directly copied into the *indirection* buffer. Dilation is implemented like in the previous kernels, by shifting the input sequence pointer over the time dimension, skipping $dilation\_rate - 1$ time slots. Padding is implemented by copying a pointer to a memory area filled with zeros beforehand.

This solution dramatically reduces the memory footprint, especially if the number of input channels is very high. It only requires to memorize a single INT-32 pointer for each required time slot, instead of all the time slot's values.

Performance, however, takes a hit when executing the *MatMul* kernels. Data access is not completely sequential like in *im2col*, therefore an extra loop is required in the computational kernel, whenever loading a new pointer from the *indirection* buffer.

Even considering this, the kernel still has better performances than the PULP-NN solution. Also, the greatly reduced memory usage makes it a very suitable solution on convolution layers involving a great number of channels.

Chapter 5

Results

5.1 Parallelization

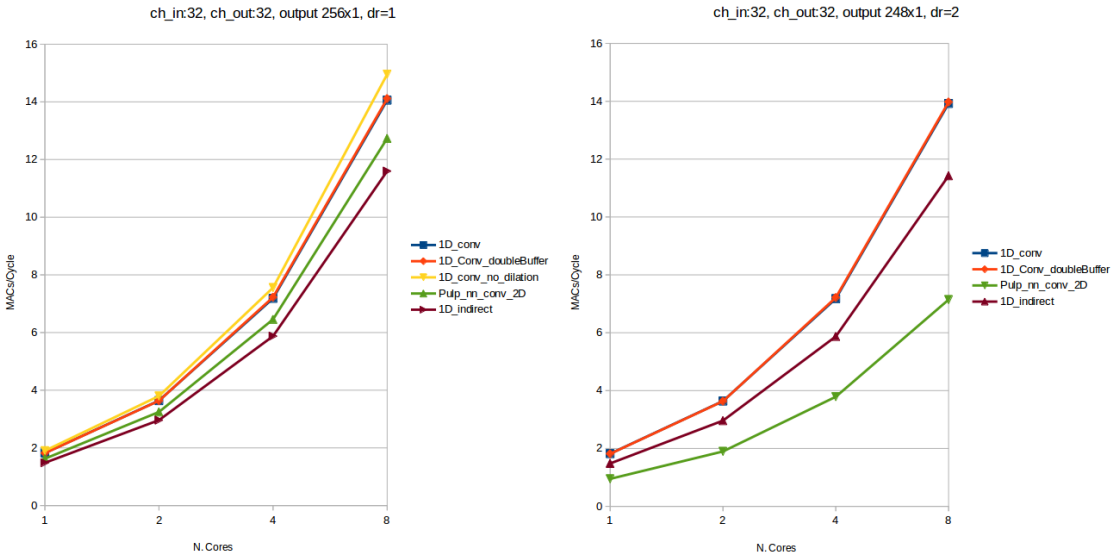

Figure 5.1: Performance of the convolution kernels considering the number of cluster cores used. Left example includes the *no_dilation* kernel since dilation rate is equal to 1.

In this section we analyze the multi-core parallelization of the kernels. In figure 5.1 are illustrated experimental results. The kernels have been tested on an input sequence of 256 time slots over 32 channels, with a filter size of $32 \times 9 \times 32$ ($outputchannels \times filtersize \times inputchannels$). The test was repeated twice, to compare performance between a normal ($dilationrate = 1$) and dilated ($dilationrate = 2$) convolutions. The tests have been deployed on the GVSOC

virtual platform, and results collected using GAP-8 built in performance counters. The PULP-NN convolution kernel was tested too, using a $256 \times 1 \times 32$ (HWC data layout) input tensor and a $32 \times 9 \times 1 \times 32$ filter.

The kernel speedup is almost linear with the number of cores, achieving up to $\sim 7.87\times$ speedup in the best case and never going below $7.5\times$ with 8 cores. Results show that complete removal of buffering leads to the highest overall performance, however this kernel does not support dilated convolution.

The indirect convolution algorithm performance are worse than the PULP-NN solution in a normal convolution. When considering a dilated convolution, however, PULP-NN has to extensively use zero padding on the weight filter bank, in this case to increase its size to $32 \times 17 \times 1 \times 32$ by simulating the dilation with zero filling. This is done to increase the history window, like dilation does (as stated in chapter 2.1.3, the effective history size of a dilated convolution layer is $(fd-1) \cdot dr$, where fd is the filter bank dimension and dr the dilation rate). Because of this, the PULP-NN has to compute a large number of useless MAC operations, making the indirect convolution a better solution.

As stated before, the double buffering solution has almost identical performance as single buffer solution, but with a slightly higher speedup rate for multi-core execution.

5.2 Memory usage

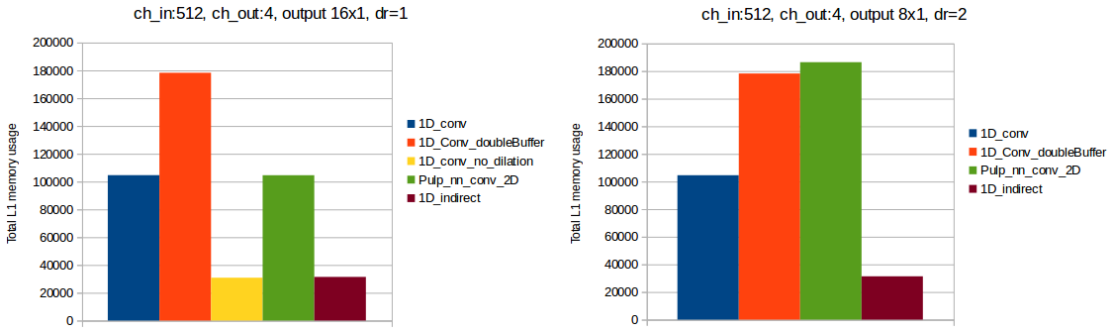

Figure 5.2: Memory usage (in bits) of the L1 memory area by dirrent kernels, using 8 cluster cores.

For profiling the memory footprint of each kernel, we consider a different convolution layer. This one has a smaller, 24 time slots long input sequence, over a large (512) number of channels, with a filter size of $4 \times 9 \times 512$. Like earlier, we compare memory footprint during a normal and a dilated convolution. Results are readable on figure 5.2. These results have been calculated considering all the 8

cluster cores available being used.

Memory footprint on PULP-TCN normal convolution kernel is almost the same as PULP-NN's, while the double buffer increases it by more than 70%. Removing buffering altogether massively reduces the memory required for a normal convolution, but precludes us from implementing a dilated one without occupying too much memory.

The indirect convolution kernel allows dilation with very minimal memory footprint over the no-buffer solution. This makes it the most optimal kernel to be used for this particular layer, memory wise.

To be noted on the right graph: increasing dilation rate does not affect memory usage in any significant way on the developed kernels. However, dilation is very impactful on the PULP-NN convolution kernel. In the example, PULP-NN requires more memory usage than the double buffer solution by just increasing the dilation rate to 2.

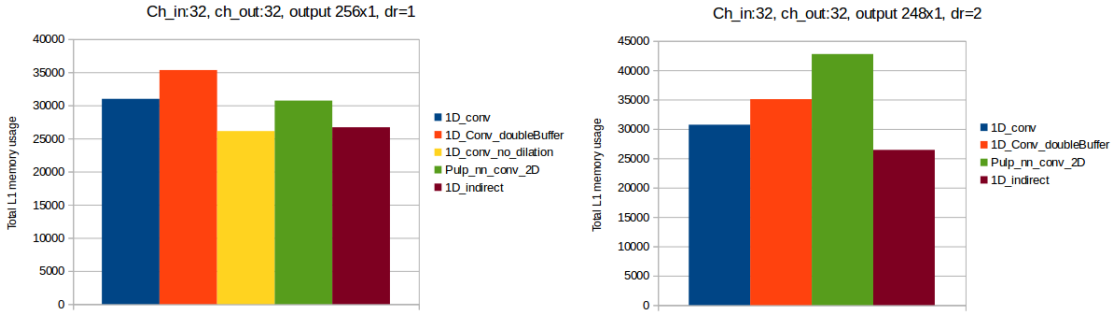

Figure 5.3: Memory usage using 8 cluster cores on a layer with a lower number of input channels.

In figure 5.3 we compare memory usage when the input layer is only over 32 channels. Memory footprint is less of a problem, in this case, and the differences between the kernels aren't as big as the previous layer. The no-buffering kernel is still the one with the lowest memory footprint in normal convolutions, followed tightly by the indirect convolution kernel. Increasing dilation rate still introduces a considerable memory footprint on the PULP-NN convolution kernel, while it doesn't affect PULP-TCN ones.

5.3 Kernel models

In order to compare the kernels' different performances, ideal single-core models have been built by studying the low-level assembly code of the solutions. These models use as base a convolution layer taking an input sequence 112 time slots long, over 32 channels, using a $32 \times 9 \times 32$ filter. We then explored changes in

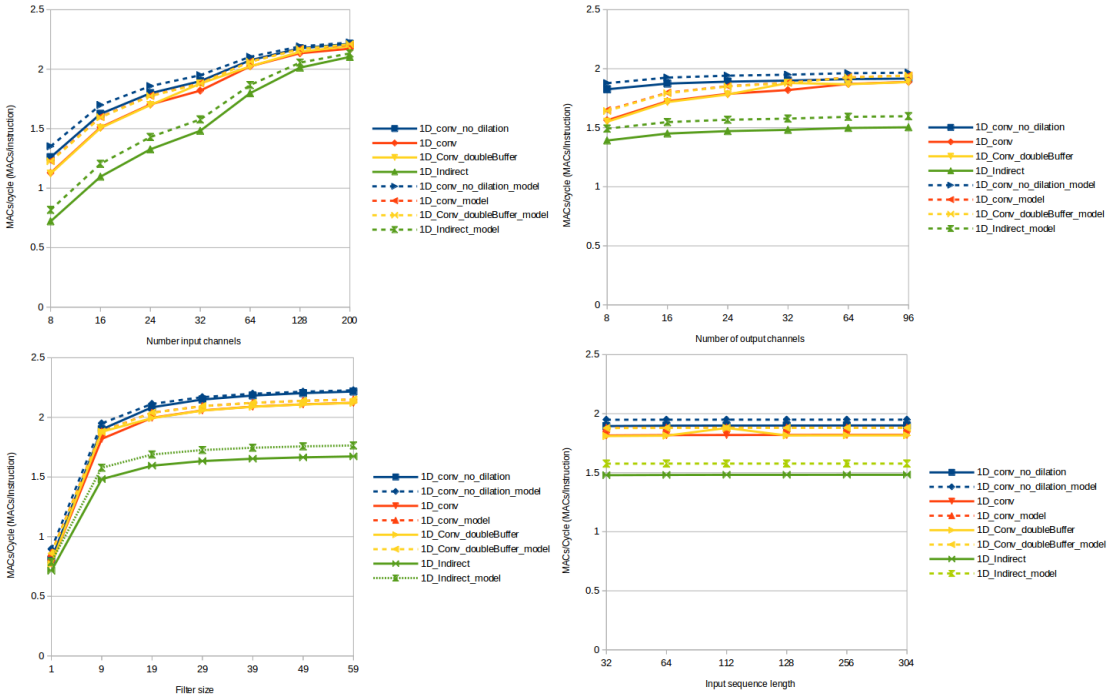

Figure 5.4: Single-core models comparison between various input parameters. Dashed lines are ideal models mathematically calculated, continued lines are experimental results.

performace by adjusting each parameter singularly. Figure 5.4 show how these models are influenced by the main layer parameters(number of channels in input, number of output channels, filter size, input sequence length). We also compared the accuracy of these models with the experimental results obtained by testing the kernels, without multi-core parallelization, on GVSOC. As shown in the graphs, the real performance follow the same trend of the ideal models, and almost overlap in certain cases. Real performance are always lower than the ideal ones, because they ignore a number of factors, such as instruction cache misses and delays, load and store stalls.

Kernels performances grow with each parameter, but with different rates. Models and experimental results show that the most influencing parameter is the number of input channels. On the other side, the input sequence size doesn't influence kernels performance by much. To be noted, if filter size is 1 performance tank to minimal levels no matter the chosen core, making it the worst-case scenario for PULP-TCN.

In figure 5.5 are shown the same models, but build while using all the 8 available cores. Even with parallelization, these models still follow the same trend as the

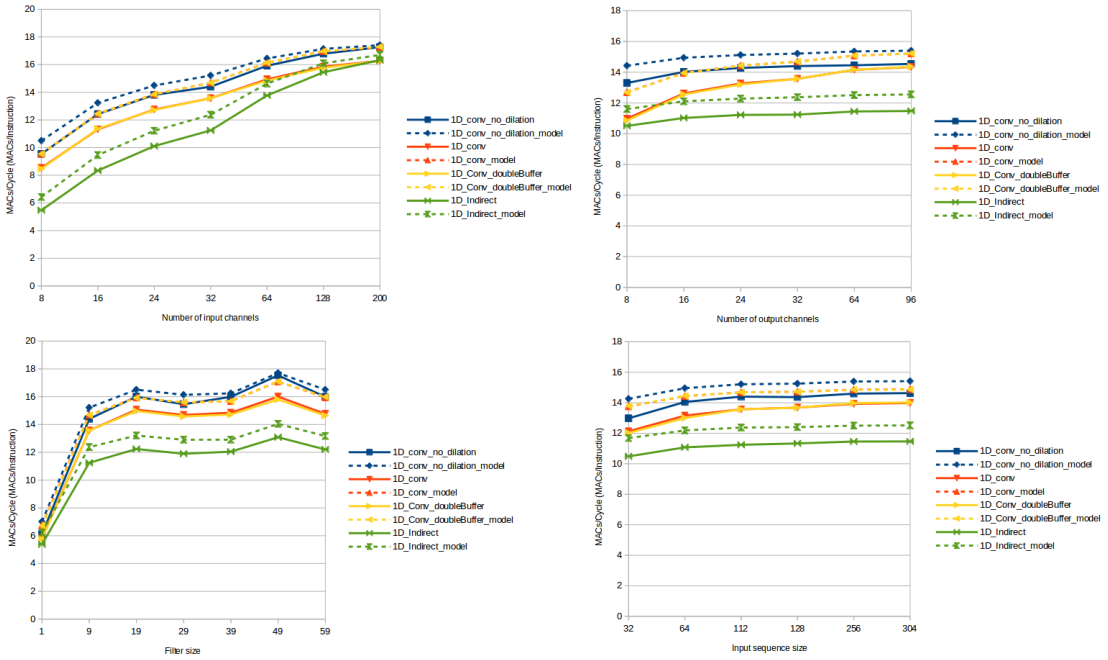

Figure 5.5: 8-Cores kernel models comparison, built on the same convolution layer as the previous figure.

single-core ones. Some features are highlighted in these models:

- The indirect convolution provides higher performance than the single and double *im2col* buffer kernels when the input layer has a large number of channels.
- Performance of the kernels implementing *im2col* buffers is very similar to the no-buffer kernel with a large number of output channels.
- The filter size model is extremely non-linear, presenting multiple local maximums.
- Input size has more influence on the kernel performance, but it's still very minimal compared to the other models.

The difference between the real results and the ideal models is higher than the single cores models. This is because the models also ignore delays due to the parallelized environment, such as stalls due to synchronization barriers and TCDM contestantions.

5.4 Tiling models

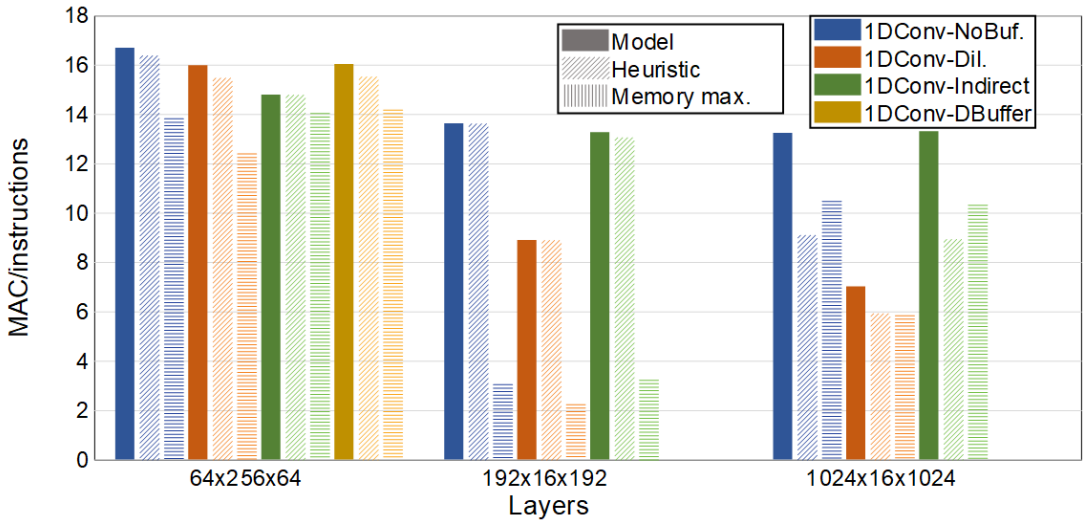

Figure 5.6: Tiling performance over different layers. 3 different tiling constraints are compared.

When testing the kernels, we used the GVSOC virtual platform's options to artificially increase the L1 memory area available, to test out larger layers. However, on a real chip, this isn't feasible. In order to work around the problem, tiling of the input layer is necessary. Tiling is the process of splitting the input sequence that doesn't fit in memory into *tiles* that do fit, and work on them separately. Finally, recollect their output into a single output sequence, equivalent to the one that would have been computed if tiling wasn't used.

We use Google's Or-tools [37] to develop an auto-tiler that calculates the most optimal tile dimensions to minimize overhead. This is done by imposing numerical constrains over the tile dimensions to achieve the following conditions:

- Each tile must be small enough to fit into L1 memory, especially if we are using all 8 cluster cores.
- The overall performance (MACs/Cycle and MACs/Instruction) must be maximized. This is equivalent to minimize the number of cycles and instructions required to compute all the tiles.

Google Or-tools offers a set of classes that allows automatic optimization of the problem. We use the Original Constant Problem solver tool [38]. We pass as argument the input layer parameters and a set of numerical constraints, such as making sure the tile dimensions fit in L1 memory. We then call the *Phase* method

to maximize the number of MACs/instruction of the entire layer. This will return the most optimal tile dimensions that meet the numerical constraints.

After calculating the tile dimensions, we can compare performance between each kernel to choose which one is more fit for computing the input layer. In fact, since each kernel has different memory requirements, each input layer geometry has a different optimal convolution kernel.

In figure 5.6 we analyze the performance of the kernels when tiling is applied on different layers. Three different optimization sets have been applied in the automatic tiling of the layers. *Model* uses as maximization criteria the 8-cores models calculated in section 5.3. *Heuristics* uses similar heuristics to the ones applied in the DORY optimizer, already cited in chapter 3.3. Finally, *Memory max.* simply forces the tile dimension to be the maximum it can fit in L1 memory. Results show that using the calculated models to compute the tile dimensions brings the best performance in each kernel, no matter the layer's dimensions. Also, by introducing tiling, the most optimal kernel is not always the same. In fact, by greatly increasing the number of channels, the indirect convolution kernel has better performances than the im2col solution, and very similar to the no-buffer kernel. This results is achieved since the creation of the im2col, given the high number of input channels, creates a very high memory overhead. Note that the input channels are never tiled in DORY and hence never reduced for im2col creation. With 1024 ch_in, the memory overhead is 1024*2*8, 16 KBs.

Chapter 6

Conclusions and Future Works

We presented PULP-TCN, a set of convolutional kernels developed for TCN inference on IoT edge nodes devices.

Compared to the PULP-NN kernels, overhead has been reduced to improve the overall computing performance for convolution over 1-dimensional input sequences. New optimization strategies, including buffering removal, integration of DMA memory transfers and double buffering, have been explored to further improve the kernels' performance. We also explored how to reduce memory footprint, by trying to remove buffering and implementing a different kernel which doesn't rely on im2col buffer, using the indirect convolution algorithm.

We also introduced dilation deployment, an alternative way to increase the history size explored by the kernel without enlarging the weight filter. This was done with very minimal overhead and no additional memory footprint. This feature is vital for TCN inference, and was not supported in PULP-NN.

Future works include integration of this library in the DORY framework, and executing the kernels as back end for complete networks. Once they have been fully optimized, we can proceed to implement PULP-TCN on bio-medical application (such as intelligent cardiograms) for IoT wearable sensors.

Appendix A

Mergesort example

```
    #include "rt/rt_api.h"
#include <stdio.h>
#include <stdlib.h>

#define STACK_SIZE      2048
#define MOUNT           1
#define UNMOUNT         0
#define CID             0

unsigned int done = 0;
#ifdef PRINTF_UART
unsigned int __rt_iodev=RT_IODEV_UART;
#endif

typedef struct {
    int *v;
    int size;
}vector;

static void merge (int *a, int n, int m) {
    rt_alloc_req_t req0;
    rt_free_req_t  req1;
    rt_alloc_cluster(RT_ALLOC_L2_CL_DATA, n*sizeof(int), &req0);
    int* x;
    x = (int *) rt_alloc_cluster_wait(&req0);
    int i, j, k;
    for (i = 0, j = m, k = 0; k < n; k++) {
        x[k] = j == n      ? a[i++]
             : i == m      ? a[j++]
             : a[j] < a[i] ? a[j++]
             :               a[i++];
```

```
32      }
33      for (i = 0; i < n; i++) {
34          a[i] = x[i];
35      }
36          rt_free_cluster(RT_ALLOC_L2_CL_DATA, x, n*sizeof(int), &req1)
        ;
37      rt_free_cluster_wait(&req1);
38  }
39
40  static void merge_sort (int *a, int n) {
41      if (n < 2){
42          return;
43      }
44
45      int m = n / 2;
46      merge_sort(a, m);
47      merge_sort(a + m, n - m);
48      merge(a, n, m);
49  }
50
51  static void parallel_merge_sort(void *arg){
52      vector* args=(vector*) arg;
53      int id=rt_core_id();
54      int i;
55      merge_sort(args[id].v, args[id].size);
56      rt_team_barrier();
57      if(id<4){
58          int size=args[((2*id))].size;
59          int size2=args[((2*id)+1)].size;
60          args[(2*id)].size+=size2;
61          for(i=0; i<size2; i++){
62              args[(2*id)].v[size+i]=args[((2*id)+1)].v[i];
63          }
64          merge(args[(2*id)].v, args[(2*id)].size , size);
65      }
66      rt_team_barrier();
67      if(id<2){
68          int size=args[((4*id))].size;
69          int size2=args[((4*id)+2)].size;
70          args[(4*id)].size+=size2;
71          for(i=0; i<size2; i++){
72              args[(4*id)].v[size+i]=args[((4*id)+2)].v[i];
73          }
74          merge(args[(4*id)].v, args[(4*id)].size , size);
75      }
76      rt_team_barrier();
77      if(id==0){
78          int size=args[id].size;
79          int size2=args[id+4].size;
```

```
80          args[id].size+=size2;
81          for(i=0; i<size2; i++){
82              args[id].v[size+i]=args[id+4].v[i];
83          }
84          merge(args[id].v, args[id].size, size);
85      }
86      rt_team_barrier();
87      printf("Core id %d finished merge_sort!\n", id);
88      return;
89  }
90
91  static void cluster_entry(void *arg)
92  {
93    printf("Entering cluster on core %d\n", rt_core_id());
94    printf("There are %d cores available here.\n", rt_nb_pe());
95
96      vector* a=(vector*) arg;
97      vector args[8];
98          int* v=a->v;
99      int n=a->size;
100     int c=n/8;
101     int r=n-(c*7);
102     printf("c=%d, r=%d\n", c, r);
103         int i;
104     int j;
105         for(i=0; i<8; i++){
106         if(i!=7){
107             args[i].v=malloc(c*sizeof(int));
108             args[i].size=c;
109             for(j=0; j<c; j++){
110                 args[i].v[j]=v[(i*c)+j];
111             }
112         }
113         else{
114             args[i].v=malloc(r*sizeof(int));
115                 args[i].size=r;
116                 for(j=0; j<r; j++){
117                     args[i].v[j]=v[(i*c)+j];
118                 }
119
120         }
121     }
122
123     rt_team_fork(8, parallel_merge_sort, args);
124
125     printf("Sorted vector:\n");
126     for (i = 0; i < args[0].size; i++)
127             printf("%d%s", args[0].v[i], i == n - 1 ? "\n" : " ");
128
```

```
129   printf("Leaving cluster on core %d\n", rt_core_id());
130 }
131
132 static void end_of_call(void *arg)
133 {
134   printf("[clusterID: 0x%x] Hello from core %d\n", rt_cluster_id(),
       rt_core_id());
135   done = 1;
136 }
137
138 int main()
139 {
140   printf("Entering main controller\n");
141
142   int a[] = {65, 4, 2, -31, 0, 99, 2, 83, 782, 1, 84, -12, 88, 23,
       92, 124, 5, 18, 182, 234, 33, 21, 2, 43, 25};
143   int n = sizeof a / sizeof a[0];
144   int i;
145   for (i = 0; i < n; i++)
146         printf("%d%s", a[i], i == n - 1 ? "\n" : " ");
147
148   vector arg;
149   arg.v=a;
150   arg.size=n;
151
152   if (rt_event_alloc(NULL, 4)) return -1;
153
154   rt_event_t *p_event = rt_event_get(NULL, end_of_call, (void *) CID)
       ;
155
156   rt_cluster_mount(MOUNT, CID, 0, NULL);
157
158   rt_cluster_call(NULL, CID, cluster_entry, &arg, NULL, 0, 0,
       rt_nb_pe(), p_event);
159
160   while(!done)
161       rt_event_execute(NULL, 1);
162
163   rt_cluster_mount(UNMOUNT, CID, 0, NULL);
164
165   printf("Test success: Leaving main controller\n");
166   return 0;
167 }
```

Appendix B

im2col buffering

```
if(i_out_y < padding_y_top){
            for(i = i_out_y * stride_y - padding_y_top; i < i_out_y * stride_y - padding_y_top + (dim_kernel_y*(1+dilation)-dilation); i += (1+dilation)){
                if(i < 0 || i >= dim_in_y)
                    pulp_zero_mem_dma(pIm2Col, ch_in, zero);
                else
                    pulp_nn_im2col_uint8((uint8_t *) pInBuffer + (i * ch_in), pIm2Col, ch_in);

                pIm2Col+=ch_in;
            }
        }
        else if(i_out_y < dim_out_y - padding_y_bottom){
            for(i = i_out_y * stride_y-padding_y_top; i < i_out_y * stride_y - padding_y_top + (dim_kernel_y*(1+dilation)-dilation); i += (1+dilation)){
                pulp_nn_im2col_uint8((uint8_t *) pInBuffer + (i * ch_in), pIm2Col, ch_in);
                pIm2Col += ch_in;
            }
        }
        else{
            for(i = i_out_y * stride_y - padding_y_top; i < i_out_y * stride_y - padding_y_top + (dim_kernel_y*(1+dilation)-dilation); i += (1+dilation)){
                if(i < 0 || i >= dim_in_y)
                    pulp_zero_mem_dma(pIm2Col, ch_in, zero);
                else
                    pulp_nn_im2col_uint8((uint8_t *) pInBuffer + (i * ch_in), pIm2Col, ch_in);
```

```
23
24                      pIm2Col+=ch_in;
25                  }
26              }
```

Bibliography

[1] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. «Deep learning». In: *nature* 521.7553 (2015), pp. 436–444 (cit. on pp. 1, 5).

[2] Riccardo Miotto, Fei Wang, Shuang Wang, Xiaoqian Jiang, and Joel T Dudley. «Deep learning for healthcare: review, opportunities and challenges». In: *Briefings in bioinformatics* 19.6 (2018), pp. 1236–1246 (cit. on p. 1).

[3] Philip Sparks. *The route to a trillion devices*. URL: `https://community.arm.com/iot/b/internet-of-things/posts/white-paper-the-route-to-a-trillion-devices` (cit. on p. 1).

[4] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. «Edge computing: Vision and challenges». In: *IEEE internet of things journal* 3.5 (2016), pp. 637–646 (cit. on p. 1).

[5] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S Emer. «Efficient processing of deep neural networks: A tutorial and survey». In: *Proceedings of the IEEE* 105.12 (2017), pp. 2295–2329 (cit. on p. 1).

[6] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. «Quantized neural networks: Training neural networks with low precision weights and activations». In: *The Journal of Machine Learning Research* 18.1 (2017), pp. 6869–6898 (cit. on pp. 2, 26, 29).

[7] Song Han, Huizi Mao, and William J Dally. «Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding». In: *arXiv preprint arXiv:1510.00149* (2015) (cit. on p. 2).

[8] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. «Sequence to sequence learning with neural networks». In: *Advances in neural information processing systems*. 2014, pp. 3104–3112 (cit. on p. 2).

[9] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016 (cit. on p. 2).

[10] Shaojie Bai, J Zico Kolter, and Vladlen Koltun. «An empirical evaluation of generic convolutional and recurrent networks for sequence modeling». In: *arXiv preprint arXiv:1803.01271* (2018) (cit. on pp. 2, 11–13, 35).

[11] Md Zia Uddin. «A wearable sensor-based activity prediction system to facilitate edge computing in smart healthcare system». In: *Journal of Parallel and Distributed Computing* 123 (2019), pp. 46–53 (cit. on p. 2).

[12] Masanari Nishimura, Kei Hashimoto, Keiichiro Oura, Yoshihiko Nankaku, and Keiichi Tokuda. «Singing Voice Synthesis Based on Deep Neural Networks.» In: *Interspeech*. 2016, pp. 2478–2482 (cit. on p. 2).

[13] Ruize Xu, Shengli Zhou, and Wen J Li. «MEMS accelerometer based nonspecific-user hand gesture recognition». In: *IEEE sensors journal* 12.5 (2011), pp. 1166–1173 (cit. on p. 2).

[14] Eric Flamand, Davide Rossi, Francesco Conti, Igor Loi, Antonio Pullini, Florent Rotenberg, and Luca Benini. «GAP-8: A RISC-V SoC for AI at the Edge of the IoT». In: *2018 IEEE 29th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE. 2018, pp. 1–4 (cit. on pp. 2, 14–17).

[15] Angelo Garofalo, Manuele Rusci, Francesco Conti, Davide Rossi, and Luca Benini. «PULP-NN: A Computing Library for Quantized Neural Network inference at the edge on RISC-V Based Parallel Ultra Low Power Clusters». In: *2019 26th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*. IEEE. 2019, pp. 33–36 (cit. on pp. 2, 29, 30, 32, 34).

[16] Francesco Conti, Davide Rossi, Antonio Pullini, Igor Loi, and Luca Benini. «PULP: A ultra-low power parallel accelerator for energy-efficient and flexible embedded vision». In: *Journal of Signal Processing Systems* 84.3 (2016), pp. 339–354 (cit. on pp. 3, 13, 14).

[17] Liangzhen Lai, Naveen Suda, and Vikas Chandra. «Cmsis-nn: Efficient neural network kernels for arm cortex-m cpus». In: *arXiv preprint arXiv:1801.06601* (2018) (cit. on pp. 3, 26–29).

[18] Davide Rossi, Igor Loi, Germain Haugou, and Luca Benini. «Ultra-low-latency lightweight DMA for tightly coupled multi-core clusters». In: *Proceedings of the 11th ACM Conference on Computing Frontiers*. 2014, pp. 1–10 (cit. on p. 3).

[19] Marat Dukhan. «The Indirect Convolution Algorithm». In: *arXiv preprint arXiv:1907.02129* (2019) (cit. on pp. 3, 40).

[20] Lin Meng, Takuma Hirayama, and Shigeru Oyanagi. «Underwater-drone with Panoramic Camera for Automatic Fish Recognition Based on Deep Learning». In: *IEEE Access* PP (Mar. 2018), pp. 1–1. DOI: 10.1109/ACCESS.2018.2820326 (cit. on p. 4).

[21] Rajat Gupta. *Getting started with Neural Network for regression and Tensorflow*. URL: https://medium.com/@rajatgupta310198/getting-started-with-neural-network-for-regression-and-tensorflow-58ad3bd75223 (cit. on p. 5).

[22] Michael Cooper. «A Deep Learning Prediction Model for Mortgage Default A Deep Learning Prediction Model for Mortgage Default». PhD thesis. May 2018. DOI: 10.13140/RG.2.2.21506.12487 (cit. on pp. 6, 7).

[23] Xiaojin Zhu and Andrew B Goldberg. «Introduction to semi-supervised learning». In: *Synthesis lectures on artificial intelligence and machine learning* 3.1 (2009), pp. 1–130 (cit. on p. 7).

[24] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. «Imagenet classification with deep convolutional neural networks». In: *Advances in neural information processing systems*. 2012, pp. 1097–1105 (cit. on p. 8).

[25] Shan-Hung Wu. *1D and 2D Convolution*. URL: https://www.youtube.com/watch?v=yi7eZ_F39UY (cit. on p. 9).

[26] Jay Riccou. *What is max pooling in convolutional neural networks?* URL: https://www.quora.com/What-is-max-pooling-in-convolutional-neural-networks (cit. on p. 10).

[27] Sergey Ioffe and Christian Szegedy. «Batch normalization: Accelerating deep network training by reducing internal covariate shift». In: *arXiv preprint arXiv:1502.03167* (2015) (cit. on p. 11).

[28] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. «Dropout: a simple way to prevent neural networks from overfitting». In: *The journal of machine learning research* 15.1 (2014), pp. 1929–1958 (cit. on p. 13).

[29] Wayne Wolf, Burak Ozer, and Tiehan Lv. «Smart cameras as embedded systems». In: *computer* 35.9 (2002), pp. 48–53 (cit. on p. 13).

[30] Mariusz Bojarski et al. «End to end learning for self-driving cars». In: *arXiv preprint arXiv:1604.07316* (2016) (cit. on p. 13).

[31] Germain Haugou. *GVSOC documentation*. 2019. URL: https://gvsoc.readthedocs.io/en/latest/ (cit. on p. 17).

[32] GreenWaves Technologies. *GAP8 SDK Manual*. URL: https://greenwaves-technologies.com/manuals/BUILD/PULP-OS/html/index.html (cit. on p. 18).

[33] Greenwave Technologies. *GAPuino User's Manual*. URL: https://gwt-website-files.s3.amazonaws.com/gapuino_um.pdf (cit. on p. 19).

[34] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. «The cifar-10 dataset». In: *online: http://www. cs. toronto. edu/kriz/cifar. html* 55 (2014) (cit. on p. 29).

[35] Alessio Burrello, Francesco Conti, Angelo Garofalo, Davide Rossi, and Luca Benini. «Work-in-progress: DORY: lightweight memory hierarchy management for deep NN inference on IoT endnodes». In: *2019 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ ISSS)*. IEEE. 2019, pp. 1–2 (cit. on p. 33).

[36] *PyTorch*. URL: `https://pytorch.org/` (cit. on p. 33).

[37] Google AI. *Google OR-Tools*. 2015. URL: `https://developers.google.com/optimization` (cit. on pp. 33, 46).

[38] Google AI. *Original CP Solver*. 2015. URL: `https://developers.google.com/optimization/cp/original_cp_solver` (cit. on p. 46).