UPPSALA UNIVERSITET

IT mINS 25 001

Degree project 30 credits

February 2025

Strength Reduction Techniques in Compilers for Optimizing Inference on Edge Devices

Alessio Petruccelli

Master's Programme in Embedded Systems

UPPSALA UNIVERSITET

Strength Reduction Techniques in Compilers for Optimizing Inference on Edge Devices

Alessio Petruccelli

Abstract

Neural networks are widely used nowadays and they start finding their applications on resource-constrained devices. This thesis will address the challenge of optimizing artificial intelligence inference focusing on compiler techniques, particularly strength reduction and input data pre-processing. This thesis explores an alternative to the traditional multiply-accumulate operations with more efficient operations, such as shift and add operations.

Even though the strength reduction optimization is a well-known technique in the field of compiler optimization, its application to neural network inference on embedded systems has not been extensively studied. The evaluation is conducted across multiple embedded architectures, including ARM Cortex-M0, AVR, MSP430, and RISC-V, utilizing hardware emulation through QEMU to benchmark performance. Experimental results demonstrate that strength reduction can significantly decrease the computational overhead of neural network inference at the cost of increased memory requirements. The findings highlight the potential of compiler-assisted optimizations in enabling efficient AI inference on edge devices, paving the way for improved TinyML applications.

Faculty of Science and Technology

Uppsala University, Uppsala

Supervisor: Stefanos Kaxiras Subject reader: Joachim Parrow

Examiner: Bengt Jonsson

Contents

List of Figures

List of Tables

1 Introduction

In the rapidly evolving era of Artificial Intelligence (AI) and Machine Learning (ML), the computation is moving from the cloud to the edge. This is comporting a new set of challenges in the field of embedded systems, such as the need of smart devices able to compute AI algorithms in a fast and efficient way without losing their characteristic features such as low power consumption and small size. However, the implementation of high computational demanding algorithms on embedded systems is not trivial, and it requires a deep knowledge of the hardware and software architecture of the target device. Moreover, the computational power of embedded systems is limited, and the execution of AI algorithms can be slow and inefficient in terms of time and energy consumption.

The field of Tiny Machine Learning (TinyML) addresses these challenges by focusing on the development of machine learning models and algorithms specifically designed for resource-constrained embedded systems. TinyML aims to enable the deployment of AI applications on devices with limited computational capabilities, such as microcontrollers and Internet of Things (IoT) devices. By leveraging techniques like model quantization, compression, and hardware acceleration, TinyML enables efficient execution of AI algorithms on embedded systems, minimizing the computational and energy requirements. The TinyML community has developed various frameworks and tools, such as TensorFlow Lite for Microcontrollers, to facilitate the development and deployment of TinyML models. However, there are still many open challenges in the field of TinyML, such as the optimization of neural network models for embedded systems, the development of efficient inference algorithms, and the integration of hardware accelerators for AI computation.

The challenge in this thesis is to perform the neural networks' inference on embedded systems by substituting the traditional operation Multiply-Accumulate (MAC) with a more efficient operation. MAC operations are computationally intensive and can overwhelm the limited resources of a microcontroller. In this work, we propose a promising solution by substituting those expensive operations with more efficient ones using the *strength reduction* optimization technique. The strength reduction optimization is a compiler optimization technique that replaces expensive operations with cheaper ones, reducing the overall computational cost of the program. In the context of neural network inference, we can apply the strength reduction optimization to replace the MAC operations with more efficient operations, such as shift and add operations. This optimization can significantly reduce the computational cost of the neural network inference, making it more suitable for resource-constrained embedded systems. Even tough the strength reduction optimization is a well-known technique in the field of compiler optimization, its application to neural network inference on embedded systems has not been extensively studied. In this thesis, we investigate the application of the strength reduction optimization to neural network inference on embedded systems, and we evaluate its impact on the computational cost and performance of the inference process.

2 Background

In this section, we will introduce the concepts of machine learning, neural networks, and compiler optimizations. We will also introduce the concept of TinyML and the deployment of machine learning models on microcontrollers. Finally, we will introduce the TensorFlow Lite framework and the compiler optimizations that we will use in this thesis.

2.1 Neural Networks

Neural networks are a computational abstraction that tries to mimic the way the human brain works. The basic computational unit of the brain is the neuron, which is a cell that receives electrical signals from other neurons and sends signals to other neurons. These electrical signals travel through the neuron's dendrites, which are the input channels, and are processed in the neuron's body, which is the processing unit. The neuron then sends the processed signal through the axon, which is the output channel, to other neurons. The neuron can be in two states: active or inactive. When the neuron is active, it sends a signal to other neurons, and when it is inactive, it does not send a signal. The strength of the signal is determined by the number of electrical impulses that the neuron receives from other neurons which are regulated by the synaptic strengths that can be excitatory or inhibitory. If the sum of the signals that the neuron receives is greater than a certain threshold, the neuron *fires* and sends the signal to other neurons.

The neurons in a neural network are modelled with the biological functioning in mind, the *firing* of a neuron is modelled with an **activation function**, which is a mathematical function that determines if the neuron is active or inactive. The synaptic strengths are modelled as **weights** w, which are real numbers that represent the strength of the signal that the neuron sends to other neurons. The threshold at which the neuron becomes active is modelled as a **bias** b, which is a real number that represents the threshold at which the neuron becomes active. In Figure 1 we can see a graphical representation of a single neuron in a neural network. The output y of a neuron is computed as follows:

$$y = f\left(\sum_{i=1}^{n} w_i x_i + b\right) \tag{1}$$

where f is the activation function. The activation function is a continuous function that maps the output of the neuron to a value, generally, between

0 and 1. The most common activation functions are the Sigmoid function, the hyperbolic tangent $\tanh$ function, and the Rectified Linear Unit (ReLU) function [6].

Similarly to our brain, a neural network is composed of layers of neurons. The external stimuli are propagated from the first set of neurons, called the input layer, to the last set of neurons, called the output layer. The layers between the input and the output layers are called hidden layers. The layering of the neurons allows the network to learn complex patterns by extracting features from the input data and passing them to the next layer. Every layer extracts different features from the data and the output layer, the last, is used to make a prediction.

In order to make our neural network able to make predictions and learn these patterns, the network goes through a process called *training*. Neural networks involve two main processes: training and inference.

- Training: this is the process where the network learns from the data. The training process involves feeding the network with a set of input data with the corresponding labels, the labels are the expected output of the network given a certain input. This learning process is called *supervised learning*. Based on a loss function, i.e. the Mean Squared Error (MSE), the network can compute the error between the predicted output and the expected output. This error is then used to update the weights and the biases of the network using an optimization algorithm, i.e. the Stochastic Gradient Descent (SDG) algorithm. The training is repeated for a certain number of iterations, called epochs, until the network converges to a solution. During the training process two possible problems can arise:
  - Overfitting: the network learns the training data too well and is not able to generalize to new data.
  - Underfitting: the network is not able to learn the training data and is not able to generalize to new data.
- Inference: this is the process where the network makes predictions on new data. In this case, the network parameters are fixed and the network is able to give a prediction based on an unseen input data.

The iterative process of training is computationally intensive and requires a large amount of data to be effective. In addition, it requires a tremendous amount of time which is unfeasible for small devices. In this thesis, we will

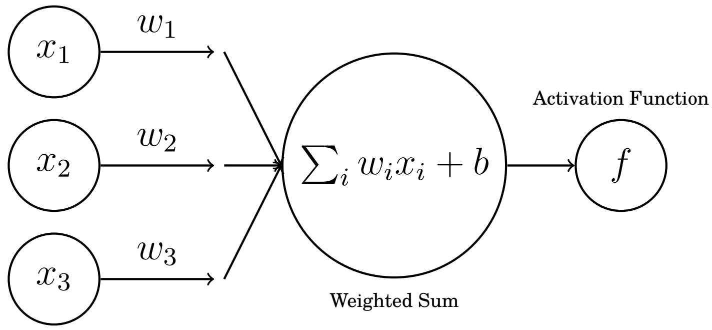


Figure 1: A single neuron in a neural network. The neuron receives signals from other neurons, computes a weighted sum of the signals, and applies an activation function to determine if the neuron is active or inactive.

focus on two types of neural networks: Multi-Layer Perceptrons (MLPs) and Convolutional Neural Networks (CNNs).

2.1.1 Multi Layer Perceptron

The Multi-Layer Perceptron (MLP), also known as *feedforward neural network* is the simplest type of neural network. It consists of an input layer, one or more hidden layers, and an output layer. Every neuron inside the network is connected to every other neuron in the next layer, in fact these networks are also called *fully connected networks*. These neurons are able to make decisions based on the weighted sum of the inputs and the activation function applied to the sum as shown in Equation (1). If we look at function, the output generated by a single neuron is a single value, this output is then repeated and provided as input to each neuron in the next layer. The adding of layers allows the network to perform more complex and abstract decisions finding different patterns on the input data [7]. The most common use case of MLPs is classification tasks, where the network is trained to categorize the input data into different classes.

2.1.2 Convolutional Neural Networks

Convolutional Neural Network (CNN) are a type of neural network designed to work with grid-like topology data, i.e. images that can be represented as a 2D grid of pixels [8]. The CNNs are so called because they implement the convolution operation instead of the general matrix multiplication. The convolution operation is a mathematical operation that takes two functions and produces a third function that represents the amount of overlap between the

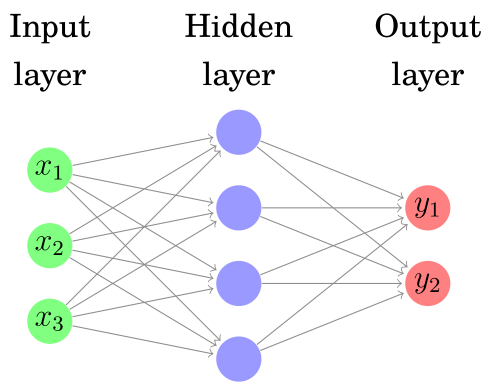


Figure 2: A Multi-Layer Perceptron (MLP) with an input layer, a hidden layer, and an output layer.

two functions. In the continuous time domain, the convolution operation is defined as:

$$(f * g)(t) = \int_{-\infty}^{\infty} f(\tau)g(t-\tau)d\tau \tag{2}$$

Where f and g are the two functions that are being convolved, and $*$ is the convolution operator. In the discrete time domain, the convolution operation is defined as:

$$(f * g)[n] = \sum_{m=-\infty}^{\infty} f[m]g[n-m] \tag{3}$$

In the context of CNNs, the first function is referred as the input, the second as the kernel or filter and the result is called *feature map*. Let us suppose we apply the convolution operation on an image, let be I the input image and K the kernel, the output of the convolution operation is:

$$(I * K)[i,j] = \sum_{m=-\infty}^{\infty} \sum_{n=-\infty}^{\infty} I[m,n]K[i-m,j-n] \tag{4}$$

We can picture the convolution operation as a sliding window that moves over the input image by a certain number of pixels (stride). At each step, the pixels of the input image that are covered by the kernel (the window) are multiplied by the kernel values and summed up to produce the output pixel. We must denote that the most available machine learning libraries (TensorFlow included), implement what is called **cross-correlation** instead of convolution, in fact the kernel is not flipped before the multiplication.

The CNNs are composed of three main types of layers: convolutional layers, pooling layers, and fully connected layers. The convolutional layers extract features from the input image, and the resulting feature map is then passed to the pooling layers, which reduce its size. Finally, the fully connected layers utilize the features extracted by the convolutional and pooling layers to make a prediction.

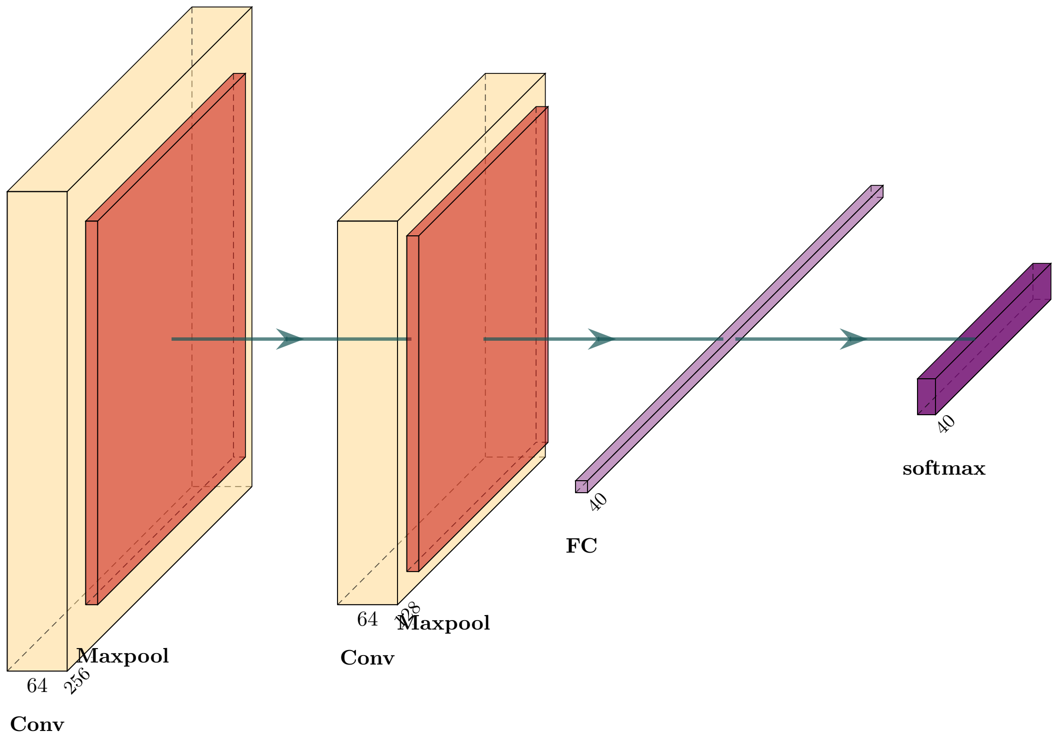


Figure 3: A Convolutional Neural Network (CNN) with a convolutional layer, a max-pooling layer, and a fully connected layer. The numbers here have no meaning, they are just placeholders.

2.1.3 Depthwise Separable Convolutional Neural Networks

Depthwise Separable Convolutional Neural Network (DS-CNN) are a type of neural network that is designed to reduce the number of parameters in a CNN. DS-CNNs are composed of depthwise convolutional layers, pointwise convolutional layers, and fully connected layers. The depthwise convolutional layers performs the convolution operation independently on each channel of the input, i.e. let us consider an RGB image. The depthwise convolutional layer will perform the convolution operation on the R, G, and B channels separately, therefore we will have three different kernels for each colour of the image. The resulting feature map, will be the indipendently combined feature maps for each colour.

The pointwise convolutional layers perform the convolution on the combined feature map of the depthwise convolutional layers projecting the output to a different space [9]. Referring to the example, the pointwise convolutional layer will take the three feature maps and combine them into a single feature map by using a 1x1 kernel with a depth equal to the number of channels of the input image, R, G, and B.

The additional layers in a DS-CNN are used to reduce the number of parameters in the network, which can lead to faster inference times and lower memory requirements. DS-CNNs are widely used in applications where the computational resources are limited, such as mobile devices and embedded systems. Let us suppose we apply the convolution operation on a feature map obtained previously from a classical convolutional layer, let be F the feature map, K the filter of the depthwise layer and $\hat{K}$ the filter of the pointwise convolutional layer. Let $O_{h,w,d}$ be the output of the depthwise separable convolutional layer and $\hat{O}_{h,w,d}$ the output of the pointwise convolutional layer, where h, w, d are respectively *height, width and depth* of an image. The operations involved in a depthwise separable convolutional layer are:

$$O_{h,w,d} = \sum_{m=-\infty}^{\infty} \sum_{n=-\infty}^{\infty} F[i,j,d]K[i-m, j-n, d] \tag{5}$$

for the depthwise convolutional layer, and:

$$\hat{O}_{h,w,d} = \sum_{m=-\infty}^{\infty} \sum_{n=-\infty}^{\infty} \sum_{t=-\infty}^{\infty} O[i,j,k]\hat{K}[i-m, j-n, k-t] \tag{6}$$

for the pointwise convolutional layer [10].

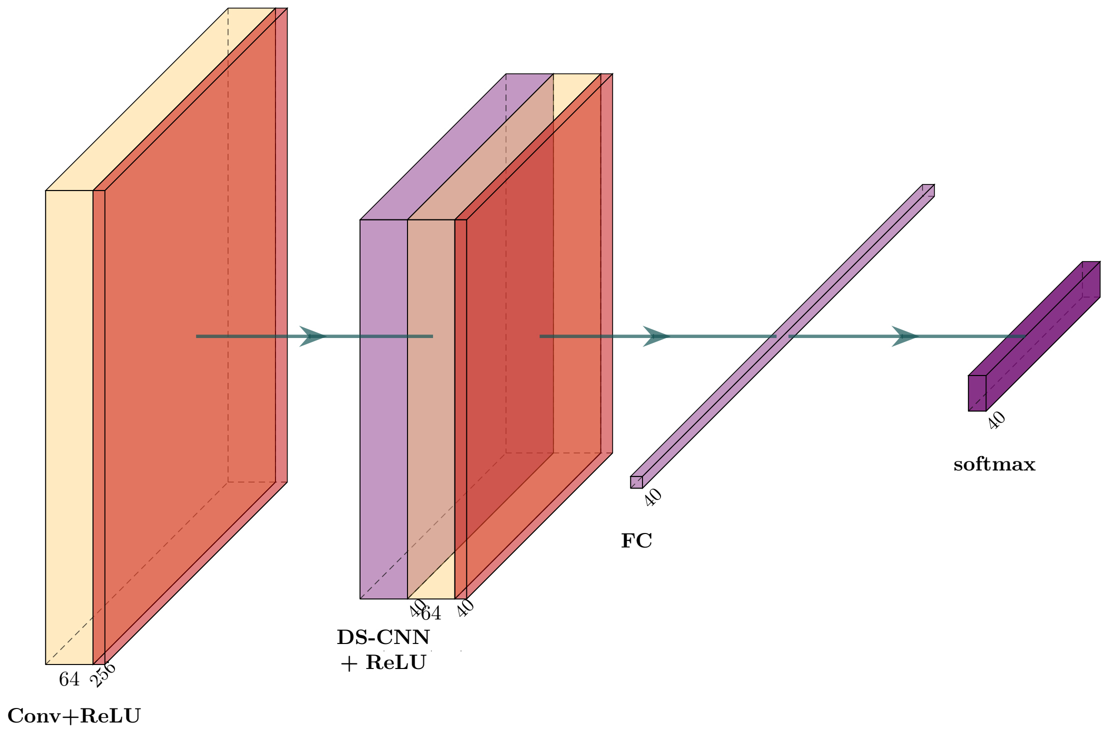


Figure 4: A Depthwise Separable Convolutional Neural Network (DS-CNN) with a first convolutional layer, then a depthwise convolutional layer, a pointwise convolutional layer and a fully connected layer. The numbers here have no meaning, they are just placeholders.

2.2 Tiny Machine Learning

Tiny Machine Learning (TinyML) is the field of machine learning that enables the deployment of machine learning models on small, low-power devices, such as microcontrollers. TinyML is used in a wide range of applications, such as predictive maintenance, anomaly detection, and environmental monitoring. TinyML models are typically trained on a powerful server or cloud platform and then deployed to a microcontroller for inference. Migrating the inference from the cloud to the edge enabled new challenges and opportunities to deploy small sized models that can run on microcontrollers with limited resources. This led to the development of new techniques for training and deploying machine learning models on resource-constrained devices such as Neural Architecture Search (NAS), Quantization, and Pruning [11]. In this thesis, we will focus on the deployment of machine learning models on microcontrollers architectures, applying further optimizations on the quantized models.

2.2.1 Quantization

Quantization is the software optimization technique that enables the deployment of neural networks to smaller architecture devices. Generally, the weights and the activations tensors are trained on servers which use 32-bit or 64-bit floating-point numbers. However, smaller architectures do not have the hardware resources to perform those floating point operations to run the inference. Moreover, the memory requirements of these models are too high to be stored in the limited memory of these devices. Quantization aids in reducing the memory requirements and the computational complexity of the model by converting the floating-point numbers to fixed-point numbers. The two main quantization techniques are Post-Training Quantization (PTQ) and Quantization-Aware Training (QAT) [12].

- Post-Training Quantization (PTQ): applies quantization to the weights and activations of the model after the model has been trained using floating point precision. The weights and activations are converted from floating-point numbers to fixed-point numbers, which reduces the memory requirements and computational complexity of the model.

- Quantization-Aware Training (QAT): involves training the model with quantization in mind. During training, the weights and activations of the model are quantized to fixed-point numbers, and the gradients are computed with respect to the quantized values. This allows the model to learn the quantized representations of the weights and activations during training.

In this thesis, we will focus on post-training quantization, which is the simplest and most widely used quantization technique.

2.2.2 Post-training Quantization

The Post-Training Quantization (PTQ) algorithm is a simple and effective way to quantize a model. It takes a pre-trained floating point model and converts it to a fixed-point model. The base idea is to minimize the loss of accuracy while reducing the memory requirements and computational complexity of the model. The base operation during the quantization operation is the *linear quantization* which is an affine mapping between the floating-point numbers and the fixed-point numbers [13]. The linear quantization is defined as follows:

$$r = S(q - Z) \tag{7}$$

with r a real number, q the quantized number, S the scale factor, and Z the zero point. When the linear quantization is applied, we want to map a floating point number to an integer range of numbers (q_{min}, q_{max}), where q_{min} and q_{max} are the minimum and maximum quantized values, the range is given by the N bit quantized representation chosen $[0, 2^N]$ for unsigned values and $[-2^{N-1}, 2^{N-1} - 1]$ for signed values. The scale factor and the zero point are computed as follows:

$$S = \frac{q_{max} - q_{min}}{r_{max} - r_{min}}$$

$$Z = round(q_{min} - \frac{r_{min}}{S})$$

where r_{min} and r_{max} are the minimum and maximum floating-point values that we are considering for the quantization. These parameters depend on the *quantization granularity* we are considering, in fact this can be *per-tensor* or *per-channel*.

- Per-tensor quantization: In this case, the scale factor and the zero point are computed for the entire tensor. This means that all the values in the tensor are quantized using the same scale factor and zero point. This is the simplest form of quantization and is widely used in practice.

- Per-channel quantization: In this case, the scale factor and the zero point are computed for each channel in the tensor. This means that each channel in the tensor has its own scale factor and zero point. This allows for more fine-grained quantization of the tensor and can lead to better accuracy than per-tensor quantization.

In this thesis, we will focus on per-tensor quantization, which is the quantization technique used in TensorFlow Lite [14], framework that we will use to quantize the models.

Per-tensor quantization

In per-tensor quantization, the scale factor and the zero point are computed for the entire tensor. This means that all the values in the tensor are quantized using the same scale factor and zero point. Here we show how a fully connected layer is quantized to an `int8` model using per-tensor quantization. The equation for a fully connected layer is:

$$Y = WX + b$$

where Y is the output tensor, W is the weight tensor, X is the input tensor, and b is the bias tensor. Applying the linear quantization in Equation (7) to each tensor we get:

$$\underbrace{S_Y(q_Y - Z_Y)}_{Y} = \underbrace{S_W(q_W - Z_W)}_{W} \cdot \underbrace{S_X(q_X - Z_X)}_{X} + \underbrace{S_b(q_b - Z_b)}_{b}$$

The input and the weights are both quantized to `int8`, so the resulting dot product is quantized to `int32`. Moreover, the bias-vector is quantized similarly to the `int32` data type. Since both the weights and the bias are quantized to a signed integer range, the zero points are set to 0. Setting $Z_W = Z_b = 0$ we get:

$$S_Y(q_Y - Z_Y) = S_W S_X(q_W q_X - Z_X q_W) + S_b(q_b)$$

The quantization scale of the bias-vector should be the same as the dot product quantization scale, so we set $S_b = S_W S_X$. We can now simplify the equation:

$$S_Y(q_Y - Z_Y) = S_W S_X(q_W q_X - Z_X q_W + q_b)$$

Since we want to compute the quantized output q_Y, we can isolate it:

$$q_Y = \frac{S_W S_X}{S_Y}(q_W q_X - Z_X q_W + q_b) + Z_Y$$

At this point, we have the term $q_b - Z_X q_W$ that can be precomputed since it does not involve any input variable, we call that q_{bias}.

$$\underbrace{q_Y}_{\texttt{int8}} = \underbrace{\frac{S_W S_X}{S_Y}}_{\text{rescale to }\texttt{int8}} \underbrace{(q_W q_X + q_{bias})}_{\text{32-bit accumulator}} + \underbrace{Z_Y}_{\texttt{int8}}$$

The term $S_W S_X / S_Y$ is the rescale factor, which is the constant used to convert the 32-bit accumulator to an `int8` value. This is only non-integer value in the equation, that can be expressed as a fixed-point multiplication[15].

Dynamic Range Quantization

Unlike the weights and the biases that are quantized to a fixed range, the activations can vary during the inference. This means that the activations can have a dynamic range that is not known a priori. To handle this there are two ways:

1. During training, the statistics of the activations are collected and then aggregated via Exponential Moving Average (EMA).
2. During inference, the statistics are gathered by running the model on a calibration dataset, which is a subset of the training dataset. Since the integer model should keep the same accuracy as the floating-point model, the chosen quantization parameters are the one that minimize the loss of accuracy which is measured by the Kullback-Leibler divergence between the floating-point and the quantized model [16].

Applying the dynamic range quantization to the activations, it is possible to retrieve the scale factor and the zero point for the activations.

2.2.3 TensorFlow Lite

TensorFlow Lite (TFLite) is a framework for deploying machine learning models on mobile and edge devices. It is part of TensorFlow [17], an open-source machine learning framework developed by Google. TFLite provides tools that enable the deployment of machine learning models on limited resource devices converting them to a FlatBuffer[18] format `.tflite`, a binary serialization format that is optimized for memory usage and speed. TFLite also provides a runtime that can run the models on a wide range of devices by providing an inference engine that can interpret the `.tflite` models and run the inference. TFLite supports a wide range of operations, although not all operations supported by TensorFlow do not have their counterpart in TFLite. Inside the TFLite framework there are two main components:

- TFLite Converter: The TFLite Converter is a tool that converts TensorFlow models to the TFLite format. The TFLite Converter takes a TensorFlow model in the `.pb` format and converts it to the TFLite format. The TFLite Converter also applies optimizations to the model, such as quantization and pruning, to reduce the memory requirements and computational complexity of the model.

- TFLite Interpreter: The TFLite Interpreter is a runtime that can run TFLite models on a wide range of devices. The TFLite Interpreter takes a TFLite model in the `.tflite` format and runs the inference.

The TFLite interpreter for microcontrollers is a subset of the full TFLite interpreter called TFLite Micro. It is written in C++ and requires 32-bit platforms. Unlike other platforms that can interprete the `.tflite` models, the model must be converted to a C array that can be compiled with the application and used by the TFLite Micro interpreter. An interesting feature of the TFLite Micro interpreter is the memory allocation, the interpreter allocates the needed memory for the model at initialization time, this memory is then treated as a stack memory, which means that the memory is not deallocated until the interpreter is destroyed and all the temporary allocations are done inside the preallocated memory space [19].

In this thesis, we will use TensorFlow Lite to convert the models to the TFLite format and run the inference on a microcontroller. However, due to the constraint that the TFLite Micro interpreter requires a 32-bit platform, we will extract the model weights and biases from the TFLite model and run the inference using a custom inference engine that we will develop, that means that every neural network inference operation will be implemented from scratch, both unoptimized and optimized versions.

2.3 Compiler Optimizations

A compiler is a specialized program equipped with tools that enable the translation of high-level languages into binary code executable by a computer, this process is called *compilation*. A compiler involves two main steps: *front-end* and *back-end*. The frontend applies an analysis on the source program, it involves all the set of operations which are needed to understand the source code, such as parsing, syntax analysis, and semantic analysis. Once the front-end phase is completed, an Intermediate Representation (IR) of the source code is generated. The back-end phase synthesize the IR generated by the frontend and constructs the final program.

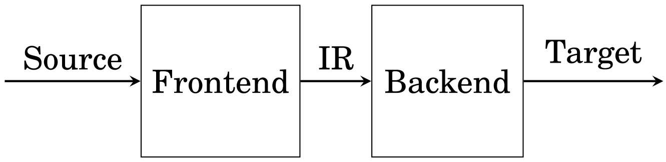


Figure 5: A 2-phase compiler with frontend and backend. In modern compilers this is equivalent as using the *-O0* flag.

The IR is a high-level representation of the code that is independent of the source code and the target architecture. With this representation is possible to add another phase called *optimization phase*, which is a set of transformations that are applied to the IR to improve the performance of the code. The optimization phase takes the IR generated by the frontend phase and generate a modified IR after multiple operations called *passes*. Every optimization pass is designed to solve the optimization problem optimally, unfortunately the optimization is highly dependent on the input code on which the pass is applied, therefore the optimality is not guaranteed and one of the challenges in compiler design is to find the sequence of passes that leads to the best optimized code maintaining its correctness [20]. In fact, the optimizer cannot change the behaviour of the program, it can only improve the performance of the code.

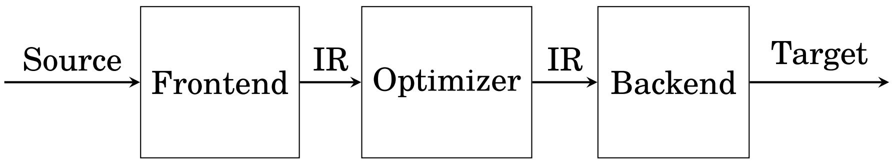


Figure 6: A 3-phase compiler with frontend, optimizer, and backend.

There are three main types of optimizations: *local optimizations*, *global optimizations*, and *inter-procedural optimizations*.

- Local optimizations: Local optimizations are optimizations that are applied to a single *basic block*. A basic block is the maximal sequence of instruction with no labels (except at the beginning) and no jumps (except the last instruction).

- Global optimizations: Global optimizations are optimizations that are extended to an entire Control Flow Graph (CFG). The CFG is a graph that represents the control flow of the program, where the nodes are the basic blocks and the edges are the control flow between the basic blocks.
- Inter-procedural optimizations: Inter-procedural optimizations are optimizations that are applied across multiple functions. These optimizations are more complex than local and global optimizations because they involve analysing the interactions between different functions, i.e. inlining functions.

In this thesis we focus on the application of the *strength reduction* optimization, which is an optimization that can be applied both locally and globally.

2.3.1 Strength Reduction

The *strength reduction* is a scalar optimization that replaces expensive operations with cheaper ones. The strength reduction optimization is based on the observation that some operations are more expensive than others, for example multiplications, divisions and exponential operations. In fact, the reduction in strength is part of the class of the *algebraic simplifications* that are applied to the code to reduce the number of operations by taking advantage of the mathematical properties of the operations such us the distributive property, the associative property, and the commutative property[21].

Expensive	**Cheaper**
$x * 2$	$x << 1$
$x/2$	$x >> 1$ or $x * 0.5$
$x * 3$	$x + x + x$
x^2	$x * x$

Table 1: Table of the cost of the operations and the cheaper operation that can replace them.

In literature, it is always coupled together with the *induction variable* optimization because it allows to drastically reduce the number of multiplication inside a loop. Let us consider the following code:

```
for (int i = 0; i < N; i++) {
        a[i] = a[i] * 2;
}
```

The code above is a simple loop that multiplies every element of the array a by 2. The multiplication by 2 can be replaced by a shift operation, which is a cheaper operation than multiplication. Moreover, the load of the array element must compute the memory address of the array as

```
a + i * sizeof(int)
```

The address computation can be replaced by an induction variable that is incremented at every iteration of the loop. The code can be rewritten as follows:

```c
int *ptr = a;
for (int i = 0; i < N; i++) {
        *ptr = *ptr << 1;
        ptr++;
}
```

By using a pointer as an induction variable, the strength reduction optimization removes the multiplication and the address computation from the loop.

In this thesis, we will focus on the application of the strength reduction optimization on the operations that are executed during the inference of a quantized neural network.

3 Hardware Emulation

In embedded systems the hardware emulation plays a crucial role, especially in the context of development time, costs and flexibility. In fact, the availability of components and the design process can turn to be time-consuming and unpredictable, and the software developers cannot wait to start working on the completed board until the design process is terminated. Therefore, using an emulator to start the application development saves time in terms of product advancement and can help detect troublesome design decisions earlier. [22]

A hardware emulator is a piece of software that mimics the behaviour of one or more hardware components. This is not to be confused with a simulator, a simulator tries to mimic the behaviour of the device and the programs are usually written in high-level languages while an emulator interprets the binary that has been written for a specific platform, it translates it to an intermediate representation and executes it on the host platform [23], an example of such emulators are *Instruction Set Simulators*. Instruction Set Simulators are particularly useful tools because they help to evaluate and validate design decisions in early stages of development offering convenient instruments such as debuggers, profiling, hardware simulators, etc. There are three types of these simulators: *interpretation-based*, *compilation-based* and *dynamic compilation-based* [24].

- *Interpretation-based* simulators recreate the target processor and simulate their execution through the *fetch-decode-execute* loop. These simulators are generally slow because the simulation happens over the representation in memory of the target CPU.

- *Compilation-based* simulators use a recompiled version of the target binary which instructions are translates into host machine instructions. These simulators keep the representation of the target machine in memory, updating it every time the equivalent block of simulator-target instructions is executed, in Figure 7 an example of simulator equivalent instruction.

- *Dynamic compilation-based* simulators perform real-time translations while the target binary is executed. They need a *Translation Cache (TC)* which is addressed by a *Translation Lookaside Buffer (TLB)*. Every time the Program Counter (PC) of the target is increased, the TLB is checked. If there is a hit, the address of the host translated instruction is returned and those are retrieved from the TC. If there is a miss, the target machine instructions are translated and both the TC and the TLB are updated.

Figure 7: Example of variable assignment *int a = 0;* compiled for the host system emulator.

In this thesis we have followed the emulation approach to extrapolate benchmarks and compare the behaviour of different architectures in terms of performance and correctness. The choice of a hardware emulator over the actual microcontrollers has been driven by the low availability of the selected boards in the market, the costs, the debugging instruments, the constraints of different Integrated Development Environments (IDEs) sometimes strongly tied to the chip vendor, and the desired analysis that we wanted to perform. Among the available hardware emulators, we have chosen the most complete one and easy to use which succeeded in our intent: Quick EMUlator (QEMU).

3.1 QEMU - Quick EMUlator

QEMU is a generic and open-source machine emulator and virtualizer [25]. QEMU provides mainly two operational modes: *system emulation* and *user-mode emulation*. The system emulation provides an emulation of an entire machine including all its hardware components (CPU, memory, peripherals, etc) while the user-mode emulation is limited to a CPU emulation, which allows running different compiled architecture on the host machine.

The system emulation provides a virtualization of the machine supporting different hypervisors. The devices are virtualized according to the Virtual I/O (VirtIO) standard [26] which allows designing look-alike physical devices to the guest virtual machine, but it also allows a pass-through to the host peripherals to reduce the emulation overhead. The user mode emulation, since it is a CPU only emulation, added some useful features such as: *system call translation*, in case of mismatch between emulated OS and host OS, *POSIX signal handling*, which allows redirecting of signal system calls to the emulated target, and *threading*.

QEMU is a *dynamic compilation-based* simulator and the translation unit component is called *tcg*. As we introduced before, also QEMU keeps a TC which contains, what in QEMU are called, the Translation Blocks (TBs). QEMU simulates the PC and the CPU information (registers) of the target architecture updating its internal state every time a basic TB block is executed. When a TB is executed, the program flow returns to the main loop and fetches the next target instruction to fetch and execute its relative TB. In order to speed up the execution, QEMU does a lookup on the next instructions and if the TB are present in the TC it concatenates them executing more instruction at once. This behaviour is done by default but is possible to instruct QEMU to execute one TB per target instruction with the flag `-one-insn-per-tb`. The Tiny Code Generator (TCG) internally uses an IR where each target architecture instruction corresponds to a list of instruction terminated by a label or a branch instruction.

Another useful feature of QEMU are the TCG plugins. The TCG plugins can be attached to the system emulation without modifying its execution state. This enables the user to retrieve information about the execution with a fine-grained resolution up to individual instructions. These plugins are loaded as external libraries from QEMU, and they are registered via a callback mechanism through the registered event, the plugin can access the internal state of the CPU and the memory. The plugins can be used for various purposes such as profiling, debugging, tracing, etc. In this thesis we have used the TCG plugins to profile the execution of neural networks on different architectures counting the number of executed instructions and memory accesses. In particular, two plugins have been used for the evaluation: `insn_count` and `mem` both compiled from the QEMU source code as shared libraries `libinsn.so` and `libexeclog.so` respectively. The `insn_count` plugin counts the number of executed instructions and the `execlog` plugin allows to log all the executed instructions and the memory accesses. However, since the generated log file was too large to be analysed, we have developed a new plugin called `memcnt` which counts the memory accesses showing the number of reads and writes. The plugins are loaded at runtime with the flag `-plugin` followed by the path of the shared library. Here is an example of how these plugins have been used:

```
qemu-arm \
        -cpu cortex-m0 \
        -one-insn-per-tb \
        -plugin /path/to/libinsncount.so \
```

```
-plugin /path/to/libexeclog.so \
-d plugin \
-D logfile.txt \
$binary
```

3.2 Reference Architectures

In this thesis we have used four different architectures to evaluate the performance of the neural networks. The architectures have been chosen based on their availability in the market, their popularity and their limitations.

3.2.1 MSP430

The first architecture we have used is the MSP430 [27], a 16-bit RISC architecture developed by Texas Instruments. The MSP430 family of microcontrollers is particularly used in low-power applications and it has been chosen for its limitations in terms of memory and computational power. The MSP430 has a 16-bit architecture with a 16-bit data bus and a 16-bit address bus. The MSP430 has 16 registers, 12 of which are general-purpose registers (R4-R15), and the remaining are used for the Stack Pointer (SP), the PC, the Status Register (SR), and the Constant Generator (CG).

Program Counter PC	R0
Stack Pointer SP	R1
Status Register SR	R2
Constant Generator CG1	
Constant Generator CG2	R3
Working Register R4	R4
Working Register R5	R5
: :	: :
Working Register R13	R13
Working Register R14	R14
Working Register R15	R15

Figure 8: MSP430 register architecture [1].

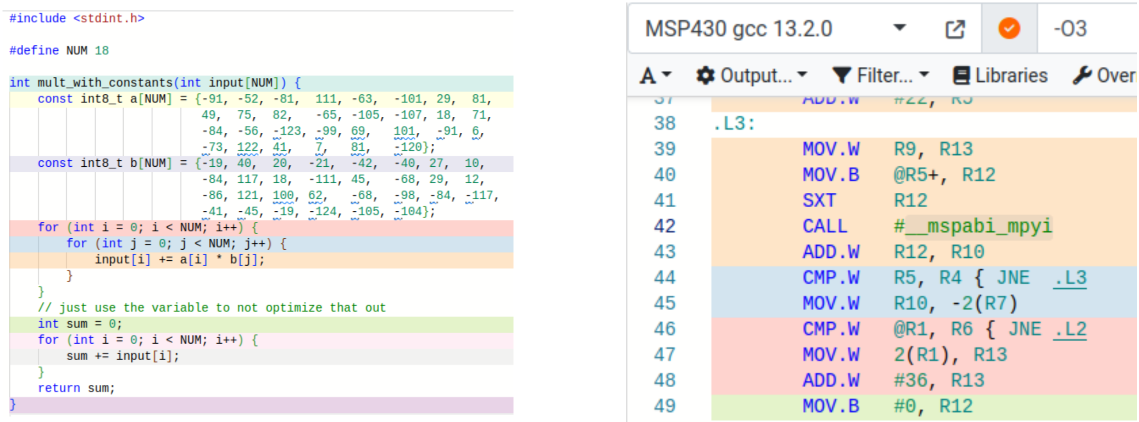
An interesting limitation of the MSP430 is the lack of a hardware multiplier, which means that the multiplication operation is done in software. In fact, the compilers implementations have to implement helper functions conforming to the MSP430 Application Binary Interface (ABI) to perform the multiplication [28]. This limitation has been taken into account to perform strength reduction on the neural network to reduce the number of multiplications. We have observed that the MSP430-GCC compiler already performs some strength reduction on the code, but this was limited to small constant arrays, which is not enough to perform the neural network inference. Moreover, the compiler was able to perform this kind of optimization only when both the arrays were constants and, with the aid of Compiler Explorer [29], we have observed that the MSP430-GCC compiler has a limitation on the size of the array that can be optimized. In fact, the compiler was able to optimize only arrays with a maximum size of 17 elements as we can see in Figure 9. This limitation has been taken into account to design the neural network to fit the MSP430 architecture.

Unfortunately, the MSP430 architecture is not supported by QEMU [30], so we used this architecture as a reference in terms of Read-Only Memory (ROM) and Random Access Memory (RAM) size, in particular we compiled the neural network for the MSP430 architecture targetting the MSP430F5529 [31] board, which is equipped with 8 KB of RAM and 128 KB of ROM. We used the size of the ROM and RAM as a reference for the other architectures. Moreover, the ROM available on this board is not contiguous, in fact we have two ROM memory regions, the low memory with a size of 48KB and the high memory with the size of 80KB. We have taken this into account when designing the neural network moving the weights in different memory regions.

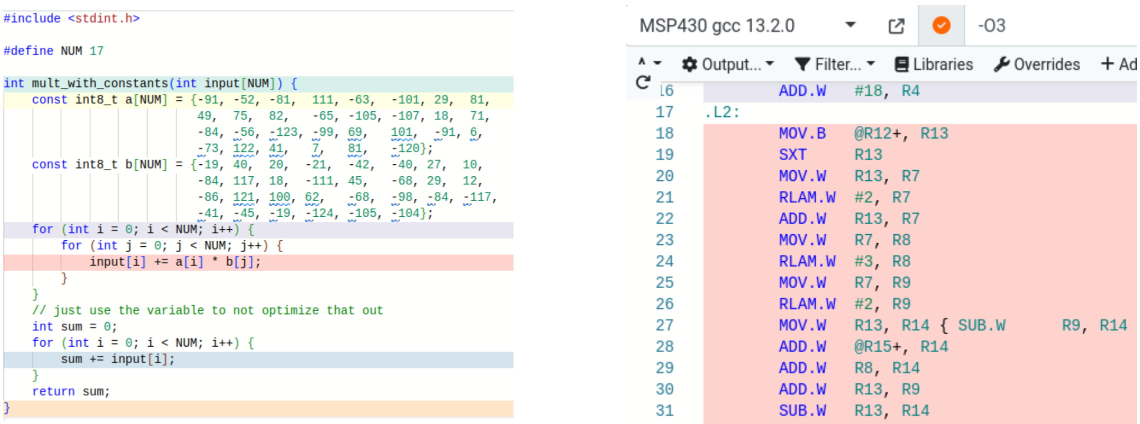
3.2.2 ARM

The ARM architecture is the most popular on the market and it is used in wide range of devices, from embedded devices to smartphones. The ARM processor that has been used in this thesis is the ARM Cortex-M0 [32] which is based on the ARMv6-M architecture. The M suffix in the ARM notation stands for microcontrollers and the Instruction Set Architecture (ISA) supported is the Thumb. This architecture's family is a 32 bit architecture particularly focused on size and determinism rather than performance. The ARM Cortex-M0 is based on a Von Neumann bus architecture with 32-bit data and address bus.

[2] `https://godbolt.org/z/TG1Y9GvaY`



(a) Assembly code generated for the multiplication of two arrays with 18 elements.



(b) Assembly code generated for the multiplication of two arrays with 17 elements.

Figure 9: Strength reduction optimization on MSP430-GCC compiler. In Figure 9a the GCC compiler invokes the `__mspabi_mpyi` routine to perform the multiplication, while in Figure 9b the GCC compiler applies the strength reduction optimization[2].

The ARM Cortex-M0 has 16 registers of which 13 are general-purpose registers (R0-R12) in addition to the SP register, the Link Register (LR) and the PC register.

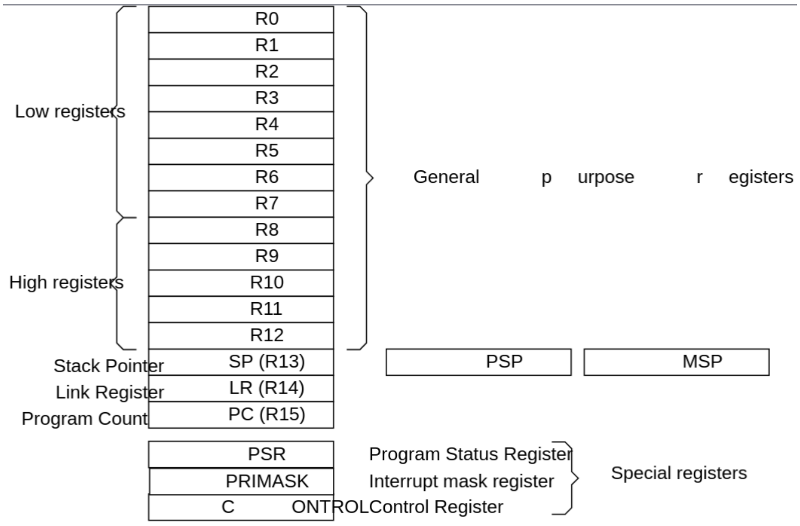


Figure 10: ARM Cortex-M0 register architecture [2].
Copyright © Arm Limited (or its affiliates).

Unlike the MSP430, the ARM Cortex-M0 has a hardware multiplier which allows performing multiplication operations in a single instruction. On the other hand, as stated on the ARM website, the processor can be shipped with two different configurations targetting different power consumptions. These two implementations differ precisely on the hardware multiplier where the "fast" implements the multiplication in a single CPU cycle while the "slow" takes up to 32 cycles.

Initially, we wanted to run the neural network on a board equipped with the ARM Cortex-M0 processor which adopted the "slow" configuration to observe a performance difference between our optimization and the default compilation, but we were not able to find a board with this configuration. Apparently, this configuration is only used on heterogeneous systems where the ARM Cortex-M0 is used as a coprocessor for the main processor.

The ARM Cortex-M0 architecture is supported by QEMU, so we have used this architecture to evaluate the performance of the neural network. We did not compile the neural network for any specific board, even though the ARM Cortex-M0 is used in many boards, we have written a simple linker file a simple startup file to run the neural network on QEMU. We did not use any advanced feature that could be present on a specific board, therefore we ran the simulation using the User Mode Emulation of QEMU. Although the user mode emulation generally provides a startup and linker file, we ran into some errors that lead us to write our own files, these errors seemed to be related to the fact that the ARM microcontrollers are adopted by different vendors and the startup and linker files are strictly tied to the desired vendor's implementation (i.e. Nordic initialize the interrupts before jumping to the main function while STMicroelectronics leaves this task to the user). In [33] is further explained also how different vendors handle reset handlers and hard faults, linking the two above-mentioned startup files and how the implementation differ from one to another.

3.2.3 AVR

The AVR architecture is an 8-bit Reduced Instruction Set Computer (RISC) architecture developed by Microchip. The AVR architecture is particularly used in low-power applications, and it is known for its simplicity and its low power consumption. Unlike the other architectures, the AVR architecture is based on a Harvard architecture with separate data and instruction memory. Since all the instructions are 16-bit or 32-bit wide, the instruction memory bus is 16-bit wide while the data memory bus is 8-bit wide [34]. Moreover, the instructions are executed with a single level pipeline so that when an instruction is fetched, the next instruction is already fetched and ready to be executed. The AVR architecture has 32 are general-purpose registers (R0-R31) of which the last six registers are used as 16-bit address pointers to address data in memory. The AVR architecture is supported by QEMU and we have compiled the binaries targetting the AtMega2560 which is the microcontroller generally know for being used on the Arduino Mega. We have chosen this microcontroller because it is the most popular in the AVR family and it is equipped with 8 KB of RAM and 256 KB of ROM. The ROM is double the size of the MSP430 but since the MSP430 has a 16-bit architecture, we expect that the binary size will be larger due to the 8-bit architecture. Unlike the other architectures, the emulation with QEMU have been performed in System Mode Emulation as the User Mode Emulation is not yet supported, but the microcontroller is completely supported and we did not need to write additional linker or startup files.

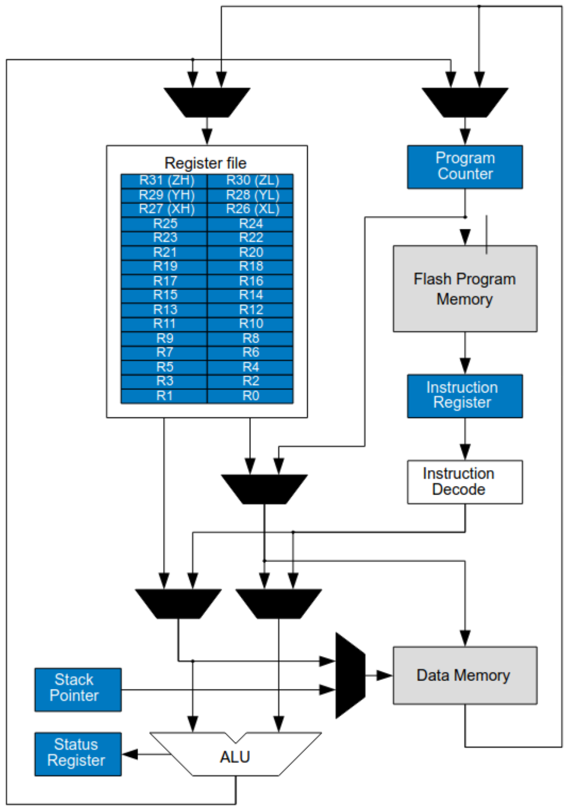


Figure 11: AVR architecture [3].

3.2.4 RISC-V

The RISC-V architecture is an open-source RISC architecture introduces by the University of California, Berkeley in 2014. The RISC-V architecture is particularly used in research and it is known for its simplicity and its flexibility. We have chosen this architecture because it is gaining popularity in the market, as that the first MCU based on the RISC-V architecture has been released in 2019 [35] and it has been recently announced the first industry's general purpose RISC-V microcontroller [36].

The RISC-V architecture comes with two different instruction set, the RV32I and the RV64I, the first one is a 32-bit architecture while the second one is a 64-bit architecture. Since we wanted to evaluate the performance of the neural network on microcontrollers, we picked the RV32I architecture which is the most used in the embedded world. In particular, we have cross-compiled a RISC-V GCC compiler based on the RV32GC architecture, a general purpose configuration that includes the base configuration, RV32I, in addition to the standard extensions MAFD, (M)ultiply, (A)tomics, (F)loating point and (D)ouble floating point. The architecture that we targeted has 32 registers (x0-

x31) of which 31 are general-purpose registers and the first one, the x0 register, is hardwired to zero. The RISC-V architecture is supported by QEMU and we

Register	**ABI Name**
x0	`zero`(hard-wired zero register)
x1	`ra` (return address)
x2	`sp` (stack pointer)
x3	`gp` (global pointer)
x4	`tp` (thread pointer)
x5–7	`t0–2` (temporary registers)
x8	`s0/fp` (saved register/frame pointer)
x9	`s1` (saved register)
x10–11	`a0–1`(function parameters/return values)
x12–17	`a2–7` (function parameters)
x18–27	`s2–11` (saved registers)
x28–31	`t3–6` (temporary registers)

Table 2: RISC-V Register overview [4]

have compiled the binaries without targetting any specific board as we have used the User Mode emulation.

4 Experiments

In this section we will describe the procedure and the methodology used to conduct the experiments focusing on two different datasets: Modified National Institute of Standards and Technology (MNIST)[37] and Speech Commands[38]. The first one is a well known dataset used to benchmark the performance of different models, while the second one is a dataset created to test the performance of the models on a real-world scenario. Furthermore, we will describe how the models were adapted to fit in the 16-bit architecture of the MSP430 and the manual optimizations applied to the code by using the Bernstein algorithm. We want to clarify that the experiments are not focused on the accuracy of the models and the different neural network architectures, but on the opportunity of further optimizations to fit the models on smaller architectures by enabling an extensive use of the strength reduction technique. Although, in order to make this kind of optimization more attractive, the models have been trained to achieve a good accuracy on the datasets.

4.1 The Bernstein Algorithm

The Bernstein Algorithm[39] is a technique used to find a sequence of cheaper operations such as addition, subtraction and shift to replace a multiplication by a constant. This algorithm has been developed in 1986 where the computational power of microprocessors was limited and the multiplication operation was performed is several clock cycles. Nowadays, the multiplication operation is performed in a single clock cycle, but the Bernstein Algorithm is still useful to reduce the number of operations and the power consumption. Although most of the architectures present in the market have a hardware multiplier, there are still some architectures that do not have it, such as the MSP430. As stated from R. Bernstein in his paper, there is no algorithm that optimally reach the minimal sequence of operations of add, shift and sub to perform the multiplication. His proposal was to combine two heuristics approaches: *the binary method* and *the factorization method*.

The binary method

The binary method takes advantage of the binary representation of the constant to be multiplied. The algorithm starts from the most significant bit of the constant, if the next bit is 1, the algorithm shifts the result of the previous operation to the left and adds the multiplicand, otherwise it only shifts the result to the left. The algorithm continues until all the bits of the constant have been processed. Here is an example of the binary method applied to the multiplication of n by $57 = (111001)_2$:

$2n \leftarrow n \ll 1$	shift left
$3n \leftarrow 2n + n$	add
$6n \leftarrow 3n \ll 1$	shift left
$7n \leftarrow 6n + n$	add
$56n \leftarrow 7n \ll 3$	shift left
$57n \leftarrow 56n + n$	add

However, in this example we did not use the subtraction operation, but is possible to reduce the number of operations if the binary representation of the constant present a long sequence on 1s. In this case, the algorithm can be optimized by using the subtraction operation.

$8n \leftarrow n \ll 3$	shift left
$7n \leftarrow 8n - n$	sub
$56n \leftarrow 7n \ll 3$	shift left
$57n \leftarrow 56n + n$	add

The factorization method

The factorization method is used to reduce the number of operations by finding the factors of the constant to be multiplied. The algorithm starts by finding the factors of the constant, then it applies the binary method to each factor and finally it combines the results. Here is an example of the factorization method applied to the multiplication of n by $57 = 3 \times 19$:

We find the sequence for $3n$

$2n \leftarrow n \ll 1$	shift left
$3n \leftarrow 2n + n$	add

We find the sequence for $19n$

$$
\begin{aligned}
16n &\leftarrow n \ll 4 && \text{shift left}\\
17n &\leftarrow 16n + n && \text{add}\\
18n &\leftarrow 17n + n && \text{add}\\
19n &\leftarrow 18n + n && \text{add}
\end{aligned}
$$

We combine the results

$$
\begin{aligned}
2n &\leftarrow n \ll 1 && \text{shift left}\\
3n &\leftarrow 2n + n && \text{add}\\
48n &\leftarrow 3n \ll 4 && \text{shift left}\\
51n &\leftarrow 48n + 3n && \text{add}\\
54n &\leftarrow 51n + 3n && \text{add}\\
57n &\leftarrow 54n + 3n && \text{add}
\end{aligned}
$$

As we can see, the factorization method does not always produce the optimal sequence of operations, in fact Bernstein has empirically discovered that this method lead to a shorter sequence of operations only if the factors are a power of 2 or a power of 2 ± 1.

The Algorithm

The Bernstein Algorithm is a combination of the binary method and the factorization method and it has been formalized by the following formula:

$$
\begin{aligned}
\textbf{Cost}(1) &= 0\\
\textbf{Cost}(\textbf{even } n) &= \textbf{Cost}(makeOdd(n)) + shiftCost\\
\textbf{Cost}(\textbf{odd } n) &= min(\\
&\quad \textbf{Cost}(n-1) + addCost\\
&\quad \textbf{Cost}(n+1) + subCost\\
&\quad \textbf{Cost}(n/(2^i-1)) + shiftCost + addCost\\
&\quad \textbf{Cost}(n/(2^i+1)) + shiftCost + subCost\\
&)
\end{aligned}
$$

The `makeOdd` function shift the number to the right until it becomes odd, the `addCost` and `subCost` are the cost of the addition and subtraction operations, while the shiftCost is the cost of the shift operation. The algorithm is recursive and terminates once the path explored by the search reach the cost limit or the node is already present in the hash table. The algorithm has a time complexity of $O(n \log n)$. The Bernstein Algorithm has been implemented in Python[3] imitating the Ada version available in the paper and following the reference implementation presented in [40], which handles the negative case and propose the use of a hash table to store the results of the recursive calls. Here is an example output of the Bernstein Algorithm applied to the multiplication of n by 57:

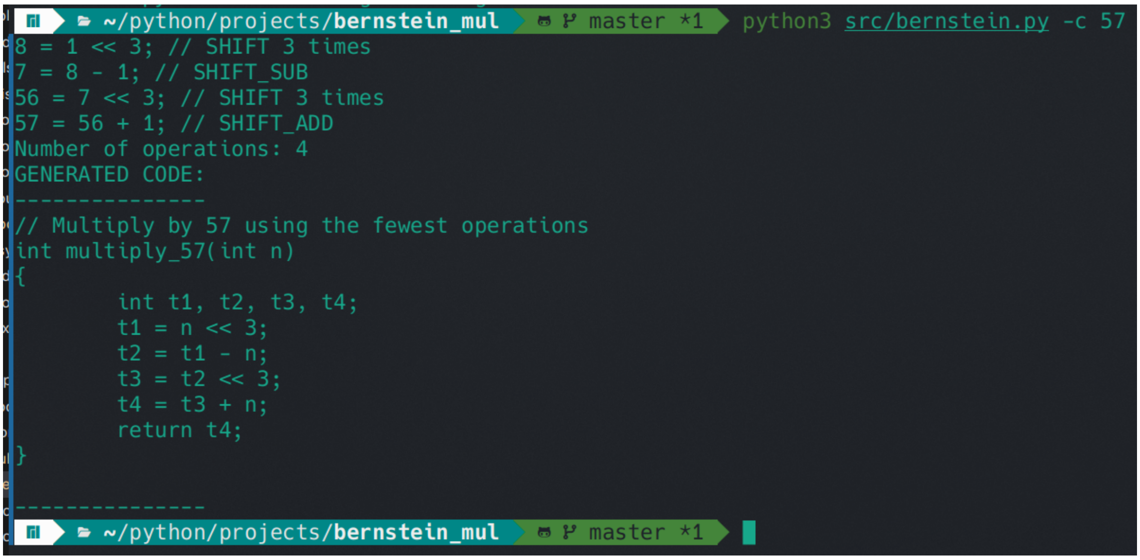


Figure 12: Output of the Bernstein Algorithm applied to the multiplication of n by 57

Considering that the weights of the Neural Networks (NNs) will be quantized to an 8-bit signed integer (int8_t), the whole multiplication operation can be performed in 16-bit without overflow. The tool has been instrumented to generate all the 8-bit integer range in a form of C source and header files. These generated files have been slightly modified to test different behaviours such as inlining and function calls. The generated files have been used to replace the multiplication operation in the code of the models and the results have been compared with the original code. The results of the comparison will be presented in the next section.

[3] https://github.com/Petrus97/bernstein_mul

4.2 MNIST

The Modified National Institute of Standards and Technology (MNIST) dataset is a dataset of handwritten digits composed of 60,000 training images and 10,000 test images. The images are 28x28 pixels in the `.pgm` format and the labels are the digits from 0 to 9. The dataset is widely used to benchmark the performance of different models and it is considered a good starting point for beginners in the field of machine learning. The images come in a greyscale format, so the pixel values range from 0 to 255. The dataset has been preprocessed by normalizing the pixel values to a range from 0 to 1 for training the models. We have applied two different NNs models to the MNIST dataset: an MLP and a CNN.

4.2.1 MLP Model

The MLP has been the starting point of the experiments. We firstly designed an MLP following the architecture of the MLP presented in [7], which is composed of 784 input neurons, 30 hidden neurons and 10 output neurons. The model present in the book has been trained using a *sigmoid* function as activation function for the hidden layer, but since our aim is to run the model on a microcontroller without floating point support, we opted to use the *ReLU* function as activation function for the hidden layer as it easier to implement and does not require floating point operations. The output layer has been implemented using the *softmax* function as activation function. We notice that the activation function in the output layer is useful only during the training phase, while during the inference phase we can simply take the index of the neuron with the highest value. Moreover, the book implement the whole neural network from scratch using the *numpy* library [41], but for simplicity and better integration with TFLite, we have used the *Keras* library[42]. We leave here the hyperparameters used to train the model:

Hyperparameter	**Value**
Optimizer	Stochastic Gradient Descend
Learning rate	0.05
Epochs	30
Batch size	100

Table 3: Hyperparameters used to train the MLP

After the training phase, the model has been converted to a TFLite model and the weights have been quantized to an 8-bit signed integer. The quantized model accept as input a 28x28 image in the `uint8_t` format and return the index of the neuron with the highest value. After the training we got these results:

Metric	Value
Training accuracy	96.88%
Test accuracy	96.25%
Quantized test accuracy	96.29%
Number of parameters	23860
Model size	93.20 KB

Table 4: Results of the MLP on the MNIST dataset

When a model is quantized with TFLite, a `.tflite` file is generated containing the weights and the operations of the model. We can visualize the model using *Netron*[4], a tool that allows to visualize the structure of the model which supports different file formats.

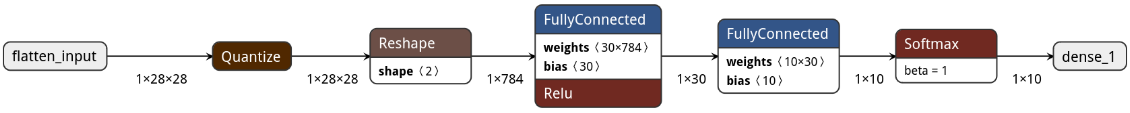


Figure 13: Structure of the MLP model

Through Netron is also possible to visualize the weights and the quantization parameters that have been calculated by Tensorflow to create the model. These quantization parameters have been first copied manually and then checked the correctness of the results by implementing manually the dot product operations on a separate Python file. The results of the manual implementation have been compared with the results of the TFLite model and the differences have been analysed. We have notice that Python and the TFLite interpreter, which is based on C++, have different rounding behaviours, but this did not affect the results of the model.

After the manual implementation of the dot product operation with Python, we have implemented the same operation in C with the Bernstein Algorithm obtaining the same results. For this purpose, a code generator has been developed Section 4.5. The Bernstein Algorithm has been applied to the multiplication of the weights and the input values and the results have been compared

[4]`https://netron.app/`

with the results of the TFLite model. Unfortunately, when we tried to compile the model that uses the Bernstein algorithm for the MSP430, the size of the compiled file was too large to fit the 128KB of ROM memory available in the reference board. We have tried to reduce the size of the compiled file by using the gcc optimization flags, but the results were not satisfactory as the size was slightly reduced. Therefore, we have decided to reduce the size of the model by removing the hidden layer and training a new model with only the input and the output layer.

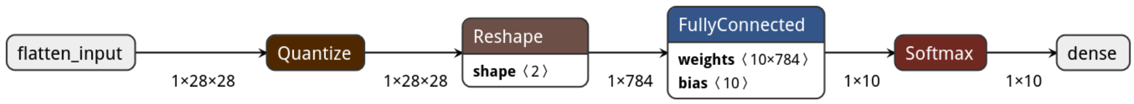


Figure 14: Structure of the MLP model without the hidden layer

The new model has been trained with the same hyperparameters of the previous model and the results have been compared with the results of the previous model. The results of the new model are presented in the following table:

Metric	Value
Training accuracy	92.26%
Test accuracy	92.14%
Quantized test accuracy	92.08%
Number of parameters	7850
Model size	30.66 KB

Table 5: Results of the MLP without the hidden layer on the MNIST dataset

The new model has been compiled for the MSP430 in two variants: in the first one we force the compiler to not inline the multiplication function we have generated with the Bernstein Algorithm, while in the second one we force the compiler to inline the function. The first approach worked to fit the model in the low memory region of the MSP430, while the second approach did not work as the size of the compiled file was too large. Therefore, we had to tweak the linker to move the whole dot product operation in the higher memory region of the MSP430 (see Section 3.2.1). As a result, we got the model occupying 74KB of higher ROM memory, which is not ideal since microcontrollers are designed to use a lot of peripherals. Since we are not able to reduce the size of the MLP model further, we have decided to move to the CNN model.

4.2.2 CNN Model

The CNN model has been designed taking inspiration from the LeNet-5 model presented in [43]. The LeNet-5 model is composed of two convolutional layers, two pooling layers and three fully connected layers, but since our experience with the MLP model has shown that the size of the model is a critical factor, we have decided to reduce the number of fully connected layers to one.

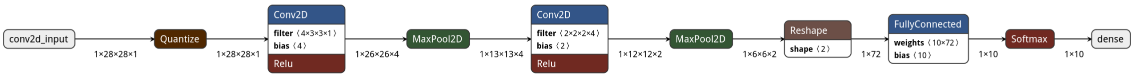


Figure 15: Structure of the CNN model

The model has been trained with the same hyperparameters of the MLP model, the results of the CNN model are presented in the following table:

Metric	Value
Training accuracy	95.70%
Test accuracy	96.15%
Quantized test accuracy	96.12%
Number of parameters	798
Model size	3.12 KB

Table 6: Results of the CNN on the MNIST dataset

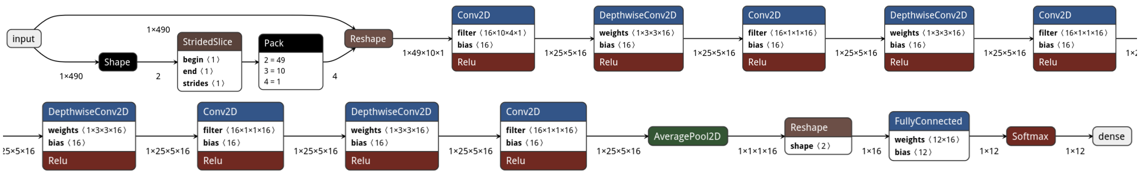
As we can see, using a CNN we have improved the accuracy and drastically reduced the size of the model. We followed the same procedure of the MLP to verify the correctness of the quantization parameters and the results of the Bernstein Algorithm. Since the size of the CNN model is small enough to fit in the 128KB of ROM memory available in the MSP430, we have compiled the model for the MSP430 by inlining the multiplication function generated with the Bernstein Algorithm.

4.3 Speech Commands

The Speech Commands dataset is a dataset that contains 105,829 1-second long utterances of 35 different words. The dataset is divided into 3 parts: training (84848 audio files), validation (9982 audio files) and testing (4890 audio files). The dataset is widely used to benchmark the performance of different models on the task of Keyword Spotting (KWS) recognition, which is becoming more and more popular with the rise of smart speakers and voice assistants to start an interaction, the most common examples are "Hey Siri" or "Ok Google". This dataset is already being used to benchmark different neural network architectures on the task of KWS, also on microcontrollers. In fact, we have been able to find pre-trained models on [44] and [45]. According to the results obtained in [44], the neural network model that was able to achieve a good accuracy maintaining the smallest size of the model was using a Depthwise Separable Convolutional Neural Network (DS-CNN) architecture. Unfortunately, the provided model has a number of parameters equal to 24,908, and it is not suitable to a strength reduction optimization since we observed that it increases the binary size. Therefore, we had to retrain the model on a slimmer DS-CNN architecture to reduce the number of parameters.

4.4 DS-CNN Model

The DS-CNN model reflects the same architecture of the models presented in [45]. We took the smallest model and retrained using the following architecture:



The model has been trained with the following hyperparameters:

Hyperparameter	Value
Optimizer	Adam
Learning rate	0.01
Epochs	30
Batch size	100

Table 7: Hyperparameters used to train the DS-CNN

The parameters that have been modified from the reference implementation are the learning rate from 0.00001 to 0.01 and we reduce the number of epochs from 36 to 30. The reason why we increased the learning rate was to speed up the training process, at the cost of ending up with a little of overfitting, see Table 8 below. The results of the model are presented in the following table:

Even though the achieved accuracy is not as good as the one presented in [44], the model is still able to achieve a good accuracy on the test data. An

Metric	Value
Training accuracy	83.16%
Test accuracy	76.91%
Quantized test accuracy	76.54%
Number of parameters	3164
Model size	12.36 KB

Table 8: Results of the DS-CNN on the Speech Commands dataset. As we can see, the model overfits the training data, but it is still able to achieve a good accuracy on the test data.

article in support of this statement is [5], where the authors have trained a DS-CNN model on the Speech Commands comparing the accuracies obtained using different combinations of filters-per-layer and number of layers. The results of the article show that the model is able to achieve a better accuracy by increasing the number of layers rather than the number of filters, comparing our results with the one of the paper using the same five layers architecture, we have obtained a better accuracy with a smaller number of filters per layer. The results are presented in the following table:

Model	Accuracy	Filters per layer	Size
Ours	83.16%(76.91%)	16	12.36 KB
DS-CNN	57.4%(54.3%)	10	8 KB
DS-CNN	74.5%(70.4%)	20	16 KB

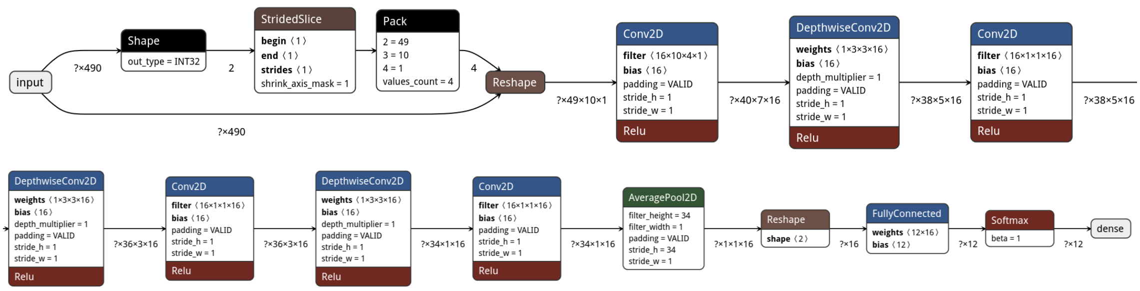
Table 9: Comparison between our results and the results from [5]

Even though the premise of reducing the number of parameters to fit the model in the MSP430 was correct, a problem arose when we tried to generate the code using the Bernstein Algorithm. The original model has been trained taking advantage of padding and strides in the convolutional layers to not lose information in the input, but the Bernstein Algorithm used to generate the code to apply the filters does not take into account the padding of the image. We had to take in consideration different viable solutions to solve this problem:

- Pad the image before applying the filters.
- Modify the model to not use padding in the convolutional layers.
- Generate the code to apply the filters taking into account the padding.

> This means that more versions of the same filter will be generated, one for each possible position of the filter in the image.

Since we are exploring the possibility of applying the strength reduction optimization to the convolutional layers, the first and the third solutions have been discarded. The first solution would require to pad the image before applying the filters, but this would increase the size of the image and the number of operations to be performed. The third solution would require to generate more versions of the same filter, but this would increase the size of the model and the number of operations to be performed. Therefore, we have decided to modify the model to not use padding in the convolutional layers. The modified model has been redesigned such that the convolutional layers do not use padding (*valid* padding in TensorFlow). The modified model has been trained with the same hyperparameters of the previous model and the results have been compared with the results of the previous model. The new model presents one convolutional layer with 16 filters, three depthwise separable convolutional layers with 16 filters each, an average pooling layer and a fully connected layer. Com-



paring the results of the new model with the results of the previous model, we have obtained a similar accuracy on the test data, but the number of parameters has been reduced. The results of the new model are presented in table Table 10.

Metric	Value
Training accuracy	80.96%
Test accuracy	85.48%
Quantized test accuracy	83.66%
Number of parameters	2604
Model size	10.17 KB

Table 10: Results of the DS-CNN without padding on the Speech Commands dataset

From the results we can see that the model is able to achieve a good accuracy on the test data and, even though the training accuracy is lower than the previous model, it does not seem to be affected by overfitting. Therefore, in this case we have been able to reduce the number of parameters and the size of the model without affecting the accuracy and made it more suitable for our optimization. We must say that, due to the lack of padding in the convolutional layers, we have observed that in the last epochs of the training the model, the accuracy was somewhat stationary, but this was expected since every layer was reducing the size of the image and the information was being lost.

4.5 Code Generation

Since the manual implementation is a tedious and more error-prone process, we have decided to automate it by introducing a code generation capability inside our Python engine to evaluate the impact of the strength reduction optimization on the computational cost and performance of the inference process. The code generator takes as input the `.tflite` model and extracts a subset of information such as the weights, the quantization parameters, strides, padding and the tensor shapes for each layer in the model. Then, the extracted information is converted to a JSON format, which is later read by the Python inference engine to create the model in memory. The code is generated dynamically at runtime. This means that the inference engine needs to perform an inference on a test image before generating the code. While the inference is performed, the same operation is generated in the C language for both the standard loop and the strength-reduced optimized version. The resulting C code is then available to be compiled for the target architecture. Moreover, we added the TFLite interpreter so that we have been able to compare the resulting inference with the one produced by TensorFlow to check the correctness of the operations.

5 Results and Discussion

In this section, we present the results of the benchmarking process and discuss the implications of the results. We compare the performance of the loop version and our version of the implemented neural networks on the different architectures running on QEMU. The evaluated parameters are the binary size, the number of instructions, and the memory usage of the two versions. The results are presented in the following subsections.

5.1 Binary Size

The binary size is an important parameter in the context of the embedded systems, as those systems have limited memory resources. After the first attempts in compiling deeper neural networks for the MSP430 architecture, we faced the problem that the strength reduced version of the binary was too large to fit in the memory of the MSP430. This lead us to choose different models that allowed a better memory management and to optimize the code to reduce the binary size. The binary size that we are going to show is the size of the `.elf` binaries obtained by removing all the symbols and the debug information from them using their toolchain version of `objcopy`. In general, when a binary is extracted with `objcopy`, the ELF header and sections are stripped out using the following command:

```
objcopy -O binary <input.elf> <output.bin>
```

However, we have decided to keep the ELF header and sections in the binary, as some toolchains were creating the same binary size by filling the binary with zeros to match the size of the memory. This is why we have decided to keep the ELF header and sections in the binary, as it is a more accurate representation of the binary size using the following command:

```
objcopy -S <input.elf> <stripped.elf>
```

We compared the binary sizes of the loop version and our version of the neural networks with the `Os` and `O3` optimization flags. The results are shown in Figures Figures 16 to 21.

We have decided to compare the binaries by compiling them with two different optimization flags available in GNU Compiler Collection (GCC)[46]:

- `Os` optimization flag: This flag is used to optimize the binary size. It applies all the optimization present in `O2` except the ones that may increase the binary size. We have chosen this flag because is the most common flag among IDEs for release builds in embedded systems.

- `O3` optimization flag: This flag is used to optimize the speed of the binary. It applies all the optimization present in `O2` and adds more aggressive optimizations. We have chosen this flag to compare the binary sizes of the two versions when the code is optimized for speed.

As we can see from the results, the strength reduced version generates a binary that is way larger than the one which uses loops, even when the code is optimized for size. This is due to the fact that the loop version is more efficient in terms of memory usage, because the constants of the neural networks are stored as they are in the `.data` section of the binary, while in the strength reduced version, the constants are embedded as assembly instructions which are stored in the `.text` section of the binary. This leads to a larger binary size, as the constants are repeated in the binary for each layer of the neural network. The binary size of the loop version is smaller than the one of the strength reduced version, because the loop version uses the same code for each layer of the neural network, while the strength reduced version uses different code for each layer.

It is worth to notice that in some presented graphs, results are missing. In particular the binary sizes of the MSP430, Arm Cortex-M0 and AVR architectures. In the case of the MLP model, Figures 16 and 17, the missing data from the AVR architecture tell us that the generated binary was bigger than the available memory of the board ATMega2560 which is 256KB. Consequently, the other two architectures, MSP430 and Arm Cortex-M0, that have less memory available, did not generate the final binary.

If we look at the DS-CNN graph in Figures 20 and 21 only in the `-O3` version the data is missing. This is due to the fact that the MSP430 has been compiled for a specific board with its own linker file. Moreover, the ARM compiler does not contain a generic linker file in its toolchain, therefore we wrote a linker file that uses the same memory sizes as the MSP430. The `-O3` showed us that the binary was too large to fit in the memory of the board, and therefore the compilation was unable to generate the final elf file. To check how much memory the binary would have occupied, we have taken the object files generated and with the aid of the `readelf` command, we have calculated manually the size of the *.text* section which prevented the linker from generating the final binary.

As we can see from Table 11, the *.text* section overflows the available 128KB

Architecture	Binary Size .text section (bytes)
MSP430	262.23 KB
ARM Cortex-M0	76.29 KB

Table 11: Binary sizes of the MSP430 and ARM Cortex-M0 architectures with O3 optimization.

of memory present on the board, but not the ARM version. The reason why the ARM version does not generate the final binary is because we have modelled the available ROM as the lower region of the MSP430 memory, which is 48KB. In the next results section, we will change this configuration to be able to generate the final binary and run the benchmark on the ARM Cortex-M0 architecture.

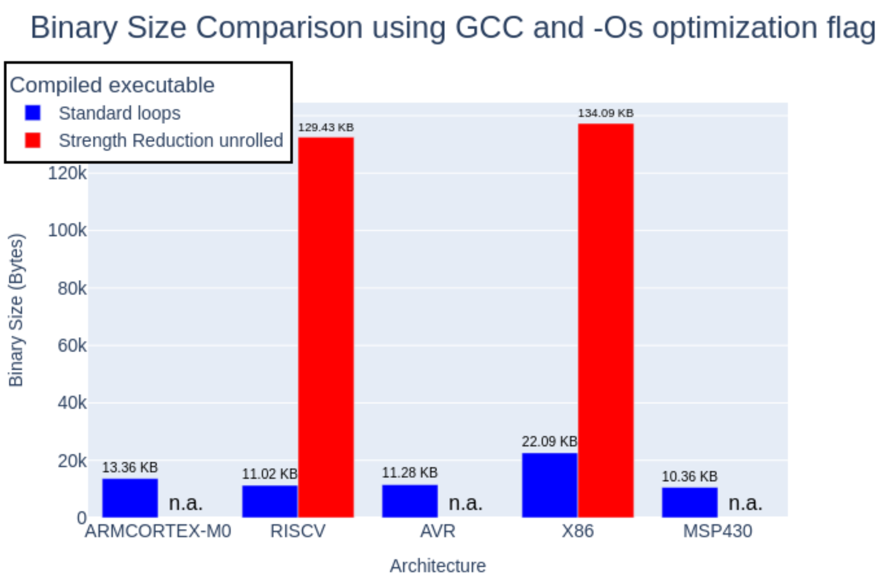


Figure 16: MLP model. Binary sizes of the loop version and our version with Os optimization.

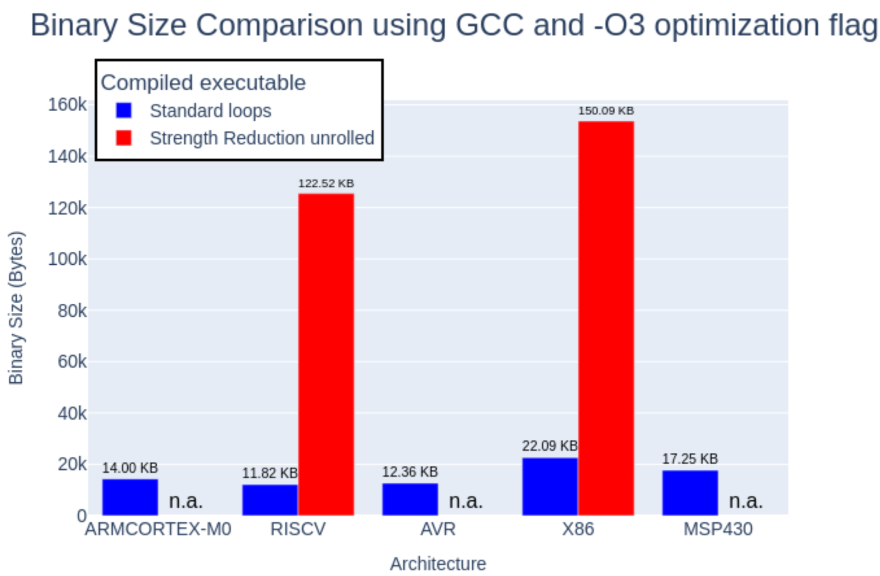


Figure 17: MLP model. Binary sizes of the loop version and our version with O3 optimization.

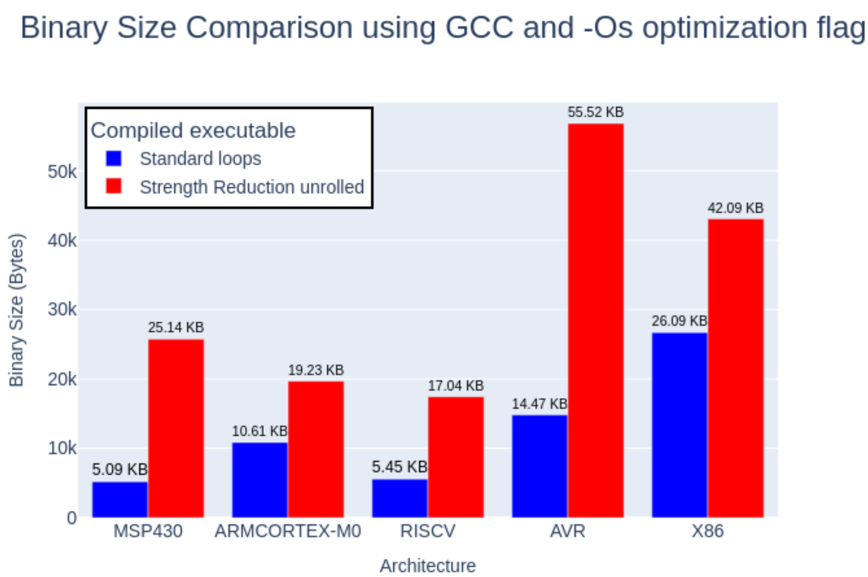


Figure 18: CNN model. Binary sizes of the loop version and our version with Os optimization.

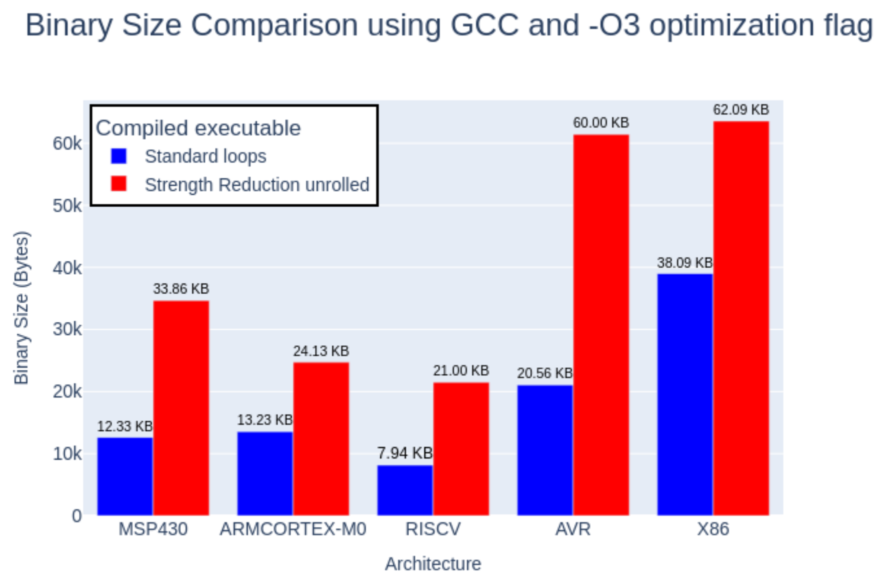


Figure 19: CNN model. Binary sizes of the loop version and our version with O3 optimization.

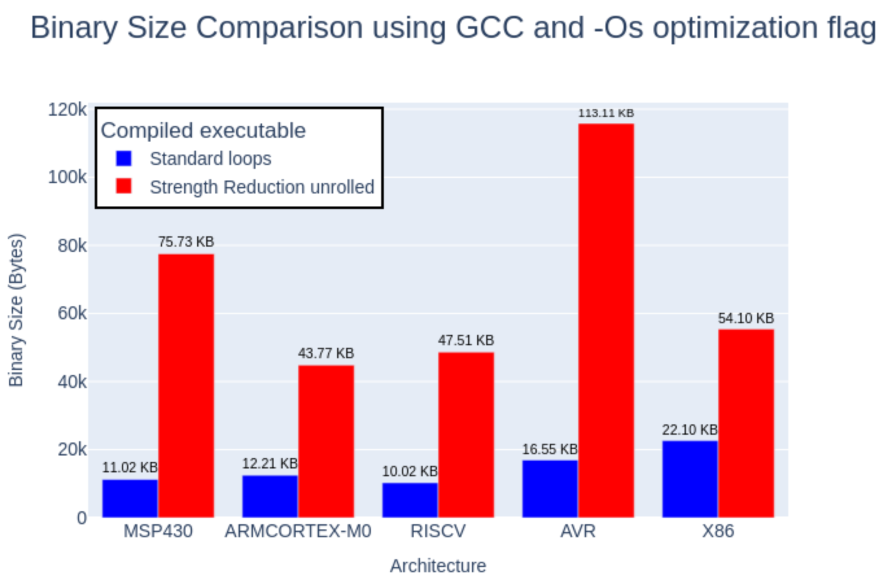


Figure 20: DS-CNN model. Binary sizes of the loop version and our version with Os optimization.

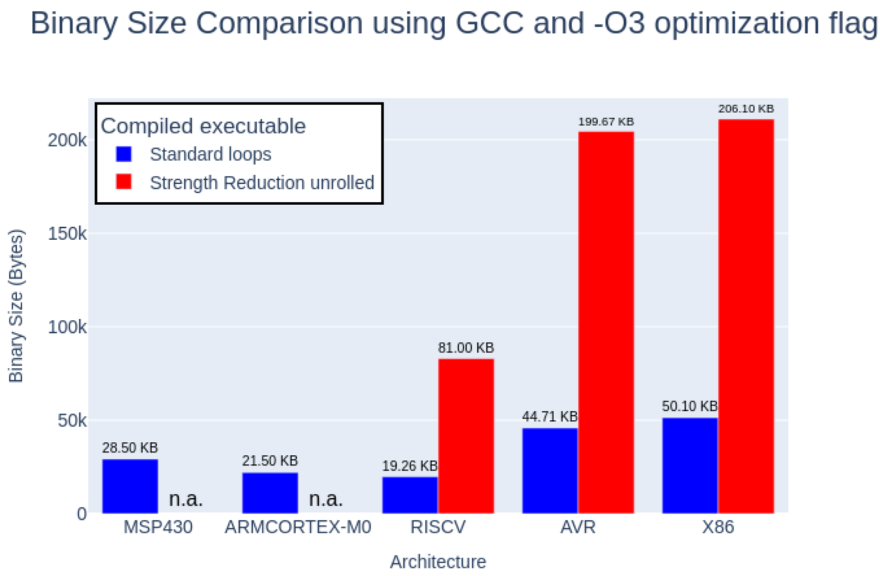


Figure 21: DS-CNN model. Binary sizes of the loop version and our version with O3 optimization.

5.2 Memory Usage and Number of Instructions

In embedded systems, the memory is not only limited but that can be also slower compared to memories available on desktop/servers systems. Although the CPU clock speed is in the order of MHz, the different in speed is still noticeable. For example, let us take the datasheet of our MSP430 reference board (MSP430F5529) used for compilation with a system clock running at 25MHz. The memory and speed of the MSP430F5529 are shown in Table 12.

Memory	Speed	Consumption
Flash	8MHz	290 μA/MHz
RAM	8MHz	150 μA/MHz

Table 12: Memory and speed of the MSP430F5529 (Active mode).

As we can see from the Table 12, the memory is slower than the CPU clock speed, which means that the memory can be a bottleneck in the system. This is why it is important to optimize the memory usage of the system. Moreover, different memories have different access times and the power consumption is affected as well, therefore limiting the number of memory access may lead to a more power efficient system. On the other hand, the number of instructions is also important, as the CPU has to execute them. The more instructions the

CPU has to execute, the more power it consumes. This is why it is important to optimize the number of instructions as well.

We have decided to compare the memory accesses and the number of instructions of the loop version and our version of the neural networks with the `Os` and `O3` optimization flags. We have emulated the execution of the neural networks on QEMU and we have used the available plugins to count the number of memory accesses and the number of instructions executed. Here is an example of how we counted the number of executed instructions:

```
qemu-arm \
    -cpu cortex-m0 \
    -one-insn-per-tb \
    -plugin ./qemu-bin/build/tests/plugin/libinsn.so \
    -plugin ./qemu-bin/build/contrib/plugin/libmemcnt.so \
    -d plugin \
    -D logfile.txt \
    $binary
```

We show here in Figures 22 to 25 the emulation results from the CNN model, the other graph results are available in the Appendix A. We also report here the exact values from the emulation of the different models in Tables 13 to 18.

The results show that the *strength-reduced* version of the neural networks has a lower number of memory accesses and a lower number of instructions executed compared to the loop version, especially in the case of the `Os` optimization flag. It is worth to notice that the `O3` optimization generally generates a lower number of instructions, this is due to the loop unrolling optimization that is applied by the compiler. The loop unrolling optimization is a technique used to reduce the overhead of the loop control instructions by replicating the loop body multiple times. This leads to a lower number of instructions executed, as the loop control instructions are executed fewer times. However, the loop unrolling optimization may lead to a larger binary size, as the loop body is replicated multiple times. This is why the binary size of the `O3` optimized version is larger than the one of the `Os` optimized version. The *strength-reduced* version of the neural networks has a lower number of memory accesses and a lower number of instructions executed compared to the loop version, especially in the case of the `Os` optimization flag. This is due to the fact that the *strength-reduced* version does not perform any **load** operations of the neural network constants from memory, but it only loads the input data as the instructions are fetched continuously and they already embed the constants.

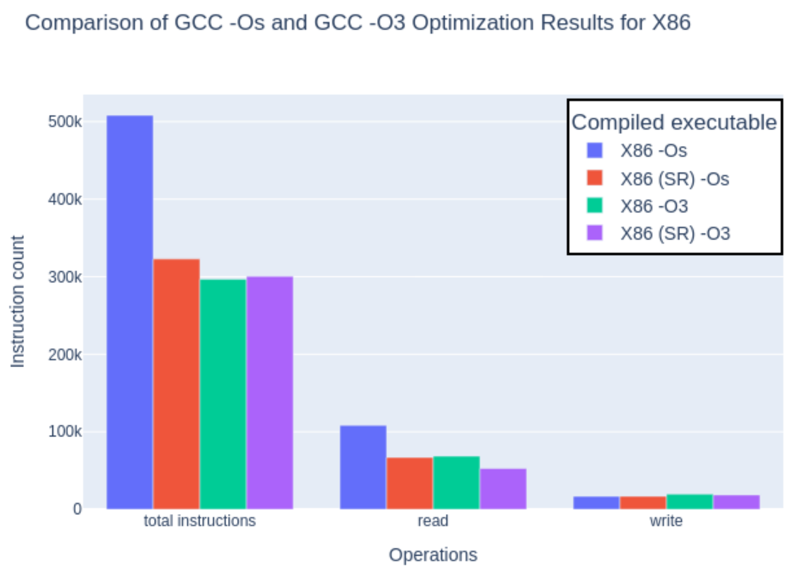


Figure 22: X86_64: CNN model. Number of instructions executed with `Os` and `O3` optimization flags. SR stands for strength reduced version.

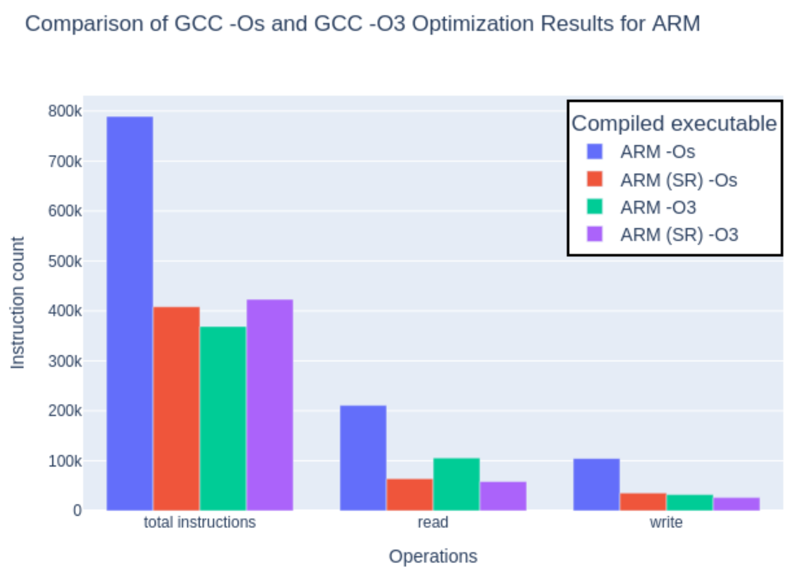


Figure 23: ARM: CNN model. Number of instructions executed with `Os` and `O3` optimization flags. SR stands for strength reduced version.

Even though the binaries generated for the AVR architecture are compiled and linked together, we have experienced some problems running the QEMU

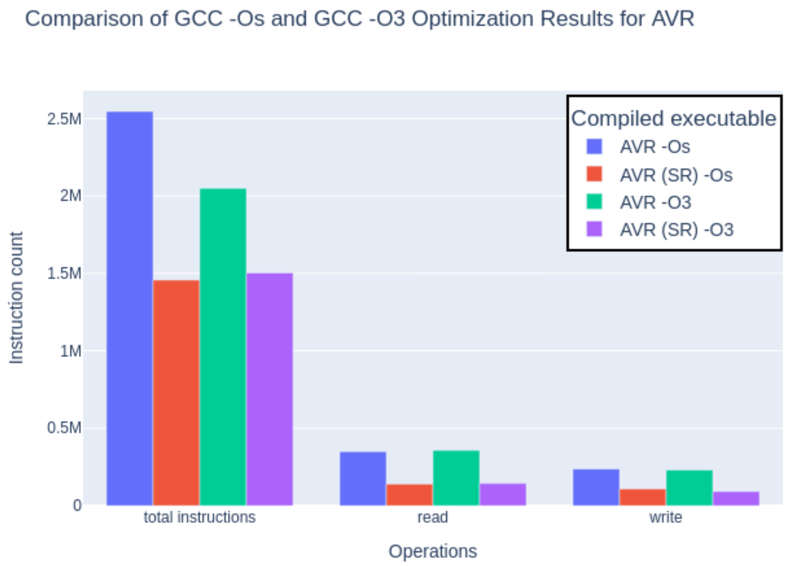


Figure 24: AVR: CNN model. Number of instructions executed with Os and O3 optimization flags. SR stands for strength reduced version.

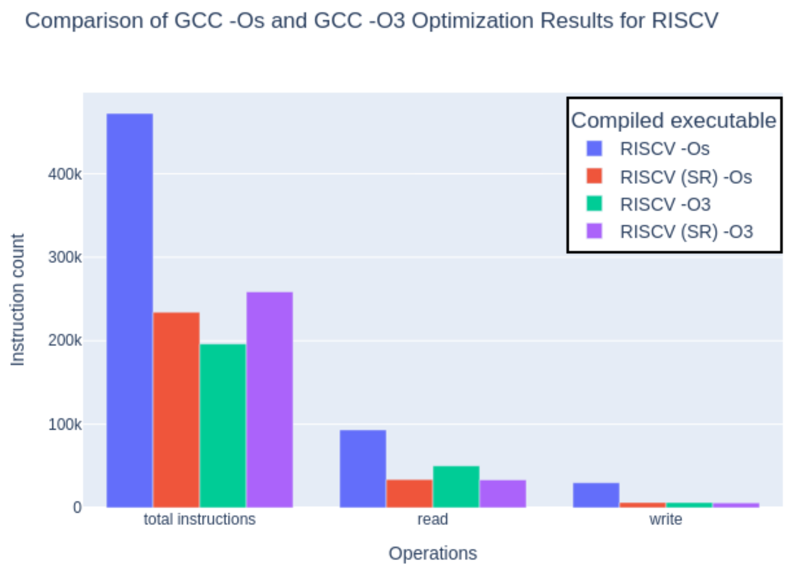


Figure 25: RISC-V: CNN model. Number of instructions executed with Os and O3 optimization flags. SR stands for strength reduced version.

Architecture	Executed instructions	Read ops	Write ops
x86_64	508157	108363	16689
x86_64 (SR)	322898	66845	16736
AVR	2545570	348707	237474
AVR (SR)	1456986	139689	107892
ARM	789188	210949	104733
ARM (SR)	408285	64107	35499
RISC-V	472276	93237	29943
RISC-V (SR)	234266	33929	6301

Table 13: GCC -Os Optimization Results on the MNIST CNN model for a single inference.

Architecture	Executed instructions	Read ops	Write ops
x86_64	296801	68485	19544
x86_64 (SR)	300669	52609	18516
AVR	2048837	357573	230636
AVR (SR)	1503686	143832	91665
ARM	368834	105681	32248
ARM (SR)	423147	58674	26343
RISC-V	196245	50177	6196
RISC-V (SR)	258730	33472	5921

Table 14: GCC -O3 Optimization Results on the MNIST CNN model for a single inference.

Architecture	Executed instructions	Read ops	Write ops
x86_64	3957011	946574	160327
x86_64 (SR)	2302025	494729	148024
AVR	n.a.	n.a.	n.a.
AVR (SR)	n.a.	n.a.	n.a.
ARM	6583484	1826174	784674
ARM (SR)	3049349	487647	176821
RISC-V	4140787	736130	92301
RISC-V (SR)	2004607	313294	19101

Table 15: GCC -Os Optimization Results on the Speech Commands DS-CNN model for a single inference.

Architecture	Executed instructions	Read ops	Write ops
x86_64	2332908	852885	145281
x86_64 (SR)	2537404	486576	147821
AVR	n.a.	n.a	n.a.
AVR (SR)	n.a.	n.a.	n.a.
ARM	3119377	1051041	234474
ARM (SR)	3238848	538156	182587
RISC-V	2132433	808093	147411
RISC-V (SR)	2225330	327464	32595

Table 16: GCC -O3 Optimization Results on the Speech Commands DS-CNN model for a single inference.

Architecture	Executed instructions	Read ops	Write ops
x86_64	188641	46623	14718
x86_64 (SR)	167160	36637	20412
AVR[1]	392684	59521	45912
AVR (SR)	n.a	n.a.	n.a.
ARM	106629	26469	3176
ARM (SR)	64152	10707	8888
RISC-V	72969	18519	3298
RISC-V (SR)	48018	8527	8981

[1] Emulation problem: using the full image of 784 pixels the emulation was stuck and it seemed to overwrite symbols in the binary, the results showed here have been taken cutting the image to 28x26 pixels.

Table 17: GCC -Os Optimization Results on the MNIST MLP for a single inference.

Architecture	Executed instructions	Read ops	Write ops
x86_64	139161	30197	12858
x86_64 (SR)	168059	35891	19539
AVR[1]	377998	59397	45788
AVR (SR)	n.a	n.a.	n.a.
ARM	63685	18039	2567
ARM (SR)	58164	10019	8247
RISC-V	62719	17918	2709
RISC-V (SR)	48657	7933	8381

[1] Emulation problem: using the full image of 784 pixels the emulation was stuck and it seemed to overwrite symbols in the binary, the results showed here have been taken cutting the image to 28x26 pixels.

Table 18: GCC -O3 Optimization Results on the MNIST MLP for a single inference.

system emulator on those. While in the MLP case the size of the binary was overwriting debugging symbols, which caused the emulator to stuck, in the DS-CNN case the emulator experienced an infinite loop in the execution of the binary. We have no infinite loops in the binary, but in case of the AVR emulation, the approach was slightly different. Embedded systems are designed to never stop, and in this particular case the architecture once the *main* function returns it ends up in an infinite loop defined by the architecture with the `__stop_program` function. Here is the code of the `__stop_program` function:

```
__stop_program:
        rjmp .-2 ; __stop_program
```

The `rjmp` instruction is a relative jump instruction that jumps to the current address minus 2. This instruction is used to create an infinite loop, as it jumps to the current address minus 2, which is the address of the `rjmp` instruction itself.

Therefore, to run the AVR architecture on QEMU, we launched the emulator with GNU Debugger (GDB) and set a breakpoint at the `__stop_program` function. Once the breakpoint is reached, we terminate the program and can see the results of the emulation. Since this breakpoint was never being reached, we enabled another QEMU plugin which dumps the executed instructions in a file. We have then analysed the file and we have noticed that the emulation was stuck in a function call.

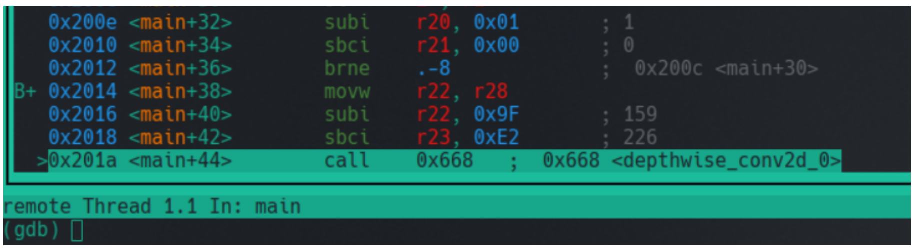


Figure 26: GDB session with the AVR architecture.

Here is a snippet of the output taken when enabling the `libexeclog.so` plugin:

```
0, 0x200e, 0x2fb93d9, "SUBI      r20, 1"
0, 0x2010, 0x2fb93d9, "SBCI      r21, 0"
0, 0x2012, 0x2fb93d9, "BRNE      .-8"
0, 0x2014, 0x2fb93d9, "MOVW      r23:r22, r29:r28"
0, 0x2016, 0x2fb93d9, "SUBI      r22, 159"
```

```
0, 0x2018, 0x2fb93d9, "SBCI      r23, 226"
0, 0x201a, 0x2fb93d9, "CALL      0x668"
0, 0x201a, 0x2fb93d9, "CALL      0x668"
0, 0x201a, 0x2fb93d9, "CALL      0x668"
0, 0x201a, 0x2fb93d9, "CALL      0x668"
0, 0x201a, 0x2fb93d9, "CALL      0x668"
0, 0x201a, 0x2fb93d9, "CALL      0x668"
0, 0x201a, 0x2fb93d9, "CALL      0x668"
// ... continues with the same instruction
```

Due to this problem, we are not able to provide the results of the AVR architecture for the DS-CNN model.

6 Conclusions

This thesis aimed to explore the potential of the *strength reduction* optimization technique in the context of neural networks inference. The results obtained show that the technique can be effectively applied to the convolutional layers of a neural network, reducing the number of multiplications and additions required to perform the inference. However, the technique is not always beneficial, as it increases the memory requirements of the system which are often limited on embedded devices. On the other hand, the reduction of the number of memory accesses and the number of executed instructions can turn beneficial for ultra-low power devices where the memory access is the main source of power consumption.

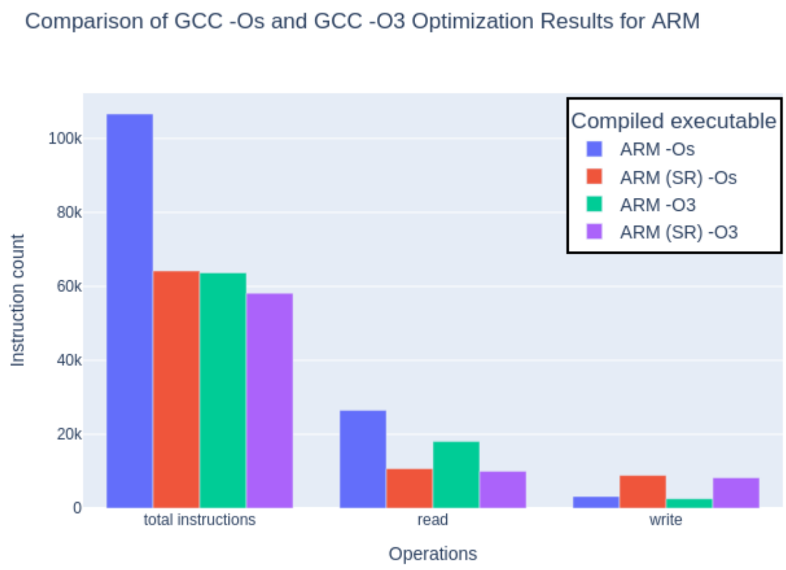
This thesis suggests that the choice of the optimization technique should be made considering the specific requirements of the target device and the application, as it is the task of the system designer to find the best trade-off between the cost, the power consumption and the performance of the system. Among the systems that can benefit from the *strength reduction* optimization technique, systems equipped with large flash memories that support the eXecute-in-Place (XiP) mode are the most promising. The XiP mode allows the system to execute the code directly from the flash memory, reducing the memory requirements of the system. The *strength reduction* optimization will limit data transfers between the flash memory and the RAM, reducing the power consumption of the system. The reduction of the number of executed instructions and memory access can be beneficial for the systems that are limited by the CPU performance. The results obtained in this thesis suggest that the *strength reduction* optimization technique can be effectively applied to the convolutional layers of a neural network, reducing the power consumption of the system and improving the performance of the system.

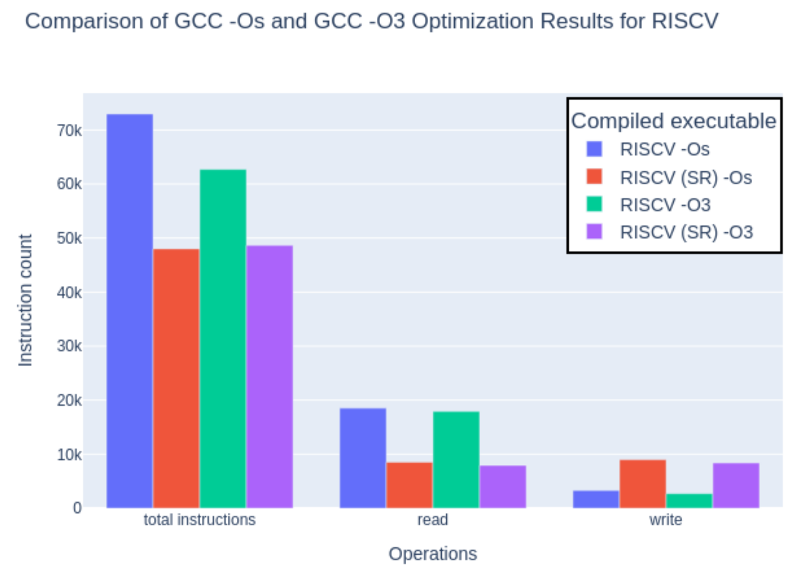
Future work to improve the efficiency of this new optimization method could be the development of a mixed approach that combines the *strength reduction* layers with the classical layers. The *strength reduction* layers would be used only for the most computationally intensive layers, while the classical layers would be used for the rest of the network. In contrary, the *strength reduction* layers could be used for less memory intensive layers, while the classical layers would be used for the most memory intensive layers. This approach would allow to balance the performance and memory usage of the system, making it more suitable for resource-constrained embedded systems.

A Appendix

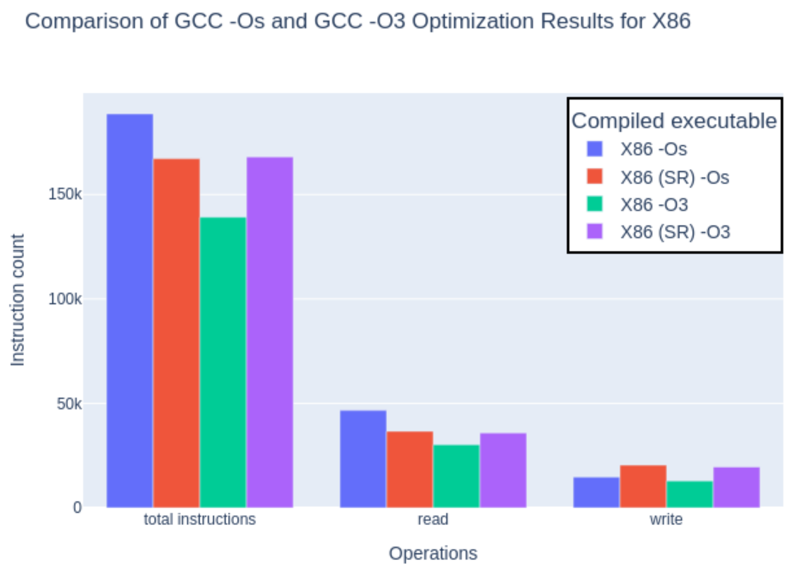
MLP results

Visualization of the results obtained for the MLP network from Tables 17 and 18.





It is interesting to see how the number of instructions, especially in the ARM architecture are drastically reduced with strength reduction optimization enabled. The instructions are reduced by almost 40% in the case of the



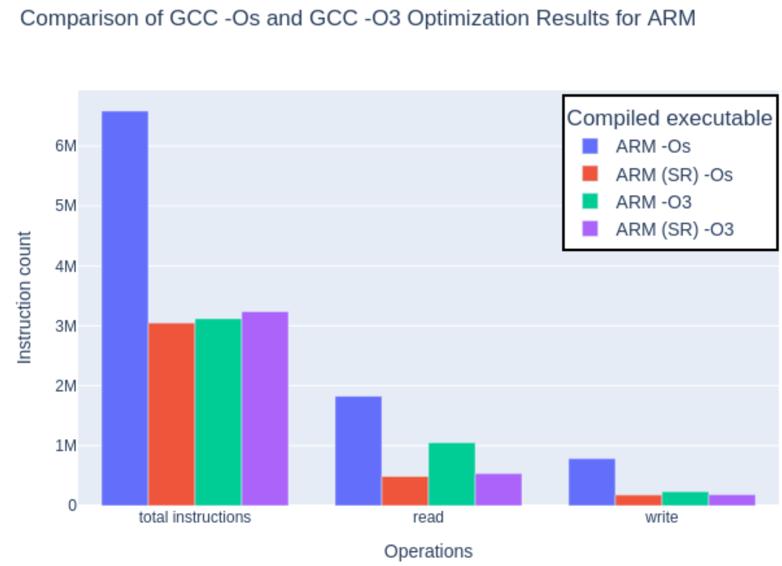
Os optimization and the number of read and write operations is reduced by almost 50%.

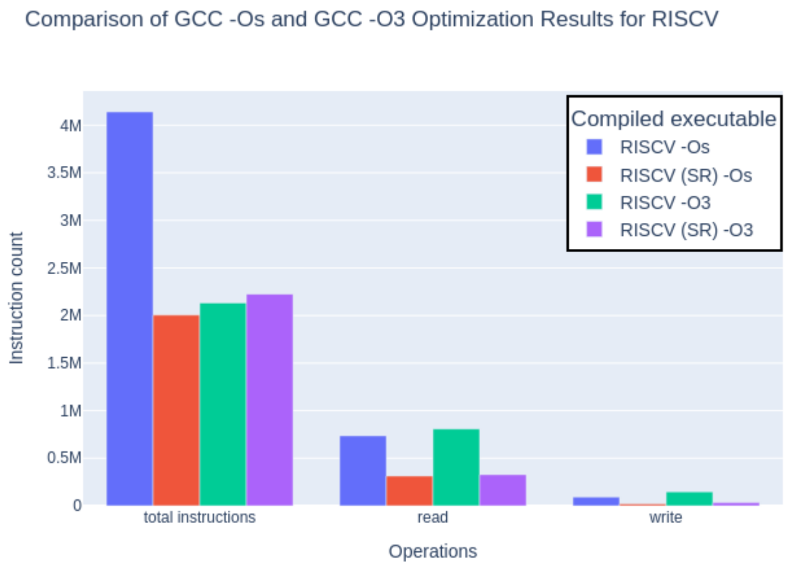
Different is the case of the x86 architecture where the number of instructions is higher when the O3 optimization is enabled. This is probably connected to the definition of the ISA, where the x86 architecture defines a Complex Instruction Set Computer (CISC) architecture, while the ARM and RISC-V architectures define a RISC architecture. The CISC architecture has a larger set of instructions, which are more complex and can perform more operations in a single instruction.

Curiously, the number of write operations is increased in each architecture, which is probably connected to the fact that the `strength reduction` optimization technique is applied to an unrolled dot product operation, which requires more write operations to store the intermediate results.

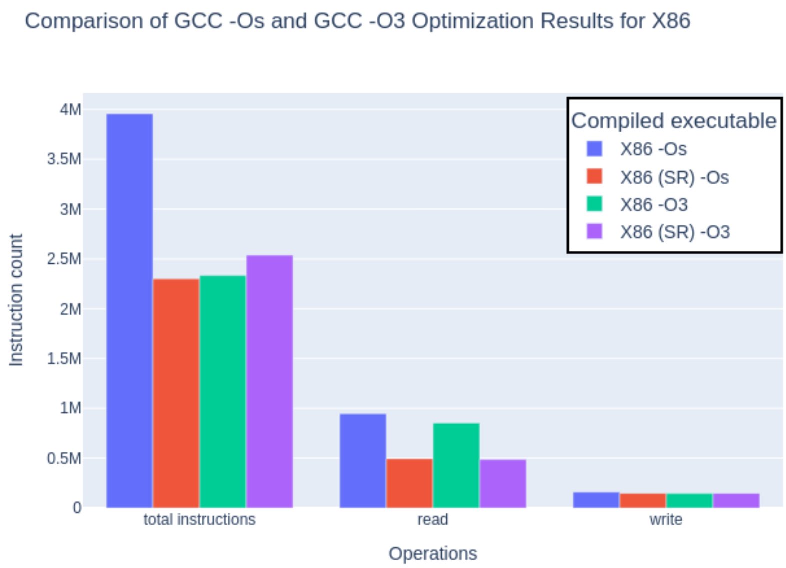
DS-CNN results

Visualization of the results obtained for the DS-CNN network from Tables 15 and 16.





Another interesting observation is that the number of instructions of the strength reduced DS-CNN network is higher when the `O3` optimization is enabled compared to the `Os` optimization. This is probably connected to the fact that the `O3` optimization enables the loop unrolling, which increases the number of instructions. This is specifically tied on how the source code has been



implemented, in our case the strength reduction is applied only on the filters operations, but the filter is moved over the image with a loop. The resulting number of read/write operations is instead comparable between the two optimizations.

Acknowledgements

I would like to express my gratitude to my supervisor, Prof. Stefanos Kaxiras, for his guidance, and his trust in my work. I recognize the effort he put in helping me improve my work and enhance its value. I also want to thank my reviewer, Prof. Joachim Parrow, for the valuable feedbacks and the patience in reviewing my thesis.

I especially want to thank my family for their support and encouragement throughout my studies in different country. They have always been with me even when I was far away from home. Thank you for ensuring I never felt distant. Their moral support has been a great source of motivation for me to keep going and to achieve my goals.

I would like to thank my friends for making the academic journey more enjoyable. Thanks to my home friends, also known as Aula Studio 010, for their unwavering support and for making me feel as if I never left. To my friends in Uppsala, the Flogsta legends, thank you for the unforgettable moments we shared. Now divided by the distance, but in life we always meet twice.

I am deeply grateful for all the support and encouragement I have received. I am thankful for the opportunity to study at Uppsala University and to work on this thesis.

List of Acronyms

ABI Application Binary Interface. 22

AI Artificial Intelligence. 1

CFG Control Flow Graph. 16

CG Constant Generator. 21

CISC Complex Instruction Set Computer. 56

CNN Convolutional Neural Network. 5–7, 32, 35, 46

DS-CNN Depthwise Separable Convolutional Neural Network. 7, 9, 36, 41

EMA Exponential Moving Average. 13

GCC GNU Compiler Collection. 40

GDB GNU Debugger. 52

IDE Integrated Development Environment. 19

IoT Internet of Things. 1

IR Intermediate Representation. 15, 20

ISA Instruction Set Architecture. 22, 56

KWS Keyword Spotting. 35, 36

LR Link Register. 24

MAC Multiply-Accumulate. 1

ML Machine Learning. 1

MLP Multi-Layer Perceptron. 5, 6, 32, 41

MNIST Modified National Institute of Standards and Technology. 28, 32

MSE Mean Squared Error. 4

NAS Neural Architecture Search. 10

NN Neural Network. 31, 32

PC Program Counter. 18, 21, 24

PTQ Post-Training Quantization. 10, 11

QAT Quantization-Aware Training. 10

QEMU Quick EMUlator. 19, 20, 22, 24, 25, 27, 40, 46

RAM Random Access Memory. 22, 25

ReLU Rectified Linear Unit. 4, 32

RISC Reduced Instruction Set Computer. 25, 56

ROM Read-Only Memory. 22, 25

SDG Stochastic Gradient Descent. 4

SP Stack Pointer. 21, 24

SR Status Register. 21

TB Translation Block. 20

TC Translation Cache. 18–20

TCG Tiny Code Generator. 20

TFLite TensorFlow Lite. 13, 14, 32, 33

TinyML Tiny Machine Learning. 1, 3, 10

TLB Translation Lookaside Buffer. 18, 19

VirtIO Virtual I/O. 19

XiP eXecute-in-Place. 54

References

[1] T. Instruments, "Msp430 family architecture guide and module library," 2024. [Online]. Available: https://www.ti.com/sc/docs/products/micro/msp430/userguid/ag_05.pdf

[2] A. Ltd, "Cortex-m0 devices generic user guide version 1.0," 2024. [Online]. Available: https://developer.arm.com/documentation/dui0497/a/the-cortex-m0-processor/programmers-model/core-registers

[3] AVR®, "Avr® db family data sheet," 2024. [Online]. Available: https://ww1.microchip.com/downloads/en/DeviceDoc/AVR128DB28-32-48-64-DataSheet-DS40002247A.pdf

[4] R.-V. International, "Isc-v abis specification, document version 1.0," November 2022. [Online]. Available: https://github.com/riscv-non-isa/riscv-elf-psabi-doc

[5] P. M. Sørensen, B. Epp, and T. May, "A depthwise separable convolutional neural network for keyword spotting on an embedded system," *EURASIP Journal on Audio, Speech, and Music Processing*, vol. 2020, no. 1, p. 10, 2020.

[6] C. D. L. for Computer Vision, "CS231n convolutional neural networks for visual recognition," 2024. [Online]. Available: https://cs231n.github.io/neural-networks-1/

[7] M. A. Nielsen, "Neural networks and deep learning," 2024. [Online]. Available: http://neuralnetworksanddeeplearning.com/chap1.html

[8] I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning*. MIT press, 2016.

[9] F. Chollet, "Xception: Deep learning with depthwise separable convolutions," *CoRR*, vol. abs/1610.02357, 2016. [Online]. Available: http://arxiv.org/abs/1610.02357

[10] Y. Guo, Y. Li, R. Feris, L. Wang, and T. Rosing, "Depthwise convolution is all you need for learning multiple visual domains," 2019. [Online]. Available: https://arxiv.org/abs/1902.00927

[11] M. Shafique, T. Theocharides, V. J. Reddy, and B. Murmann, “Tinyml: Current progress, research challenges, and future roadmap,” in *2021 58th ACM/IEEE Design Automation Conference (DAC)*, 2021, pp. 1303–1306.

[12] M. Nagel, M. Fournarakis, R. A. Amjad, Y. Bondarenko, M. van Baalen, and T. Blankevoort, “A white paper on neural network quantization,” *CoRR*, vol. abs/2106.08295, 2021. [Online]. Available: https://arxiv.org/abs/2106.08295

[13] S. Han, H. Cai, and J. Lin, “Mit 6.5940 fall 2023 tinyml and efficient deep learning computing,” 2024. [Online]. Available: https://hanlab.mit.edu/courses/2023-fall-65940

[14] TensorFlow, “Tensorflow lite 8-bit quantization specification,” 2024. [Online]. Available: https://www.tensorflow.org/lite/performance/quantization_spec

[15] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. G. Howard, H. Adam, and D. Kalenichenko, “Quantization and training of neural networks for efficient integer-arithmetic-only inference,” *CoRR*, vol. abs/1712.05877, 2017. [Online]. Available: http://arxiv.org/abs/1712.05877

[16] S. Migacz, “8-bit inference with tensorrt,” in *GPU technology conference*, vol. 2, no. 4, 2017, p. 5.

[17] Tensorflow, “Tensorflow,” 2024. [Online]. Available: https://www.tensorflow.org/

[18] FlatBuffers, “Flatbuffers: Flatbuffers,” 2024. [Online]. Available: https://flatbuffers.dev/

[19] R. David, J. Duke, A. Jain, V. Janapa Reddi, N. Jeffries, J. Li, N. Kreeger, I. Nappier, M. Natraj, T. Wang, P. Warden, and R. Rhodes, “Tensorflow lite micro: Embedded machine learning for tinyml systems,” in *Proceedings of Machine Learning and Systems*, A. Smola, A. Dimakis, and I. Stoica, Eds., vol. 3, 2021, pp. 800–811. [Online]. Available: https://proceedings.mlsys.org/paper_files/paper/2021/file/6c44dc73014d66ba49b28d483a8f8b0d-Paper.pdf

[20] K. D. Cooper and L. Torczon, *Engineering a compiler*. Morgan Kaufmann, 2022.

[21] V. A. Alfred, S. L. Monica, and D. U. Jeffrey, *Compilers Principles, Techniques & Tools*. pearson Education, 2007.

[22] P. Sampath and B. R. Rao, "Efficient embedded software development using qemu," in *13th Real Time Linux Workshop*, 2011.

[23] S. Das, "Emulator vs simulator: Know the differences | browserstack," *BrowserStack Blog*, 2024. [Online]. Available: https://www.browserstack.com/guide/difference-between-emulator-and-simulator

[24] J. Zhu and D. Gajski, "An ultra-fast instruction set simulator," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 10, no. 3, pp. 363–373, 2002.

[25] T. Q. P. Developers, "Qemu documentation," 2024. [Online]. Available: https://www.qemu.org/docs/master/about/index.html

[26] M. S. Tsirkin, R. Hat, C. Huck, and E. D. Listing, "Virtual I/O device (VIRTIO) version 1.3," 2023.

[27] L. Bierl, "Msp430 family architecture guide and module library," 2000. [Online]. Available: https://www.ti.com/sc/data/msp/databook/chp8.pdf

[28] T. Instruments, "Msp430 embedded application binary interface," 2022. [Online]. Available: https://www.ti.com/lit/an/slaa534a/slaa534a.pdf

[29] M. Godbolt, "Compiler explorer," 2024. [Online]. Available: https://godbolt.org/

[30] Q. documentation, "Emulation — qemu documentation," 2024. [Online]. Available: https://www.qemu.org/docs/master/about/emulation.html

[31] T. Instruments, "Msp430f5529 data sheet, product information and support," 2024. [Online]. Available: https://www.ti.com/product/MSP430F5529

[32] A. Ltd, "Cortex-m0 technical reference manual," 2024. [Online]. Available: https://developer.arm.com/documentation/ddi0432/c

[33] P. Johnston, "Ending the embedded software dark ages: Let's start with processor fault debugging! - embedded artistry," 2024. [Online]. Available: https://embeddedartistry.com/blog/2021/01/11/hard-fault-debugging/

[34] M. Reynolds, “Avr® memory - developer help,” 2024. [Online]. Available: https://developerhelp.microchip.com/xwiki/bin/view/products/mcu-mpu/8-bit-avr/structure/memory/

[35] OnChip, “Open-v | crowd supply,” 2016. [Online]. Available: https://www.crowdsupply.com/onchip/open-v

[36] Renesas, “Renesas introduces industry’s first general-purpose 32-bit risc-v mcus with internally developed cpu core | renesas,” 2024, [Online]. Available: https://www.renesas.com/us/en/about/press-room/renesas-introduces-industry-s-first-general-purpose-32-bit-risc-v-mcus-internally-developed-cpu-core .

[37] L. Deng, “The mnist database of handwritten digit images for machine learning research,” *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 141–142, 2012.

[38] P. Warden, “Speech commands: A dataset for limited-vocabulary speech recognition,” *CoRR*, vol. abs/1804.03209, 2018. [Online]. Available: http://arxiv.org/abs/1804.03209

[39] R. Bernstein, “Multiplication by integer constants,” *Software: practice and experience*, vol. 16, no. 7, pp. 641–652, 1986.

[40] P. Briggs and T. Harvey, “Multiplication by integer constants,” *This is a “web”, a literate programming document. See http://softlib. rice. edu/M-SCP*, 1994.

[41] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. Fernández del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant, “Array programming with NumPy,” *Nature*, vol. 585, p. 357–362, 2020.

[42] F. Chollet *et al.* (2015) Keras. [Online]. Available: https://github.com/fchollet/keras

[43] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, “Gradient-based learning applied to document recognition,” *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[44] Y. Zhang, N. Suda, L. Lai, and V. Chandra, “Hello edge: Keyword spotting on microcontrollers,” *arXiv preprint arXiv:1711.07128*, 2017.

[45] C. Banbury, V. J. Reddi, P. Torelli, J. Holleman, N. Jeffries, C. Kiraly, P. Montino, D. Kanter, S. Ahmed, D. Pau *et al.*, “Mlperf tiny benchmark,” *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*, 2021.

[46] G. Project, “Gnu compiler collection (gcc),” https://gcc.gnu.org/.