Univerza v Mariboru

Fakulteta za elektrotehniko,
računalništvo in informatiko

Mario Gavran

ARM-based Video Intercom System with Next-Gen Human Presence Detection using Deep Learning

Master's thesis

Maribor, July 2022

Univerza v Mariboru

Fakulteta za elektrotehniko,
računalništvo in informatiko

Mario Gavran

ARM-based Video Intercom System with Next-Gen Human Presence Detection using Deep Learning

Master's thesis

Maribor, July 2022

ARM-based Video Intercom System with Next-Gen Human Presence Detection using Deep Learning

Master's thesis

Student:	Mario Gavran
Study program:	študijski program 2. stopnje Elektrotehnika
Program module:	Elektronika
Mentor:	izr. prof. dr. Matej Rojc

Acknowledgments

I would like to thank my family for their support and patience, and I would also like to thank assoc. prof. dr. Matej Rojc for his help and guidance.

Video domofonski sistem na ARM platformi z naprednim zaznavanjem prisotnosti ljudi z uporabo globokega učenja

Ključne besede: TensorFlow Lite Micro, Video domofonski sistem, ARM Cortex-M mikrokrmilnik, Zaznavanje prisotnosti ljudi, nevronske mreže

UDK: 000.000.0:000.0(000.0)

Povzetek

V magistrski nalogi predstavimo razvoj naprednega video sistema z zaznavanjem prisotnosti ljudi, ki temelji na uporabi globokega učenja in mikrokrmilnika z nizko porabo. Cilj naloge je razviti sistem, ki lahko deluje kot pameten video domofon. Sistem se lahko namesti na vhodna vrata in zna samodejno obvestiti lastnika o osebi pred vrati. Glavni cilj magistrskega dela je tako uporaba AI, in zlasti agoritmov z globokimi nevronskimi mrežami, ki se morajo izvajati tudi na končnih napravah z ARM mikrokrmilnikom. Pri tem je velika prednosti predvsem manjša poraba energije in pa cena.

Naloga naslavlja najpogosteje uporabljene metode za zmanjšanje porabe energije in pomnilnika, ter metode za učinkovito izvajanje in optimizacijo algoritmov globokega učenja. Podrobneje predstavljamo algoritem "Deep Compression", kvantizacijo in ternarno kvantizacijo parametrov, rezanje uteži (ang. pruning), souporabo uteži (ang. weight sharing), uporabo Fourierjevih transformacij itd. Predstavljamo tudi najbolj uporabljane strojne platforme za globoko učenje v okoljih z omejenim pomnilnikom in porabo energije. Podrobneje predstavimo nov in inovativen strojni pospeše-

valnik imenovan EIE (ang. Efficent Inference Engine), ki deluje neposredno na nevronskih mrežah stisnjenih z algoritmom "Deep Compression". Predstavljamo tudi bolj splošne platforme, ki temeljijo na ARM Cortex-M procesorskem jedru, namreč Cortex-M4 in -M7, ter novo -M55 jedro z vektorsko razširivijo Helium, in Ethos-U55 jedro, ki je zasnovano posebej za pospeševanje strojnega učenja za naprave na robu (angl. edge).

Magistrska naloga predstavlja tudi razvojni proces in funkcionalno zasnovo predlaganega sistema, ki temelji na ARM Cortex-M4 mikrokrmilniku. V sistemu je tako CMOS kamera z VGA ločljivostjo in 3,97" in LCD zaslon z WVGA ločljivostjo za prikaz zajetega videa. ARM mikrokrmilnik poleg izvajanja AI algoritma za prepoznavo obraza, krmili tudi kamero in zaslon. Periferni vmesnik mikrokrmilnika, ki se uporablja za povezovanje zunanjih pomnilnikov, je uporabljen tudi za povezavo z LCD zaslonom. Periferni vmesnik FSMC (ang. flexible static memory controller), deluje s signali, ki se nekoliko razlikujejo od signalov za vodilo i80, ki ga uporablja LCD zaslon. Je pa dovolj prilagodljiv, da ga je mogoče konfigurirati tudi za uporabo z vodilom i80. Ker LCD zaslon zaseda edini paralelni vmesnik, je povezava s kamero izvedena z uporabo GPIO vrat in DMA krmilnika. V nalogi je tako podrobneje predstavljena tudi povezava s kamero in LCD zaslonom na nivoju programske opreme.

Zaradi raznolikosti in razdrobljenosti med proizvajalci strojne opreme na področju vgrajenih sistemov, ter pomanjkanja standardov in interoperabilnosti, so do nedavnega inženirji razvijali lastne ekosisteme za strojno učenje. Takšne rešitve so običajno brez podpore, so slabo prenosljive in so pogosto optimizirane na specifično platformo. Tensorflow Lite Micro je C++ odprto kodno ogrodje za poganjanje modelov globokega učenja na vgrajenih platformah. To ogrodje odpravlja kar nekaj omenjenih problemov kot so razdrobljenost, raznolikost in izpolnjevanje zahtev glede učinkovitosti, ki so potrebne za vgrajene sisteme, ter tako omogoča združljivost tudi s platformami, ki je bila prej praktično nemogoča. Vsi problemi in rešitve, ter delovanje in struktura ogrodja so podrobneje predstavljeni tudi v nalogi. Razen ogrodja, ki je razvito v programskem jeziku C++, je preostala programska oprema napisana v programskem jeziku C.

Rezultati magistrske naloge v celoti izpolnjujejo pričakovanja in cilje naloge.

Uporabili smo predhodno naučen model globoke nevronske mreže za zaznavanje prisotnosti ljudi na podatkovni množici slik, ki je tudi po testiranju optimizirane različice, podajal pričakovane rezultate tako glede natančnosti kot tudi zakasnitev. Poraba pomnilnika in struktura končnega programa je v nalogi prav tako podrobneje predstavljena.

Pri razvoju sistema so bile tudi težave, ki so od nas zahtevale nove rešitve. Uporaba AI v vgrajenih sistemih se je vsekakor pokazala kot zelo zanimiva saj odpira ogromno možnosti, zato bo prihodnje delo usmerjeno v nadaljne raziskave in razvoj še učinkovitejših modelov. Pri tem učinkovitost modela v veliki meri zavisi od aplikacije in tudi strojne platforme. Nove razpoložljive arhitekture strojne opreme so zasnovane posebej za boljše obvladovanje specifičnih zahtev, ki jih nalagajo nevronske mreže in jih je potrebno upoštevati pri prihodnjem delu.

ARM-based Video Intercom System with Next-Gen Human Presence Detection using Deep Learning

Keywords: TensorFlow Lite Micro, Video intercom system, ARM Cortex-M microcontroller, Human presence detection, Neural network

UDC: 000.000.0:000.0(000.0)

Abstract

This master's thesis presents an advanced video system with human presence detection based on deep learning and an ARM microcontroller. The objective of the thesis is to develop a system that works as a smart video intercom, which could be installed, e.g. on the entrance door, and autonomously alert the owner that a guest is in front of the door. The main goal is to use an AI algorithm, namely the neural network model on a constrained device, such as an ARM microcontroller, as their main advantage is lower power consumption and cost.

The thesis also describes commonly used methods to reduce the power and memory footprint and to implement and accelerate the deep learning algorithms more effectively. Further, the most notable deep learning hardware and some general platforms are described in more detail. The thesis also presents the development of a human presence detection system based on an ARM microcontroller, VGA camera, and LCD, where TensorFlow Lite Micro, an open-source C++ framework for deploying deep learning models to embedded platforms and a pre-trained neural network model for person presence detection are used.

Univerza v Mariboru

Fakulteta za elektrotehniko,
računalništvo in informatiko
Koroška cesta 46
2000 Maribor, Slovenija

FERI

IZJAVA O AVTORSTVU ZAKLJUČNEGA DELA

Ime in priimek študent-a/-ke: Mario Gavran
Študijski program: Elektrotehnika
Naslov zaključnega dela: ARM-based Video Intercom System with Next-Gen Human Presence Detection using Deep Learning (Video domofonski sistem na ARM platformi z naprednim zaznavanjem prisotnosti ljudi z uporabo globokega učenja)

Mentor: izr. prof. dr. Matej Rojc
Somentor: _____

Podpisan-i/-a študent/-ka Mario Gavran

- izjavljam, da je zaključno delo rezultat mojega samostojnega dela, ki sem ga izdelal/-a ob pomoči mentor-ja/-ice oz. somentor-ja/-ice;
- izjavljam, da sem pridobil/-a vsa potrebna soglasja za uporabo podatkov in avtorskih del v zaključnem delu in jih v zaključnem delu jasno in ustrezno označil/-a;
- na Univerzo v Mariboru neodplačno, neizključno, prostorsko in časovno neomejeno prenašam pravico shranitve avtorskega dela v elektronski obliki, pravico reproduciranja ter pravico ponuditi zaključno delo javnosti na svetovnem spletu preko DKUM; sem seznanjen/-a, da bodo dela deponirana/objavljena v DKUM dostopna široki javnosti pod pogoji licence Creative Commons BY-NC-ND, kar vključuje tudi avtomatizirano indeksiranje preko spleta in obdelavo besedil za potrebe tekstovnega in podatkovnega rudarjenja in ekstrakcije znanja iz vsebin; uporabnikom se dovoli reproduciranje brez predelave avtorskega dela, distribuiranje, dajanje v najem in priobčitev javnosti samega izvirnega avtorskega dela, in sicer pod pogojem, da navedejo avtorja in da ne gre za komercialno uporabo;
- dovoljujem objavo svojih osebnih podatkov, ki so navedeni v zaključnem delu in tej izjavi, skupaj z objavo zaključnega dela.

Uveljavljam permisivnejšo obliko licence Creative Commons: _____ (navedite obliko)

Kraj in datum: Maribor, 19.06.2022

Podpis študent-a/-ke:

Gavran

Table of Contents

List of Figures

List of Tables

List of abbreviations

AI	Artificial intelligence
NN	Neural network
ML	Machine learning
CNN	Convolutional neural network
MCU	Microcontroller
FPU	Floating-point unit
SGD	Stochastic gradient descent
CSC	Compressed-sparse-column
DFT	Discrete Fourier transform
FFT	Fast Fourier transform
EIE	Efficient inference engine
PE	Processing element
LNDN	Leading non-zero detection node
CCU	Central control unit
MAC	Multiply-accumulate
ISA	Instruction-set architecture
SIMD	Single-instruction-multiple-data
DSP	Digital signal processor
MVE	M-profile vector extension
DMA	Direct memory access
TFLM	TensorFlow Lite for Microcontrollers
TFL	TensorFlow Lite
VWW	Visual-wake-word
HWD	Hot-word detection
FSMC	Flexible-static-memory-controller
SCCB	Serial-camera-control-bus
EXTI	External interrupt controller

1 INTRODUCTION

The foundation of edge computing was formed at the turn of the century when content delivery networks evolved from serving web and video content from the edge servers to serving hosted applications from the same servers using the same infrastructure. Today, as the field of artificial intelligence (AI) expands and the technology evolves, researchers and engineers begin to realize that many problems of the future will be most effectively addressed by a distributed computing paradigm and AI. This approach brings computation and storage capabilities to the very source of the data and to the location where the result of an AI computation is needed.[8] The existing infrastructure of mobile phones, cameras, home-control systems and other edge devices is beginning to be reused for this purpose. This approach saves a lot of bandwidth and energy, especially in the case of inferencing image or video data, where neural network (NN) models have very large number of input layer nodes. This image data needs to be sent to the server application to be processed, and the result sent back to the origin of data or the user. If an AI application is used to detect human presence in some video content, the video must be sent constantly to the server for inferencing, while the result of virtually every frame of the video must be sent back to the user. This raises questions about privacy and data security, as well as power and bandwidth consumption.[3] Although edge AI computing can reduce concerns about privacy and data security, the distributed paradigm can increase them because edge devices would likely interact with each other and share data. In this case, the data are stored in these micro-data centres, which can be more vulnerable to different sorts of cyber-attacks, illegal and corrupt exploitation and ignorant abuse.

The distributed computing paradigm can be used partially as well, e.g. devices like Amazon Alexa and Google Home. These devices perform local keyword detection to initiate communication to the server for further processing, thus saving a lot of bandwidth and power. As the field of AI progresses, applications like keyword detection or image recognition are not even considered AI in present times, but routine technology.

It is widely known today, that AI applications require a lot of computing power and energy. The AlphaGo computer, for example, used almost 2000 CPUs and 300 GPUs, resulting in a $3000 cost per game.[10] Additionally, in this pursuit of achieving better accuracy, capability and performance, the deep learning models are getting larger and more demanding. For example, the winner of the ImageNet recognition challenge for image recognition in 2015 was 16 times larger than the one from 2012, and the winner for speech recognition required ten times more training operations than the one from 2014.[10] These facts encourage investment in developing efficient methods for downsizing tremendous memory usage demands, increasing the efficiency and computational cost of the inference process and bringing machine learning (ML) even to the smallest and most efficient hardware devices. Some market research reports predict that the global edge AI software market size is projected to reach hundreds of millions of USD by the quarter of this century. This rapid growth is attributed mostly to the emergence of the 5G network. Many reports also predict that the video surveillance segment would have the largest market size along with autonomous vehicles, access management and predictive maintenance segments.[15]

This thesis presents concepts and a broad overview of the edge AI and ML. Some useful and popular methods, available hardware and software infrastructure for enabling the use of these technologies in the constrained environment of the embedded systems, are also presented. Specific development of human presence detection is then described and tested.

2 EDGE ARTIFICIAL INTELLIGENCE

Edge AI is a popular name for the computer science research field that emerged at the intersection of embedded systems development and AI. Namely, the resource-efficient ML delivers an entirely new branch of computer science and AI to engineers and researchers, even though it is still in its early development stages. Despite its immaturity, it is already being applied to many key applications like keyword and acoustic anomaly detection, motion gesture recognition, visual object classification, etc. It is important to understand that the so-called TinyML research field is not intended to be used in more complex applications and with more deep NN models, where these modern AI models are used in general computing platforms based on x86 and/or GPU. Namely, TinyML has its applications and advantages over general AI computing in its lightness, which means it can be powered even by a single coin cell battery for months and years[2], or even be powered by vibrations or heat.[19] Therefore, wherever the power supply is inconvenient, like in factories with thousands of sensors in harsh environments, remote rural and o-grid areas or seas, applications with sensors sensitive to EM noise or medical equipment and implants, the TinyML solution is perfect to deal with those constraints. Further, applications like the full garbage can detection, recognizing and counting birds by their bird call, other animal tracking applications, soil moisture reporting on farms, sea temperatures, and others, are just a fraction of more intelligent IoT solutions and use cases.[19] Further, some other advantages are lower latency and lower-to-none bandwidth usage compared to traditional cloud computing solutions. This is very important because some applications, e.g. augmented reality glasses or self-driving cars cannot afford much latency or unreliable network connection (critical IoT). In short, edge AI is aiming to do as much processing as possible, at lower latency, at lower/no bandwidth usage, with very low power consumption, and to guarantee more reliable real-time operation.

ML can be improved in many different ways. In essence, the optimization problem and better performance, power consumption and latency can all be observed from four basic aspects. These are roughly divided into categories as shown in figure

2.1. The inference and training of deep models can be improved from a hardware and/or software standpoint. Nevertheless, in this thesis, the focus is mostly on the inference side of the edge; however, it is important to at least present the broader picture, namely for a better understanding of the core problem.

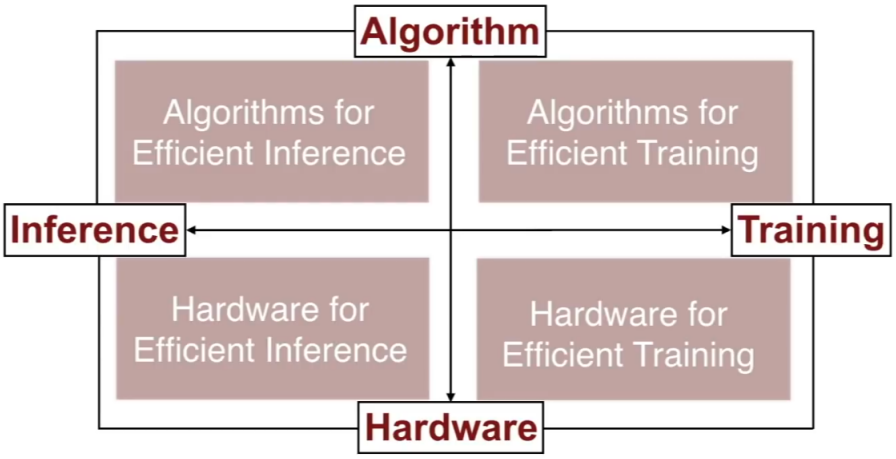


Figure 2.1: Algorithm-Hardware and Inference-Training compass [10]

In the ML industry, there are four basic demand categories. The first of them is the one that doesn't need to be low cost and needs high power computing, which is used for training large ML systems required for research and exploration of what can be done with ML. The second type of demand is for training already-designed and used systems and models with new data or adding new features or labels and objects that, for example, need to be recognized in existing image recognition systems. This consumes less power, uses lower precision arithmetic, and is cheaper. The third category is running ML on powerful servers in data centres used, e.g. newsfeeds on news and social media sites, filtering search results, etc. In this case, power consumption and latency are the main concerns because the trend shows there is a fast-growing number of services using this technology and a growing number of users using them. The last category is embedded devices, cars, phones, smart cameras, etc. All these edge devices have less power available, have a smaller memory footprint, and usually have some sort of accelerated arithmetic.

2.1 Efficient methods and models' size reduction on the edge

Ultra-low-power ML requires vast optimization for efficient run-time performance, the size of the model, and a significant reduction of needed inference engine operations. Most of the ML algorithms, especially convolutional NN (CNN), are based on matrix multiplications, matrix additions, and convolution computations. These operations, on the other hand, have large memory and power consumption. State-of-the-art DL models themselves can be huge and consume a lot of memory. Further, arithmetic operations within these models in the inference process are very demanding regarding processing power, energy, and memory. Therefore, edge optimization aims to build a more compact model, reduced in size as much as possible, and to develop an inference engine that will fit into, often very memory-constrained, embedded platforms on the edge. Also, the aim is to utilize the efficient and fast hardware of these processors to allow the use of high performance and high throughput vectorized and parallelized operations. In the following sections, several methods for DL models' optimization and compression are described.

2.1.1 Quantization

In general-computing platforms, real numbers usually use floating-point representation with separate exponent and mantissa information. It allows representing values with a high dynamic range and at the same time with high accuracy for values near zero. Single-precision or 32-bit floating-point numbers are proved to be more than enough for training and inference. The double-precision representation uses additional memory resources, while half-precision representation can further improve the performance without significant loss in accuracy. However, low-power microcontrollers (MCU) usually support only single-precision floating-point units (FPU), meaning that half-precision numbers will be promoted to single-precision, thus losing the benefits of improved performance of executing an operation on two half-precision values in the same number of clock cycles. Very low-power MCUs mostly don't even have a floating-point unit (FPU). Additionally, single-precision floating-point representation has a dynamic range far greater than what is typical weights' values in NNs; therefore, the fixed-point and integer representations seem

to be much more suitable.[20]

Quantization is a process or method for preserving high or best possible accuracy, but with lower precision and a much smaller memory footprint, which allows performing inference computations at acceptable latency and using values represented with low-bit widths. Low precision refers to the numerical representation of the weights, biases and other NN parameters. High accuracy refers to the result of the inference computation.

When NNs are deployed on some platform, all parameters are usually represented as floating-point values. Nevertheless, when these parameters are represented with low bit-width fixed-point or integer representation, the accuracy does not suffer much, can sometimes even be improved, and can even help avoid overfitting during retraining.[16] Figure 2.2 shows the range and accuracy for different number representations.

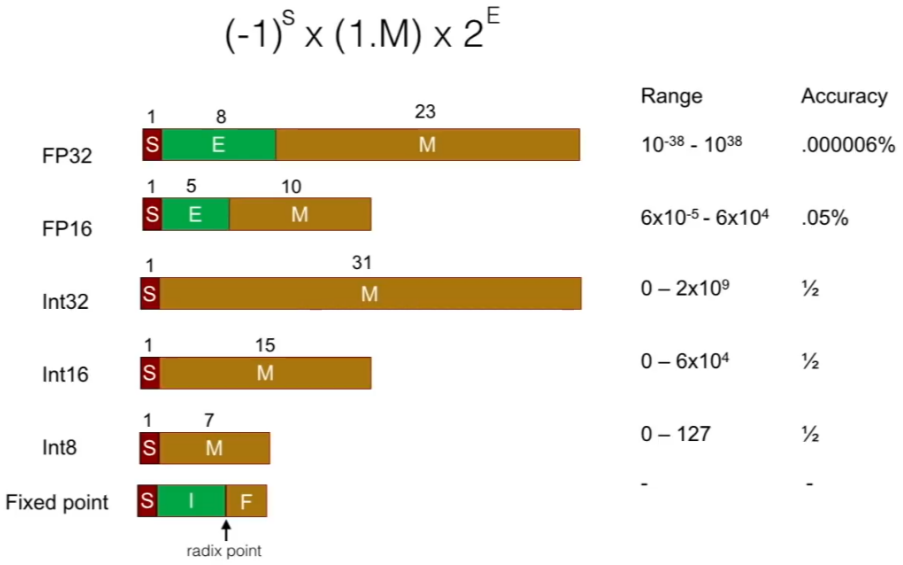


Figure 2.2: Range and accuracy of common number representations [10]

NNs deployed on efficient embedded platforms can still be trained on more powerful general-computing platforms using floating-point representation. Since the floating-point arithmetic on modern desktop CPU is as fast as integer arithmetic and GPUs are optimized toward single-precision floating-point calculations. Nevertheless, NNs' parameters can then be quantized and retrained to optimally reduce the model size and computational requirements and maintain the original accuracy.

The reason for this is because embedded edge processors mostly don't have efficient support for floating-point computations, while the computations with integer numbers offer higher throughput, and are thus faster and more energy-efficient. Other reasons are that smaller word sizes reduce memory bandwidth pressure and thus improve the performance for bandwidth-limited computations and lower memory size requirements, which can improve cache utilization, as well as other aspects of the memory system operation. Figure 2.3 is an example of a 45nm technology chip and shows how much the number representation, e.g. affects energy consumption and silicon area. It shows that the difference in energy consumption between 16-bit floating-point multiplication and 8-bit integer multiplication is around five times less in favour of integer.

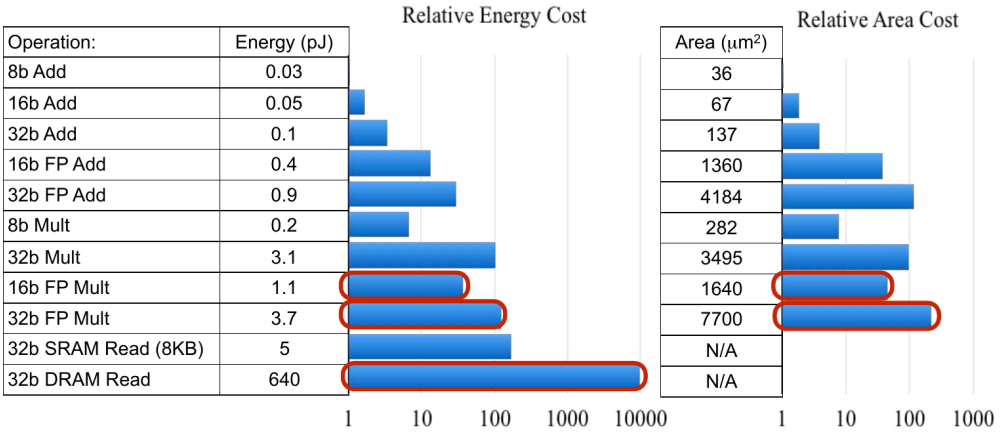


Figure 2.3: 45nm tech node relative energy and area costs for different number representations [16]

Many different quantization methods differ concerning different aspects. Some of them are unified and non-unified, fixed precision and mixed-precision, layer-wise, channel-wise and kernel-wise, floating-to-fixed-point and floating-point-to-integer. Almost any combination of training and inferencing with different number representations is successful, but it depends on the specific application and targeted device. E.g., a non-uniform quantization scheme with a modified fixed-point format was implemented in [26] on an FPGA device, which showed improvement in power consumption, but also noticeable degradation of accuracy. In this thesis, the focus is on uniform affine and scale integer quantization methods, because they are most suit-

able for matrix and convolution computations on an efficient embedded processor.

Affine quantization, described in figure 2.4, maps real values $x \in \mathbb{R}$, with its range defined in (2.1), to a b-bit signed integer values $x_q \in \mathbb{Z}$, with its range defined in (2.2). The real values that lie outside the range of x_q are clipped to the nearest bound.

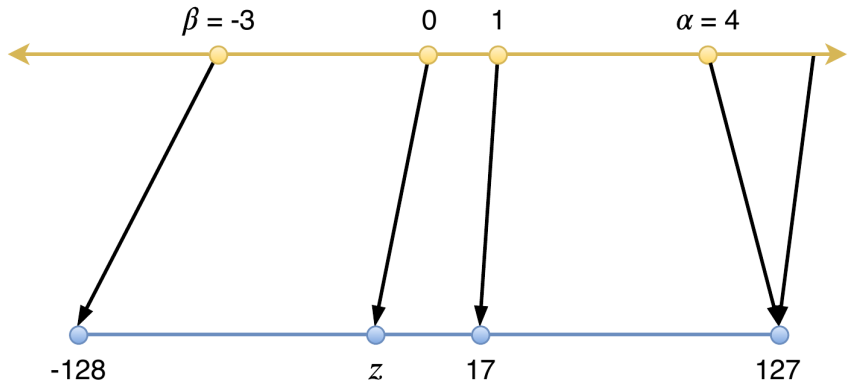


Figure 2.4: Affine quantization [27]

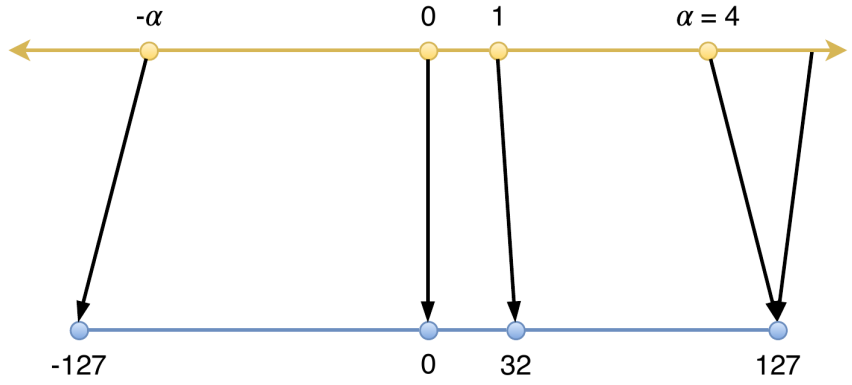


Figure 2.5: Scale quantization [27]

$$x \in [\beta, \alpha] \tag{2.1}$$

$$x_q \in [-2^{b-1}, 2^{b-1} - 1] \tag{2.2}$$

The quantization and de-quantization equations are defined in (2.3) and (2.4), respectively, where s is the scale factor, and z is the zero-point, defined in (2.5) and

(2.6) respectively. The *round*() function rounds the real value to the closest integer value, and b is the number of bits chosen for the quantized value.

$$x_q = round\,(s \cdot x + z) \tag{2.3}$$

$$x = \frac{1}{s}\,(x_q - z) \tag{2.4}$$

The scale factor s is defined as the ratio between the integer-representable and the real value ranges, and zero-point z is defined as the quantized value to which the real zero value is mapped or simply stated, the zero-crossing point in a real value space.

$$s = \frac{2^b - 1}{\alpha - \beta} \tag{2.5}$$

$$z = -round\,(s \cdot \beta) - 2^{b-1} \tag{2.6}$$

Zero-point must be represented with no error in the quantized space because zero padding is commonly used in many CNNs. If it isn't represented exactly as 0, it will result in accuracy errors. Thus z is rounded in (2.6) to the nearest integer value, while scale factor s can have a real value.

2.1.2 Pruning, weight sharing and Huffman coding

Usually, when the optimization methods are utilized, the method itself is only one part of the optimization pipeline. It is often combined with other optimization and acceleration methods, and also with methods for efficient data storage. Such a solution can be found in Deep Compression algorithm shown in figure 2.6.[12]

In this algorithm, the pruning method is used for size reduction, as well as to speed up the inference process. Next, trained quantization or weight sharing coding is combined with it to further reduce the size of the NN model stored in the memory, and improve efficiency and performance. Additionally, the Huffman coding to reduce the memory footprint is also used.

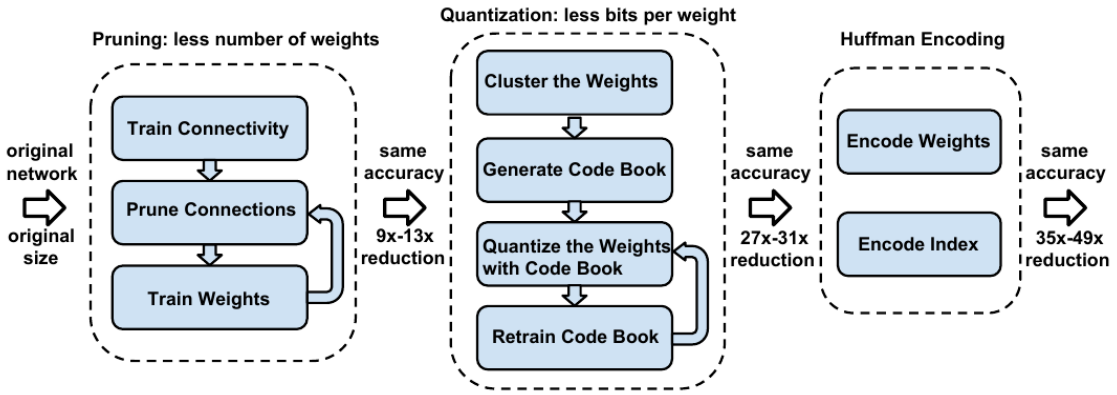


Figure 2.6: Deep Compression pipeline [12]

Pruning was inspired by the human brain where the number of connections grows from 50 to 1000 trillion in the first year of life and then reduces to cca. 500 trillion by the age of 10. The method was explored in the early 90s by Yann LeCun who states that the growing number of parameters leads to overfitting problems and bad generalization performance.[17] The technique is called Optimal Brain Damage, and it solves these problems by selectively deleting half or more of the weights in a NN model without losing its original performance. The idea was revisited by Song Han in 2015 who pruned state-of-the-art CNN models like AlexNet and VGGNet, with no loss in accuracy.[12] The general idea for pruning is shown in figure 2.7, where the original dense network on the left is pruned by removing redundant small-weight connections and producing the sparse network on the right, keeping only the most informative connections.

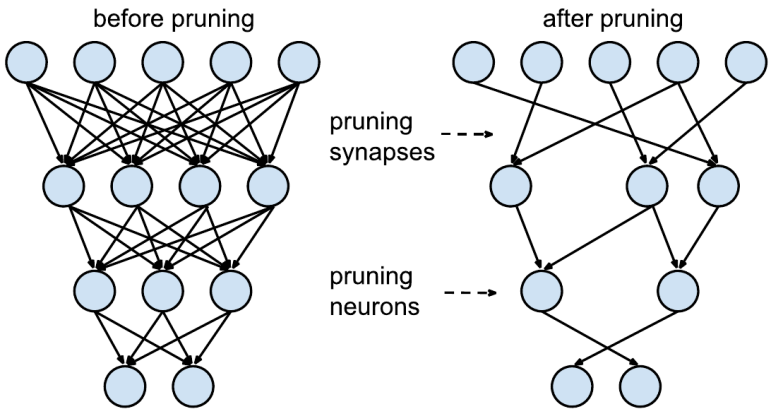


Figure 2.7: Pruning synapses and neurons [9]

The pruning procedure is the first step in the Deep Compression pipeline, presented in figure 2.6. After learning the connectivity with the usual network training, all small-weight connections below a certain threshold are removed. Then the network is retrained to adjust the remaining weights. In figure 2.8, it can be seen that small weights make up the majority in the distribution of weights and that after retraining, the quantity of weights is much smaller. The process is repeated iteratively, and the result is a sparse NN model shown on the right side in figure 2.7.

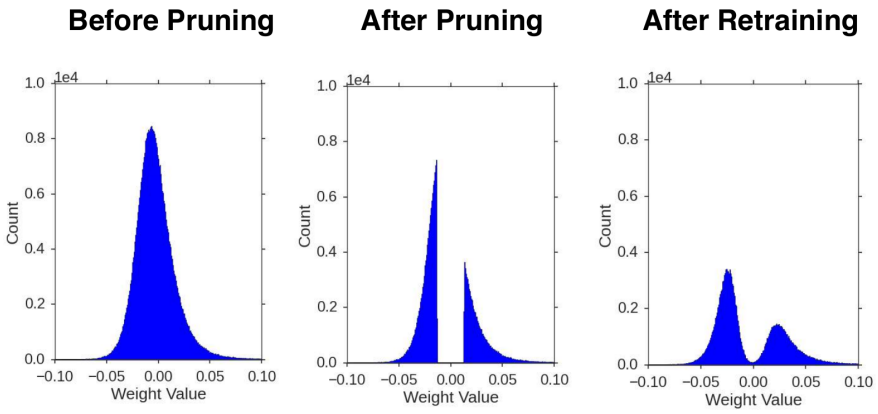


Figure 2.8: Distribution of weights before and after pruning and after retraining [10]

E.g., in the AlexNet model, the size of convolutional layers can be reduced 3 times, while fully connected layers by 10 times. This results in more than 90% reduction in size, or from 60 million to only about 6 million weights, with no loss in accuracy as is shown in figure 2.9.[10] [12]

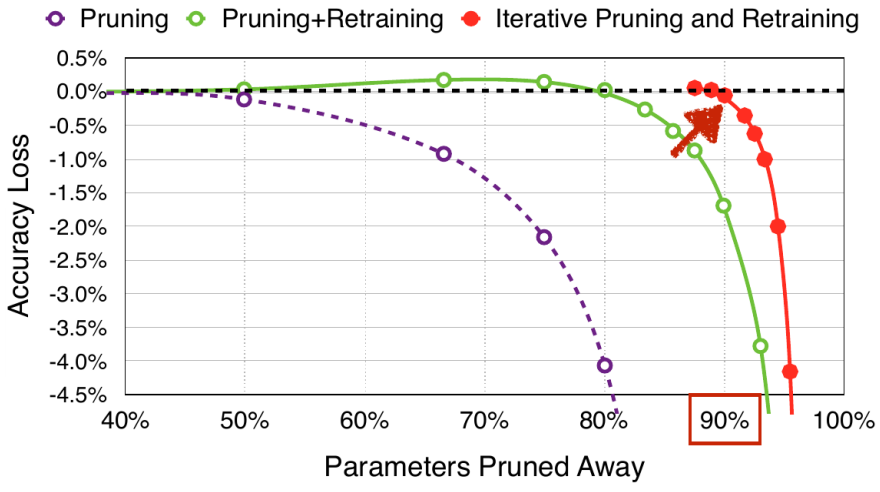


Figure 2.9: Accuracy of the model after pruning [10]

With only one cycle of pruning and retraining, the accuracy can be fully restored at about 80% of the parameters pruned away. When the process is iteratively repeated, even more than a 90% reduction in model size can be achieved. Nevertheless, if the pruning is applied too aggressively, this can lead to overfitting; therefore, there is also a limit we have to take into consideration, and it much depends on the model itself.[10] [12]

2.1.3 Weight sharing or trained quantization

Weight sharing is presented in figure 2.10. This process represents the second stage in the Deep Compression pipeline, as shown in the middle in figure 2.6, and is also used to further compress the pruned network by reducing the number of stored weights. The general idea is that after removing the weights with pruning, the remaining weights with similar values can be clustered and represented with a single value. This way, the number of weights is reduced and also the number of bits required for storing the weights is reduced heavily. To better understand this process, the representation in figure 2.10 is shown.

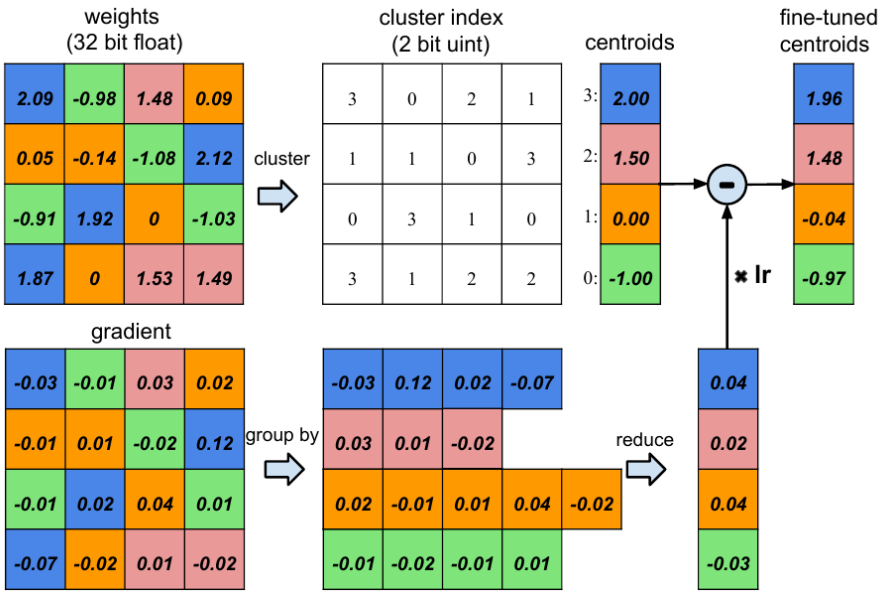


Figure 2.10: Weight sharing and SGD training [12]

Let's suppose that the DL model has 4 inputs, 4 outputs and a 4×4 weight matrix W with 16 weights. The matrix W is generally described with (2.7). The

weights are clustered into k clusters (in this case 4 clusters), and all the weights in a cluster will share the same value, named the centroid. Therefore, in order to represent each weight in a matrix, only a small index needs to be stored, in this case, a 2-bit index. The weight is then stored next to its index in the centroid vector C represented in (2.8), while the new matrix of weights that needs to be stored in the memory only stores the indices. E.g., blue colour values in the weight matrix are represented with 3, which is the index in the centroid vector; therefore, all those values in the weight matrix are replaced with only a 2-bit index.

$$W = \{w_1, w_2, \ldots, w_n\} \tag{2.7}$$

$$C = \{c_1, c_2, \ldots, c_k\} \tag{2.8}$$

$$r = \frac{n \cdot b_w}{n \cdot \log_2(k) + k \cdot b_c} \tag{2.9}$$

Only $\log_2(k)$ bits are needed to encode the indices for k clusters. Thus, in this example, we can achieve a 16 times reduction in memory consumption per stored weight, when the original weight is 32 bits long. The compression rate is calculated as proposed in (2.9), where b is the bit width of the original weights, k is the number of clusters, and n is the number of original weights. In this case, the compression rate of 3.2 is calculated, when 16 different original weights are represented with 32 bits. A similar approach can be used on the gradient matrix during the stochastic gradient descent algorithm (SGD). The difference, in this case, is that we are not clustering the gradient values by their value but by their position in the matrix since they only update the values stored in those matrix positions. Namely, the gradient values in the same cluster are summed together, then multiplied by the learning rate, and at the end, subtracted from the centroids. The weight distribution is shown in figure 2.11, where it can be seen that the weights are discrete and non-linearly distributed.

Same as the NN is retrained after the pruning process, the retraining can also be done after weight sharing, and as is shown in figure 2.6, it can be done iteratively. After retraining, only the centroids are adjusted, and because there are much fewer centroids than the actual weights, the process is much faster.

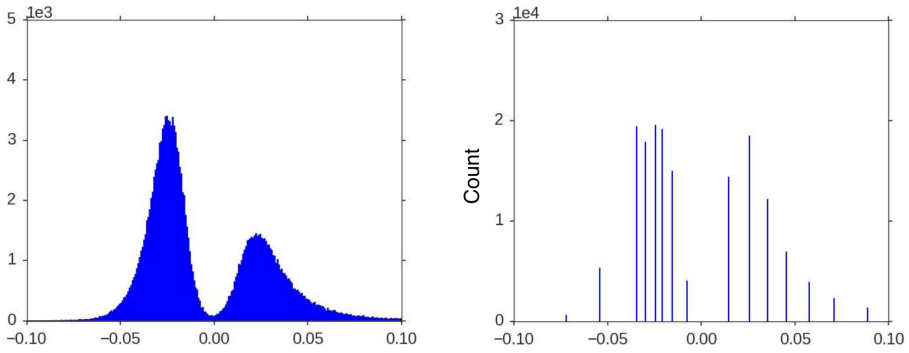


Figure 2.11: Weight distribution before and after trained quantization [12]

Deep Compression uses k-means clustering to cluster the weights in the model, but other methods are also viable. Namely, centroid values are identified for each layer and are not shared between the DL model's layers. Further, centroid initialization for k-means clustering affects the DL accuracy and can also be performed with other methods, such as random, density-based and linear operations, as proposed in [12]. Additionally, the authors report that in AlexNet, only 8 bits per convolution layer and only 5 bits per fully connected layer are already enough to retain the same accuracy as in the original network. Further, using only 4 bits for convolutional and 2 bits for fully-connected layers results in only a 2% loss of accuracy.

After pruning and weight sharing, the last step in the optimization pipeline is the Huffman coding, as presented in figure 2.6. It is used for both the shared weights and their indices. It is a well-known prefix code used for lossless data compression that uses variable-length code words to encode the input values. The more frequent indices are coded with a smaller number of bits, while less frequent indices with more bits. In [12], it is stated that results show that 20-30% of the network storage can be saved this way.

After pruning and weight sharing, the remaining data is very sparse. To further compress data and exploit sparsity, a variation of the compressed sparse column (CSC) format is used.[11] This will be discussed further in section 2.2.1, where custom hardware for working directly on NNs compressed with the Deep Compression approach is described.

2.1.4 Ternary and binary quantization

More aggressive quantization than described in section 2.1.1 can be performed by using only three values for the weights in a neural network. The weights in a ternary network are usually described with (2.10), where H is the absolute value of the mean weight value, or simply -1 or 1. However, in the Trained Ternary Quantization algorithm, for example, as proposed in [28], two full-precision parameters W_l^p and W_l^n, whose values are trained and their absolute values are not equal, are used. Firstly, the full-precision weights are normalized into the range $[-1, +1]$, and then the intermediate weights are quantized into $\{-1, 0, +1\}$ by using a threshold factor that is the same across layers. Finally, backpropagation is performed separately on the full-resolution weights in order to learn their ternary assignments and then on the coefficients W_l^p and W_l^n to learn the final ternary values. In other words, during the SGD, values for the ternary weights are learned, as well as which original weight has to be assigned to which specific ternary weight. Namely, during training, ternary and full-resolution weights are used, while during inference time, only the ternary values are used. Figure 2.12 shows how the ternary weight values and their distribution might change during training.[11]

Other ternary NNs can use many different combinations of methods for determining and training the weights. Ternary Weight Network uses two symmetric thresholds and zero values. TernaryNet replaces floating-point values with only $\{-1, 0, +1\}$ values, thus introducing even more efficient ways for multiplication and addition operations by exploiting the two's complement operation

$$W_t = \begin{cases} +H & \text{if } W > 0 \\ -H & \text{if } W < 0 \\ 0 & \text{else} \end{cases} \tag{2.10}$$

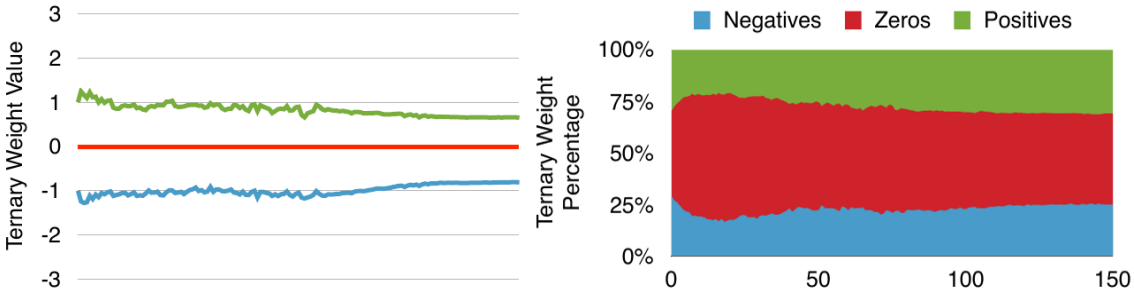


Figure 2.12: Ternary weight values and distribution during training [28]

Figure 2.13 presents an example of CNN kernels before and after ternarisation where it can be seen that for some applications, only three different weights might be enough for detecting corners, edges, and other simple patterns because it leads to only a minor degradation and high computational efficiency.

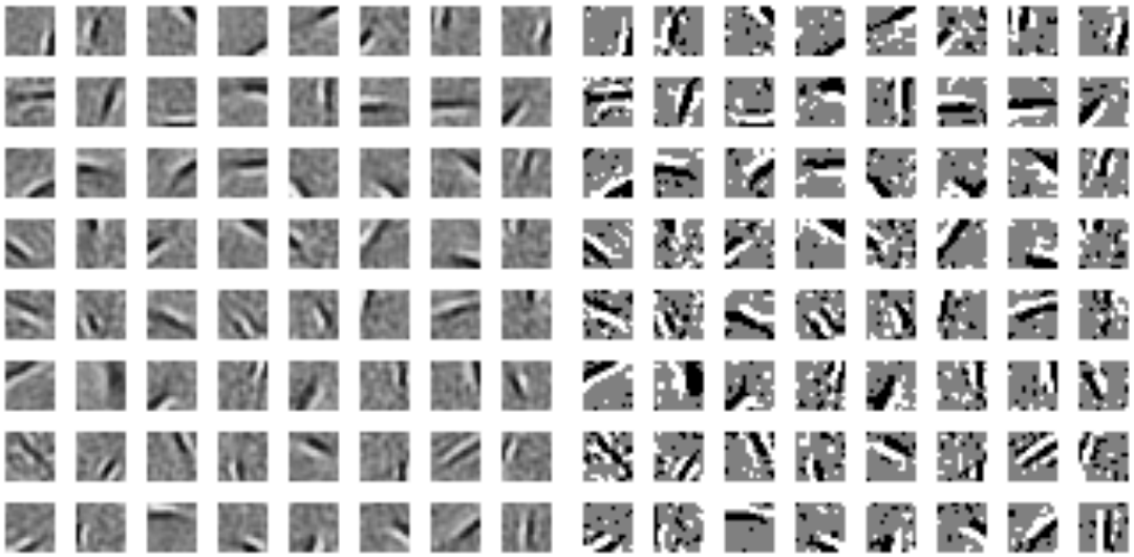


Figure 2.13: Weights before and after ternarisation [13]

Even more aggressive approach than the ternary is the binary quantization with only two different values for weights as proposed in (2.11). Binary Connect architecture from 2015 is one of the first ones that use weights $W \in [-1, 1]$.[6] The network uses full precision numbers for activations. In this case, the multiplication operation is very simple because only the sign bit determines the result, but the addition is still performed with full precision. BinaryNet and XNOR-Net use binary weights and activations. In this case, value 1 is represented as binary 1 and value -1 as binary

0. Therefore, the multiplier can be replaced with an XNOR gate. Only the first and last layers are based on full precision math. Thus, for addition in the last layer, the so-called population count (popcount) block can be implemented in software or hardware.[13]

$$W_b = \begin{cases} +1 & \text{if } W >= 0 \\ -1 & \text{else} \end{cases} \tag{2.11}$$

2.1.5 Fourier transformations

In a CNN model, most of the memory is consumed by the fully connected (FC) layers. Nevertheless, the convolutional layers account for more than 90% of overall computational complexity and thus dominate runtime and energy consumption. As CNNs are the most popular approach to computer vision in ML applications, and because spatial convolution is the most computationally expensive part of the network, it is advisable to explore possible circumventions around computing spatial convolutions and find more efficient ways to implement them.

The convolution operation is usually performed by sliding a filter matrix across the input matrix in order to compute the element-wise multiplication between each element and sum the results. This way, we form one element of the output matrix. Because there are $N \times N$ elements in the input matrix, for each element $N \times N$ multiplications are needed to compute the output matrix. This naive approach has the complexity of computation described in (2.18). For most of the applications in ML, especially for embedded platforms, this number of operations is not appropriate.[23]

Fourier transform can be used to significantly reduce the number of operations needed to compute the convolution between two functions. The convolution theorem states that under suitable conditions, the Fourier transform of a convolution of two functions is equal to the point-wise product of their Fourier transforms, as described (2.12) - (2.14) and figure 2.14. Here $\mathcal{F}$ denotes the Fourier transform, where x is the input, w is the filter, and y is the output function.[18] [23]

$$y(x,y) = x(x,y) * w(x,y) \tag{2.12}$$

$$= \mathcal{F}^{-1}\left(\mathcal{F}(x) \circ \mathcal{F}(w)\right) \tag{2.13}$$

$$= \mathcal{F}^{-1}\left(X(v_x, v_y) \cdot F(v_x, v_y)\right) \tag{2.14}$$

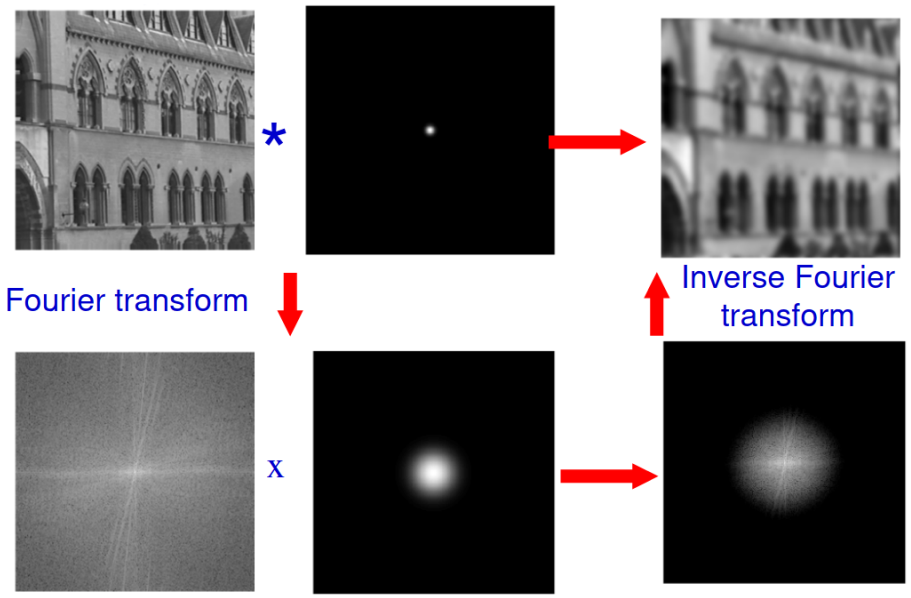


Figure 2.14: Computing 2D convolution in Fourier space [29]

The discrete Fourier transform (DFT) converts a finite sequence of values, e.g., a grey-scale picture, into a same-length sequence of equally-spaced samples of the discrete-time Fourier transform, which is a complex-valued function of frequency. Since it deals with finite sequences of values, it can be implemented by numerical algorithms. This is usually best done by efficient Fast Fourier transform (FFT) algorithms. Namely, the FFT algorithm reduces the number of required computations from N^2 to $N \log_2 N$. The convolution can be computed on discrete signals as described with (2.15) and (2.16). Similarly to the continuous convolution theorem, the convolution between two matrices can be computed efficiently by using FFT as described in (2.17), where $x \in \mathbb{R}^{M \times M}$, $w \in \mathbb{R}^{N \times N}$ and $y \in \mathbb{R}^{E \times F}$ are input, filter and output matrices, respectively. Thus, the important reduction in computational complexity can be achieved in this way. This process is described as: (2.19).

$$y[i][j] = (x * w)[i][j] \tag{2.15}$$

$$= \sum_{k=0}^{i} \sum_{l=0}^{j} x[k][l] \cdot w[i-k][j-l] \tag{2.16}$$

$$= IFFT\left(FFT(x^{(M\times M)}) \circ FFT(w^{(N\times N)})^*\right) \tag{2.17}$$

Another optimization possibility is that the FFT of filers can be pre-computed and stored in memory. Nevertheless, even small-sized filters are still large after the transformation. Because the filters in CNNs are often very small, the benefit of reduced complexity is insignificant, as can be deduced from (2.19). Also, if the input and filter matrices are sparse, after the transformation, the sparsity is lost; therefore, it cannot be exploited.

$$\mathcal{O}(M^2 \cdot N^2) \tag{2.18}$$

$$\mathcal{O}(M^2 \log_2 N) \tag{2.19}$$

Because the result of the DFT algorithm is a matrix with complex value elements, it is not appropriate for many different applications. Complex values take up more memory space and also add additional computational overhead. Besides the regular complex DFT, a commonly used transform is the real discrete cosine transform, called DCT. Since this transformation deals only with the real part of the spectrum, it only results in the amplitude part, while the phase part is ignored. When input values and filter matrices are only integers, then the floating-point arithmetic can be replaced by Number Theoretic Transform, which is a variant of the FFT that uses modulo instead of complex numbers; therefore, calculations are done only with integer numbers. This can significantly increase efficiency and even allow computing on very restricted MCU platforms.

2.2 Efficient hardware for TinyML

In this section, the inference-hardware standpoint shown in figure 2.1 will be presented in more detail. The focus will be specifically on new and innovative, as well

as on more general hardware developed for TinyML tasks. Although the TinyML research field is still largely unexplored, the hardware vendors and researchers are already trying to optimize and upgrade their hardware for these relatively new applications.

2.2.1 Efficient Inference Engine (EIE)

The Efficient Inference Engine (EIE) is the first hardware accelerator that works directly on deeply compressed models; specifically, it works on networks compressed with Deep Compression presented in sections 2.1.2 and 2.1.3. Deep Compression makes the networks much smaller but makes their processing also pretty challenging. Namely, by pruning, their matrices become sparse. Further, weight sharing adds overhead to fetching the weight values from the codebook. Additionally, storing everything in a modified CSC format adds an overhead of decoding the position from relative indices. All this adds extra complexity and contributes to the computational inefficiency of CPUs and GPUs. EIE performs inference directly on those compressed networks and takes advantage of static weight matrix sparsity, dynamic input vector sparsity, weight sharing, relative indexing and narrow weight bit lengths.

The EIE is designed as a scalable array of processing elements (PE). Each PE stores a partition of a network, with a maximum of 131000 weights and performs computation on that partition. Therefore, the EIE is considered to be a distributed storage and distributed computation engine that parallelizes sparse matrix computation. It is able to perform 800 million weight calculations per second.[11]

A matrix-vector multiplication, the basic building block of many NNs, is briefly described with (2.20) and (2.21), where b is the output vector with elements b_i, a and a_j are input vector and input activation, respectively. W and W_{ij} are the weight matrix and matrix element, respectively, and f is the non-linear output function, typically rectified linear unit commonly known as ReLU.

$$b = f(W \cdot a) \tag{2.20}$$

$$b_i = ReLU\left(\sum_{j=0}^{n-1} W_{ij} \cdot a_j\right) \tag{2.21}$$

The deep compression method replaces each weight W_{ij} with a 4-bit index I_j and stores the weights in a shared table S. Thus, the equation (2.21) becomes (2.22). Here I_{ij} is the index of the weight W_{ij}, S is the codebook, X_i is the set of columns j for which weights W_{ij} are non-zero. It represents static sparsity and is fixed for a given model. Y is the set of indices j for which the activations a_j are non-zero values. It represents the dynamic sparsity of the input vector a. Computing with only non-zero elements in W_{ij} and a_j exploits the sparsity but makes the computation dynamically irregular.

$$b_i = ReLU\left(\sum_{j \in X_i \cap Y} S[I_{ij}] \cdot a_j\right) \tag{2.22}$$

To exploit the memory usage sparsity, each column W_j in matrix W is stored in a vector v that contains only the non-zero weights. Vector z encodes the number of zeros before the corresponding weight and it is also stored. This vector has the same length as v. If more than 15 zeros appear before a non-zero element, a zero is added in vector v and the number 15 is added in vector z because the elements are encoded with a 4-bit value.

Both vectors for each column are stored in a pair of two large arrays with an additional column pointer vector p that points to the start of each column. Now, the sparsity is simply exploited by multiplying only the non-zero activations with every element in the corresponding column, except the padding zeros. Further, the matrix W is distributed and computations are parallelized by interleaving the rows of a matrix among multiple PEs, as is shown in figure 2.15. Each colour corresponds to one PE.

If i is the row counter and N is the number of PEs, then $\mathrm{PE_k}$ holds all rows $\mathrm{W_i}$ for which (2.23) holds.

$$i \bmod N = k \tag{2.23}$$

Each PE has its own fraction of W columns in CSC format as described for $\mathrm{PE_0}$ in the bottom portion in figure 2.15. Each PE is also assigned a portion of input vector

a and output vector b. Every new operation starts with scanning vector a to find its first non-zero element. Then this element is broadcasted to all PEs, which then multiplies this value with its portion of nonzero weights, and accumulates the partial sums in the vector b. The interleaved CSC format allows for fast zero detection in both input vector and weights.

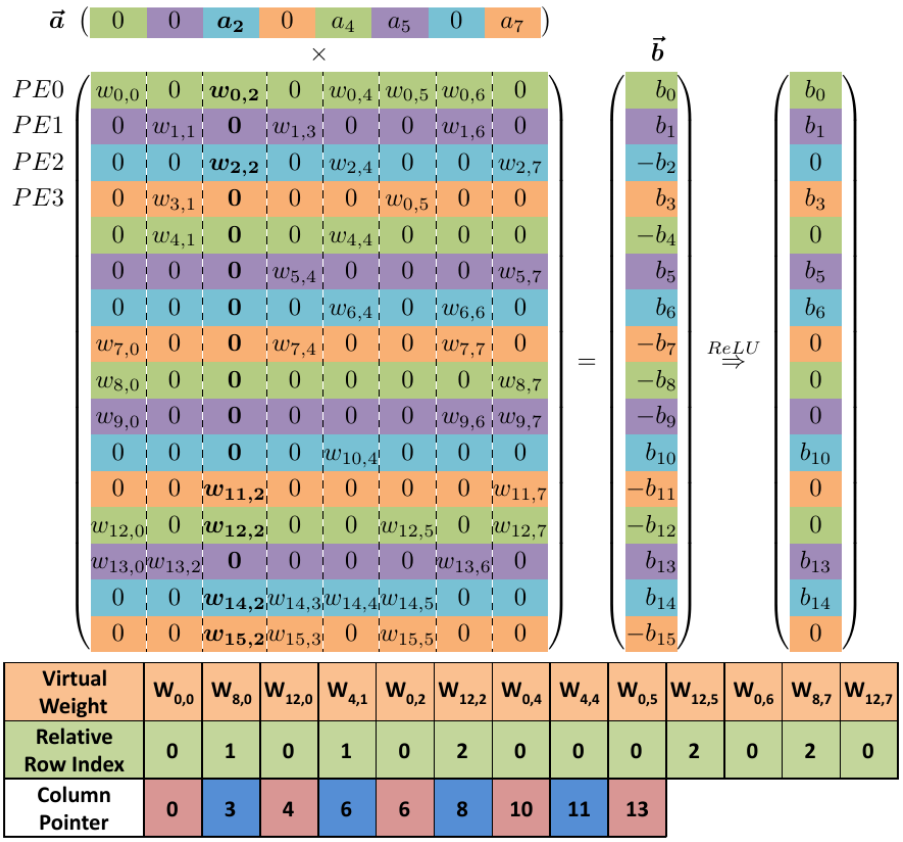


Figure 2.15: Matrix W, input vector a, output vector b and memory layout corresponding to PE_0 [11]

PE architecture and Leading Non-zero Detection Node (LNDN) is shown in figure 2.16. The Central Control Unit (CCU) controls the PE array, receives non-zero input activations from the LNDN network, communicates with the master (e.g. a CPU), and monitors each PE's state. CCU broadcasts the non-zero input element and its index j to each PE. Each PE has its Activation Queue, which can disable a broadcast from CCU when it is full. The queue is used to even out the load imbalance due to the imbalance of non-zero elements in its portion of weights W. Pointer Read Unit is used to look up the start and end pointers for the v vector.

The pointers will always be stored in different SRAM banks in order to be readable during the same clock cycle. Sparse Matrix Read Unit reads non-zero elements in the corresponding partition. SRAM is 64 bit wide to allow eight entries to be fetched on a single read. The Arithmetic Unit (AU) performs a multiply-accumulate (MAC) operation between elements fetched using vector v and the head of the queue.

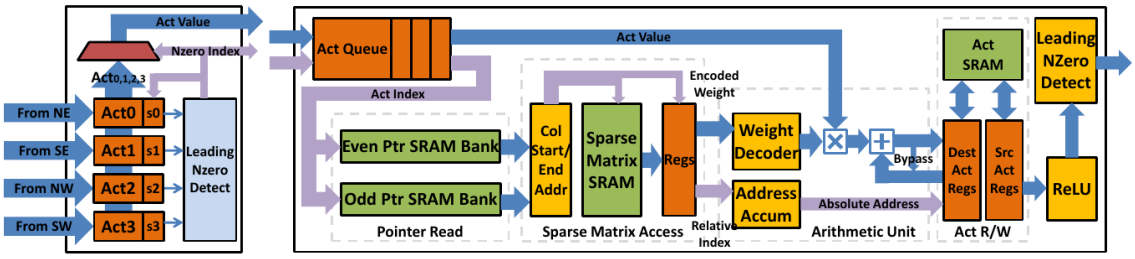


Figure 2.16: The architecture of Leading non-zero detection node and the PE architecture [11]

2.2.2 ARM Cortex-M4/7, Cortex-M55 and Ethos-U55

The instruction-set architecture (ISA) of a given processor is a substantial limitation for running NN inference for any available microprocessor. Namely, the ML performance is dictated by the efficiency and the speed of processing numerical linear-algebra operations, like element-wise multiplications and other matrix operations. Thus, the single-instruction-multiple-data (SIMD), MAC, and vectorized instructions are crucial elements for ML processors. Most of the resource-constrained microprocessors, namely MCUs, do not even support SIMD instructions in their ISA, which makes them even more inappropriate. Namely, most MCUs operate on fixed-length registers. Therefore, using lower bit widths in these MCUs, e.g. in the case of aggressively quantized NNs, the parameters and data will be promoted to higher precision for computation. Therefore, some of the benefits from the quantization process will be lost since the ISA does not support sub-byte computations.

ARM addressed some of these problems with Cortex-M4 and Cortex-M7 processor cores, which are very similar with notable differences in pipelining and floating-point precision. The 32-bit Cortex-M4 processor core is the first AMRs core of the Cortex-M line that is built on an Armv7E-M architecture and features a dedi-

cated Digital Signal Processor (DSP) IP block and an optional FPU. The addition of DSP and FPU extensions to ISA greatly improves the performance of numerical algorithms by processing operations directly on the Cortex-M processors while maintaining the Cortex-M programmer's model. Some of the additions are SIMD instructions that operate on 8- or 16-bit integers, saturation instructions, a wide range of MAC instructions, hardware divide in 2-12 cycles, etc. All registers are still 32-bits wide, but the SIMD instructions operate on two 16-bit or four 8-bit values at the same time. Further, the SIMD instructions allow these 2x16-bit or 4x8-bit operations to be performed in parallel. Instructions that work on 8- or 16-bit data types are also useful for processing data such as video or audio, as they do not require full 32-bit precision. Figure 2.17 presents some features of the Cortex-M4 as a block diagram.

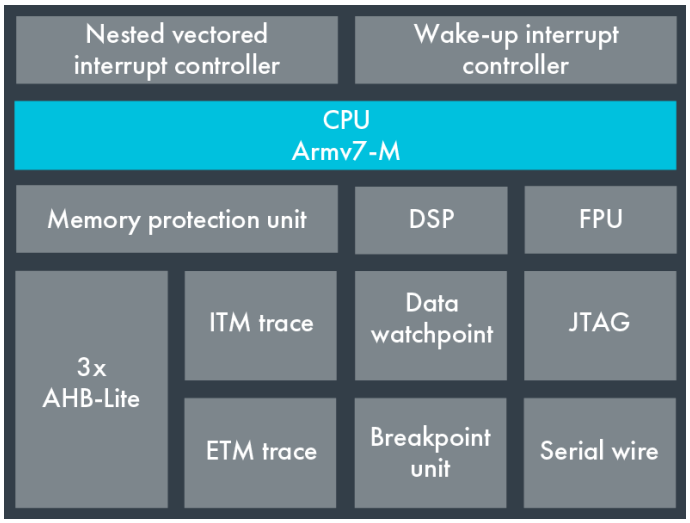


Figure 2.17: The Cortex-M4 features block diagram [5]

The Cortex-M55 is the latest low-power microprocessor core from ARM and is the first to be based on Armv8.1-architecture. It incorporates the M-Profile Vector Extension (MVE), also known as Helium, and it is a vector architectural extension introduced as a part of the Armv8.1-M architecture. Figure 2.18 shows most of the new features in the Cortex-M55 MCU, and there are different configurations of those features; for example, the Helium extension is optional.

The Cortex-M55 is a 4-stage pipeline design. Nevertheless, when either the FPU or MVE are implemented, the design must include the extended processing unit,

and the pipeline is extended to a 5-stage. The extended pipeline stage is separate and allows the core to switch to the retention or power-down state when not used.

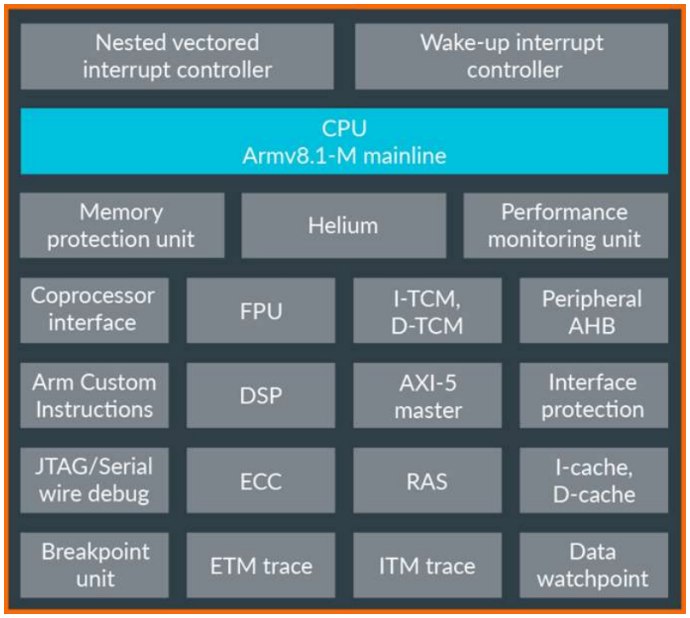


Figure 2.18: The Cortex-M55 features block diagram [25]

The MVE comes in two distinct versions. The MVE-I operates on 8-, 16- and 32-bit integer values, while the MVE-F on half- and single-precision floating-point values. Further, the MVE-I can be implemented without the FPU or MVE-F, while MVEF must be implemented with the FPU. All MVE instructions operate on 128-bit vectors. There are 8 MVE vectors (Q0-Q7) that are defined as aliases to the FPU register file (S0-S31) to increase area reuse. Many instructions can operate on a general-purpose register file to reduce the pressure on the FPU registers. Even though the registers' width is 128-bit, only half of that can be processed in one clock cycle. Nevertheless, the instructions can be overlapped. E.g., when a 128-bit load and a 128-bit MAC operation are overlapped. In one clock cycle, lower 64 bits of data are loaded into the register. In the next cycle, while still loading the upper 64 bits of data, the MAC instruction can already be executed on the previously loaded 64 bits. Nevertheless, the throughput of a MAC instruction is 2x32-bit, 4x16-bit and 8x8-bit. This is doubled when compared to Cortex-M4 and -M7.

Ethos-U55 is ARM's new type of processor, called microNPU, which is designed specifically for ML inference acceleration in constrained embedded devices. It is the

first of that kind that is available at the industry level. It is designed for integration with the Cortex-M line of MCUs, to provide additional performance for ML tasks. Figure 2.19 shows features of the Ethos-U55 NPU and the integration with a Cortex-M processor.

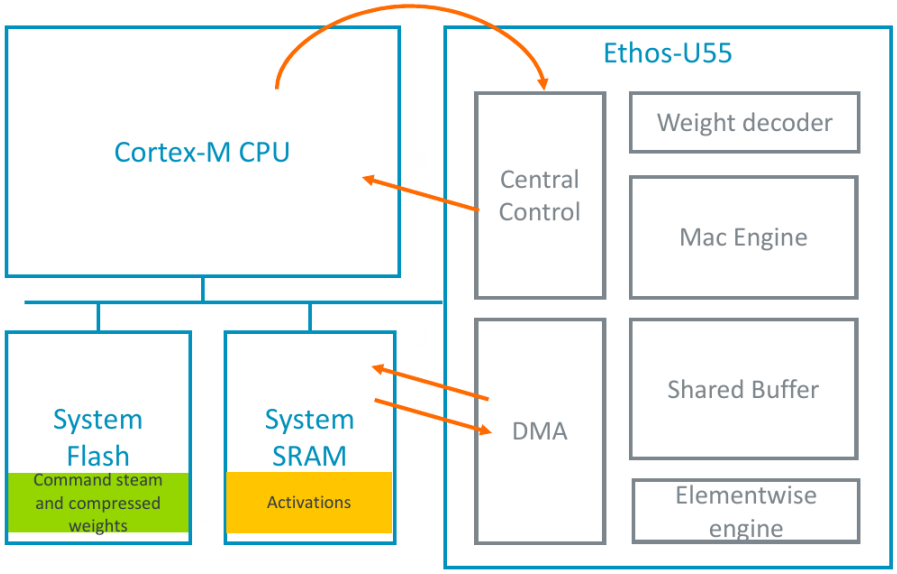


Figure 2.19: The Ethos-U55 features block diagram [25]

The NPU shares the flash and SRAM memory with the host MCU and supports 8-bit weights and 8- and 16-bit activations. The configurable MAC engine supports 32, 64, 128 or 256 MAC instructions per clock cycle for 8-bit activations, while 16-bit activations use two clock cycles per MAC operation. The Elementwise engine is designed to optimize commonly used matrix operations, while the weight decoder opens memory efficiency opportunities. The command stream is stored in the flash with the corresponding weights of a model, while the activations are stored in the SRAM. Ethos-U55 supports a fixed set of operators, while those not supported are just transferred to the host Cortex-M processor for their execution. Nevertheless, the interaction between the processors is simple. Namely, a trained NN has to be converted into a ".tflite" FlatBuffer file. The NN optimizer, a tool provided by ARM, then reads this ".tflite" file and decides which NN parts can be executed and accelerated on the Ethos-U55. The rest of the NN is executed on Cortex-M and accelerated through the CMSIS-NN library kernels. The host MCU also defines all needed memory regions. Namely, the location of the commands, weights and input

activations. The host then starts the NPU, which starts executing the commands using the SRAM regions, defined by the host through its direct memory access (DMA) engine. When the final result of the inference process is calculated, an interrupt is generated to the host MCU. The general workflow is depicted in figure 2.20.

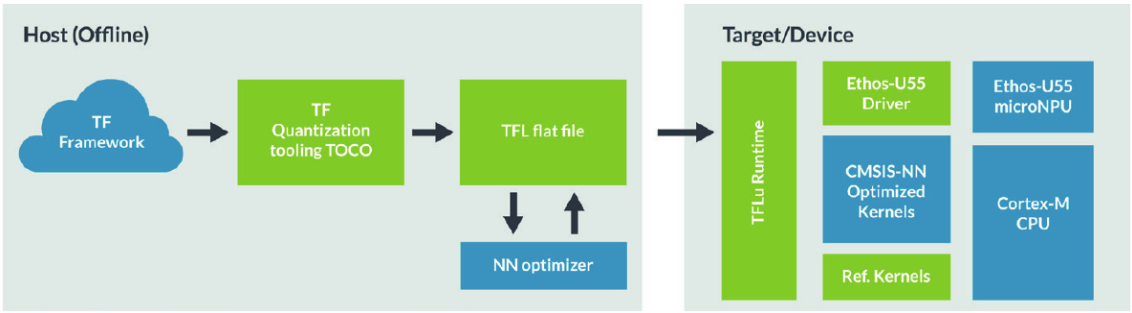


Figure 2.20: Ethos-U55 workflow with TensorFlow NN model [1]

3 TENSORFLOW LITE MICRO

Embedded platforms have always lacked the benefits of many advances in technology that made software development easier and faster due to high diversity and fragmentation among different hardware vendors. Unlike mainstream general-computing platforms that have unified, backward compatible and standardized ISA, virtual memory, operating system and other similar benefits, embedded platforms are missing all of them because of the highly-specific nature of their applications and their close relation to the hardware architecture and low memory and performance capabilities. Almost all software running on embedded systems is compiled from the source code; therefore, there is a lower demand for unification and cross-compatibility that leads to proprietary and closed ecosystems.

Until recently, engineers, developers, and researchers had to build their own ML ecosystems that had to be manually optimized for each hardware platform. Thus, they were narrowly focused, were missing support for other applications, and were hardly portable across different hardware platforms with different optimization possibilities. This results in a slow and difficult development process; therefore, a consequence is unjustifiably high costs regarding time and resources. Furthermore, hardware vendors do not have a fair, neutral, and comparative manner of benchmarking for testing the performance of their solutions. Further, there is also a problem of incompatible and incomplete infrastructure for pipelining of training, conversion, debugging, optimization, and deployment for TinyML use cases. The lack of these productivity tools makes everything even harder for the developers. All this leads to a lack of testing in real-world applications, and in many cases, to commercial licensing of these tools and even more fragmentation.

At the Google I/O 2019 event, it was announced that Google is partnering with ARM to develop TensorFlow Lite for Microcontrollers (TFLM). Therefore, the ARM's uTensor, one of the first lightweight inference libraries for embedded systems, became a part of the project presented in [24]. TFLM is an open-source ML inference framework for running deep-learning models on embedded platforms. It can be used to solve some of the fragmentation problems, make the efficiency re-

quirements imposed on embedded systems reachable, and guarantee cross-platform compatibility, which was so far practically impossible. The authors in [7] claim that TFLM is portable, flexible and easily adaptable to new applications. It minimizes the need for external dependencies and library requirements and does not require any standard C or C++ libraries or dynamic memory allocation.

3.1 Portability

The TensorFlow Lite (TFL) is a set of tools that enables on-device ML for mobile devices, while the TFLM is its subset meant to be used for 32-bit processors, namely MCUs. It is tested on different processors, most extensively on ARM Cortex-M architecture, but also on other popular platforms like ESP32, ARC and RISCV. It is written in C++ and intended to be used as a library, included in an application project and compiled with the rest of the application, unlike some other inference engine tools like GLOW AOT, which is a compiler that generates object files that can be used dynamically or statically in a project.

As the ML field is still growing and not entirely explored, it is subjected to research, changes and advances. Fundamental changes and new developments in network architectures, numerical algorithms and calculation techniques are hard to predict and follow. All these demand frequent software changes. Because of the mentioned fragmentation among hardware vendors and because hardware developers do not always have suitable qualifications for the ML field, it is hard to exploit the maximal possible performance of their devices, namely embedded processors, by the library itself. Hardware developers can e.g. deliver an optimized design for some numerical calculation, but it may not be suitable for ML. To address these issues, the TFLM team keeps close relations with hardware developers by encouraging submission to the library repository with essential technical support, tests and benchmarks. Because ML performance is mostly affected by the implementation of the mathematical operators like linear algebra computation, pooling, activation, convolution, and others, the hardware implementation is crucial. These operations are well defined and the implementation details can be hidden behind abstraction. The TFLM, therefore, allows the hardware manufacturers to modify the library by

developing their own implementation of a specific kernel function that exploits their specific hardware design. The C++ source code of the corresponding implementation can then override the default implementation. As it is shown in figure 3.1, default kernel function implementations are in a directory, while optimized versions are organized in subdirectories named by the specific hardware platform.

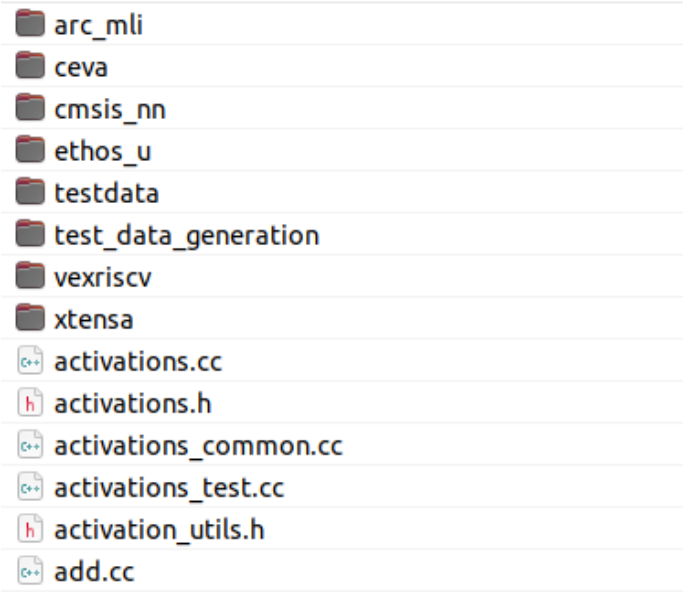


Figure 3.1: An example of kernel folder structure

Whoever uses this framework can then decide to use the optimized versions during the build process. Each platform is given a unique tag that is used as a command-line argument for building a final system. E.g., ARM is developing a CMSIS-NN library for Cortex-M MCUs, a collection of efficient NN kernels. In order to use this library, we only have to include the "cmsis-nn" tag during the building process in the TFLM framework, which in turn replaces the default kernels with the optimized versions during the compilation.

As can be deduced from the previous paragraph, TFLM uses a build system to determine which files will be used during the compiling process. This also helps to address the hardware fragmentation, because the developers often use platform-specific toolchains for the development of the software. This approach reduces the amount of work needed to include the TFLM library in a project. It is based on Makefile and Python scripts in order to create a project template, which is then

filled with required platform-specific files. When required, it is also converted into a specific format that is suitable for a specific toolchain. Therefore, the build system must support a highly fragmented ecosystem and avoid those features that do not generalize well across the platforms. Features like the customization of include paths, different compilers, and definitions of preprocessor macros must be supported. Otherwise, they will not be accepted by developers and vendors. The main principle is that the TFLM generates the required source and header files and stores them into a folder hierarchy that can then be easily included in a project and compiled all together.

To improve the framework's portability, it is assumed that the model, input data, and output arrays are already placed into memory, and it is the developer's job to load them before. The ML models have well-defined inputs and outputs. The library excludes the possibility of loading a model from a file system or accessing the peripheral for input or output data. The reason is the lack of unification and standardization among vendors. Making it a part of the framework would seemingly hurt portability.

3.2 Scalability

While the TensorFlow training environment supports more than 1400 operations, the TFL supports only its subset, about 130 operations, since not all operations are useful on edge and embedded devices. That makes exporting the model even more complicated. Developers often want to convert a plethora of potential models to be used with embedded and mobile devices, but manual conversion is unacceptable. Changing the model to meet the framework's requirements is almost always required because some operations may not be compatible with the target. Even if the operation is supported, the data type or range of values may not be. Most training environments use floating-point representation since they are well optimized for general computing platforms, and although they can be converted into quantized representation, they increase the complexity, and some may even require support during training. Using the high-level APIs only complicates the task because they often hide the operations behind the abstraction.

Figure 3.2 presents a typical workflow of training, exporting, and deploying a model. Namely, the exporter receives a trained TensorFlow model and generates a TFL model file with the ".tflite" extension. After exporting, the ".tflite" file can already be deployed on some edge devices, e.g. mobile phones, to run inference locally using the TFL interpreter. The TFL toolchain is reused, and the TFLM workflow is built on top of it. The converter from the TFL Python API converts the ".tflite" file into the "FlatBuffer" file that can then be converted into a C array. Namely, the C array is used by the TFLM to load the model and to be used by the interpreter. Strong integration of the TFL with the TensorFlow training environment enables this extending of the TFL toolchain to support embedded ML systems by reusing useful tools.

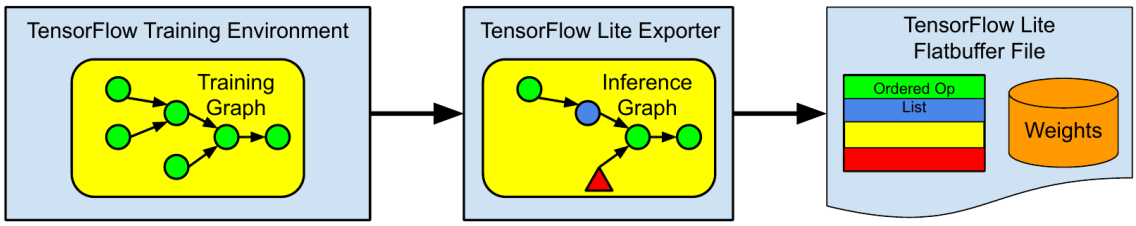


Figure 3.2: Model export workflow [7]

3.3 Implementation

Figure 3.3 shows the basic structure of the TFLM implementation. First, the embedded application creates a NN model object in the memory. The TFLM interpreter expects that the model is provided as a C++ array. The model is defined in model.h and model.cc files, and the header is included in the application. The developer has to produce the Operator Resolver object through the client API. This API call controls which operators will be linked to the final binary to minimize the size of the model. Then a contiguous part of the memory is allocated to the Tensor Arena object. Tensor Arena is used by the interpreter and holds intermediate calculation results and other variables. The size required for the Tensor Arena depends on the model and needs to be determined through the experimental work. The Tensor Arena, in this way, eliminates the need for dynamic memory allocation and possible errors caused by heap fragmentation. Then the interpreter's instance is supplied

with the model, the operator resolver and the arena as function arguments. The interpreter can then allocate the needed memory from the Tensor Arena. The application then populates the input memory region with the input data and invokes the interpreter to perform inference calculations. The invocation is a simple blocking call. The interpreter iterates through the topologically sorted operations and returns control to the application. Then the application can fetch the inference results from the output location.

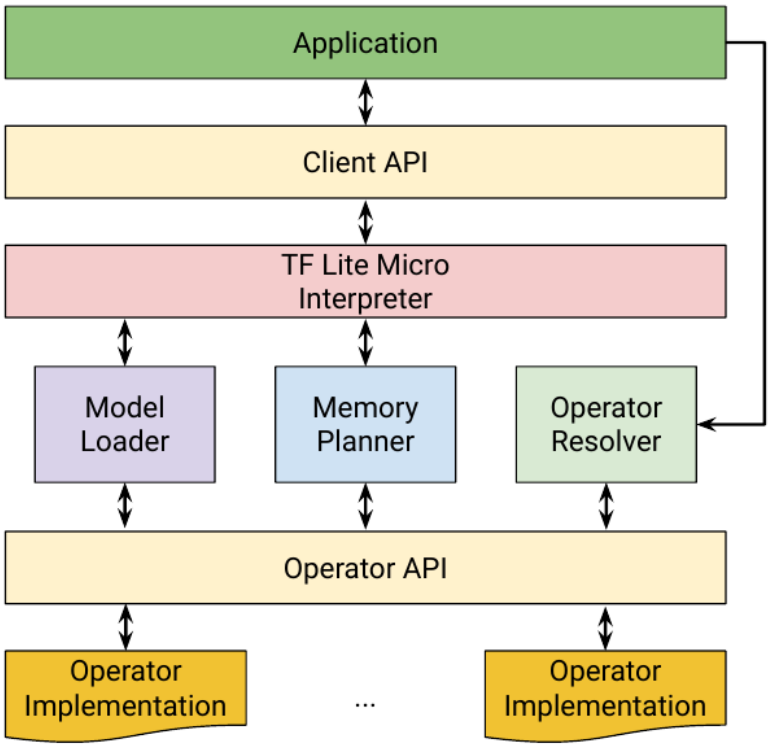


Figure 3.3: The implementation module [7]

3.4 The interpreter

The interpreter-based approach is usually viewed as a less efficient approach for ML in embedded applications when compared to other available methods. The interpreter loads a data structure that defines a model and handles that data during run time. The data structure determines which operators to use and all parameter locations. During the initialization, the communication between the interpreter and the operator resolver determines which operator implementations should be used

and their locations. The interpreter then uses this information to iterate through the topologically sorted operations. Nevertheless, the alternative is code-generation-based or compilation-based inference. These two use target's native machine code to describe the model and its execution by intertwining models architecture, operator function calls, and parameters to form a binary image.

The major advantages of interpreter-based over compilation-based frameworks are the ease of updating the models, their maintenance, and the code's portability. It allows updating only the model. On the other hand, when using the code-based approach, the entire executable must be replaced in order to update the model. The interpreter-based approach also keeps the model separate from the rest of the engine. Thus, it allows modifying only one piece of the memory without the recompilation process. Additionally, the same model file can be reused on multiple targets.

The slight drawback of the interpreter approach is the interpreter's overhead. Nevertheless, since the ML applications consist mostly of long-running linear-algebra operations, the operator branching overhead during run-time is comparably small. The authors in [7] demonstrated that this overhead represented only a minor issue and published the experimental results for two different hardware platforms and two different ML models. The experiments were done on both ends of the 32-bit microprocessor spectrum, namely Ambiq Apollo Cortex-M4 MCU and Xtensia mini DSP. The models used are also different: the relatively large and long-running visual-wake-word (VWW) person-detection model and a small and short-running hot-word detection (HWD) model. The details of the platforms used are presented in table 3.1.

Table 3.1: The hardware platforms used in evaluation [7]

Platform	Processor	Memory: Flash / RAM
Ambiq Apollo	ARM Cortex-M4 96 MHz	1 MB / 0.38 MB
Tensilica HiFi	Xtensia mini DSP 10 MHz	1 MB / 1 MB

The evaluation was done with TFLM reference kernels and with high-performance optimized kernels. Namely, the CMSIS-NN library for the Cortex-M4 MCU and the

Cadence library for the Xtensia DSP. Results are listed in tables 3.2 and 3.3. The second column in these tables shows the number of cycles needed to complete one inference instance with and without the interpreter's overhead. Thus, the difference between these values is the overhead. The results also show that the overhead is insignificant, especially for the VWW model, since it is less than 0.1%. Further, the difference in performance between reference and optimized kernels is considerable and results in 4 times increase in speed for Cortex-M4 and even a 7.7 times increase in speed for Xtensia DSP for the VWW model. For the HWD model with the optimized kernel, the Cortex-M4 shows roughly only a 25% increase in speed, while the DSP shows more than 10 times increase in speed.

Table 3.2: Evaluation results for the Cortex-M4 [7]

Model	Cycles: with overhead / only inference	Overhead
VWW - reference	18,990.8K / 18,987.1K	$< 0.1\%$
VWW - optimized	4,857.7K / 4,852.9K	$< 0.1\%$
HWD - reference	45.1K / 43.7K	3.3%
HWD - optimized	36.4K / 34.9K	4.1%

Table 3.3: Evaluation results for the Xtensia DSP [7]

Model	Cycles: with overhead / only inference	Overhead
VWW - reference	387,342K / 387,330K	$< 0.1\%$
VWW - optimized	49,952K / 49,946K	$< 0.1\%$
HWD - reference	990.4K / 987.4K	3.3%
HWD - optimized	88.4K / 84.6K	4.1%

3.5 Model representation

The "FlatBuffer" serialization format that holds the model is a memory-mapped representation. The operations of a model are in a topologically sorted list and do not require unpacking. The format is converted into a C++ source file that contains

a data array and is compiled into binary form, to which the interpreter can refer simply by iterating through the list. The accessor's code is very small and is written in the header-only library, which makes it memory efficient and easy to compile.

The model representation is the same as for TFL and is reused for TFLM for scalability reasons. As TFL is designed to be portable across systems, it abstracts the operator parameters from the arguments passed to the implementation functions of those operators. Therefore, during run-time, each operation requires some processing code to convert the parameters from serialized representation into the one required by the underlying implementation. Although the overhead is small, it does increase the size and complicates the operator implementations.

3.6 Memory management

The application supplies a fixed-size memory arena to the interpreter during initialization. The interpreter determines the lifetime and the size of all necessary buffers, including run-time tensors, memory for meta-data, scratch memory, etc. The interpreter treats allocations within the arena as a stack. Since the allocations overlap during their lifetime, the stack is divided into two parts, as is shown in figure 3.4. The objects needed during initialization and operator evaluation are stacked on the Head stack that increments from the arena's lowest memory address. The interpreter-lifetime objects are stacked on the Tail stack that decrements from the arena's highest memory address. The model preparation allocations are placed between the two stacks. If the two stack pointers cross each other, the application-level error occurs in order to indicate a lack of memory for the arena. With this approach, the arena size can be significantly reduced.

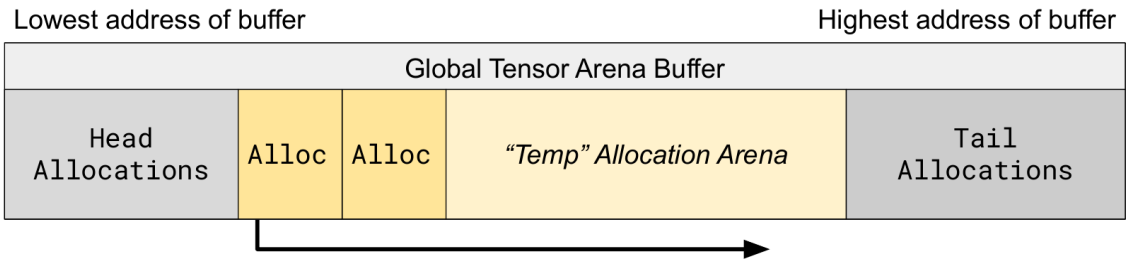


Figure 3.4: The arena buffer allocations [7]

The memory planner shown in figure 3.3 creates a memory plan that reuses memory space if possible while ensuring that its contents are valid during their lifetime. One operator may write to one or more buffers, while the following operators may read them as their inputs. If the buffer is not defined as a model output, it needs to remain intact only until the last operator that needs it has finished its operation. Here, memory reuse is possible when the allocation that is not needed anymore is replaced with a new allocation. Figure 3.5 shows the naive approach to allocation on the left and a more efficient approach on the right side that is named bin packing. In figure 3.5, one dimension represents the memory, while the other represents the time with the operators taking place in sequential.

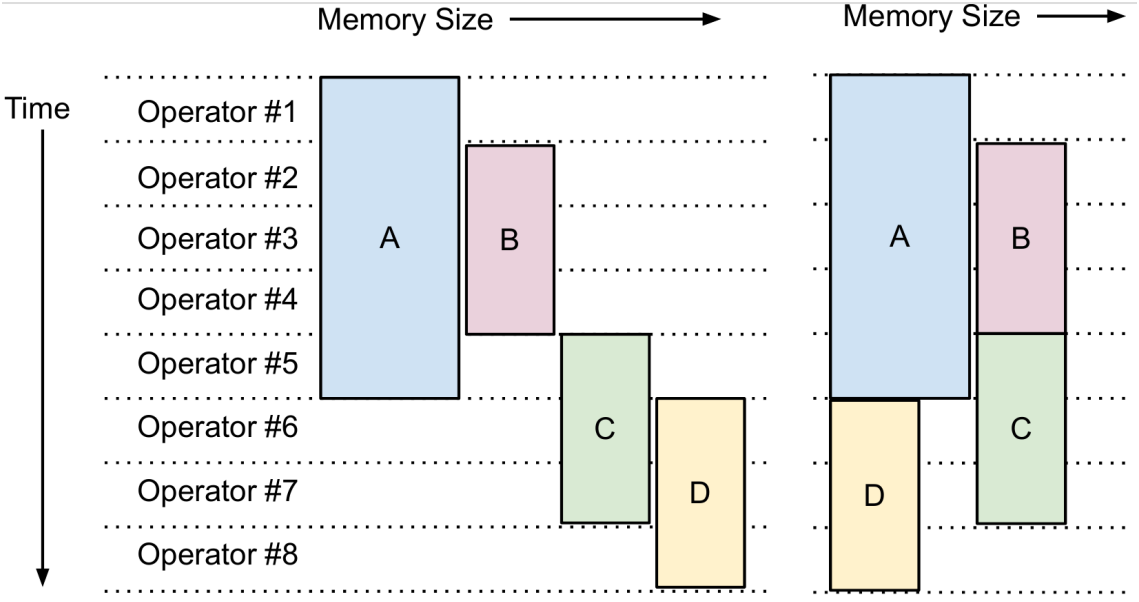


Figure 3.5: Intermediate allocations [7]

As the interpreter has gathered a list of all buffers with their lifetimes and sizes, the obtained list is sorted by size in descending order and used to place each allocation in the first sufficiently large memory gap. For example, in figure 3.5, when operator 4 finishes its execution, the B allocation is not needed anymore; therefore, the allocation C is placed in its memory space. By using this memory planning method, the arena's memory can be reduced even further. Table 3.4 shows how much persistent and non-persistent memory is used by different models. The tests were done on the Cortex-M4 MCU with the same models as before.

Table 3.4: Memory consumption on the Cortex-M4 device [7]

Model	Persistent	Non-persistent	Total
VWW - reference	26.5 kB	55.3 kB	81.8 kB
HWD - reference	12.12 kB	680 B	12.8 kB

TFLM also allows the end user to create an offline-planned tensor allocation layout. It enables more compact memory plan designed before the run time. It imposes less overhead during initialization by allowing different memory banks to hold certain memory areas.

3.7 Multitenancy and multithreading

Many embedded ML applications need to support multiple specialized models on the same system. TFLM allows this if the models don't need to run simultaneously. Instead of separating the models to run completely independently, the TFLM supports multitenancy to allow the reuse of temporary memory space. The single-model use of the arena is depicted on the top, and the multiple-model reuse of the arena on the bottom in figure 3.6.

The reuse is enabled by allowing multiple interpreters of different models to allocate memory from a single arena. The interpreter-lifetime objects stack on each other, while the function-lifetime stack is reused by different models. The reusable stack is set to the largest requirement, and the persistent stack grows with each model invocation.

TFLM can safely run on multiple threads or multiple MCU cores, if there is no state kept outside the model's interpreter, and allocations within the arena. It supports multiple interpreter instances to run from different threads. Multiple cores can share the code because the arena holds the interpreter's variables.

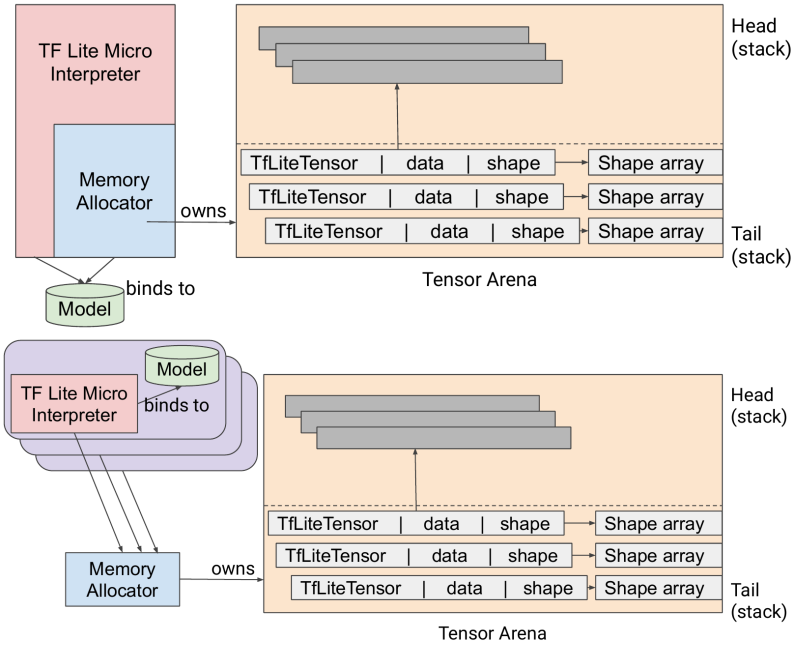


Figure 3.6: The use of arena space by single model, and multiple models [7]

4 VIDEO INTERCOM SYSTEM WITH HUMAN DETECTION

Intercom systems have been a part of all kinds of properties for decades, from homes and offices to industrial facilities and any other property that requires managed visitor access, helping their owners to increase security and convenience by simplifying property access. Regular intercom systems enable two-way communication between a tenant and a visitor and allow the tenant to grant access by opening a door remotely. Looking at history, one can find that the first telephone-based intercom system was patented in 1894. But even an earlier version existed in the form of metal tubes that carried sound and were used between offices. Today's modern electronic intercoms are becoming increasingly popular and more versatile, using contemporary technology and enabling features like internet connectivity to be able to grant access from anywhere in the world. More advanced and expensive systems feature facial recognition and authentication and enable touchless access control. Although there are already many smart intercoms available, there is still room for novel experiments and improvements. Most of these devices utilize multi-core application processors, which require a more complex design and increase power consumption. They often require internet connectivity to send a request to the inference server and could thus hurt the clients' trust.

To use the TFLM library and to deploy a CNN model on some constrained MCU device, the smart video intercom system is proposed and developed in this master's thesis. This is a much simpler and cheaper smart intercom device, intended for easy and more convenient installation on, e.g. entrance doors. The design is simpler than most smart intercoms on the market today and features human presence detection to automatically alert the user that someone is at the door but does not have facial authentication because of simplicity and cost reasons. It also incorporates an LCD display that automatically turns on and off, depending on the detection result, for the sake of using less energy. The system is built around mainstream Cortex-M4 MCU. Its design is described in more detail in the following section.

4.1 The Intercom system's hardware design

In this section, the system's components, their application and their integration into the Intercom system are described in more detail. Figure 4.1 shows the functional system block diagram used to develop the Intercom system. As can be seen, there are 3 main components: Cortex-M processor, camera, and the LCD display

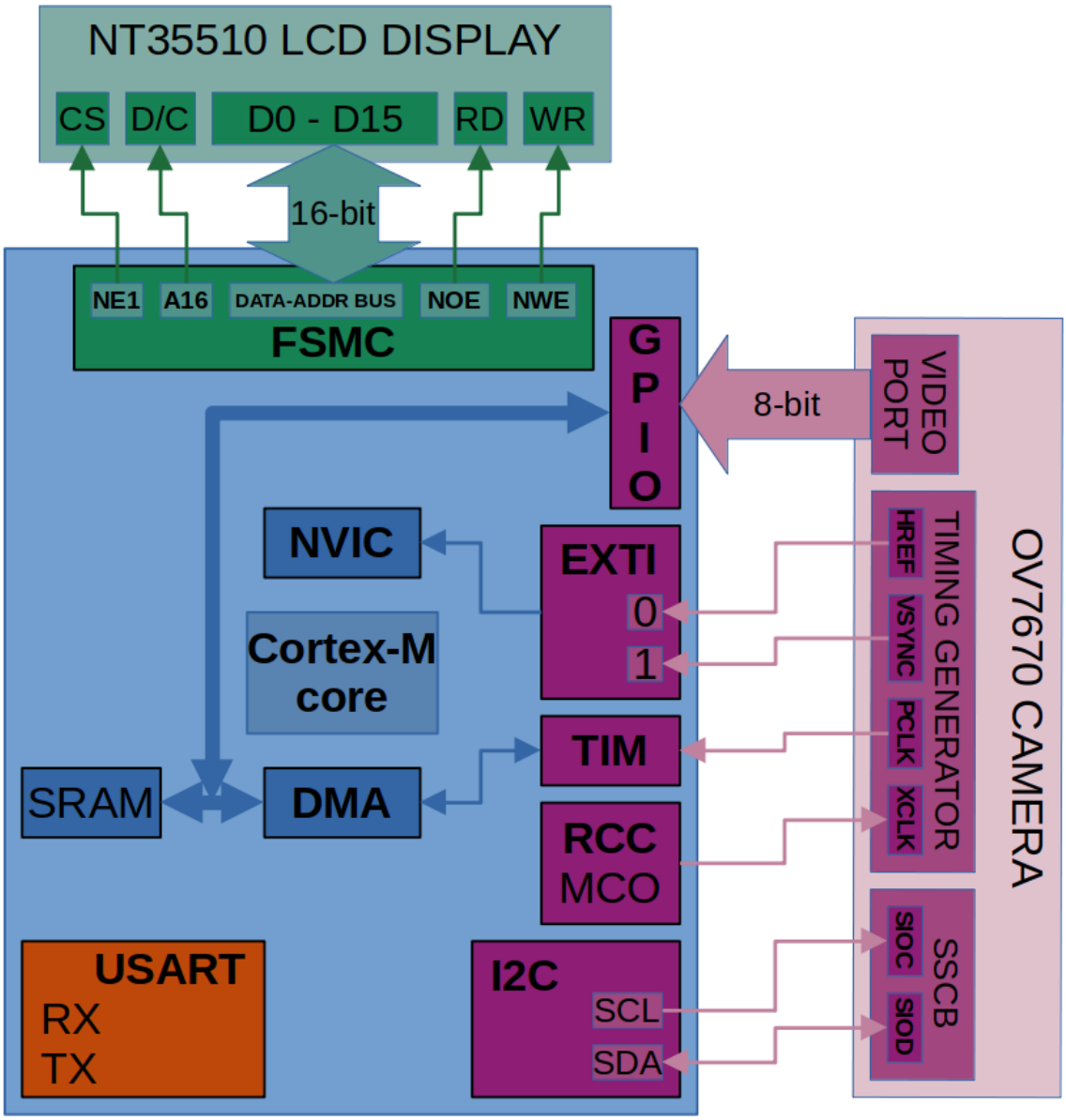


Figure 4.1: The system's high-level block diagram

Further, the proposed Intercom system also includes the power supply and serial interface used for communication with a personal computer. The serial interface is needed due to the possible need to update the NN model and also for debugging and

testing purposes during development. Both camera and the LCD display are connected by a parallel bus. Nevertheless, they are implemented in different ways. The LCD bus uses a dedicated peripheral block called flexible-static-memory-controller (FSMC), while the camera bus is implemented using a GPIO port and the DMA. Both are described later in this section.

The MCU used for this thesis is an STM32F413 device based on Cortex-M4 32-bit RISC MCU operating at 100 MHz through the PLL and an internal 16 MHz RC oscillator. It supports DSP instructions and has 1.5 MB of flash memory and 320 kB of SRAM.

The LCD display used in this thesis is an EastRising ER-TFT040-1 TFT LCD module with an NT35510 controller that incorporates a gate driver and an internal GRAM. The display has a WVGA resolution of 480x800, while the diagonal size is 3.97 inches. The NT35510 controller itself supports several different communication interfaces, including the i80 parallel interface, I2C, MIPI DSI and MDDI. The module manufacturer has limited the choice to 8- or 16-bit i80 and 16-bit SPI interfaces. The colour depths are also configurable, while the maximum depth is 16.7M colours. For this thesis, the 16-bit i80 parallel interface is selected for the communication. Further, 16-bit colour depth or 65k colours has been selected. In this way, each pixel can be written in only one write cycle.

The display controller acts as a slave device to the i80 communication interface and uses a 16-bit multiplexed bi-directional bus for both the register addressing and for data written to or read from these registers. It also uses 4 control signals, the data/command (D/C), the read (RDX) and write (WRX) strobe, and the chip-select (CS) signals. Figures 4.2 and 4.3 show a diagram of write and read cycles for single and multiple parameters. The CS signal enables and disables the interface. The D/C signal determines whether the address of a register or data for the register is on the bus. When D/C is low, the data on the bus is interpreted as a register's address, and when it is high, the data is interpreted as the register parameter or GRAM data. The RDX and WRX are the read and write signals, respectively. The master indicates a read cycle with a falling edge of the RDX signal. Then, the LCD controller has some time to put the data on the bus. On the rising edge of the RDX

signal, the master reads the bus. The master indicates a write cycle with a falling edge of the WRX signal and starts assembling data on the bus. The LCD controller reads the data on the rising edge of the WRX signal.

The STM32F413 MCU has the FSMC used for driving external memories, and it can be configured to interface with the LCD display. The signals used in FSMC are somewhat different from those in the i80 interface, but due to its flexibility, the signals can be seamlessly repurposed.

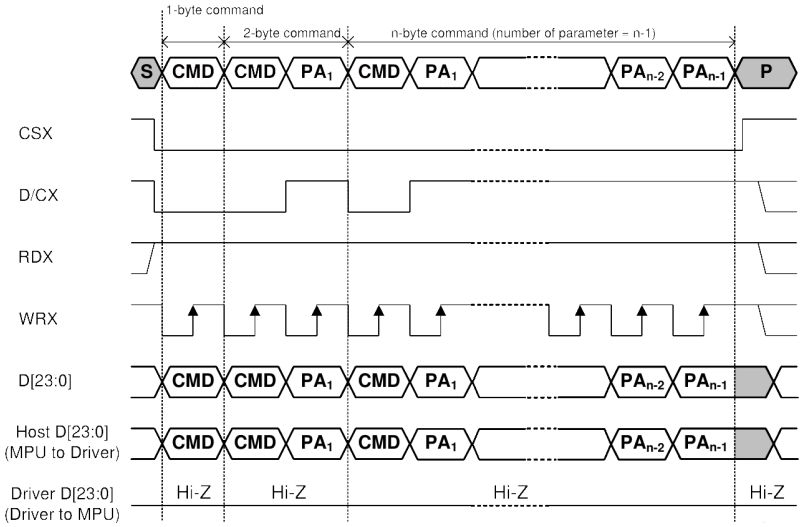


Figure 4.2: The i80 timing diagram for write cycle [21]

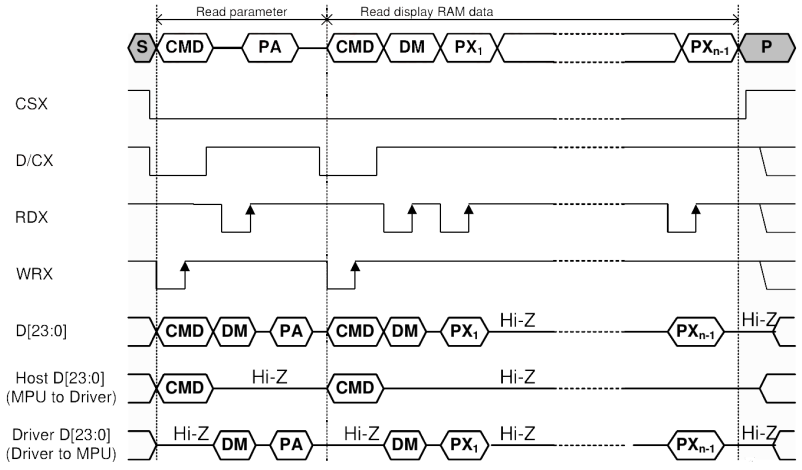


Figure 4.3: The i80 timing diagram for read cycle [21]

The FSMC is mapped to a range of MCU addresses. When writing to these addresses, the FSMC generates appropriate signalling to transfer the data to the

external memory or the LCD. Thus, writing to the LCD controller is just writing a word into the MCU memory. The address space is divided into 4 banks, where each bank has its enable signal, called NE, and can be repurposed for the i80's CS signal. Depending on which quarter of the memory space is used, a different NE signal is issued. It uses NOE and NWE signals in the same way as the RDX and WRX are used in i80.

The FSMC uses separate address and data buses and thus has no D/C signal. The i80 interface multiplexes address and data on the same bus and use the D/C signal to distinguish between them. Therefore, the FSMC's address bus is reused as a D/C signal for the i80 interface. Only one line of the FSMC's address bus is physically implemented, meaning that only two addresses from the whole address space are used. Writing data to one of these addresses turns on this address line while writing to the other address turns it off. This way, the address line is reused as a D/C signal. From a software perspective, there is one address where commands are written and one address where data is written. For example, if bank 1 is used, the MCU address space is limited to an address range from 0x6000 0000 to 0x6FFF FFFF, and if address line 16 (A16) is used as a D/C signal, the address space is limited to two addresses, namely 0x6000 0000 and 0x6002 0000. The 17th bit is used instead of the 16th bit because the MCU address space is byte-addressable, and the FSMC bus width is 16 bits long. Therefore, the actual address is shifted left for one bit, and the address issued to the address bus is shifted back. Because only one line of the address bus is used, this addressing caveat is important only for determining which physical line is going to be used as a D/C signal. When 8-bit bus width is used instead of a 16-bit, this shifting will not occur. Namely, the software can actually use any other address from this memory bank space to write a registered address to the LCD because other physical address lines are not used at all, and only 0x6002 0000 must be used for the data. Therefore, the software can define a structure with two half-word elements and point the structure to the address of 0x6001 FFFE. The first element of the structure will be used for address transmission, as the 17th bit is low. The second element will be 2 addresses higher and will have the 17th bit high. The software can now conveniently use this structure to interface with the

LCD controller.

The camera used for this thesis is an OmniVision OV7670 CMOS image sensor with a VGA resolution of 640x480 pixels. It is capable of operating at 30 fps and has configurable image quality, frame rate, image size and output data formatting. In this thesis, the frame rate is reduced to 12 fps, while the resolution is downscaled to QVGA or 320x240. The output format used is RGB565, the same as for the LCD. The configuration of the camera can be performed over the serial-camera control bus (SCCB), which is similar to the I2C bus. The image data is transferred through the synchronous 8-bit parallel interface. In addition to 8 data lines, the parallel interface defines 4 additional signals: VSYNC, HREF, XCLK and PCLK. Figures 4.4 and 4.5 show the timing diagrams of these signals, but their polarity and gating can be configured differently.

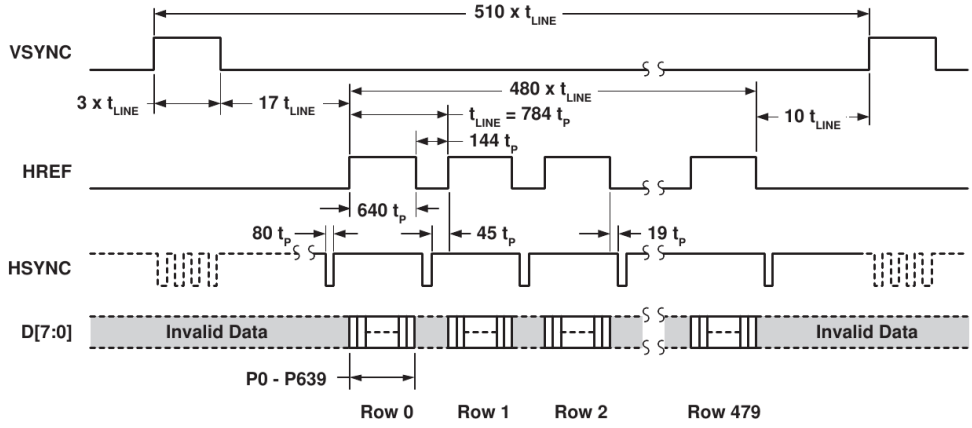


Figure 4.4: The OV7670 vertical timing diagram [22]

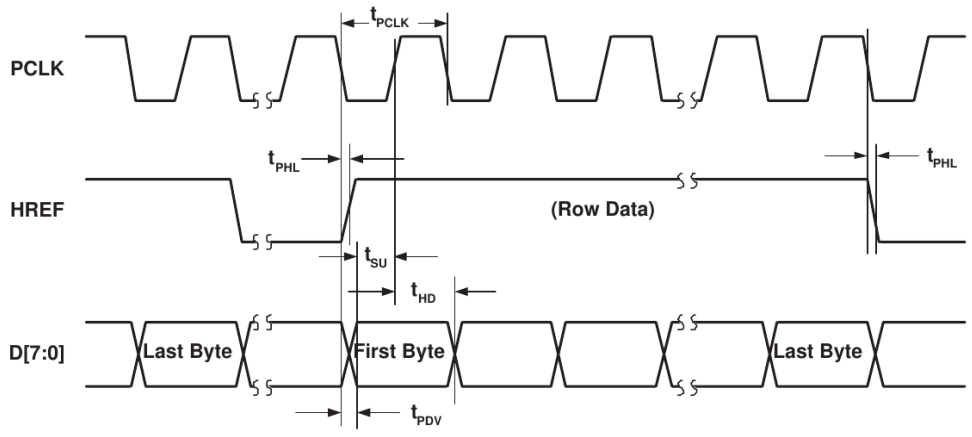


Figure 4.5: The OV7670 horizontal timing diagram [22]

The VSYNC signal is transmitted after each frame. The HREF signal is high during the transmission of each frame's line. Data is transmitted over 8-bit parallel bus, while each byte is sampled on the PCLK rising edge. The PCLK is configured to be gated by the HREF signal, meaning that there is no PCLK signal during the horizontal blank period. Each pixel is divided into two bytes that are transmitted consecutively. Thus, one line in QVGA has a total of 640 bytes.

Because the LCD controller has occupied the only available MCU's parallel interface, the camera interface is implemented by using a GPIO port, timer, external interrupt controller (EXTI), and DMA controller. The camera requires a continuous clock signal feeding over the XCLK line. The MCU's clock control circuitry can output a pre-scaled clock signal from different sources on a selected GPIO pin. The MCU system's clock is selected and down-scaled to 20 MHz. The implementation of the interface is fairly simple. The PCLK signal from the camera is scaled down to 5 MHz through the camera's control registers over the SCCB and fed to the MCU's timer peripheral. The timer is configured to generate a DMA request on each PCLK rising edge. The DMA then transfers the GPIO input register into the memory and increments the memory pointer. The VSYNC and HREF signals are fed to the MCU's EXTI inputs. The VSYNC signal is configured to trigger an interrupt on each rising and falling edge. The falling edge is used to enable the HREF interrupt and set the DMA memory pointer to the initial value. The rising edge is used to disable both the DMA and the HREF interrupt. The HREF interrupt is triggered only on the falling edge and is used to reconfigure the DMA and increment the line pointer. In this way, the DMA is configured to transfer 640 bytes and is reconfigured on each HREF interrupt after the line pointer is incremented.

4.2 The Intercom system's software design

The development environment used for this thesis is STM32CubeIDE. It is a C/C++ development platform based on the Eclipse framework that enables peripheral driver configuration, project creation, code generation, compilation and debugging. It uses GCC toolchains for development and GDB for debugging purposes. It helps the developer to analyze memory usage and debug code and significantly save development

time spent on the peripheral device configuration and its initialization.

Although the TFLM library is written in C++, the rest of the code, including the low-level peripheral drivers and application, is written in C, except for the startup code, which is written in assembly. The low-level drivers for the FSMC, DMA, GPIO, the timer, EXTI, NVIC, I2C, USART, and their initialization, are configured through the STM32CubeIDE Device Configuration Tool. The low-level drivers for these devices, as well as the project structure, startup code, linker script, and system and clock initialization, are generated by this tool. The high-level drivers for using these peripherals and the application are then developed, and the TFLM library is included in the project. The high-level driver for using the FSMC with the LCD display includes several useful functions, such as drawing lines and dots, printing characters and strings with variable font size, setting the screen orientation, clearing and filling desired areas of the screen, etc. The driver for the camera includes functions for configuring the camera's resolution, output format, frame rate, colour saturation, brightness, etc. It also includes the interrupt routines and communication with the camera's configuration register over the I2C bus.

The model used in this thesis is already pre-trained by the TensorFlow team. They used the VWW dataset [4] for training and the MobileNets [14] CNN architecture class to make this model. The trained model is converted to a ".tflite" file and then converted into a constant unsigned char array.

As described in section 3.1, the TFLM framework uses a combination of Makefile and Python scripts to generate the library structure that is included in a project and fills it with necessary and relevant files. The scripts are able to modify the generated structure to support different development environments and add optimized code to exploit certain platforms by specifying the IDE and target platform during the generation process.

The TFLM file structure is relatively simple. The micro directory contains the kernels, memory planner, and model directories. The kernels directory contains reference and optimized versions of the kernels. The memory planner's directory contains code for optimizing the memory usage described in section 3.6. The models' directory contains the C array model. The micro directory also contains several

important source files. The all_ops_resolver.cc and micro_mutable_op_resolver.h is used to pull in the operations needed by the model. The micro_interpreter.cc contains the code for running the model. Outside the micro directory are other important files that are shared with the TFL, e.g. the schema directory contains the code used to interpret the FlatBuffer model's file format, while the core directory contains the code for error reporting.

To run inference on the MCU, several header files, including the model header, must be included in the main task file. Memory for the tensor arena needs to be allocated. Then the model is loaded and checked if it is compatible with the schema version. The AllOpsResolver or the MicroMutableOpResolver instance is then used to load the kernel operations. The AllOpsResolver simply loads all available operations, while MicroMutableOpResolver loads only the ones used by the model, but the operations need to be manually listed. The MicroInterpreter instance is created, and the model, MicroMutableOpResolver, tensor arena and arena size are passed to it as arguments. Finally, the interpreter allocates the memory from the arena needed for the tensors. After this procedure, the interpreter is ready to be invoked, and the result of the inference can be retrieved.

The model expects a grey-scale image with a size of 96x96. The grey-scale is 8 bits in depth. Thus, the image obtained from the camera must be cropped and converted into an 8-bit grey-scale. Because both the camera and the LCD are using a 16-bit RGB format, after converting to grey-scale by averaging the R, G, and B components of a pixel, the value must be scaled into 8-bit depth; otherwise, the values would never be greater than 31, and that would impact the accuracy of the model.

4.3 Tests and results

This section overviews the proposed Intercom system, the testing approach, and the measurements. To evaluate the overall system performance and the applicability of the TFLM solution to the image classification problem in embedded systems, the system was analyzed for accuracy, memory and power consumption, and latency. All tests and measurements were done on the same physical system built as a part

of this master's thesis and for this purpose.

Figure 4.6 shows memory usage for the Intercom system's flash and SRAM. This information was obtained by the STM32CubeIDE tool, Build Analyzer, and the generated memory map. The system flash memory used about 25%, mostly by the person detection model array in the ".rodata" section, which is just under 300 kB. Only about 65 kB is used by the .text section compared to the model size that includes all drivers and kernel implementations. The flash memory on this particular MCU is relatively large; thus, a much larger model could fit in. The usage of the SRAM is about 93%, and most of it is spent on the tensor arena and the camera buffers. The camera buffer is a 16-bit array and consumes 150 kB because the application prints coloured pictures in QVGA size on the LCD. But if that wasn't needed, the camera could be configured in lower resolution than QVGA, resulting in even faster acquisition. Further, the data could be converted to a grey-scale during frame acquisition, and only a small 8-bit grey-scale buffer could be used for the inference. Almost all remaining RAM memory was used by remaining .bss, heap and stack sections.

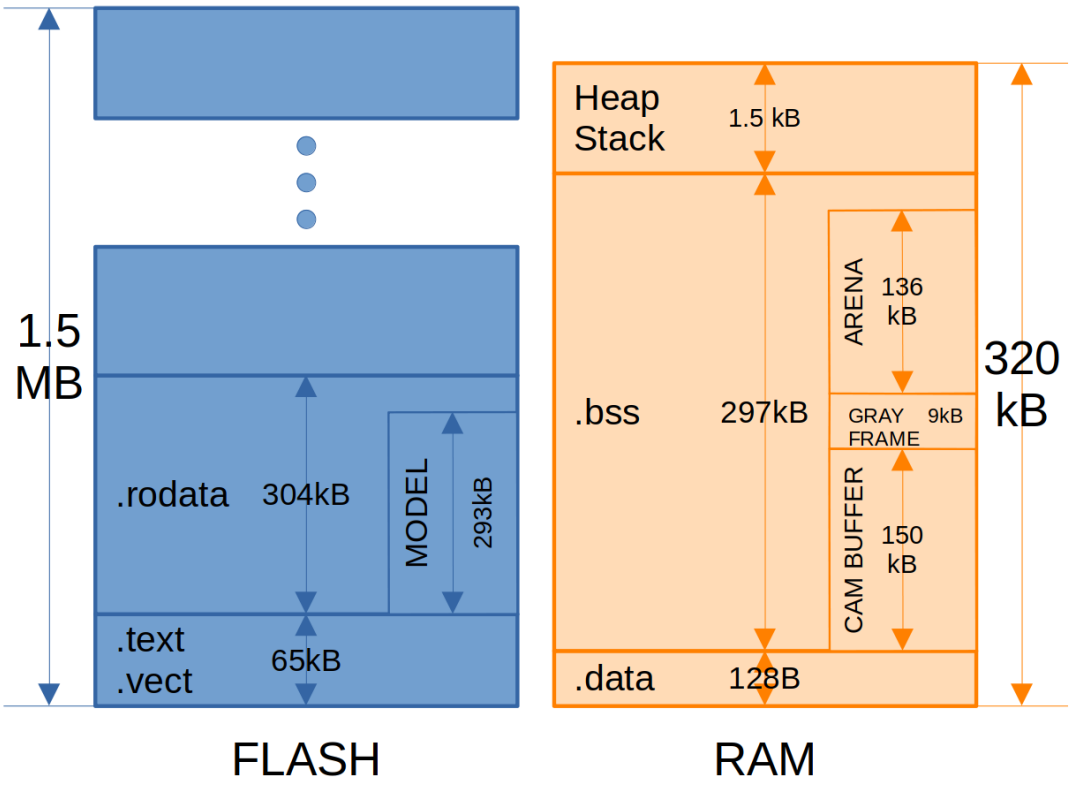


Figure 4.6: The flash and SRAM usage of the system

To test the TFLM library's work, a testing dataset was compiled and ran through the model. The dataset was small, consisting of 30 different images with a person present and 30 images without. Figure 4.7 shows a part of the dataset, while table 4.1 contains the resulting scores after the inference. The images in figure 4.7 and the data from the table are ordered in the same order, with images containing a person on the top two rows and without a person on the bottom.

Figure 4.7: A part of the dataset used for testing

Table 4.1: A comparison of classification results for 20 different images as displayed in Figure 4.7

person	95	61	-14	97	116
person	57	58	112	78	110
no person	-77	30	-33	-17	10
no person	-95	-83	74	102	10

The scores for the images with a person should be positive, and the scores for images without a person should be negative; otherwise, the model has failed to classify the image correctly. From this part of the dataset, the model failed to

correctly classify 6 out of 20 images. The whole dataset evaluated the model with an accuracy of 73%. The images used were downloaded from the internet and transferred to the system via serial communication. Thus, using additional memory in flash. This is not considered in the memory evaluation above because it was done only during this test.

Figure 4.8 shows the current measurement. A 1 Ω resistor, rated at 10 W, is connected between the ground of the power supply, and the ground of the system. The voltage across the resistor is representing the current that the system is using. The power usage is then calculated by multiplying the RMS current with the nominal voltage. The average current is roughly 110 mA, mostly due to the LCD backlighting. The graph was recorded using the OWON VDS1022 PC oscilloscope.

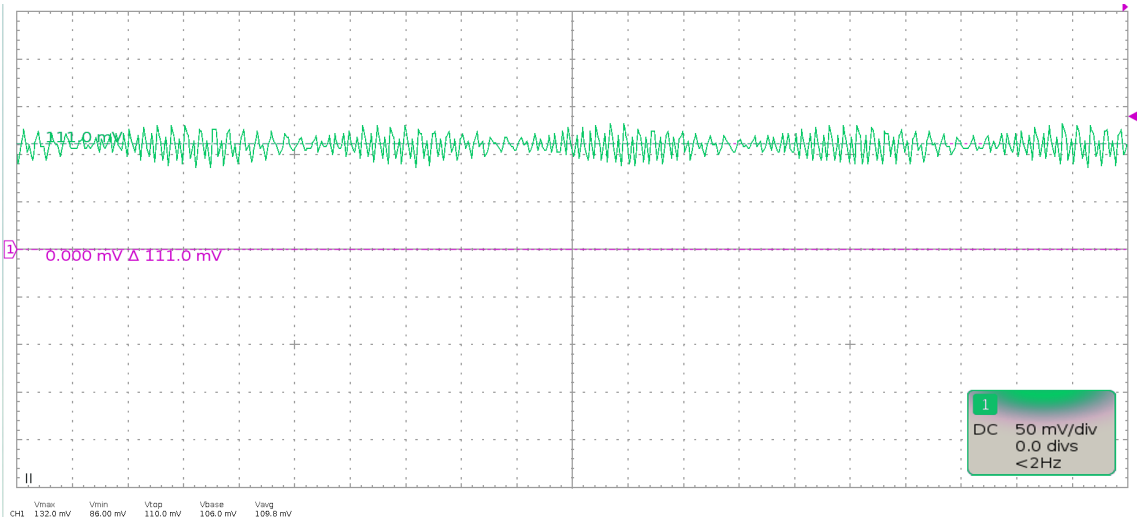


Figure 4.8: The power consumption measurement

The latency of the inference process was measured by using an additional GPIO pin. The pin was turned on just before and turned off just after the inference process. The time between toggling of the pin was then measured. Namely, this time roughly represents the inference time, or the latency. We measures roughly 380 ms.

5 CONCLUSION

In this thesis, a specific application of the edge ML technology was presented, in particular the AI machine vision implementation of the video Intercom system with human presence detection. As is assumed in the opening chapters, the problem was relatively simple and thus applicable in the constrained embedded environment. Local offline AI computation on a general MCU device is manageable and achievable for tasks, such as image classification and word detection, but also suitable for preprocessing and initiation of more complex tasks, such as facial authentication.

The testing results show that the RAM is the biggest bottleneck for such applications, but it also depends on the task and available hardware. The methods for compressing the models and accelerating typical algorithms are relatively effective and provide a reasonably small code size for the flash memory capacity of the contemporary MCU devices. The MCU device used in testing was evaluated to be capable for needed computational complexity, achieving decent results in terms of latency and power consumption. The LCD power consumption prevailed the overall consumption, making the AI computation consumption unnoticeable. The accuracy of the model was also acceptable. TFLM framework proved to be simple to use and to implement.

The application for the edge AI seems to be endless, and future work would be to research, explore, and develop more effective model architectures and find new ways of reducing model size and complexity. Namely, the effectiveness of a particular method greatly depends on the application and underlying hardware platform. Newly available hardware architectures are built just for this purpose and should be taken under consideration in future work.

References

[1] Arora T., ARM, Accelerating Machine Learning Compute for IoT and Embedded Market white paper. ARM Ltd, 2020.

[2] Arunkumar J. An Introduction to TinyML. Available at: https://towardsdatascience.com/an-introduction-to-tinyml-4617f314aa79 [28.9.2021]

[3] Banbury C.R., Reddi V.J., Lam M., Fu W., Fazel A., Holleman J., Huang X., Hurtado R., Kanter D., Lokhmotov A., Patterson D.A., Pau D., Seo J., Sieracki J., Thakker U., Verhelst M., Yadav P., Benchmarking TinyML Systems: Challenges and Direction. Computing Research Repository, abs/2003.04821, (2020)

[4] Chowdhery A., Warden P., Shlens J., Howard A., Rhodes R., Visual Wake Words Dataset. Computing Research Repository, abs/1906.05721, (2019)

[5] Cortex-M4. Available at: https://developer.arm.com/ip-products/processors/cortex-m/cortex-m4

[6] Courbariaux M., Bengio Y., David J., BinaryConnect: Training Deep Neural Networks with binary weights during propagations. Computing Research Repository, abs/1511.00363, (2015)

[7] David R., Duke J., Jain A., Reddi V.J., Jeffries N., Li J., Kreeger N., Nappier I., Natraj M., Regev S., Rhodes R., Wang T., Warden P., Tensorflow Lite Micro: Embedded machine learning on TinyML systems. Computing Research Repository, abs/2010.08678, (2020)

[8] Gill B., Smith D., The Edge Completes the Cloud: A Gartner Trend Insight Report. Gartner: 14 September 2018

[9] Han S. Deep Compression and EIE. CVA group, Stanford University, 2015. Available at: https://web.stanford.edu/class/ee380/Abstracts/160106-slides.pdf [28.9.2021]

[10] Han S. Efficient Methods and Hardware for Deep Learning. Stanfor University, 2017. Available at: http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture15.pdf [28.9.2021].

[11] Han S., Liu X., Mao H., Pu J., Pedram A. Horowitz M.A., Dally W.J., EIE: Efficient Inference Engine on Compressed Deep Neural Network. Computing Research Repository, abs/1602.01528 (2016)

[12] Han S., Mao H., Dally W.J., Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding. 4th International Conference on Learning Representations 2016, San Juan, Puerto Rico, May 2-4, 1510.00149, (2016)

[13] Heinrich M.P., Blendowski M., Oktay O., TernaryNet: Faster Deep Model Inference without GPUs for Medical 3D Segmentation using Sparse and Binary Convolutions. Computing Research Repository, abs/1801.09449 (2018)

[14] Howard A.G., Zhu M., Chen B., Kalenichenko D., Wang W., Weyand T., Andreetto M., Adam H., MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. Computing Research Repository, abs/1704.04861, (2017)

[15] Khan A. Machine Learning Model Optimization for Intelligent Edge. Available at: https://medium.com/swlh/machine-learning-model-optimization-for-intelligent-edge-d0f400111002 [28.9.2021].

[16] Le J. The 5 Algorithms for Efficient Deep Learning Inference on Small Devices. 2019. Available at: https://heartbeat.fritz.ai/the-5-algorithms-for-efficient-deep-learning-inference-on-small-devices-bcc2d18aa806 [28.9.2021].

[17] LeCun Y., Denker J.S., Sola S.A., Optimal brain damage. In: Touretzky D. (editor), Advances in Neural Information Processing Systems, 1990. Morgan-Kaufmann, volume 2

[18] Lin S., Liu N., Nazemi M., Li H., Ding C., Wang Y., Pedram M., FFT-Based Deep Learning Deployment in Embedded Systems. Computing Research Repository, abs/1712.04910, (2017)

[19] Loukides M. TinyML: The challenges and opportunities of low-power ML applications. Available at: https://www.oreilly.com/radar/tinyml-the-challenges-and-opportunities-of-low-power-ml-applications/ [28.9.2021]

[20] Novac P.E., Hacene G.B., Pegatoquet A., Miramond B., Gripon V., Quantization and Deployment of Deep Neural Networks on Microcontrollers. Computing Research Repository, abs/2105.13331, (2021)

[21] Novatek, NT35510 Datasheet. Version 0.80, 2011.

[22] OmniVision, OV7670/OV7171 CMOS VGA CameraChip Advanced Information Datasheet. Sunnyvale, CA USA: Version 1.4, 2006.

[23] Pratt H. Williams B.M., Coenen F., Zheng Y., FCNN: Fourier Convolutional Neural Networks. In: Ceci M. Hollmén J., Todorovski Lj. Vens C., Dzeroski S. (Editor), Machine Learning and Knowledge Discovery in Databases - European Conference, Skopje, Macedonia, September 18-22, 2017, Proceedings, Part I, Springer 2017, page 786-798, vol. 10534

[24] Shelby Z. uTensor and Tensor Flow Announcement. Available at: https://os.mbed.com/blog/entry/uTensor-and-Tensor-Flow-Announcement/ [28.9.2021]

[25] Skillman A., Edsö T., A Technical Overview of Cortex-M55 and Ethos-U55: Arm s Most Capable Processors for Endpoint AI. ARM Ltd, 2020.

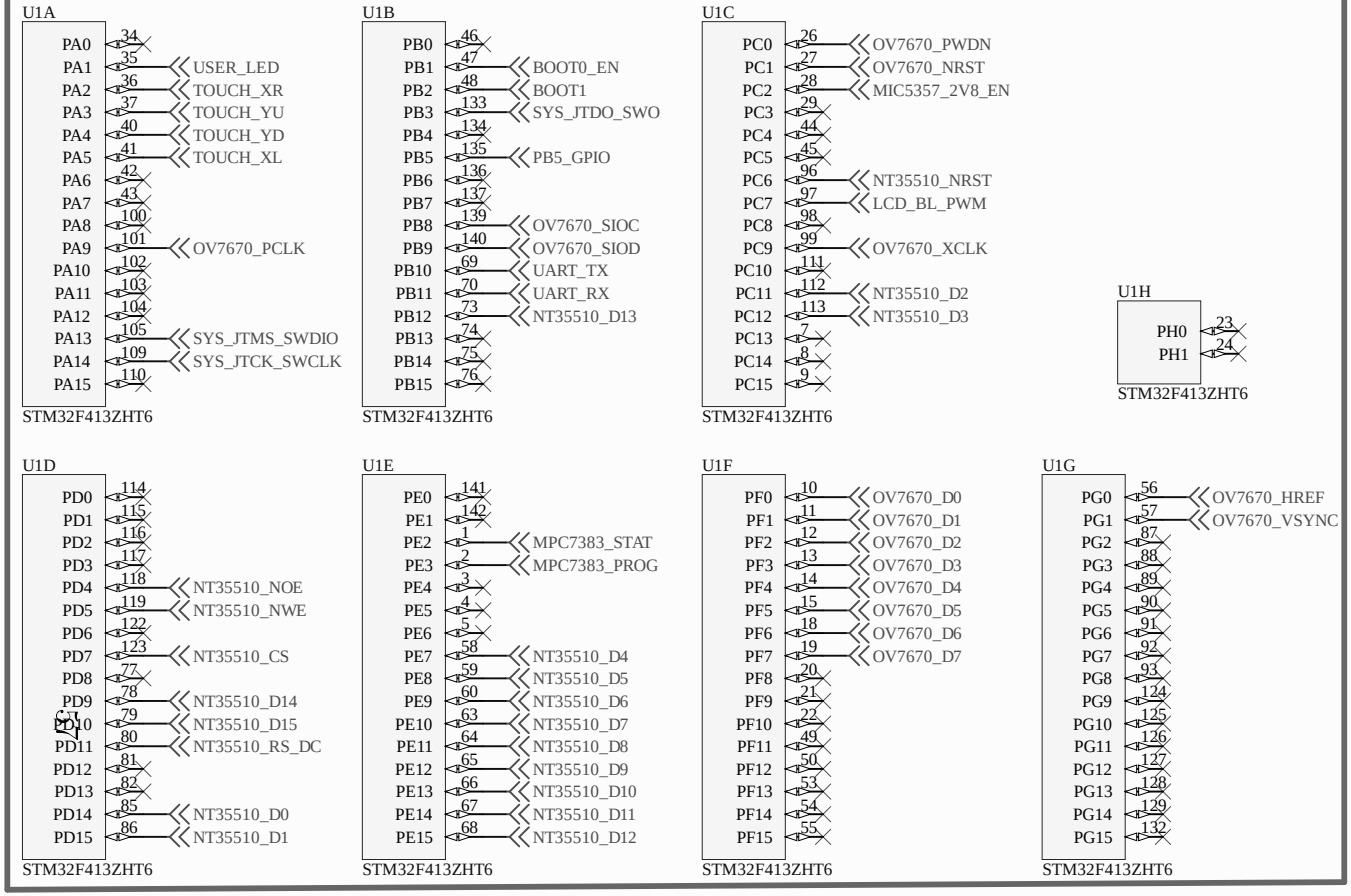
[26] Vogel S, Raghunath R. B., Guntoro A., Van Laerhoven K., Ascheid G., Bit-Shift-Based Accelerator for CNNs with Selectable Accuracy and Throughput. 2019 22nd Euromicro Conference on Digital System Design (DSD), 2019, page 663-667

[27] Wu H., Judd P., Zhang X., Isaev M., Micikevicius P., Integer Quantization for Deep Learning Inference: Principles and Empirical Evaluation. Computing Research Repository, abs/2004.09602, (2020)

[28] Zhu C., Han S., Mao H., Dally W.J., Trained Ternary Quantization. Computing Research Repository, abs/1612.01064 (2016)

[29] Zisserman A. 2D Fourier transforms and applications. Available at: https://www.robots.ox.ac.uk/ az/lectures/ia/lect2.pdf [28.9.2021]

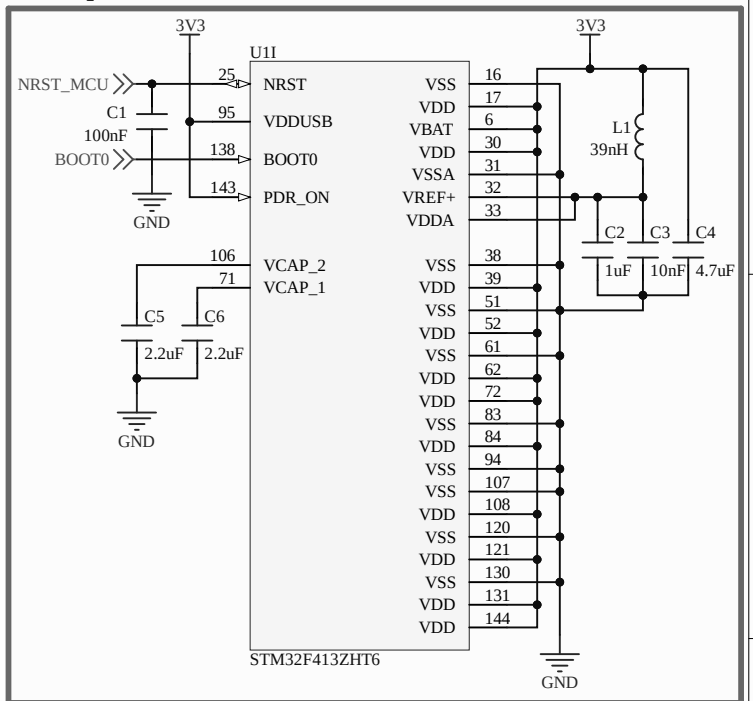
MCU GPIO ports



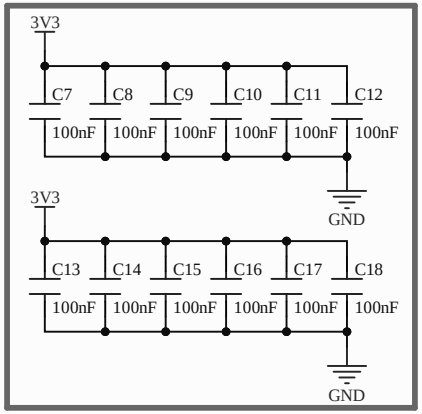
Net name	Pin	Mode	Pull-Up-Down	
BOOT0_EN	PB1	Output, PP	No	Active high
NT35510_NRST	PC6	Output, OD	No	Active low
LCD_BL_PWM	PC7	TIM3 PWM CH2, OD	No	
OV7670_NRST	PC1	Output, OD	No	Active low
OV7670_PWDN	PC0	Output PP	No	Active high
MIC5357_2V8_EN	PC2	Output PP	No	Active high
MCP73831_PROG	PE3	Output OD	No	Active low
MCP73831_STAT	PE2	Input	No	Tri-state
TOUCH_XR	PA2	ADC1_IN2 & Output PP	No	High/Analog
TOUCH_YU	PA3	ADC1_IN3 & Output PP & EXTI3	Yes	High/Analog
TOUCH_YD	PA4	Output & Input	No	Low/Open
TOUCH_XL	PA5	Output & Input	No	Low/Open
USER_LED	PA1	Output pp	No	Active high

Net name	Pin	Mode
USART6_TX/RX	PB10/11	USB serial comm & bootloader interface
OV7670_SIOC/D	PB8/9	I2C1 SCL/SDA - SCCB
OV7670_XCLK	PC9	RCC_MCO2
OV7670_PCLK	PA9	TIM1_CH2 Trigger source
OV7670_VSYNC	PG1	EXTI1
OV7670_HREF	PG0	EXTI0

MCU power, boot & reset



MCU decoupling capacitors



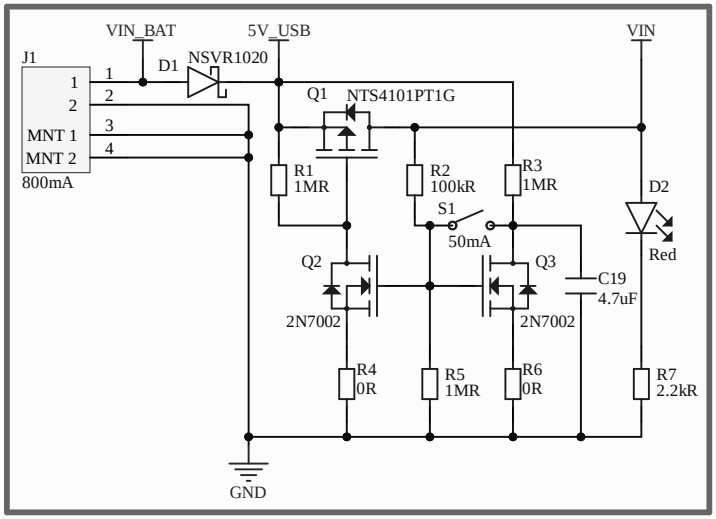
A: Schematics

Title	***ARM based Smart Video Intercom System***		
Size: A4	Number: 1	Revision: 0.1	
Date: 5/23/2022		Sheet 1 of 2	

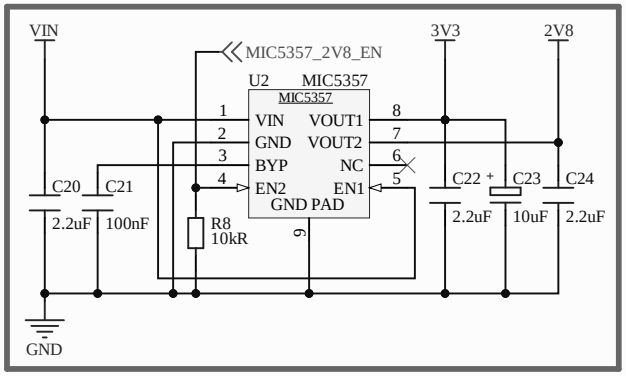
FERI UM
Koroška cesta 46
2000 Maribor
Slovenia
mario.gavran@student.um.si

University of Maribor
Faculty of Electrical Engineering and Computer Science

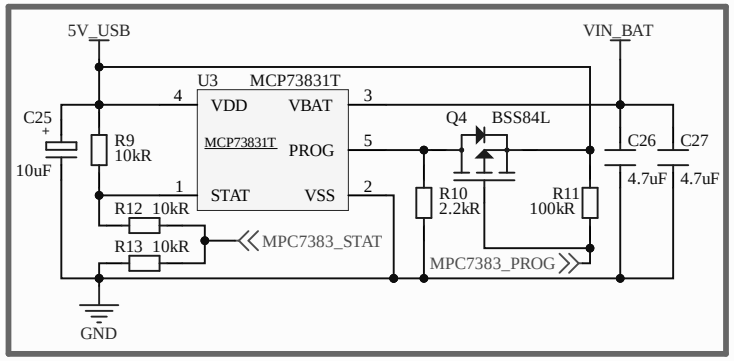
Battery connector & push button latch



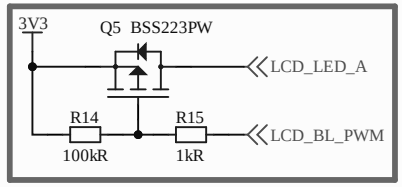
LDO power supply



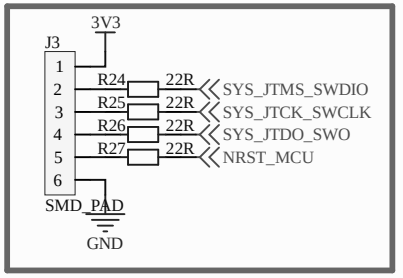
Battery charger



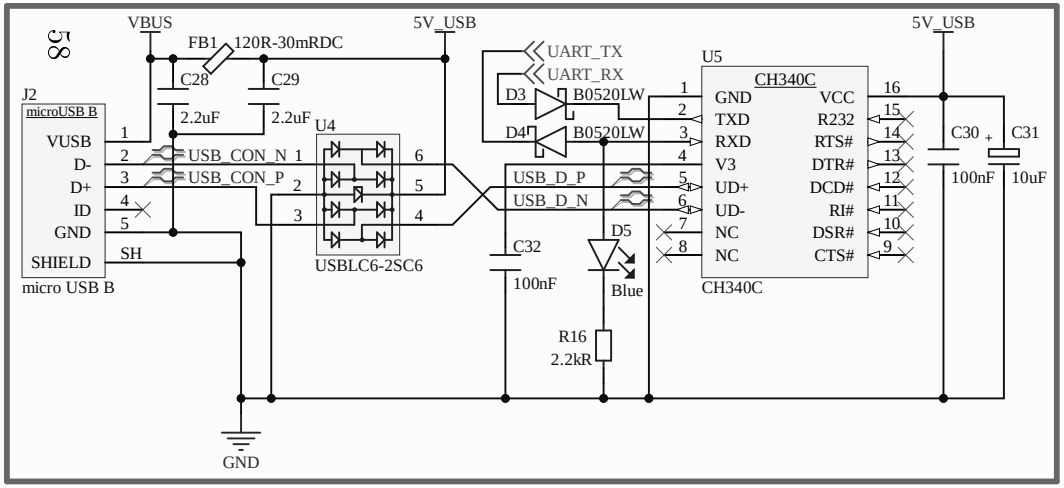
PWM LCD backlight control



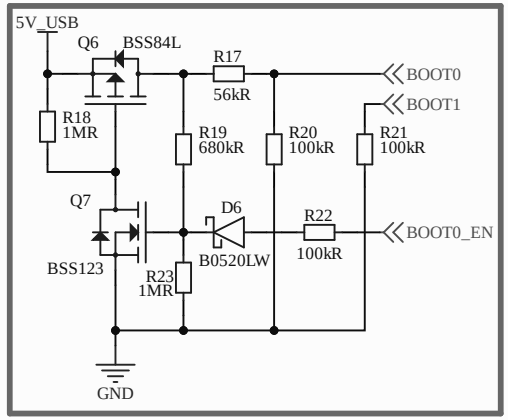
SWD connector



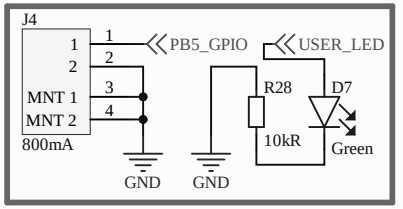
USB to serial communication



BOOT0 select latch



GPIO connector and user LED



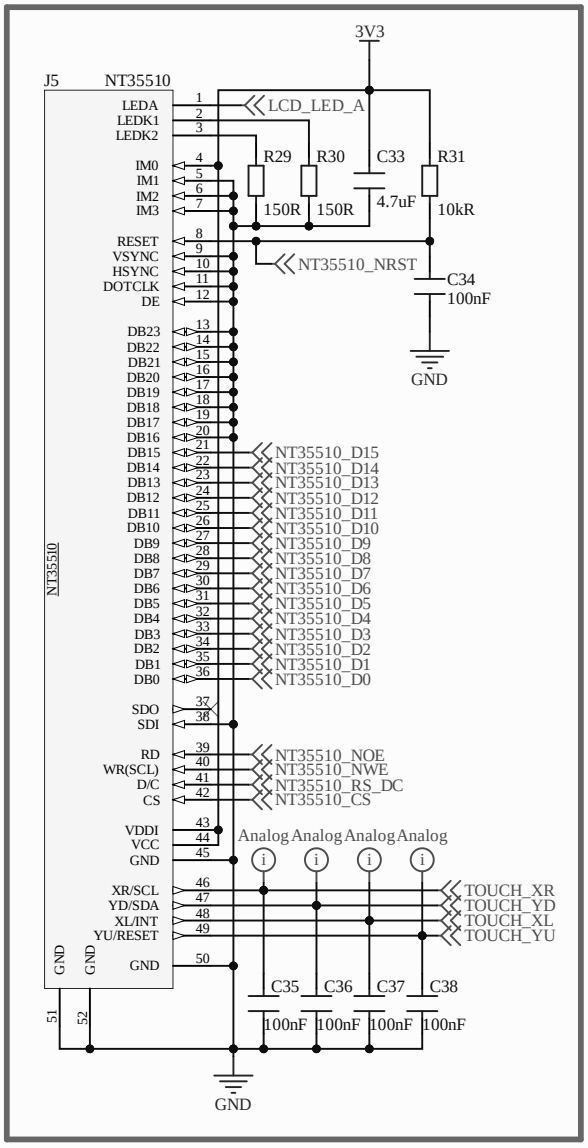
A: Schematics

Title	***ARM based Smart Video Intercom System***		
Size: A4	Number: 1	Revision: 0.1	
Date: 5/23/2022		Sheet 2 of 3	

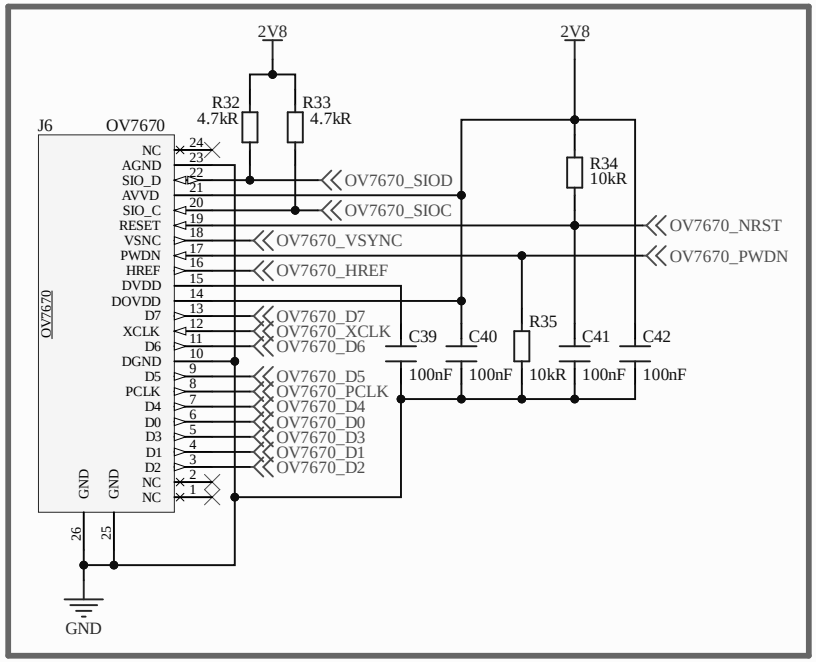
FERI UM
Koroška cesta 46
2000 Maribor
Slovenia
mario.gavran@student.um.si

University of Maribor
Faculty of Electrical Engineering and Computer Science

WVGA LCD controller



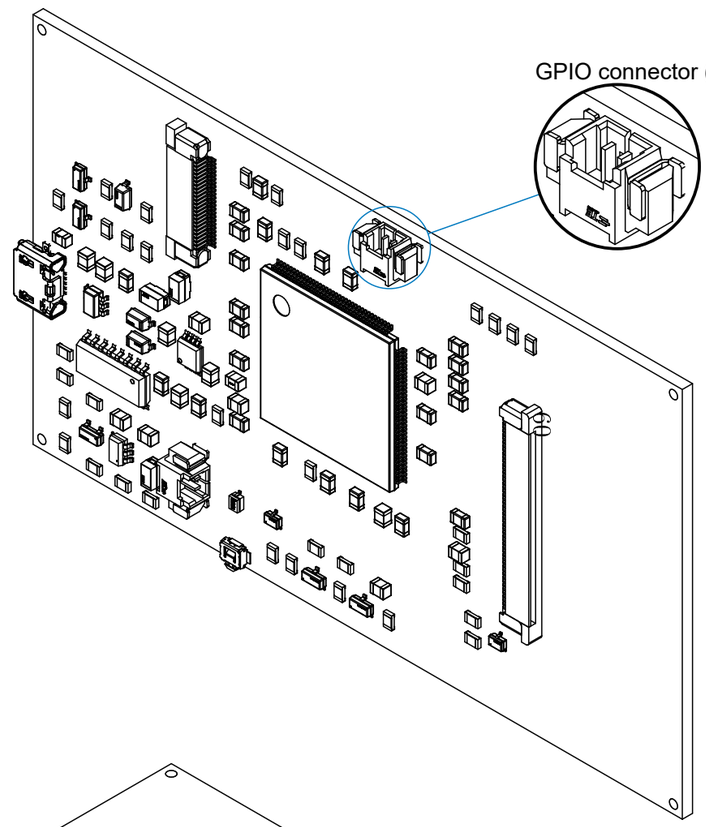
VGA Camera

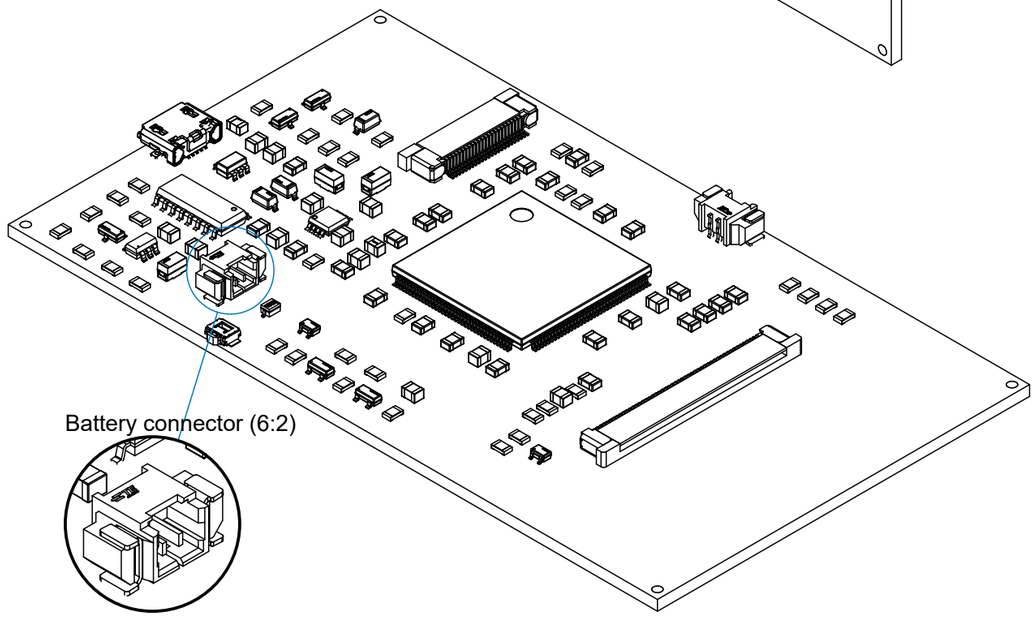


A: Schematics

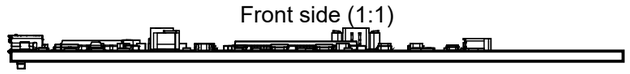
Title ***ARM based Smart Video Intercom System***			*FERI UM* *Koroška cesta 46* *2000 Maribor* *Slovenia* *mario.gavran@student.um.si*
Size: A4	Number: 1	Revision: 0.1	
Date: 5/23/2022		Sheet 3 of 3	

University of Maribor
Faculty of Electrical Engineering and Computer Science





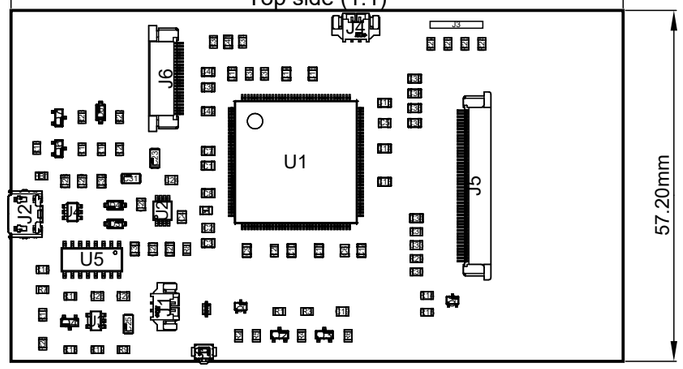
Front side (1:1)



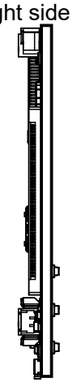
99.80mm

Left side (1:1)

Top side (1:1)

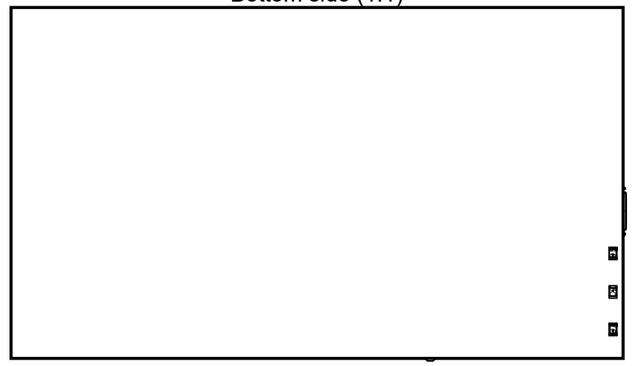


Right side (1:1)



Back side (1:1)

Bottom side (1:1)

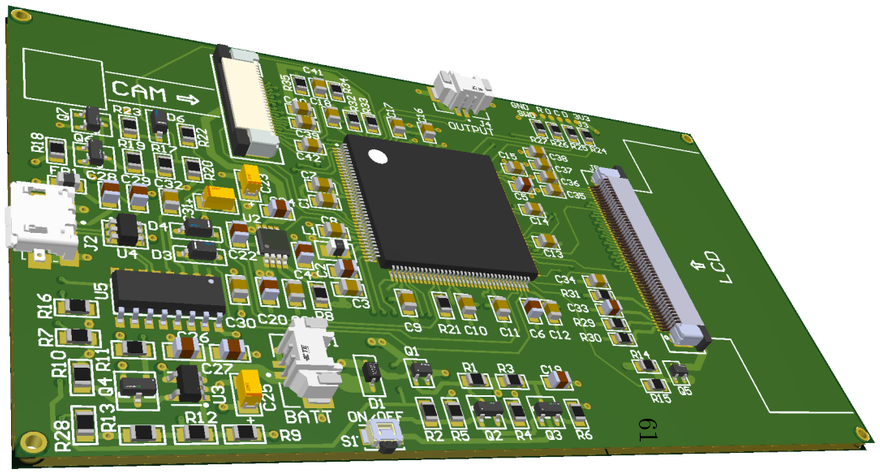


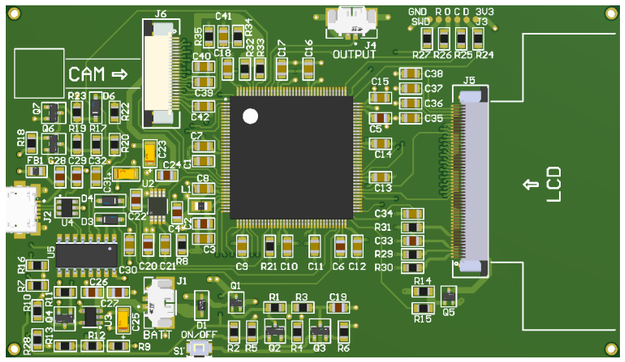
B: Assembly and PCB drawings

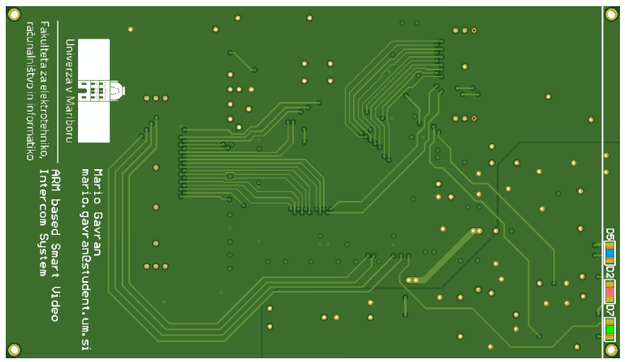
Title:	*ARM based Smart Video Intercom System*		
Size: A3	Number: 1	Revision: 0.1	
Date: 5/23/2022		Sheet: 1 of 3	

UM FERI
Koroška cesta 46
2000 Maribor
Slovenia
mario.gavran@student.um.si

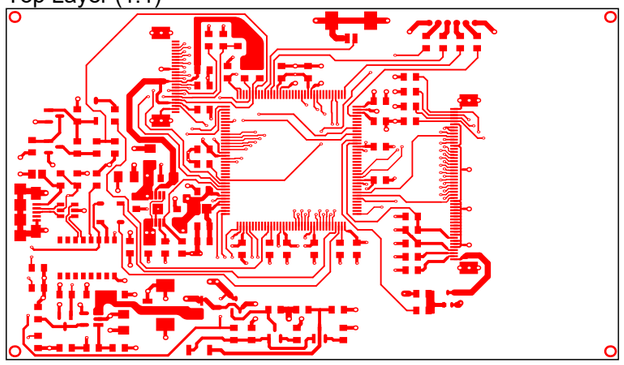
Univerza v Mariboru
Fakulteta za elektrotehniko, računalništvo in informatiko



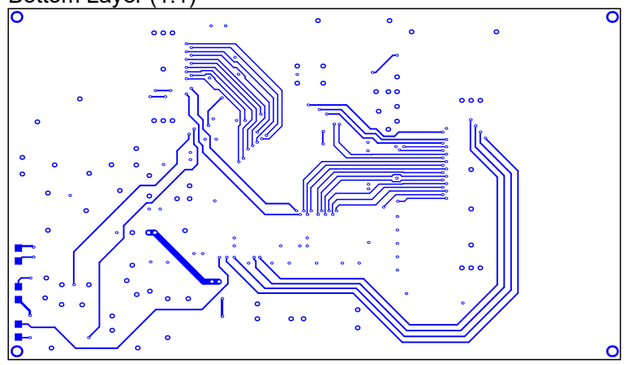




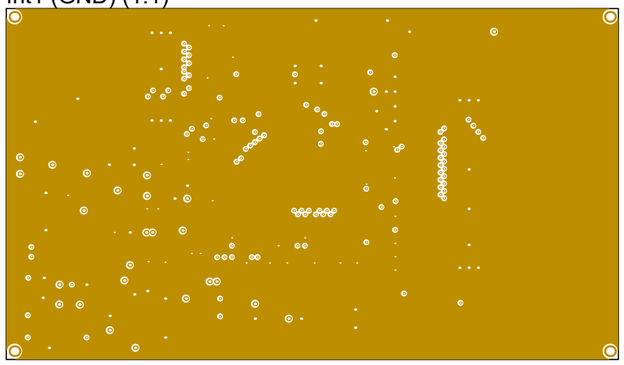
Top Layer (1:1)



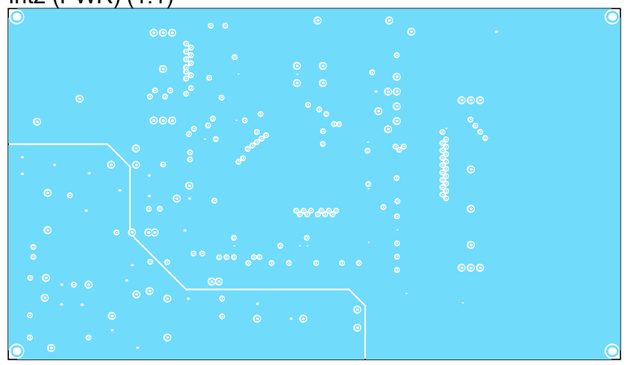
Bottom Layer (1:1)



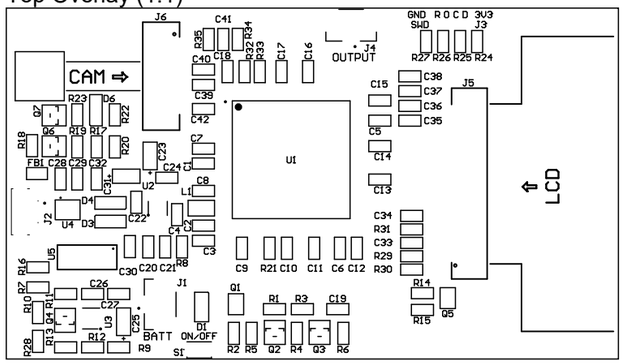
Int1 (GND) (1:1)



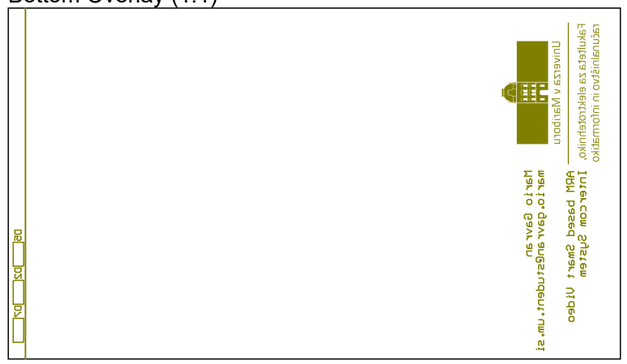
Int2 (PWR) (1:1)



Top Overlay (1:1)



Bottom Overlay (1:1)



B: Assembly and PCB drawings

Title:	*ARM based Smart Video Intercom System*		
Size: A3	Number: 1	Revision: 0.1	
Date: 5/23/2022		Sheet: 2 of 3	

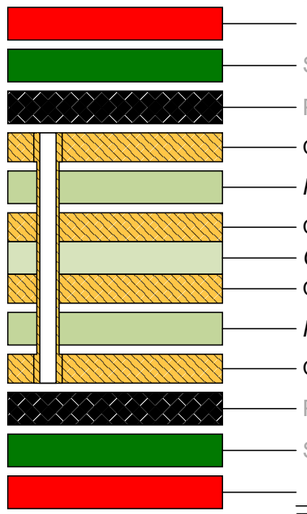
UM FERI
Koroška cesta 46
2000 Maribor
Slovenia
mario.gavran@student.um.si

Univerza v Mariboru
Fakulteta za elektrotehniko, računalništvo in informatiko

Bill Of Materials

Line #	Designator	Comment	Quantity
1	C1, C7, C8, C9, C10, C11, C12, C13, C14, C15, C16, C17, C18, C21, C30, C32, C34, C35, C36, C37, C38, C39, C40, C41, C42	100nF	25
2	C2	1uF	1
3	C3	10nF	1
4	C4, C19, C26, C27, C33	4.7uF	5
5	C5, C6, C20, C22, C24, C28, C29	2.2uF	7
6	C23, C25, C31	10uF	3
7	D1	NSVR1020	1
8	D2	Red	1
9	D3, D4, D6	B0520LW	3
10	D5	Blue	1
11	D7	Green	1
12	FB1	120R-30mRDC	1
13	J1, J4	800mA	2
14	J2	micro USB B	1
15	J3	SMD_PAD	1
16	J5	NT35510	1
17	J6	OV7670	1
18	L1	39nH	1
19	Q1	NTS4101PT1G	1
20	Q2, Q3	2N7002	2
21	Q4, Q6	BSS84L	2
22	Q5	BSS223PW	1
23	Q7	BSS123	1
24	R1, R3, R5, R18, R23	1MR	5
25	R2, R11, R14, R20, R21, R22	100kR	6
26	R4, R6	0R	2
27	R7, R10, R16	2.2kR	3
28	R8, R9, R12, R13, R28, R31, R34, R35	10kR	8
29	R15	1kR	1
30	R17	56kR	1
31	R19	680kR	1
32	R24, R25, R26, R27	22R	4
33	R29, R30	150R	2
34	R32, R33	4.7kR	2
35	S1	50mA	1
36	U1	STM32F413ZHT6	1
37	U2	MIC5357	1
38	U3	MCP73831T	1
39	U4	USBLC6-2SC6	1
40	U5	CH340C	1

Layer Stack Legend



Material	Layer	Thickness	Dielectric Material	Type	Gerber
	Top Overlay			Legend	GTO
Surface Material	Top Solder	0.02mm	SM-001	Solder Mask	GTS
PbSn	Top Surface Finish	0.02mm		Surface Finish	
CF-004	Top Layer	0.04mm		Signal	GTL
Prepreg		*0.20mm*	*PP-017*	*Dielectric*	
CF-004	Int1 (GND)	0.02mm		Signal	G1
Core		*1.06mm*	*Core-039*	*Dielectric*	
CF-004	Int2 (PWR)	0.02mm		Signal	G2
Prepreg		*0.20mm*	*PP-017*	*Dielectric*	
CF-004	Bottom Layer	0.04mm		Signal	GBL
PbSn	Bottom Surface Finish	0.02mm		Surface Finish	
Surface Material	Bottom Solder	0.02mm	SM-001	Solder Mask	GBS
	Bottom Overlay			Legend	GBO

Total thickness: 1.65mm

B: Assembly and PCB drawings

Title: *ARM based Smart Video Intercom System*		
Size: A3	Number: 1	Revision: 0.1
Date: 5/23/2022		Sheet: 3 of 3

UM FERI
Koroška cesta 46
2000 Maribor
Slovenia
mario.gavran@student.um.si

Univerza v Mariboru
Fakulteta za elektrotehniko, računalništvo in informatiko