Delft University of Technology
Master of Science Thesis in Embedded systems

Split Inference on Networked Microcontrollers

Junyu Lu

TU Delft Delft University of Technology

Split Inference on Networked Microcontrollers

Master of Science Thesis in Embedded systems

Embedded Systems Group
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Van Mourik Broekmanweg 6, 2628 XE Delft, The Netherlands

Junyu Lu
j.lu-17@student.tudelft.nl
thisislujy@163.com

Thursday 20^{th} June, 2024

Author
Junyu Lu (j.lu-17@student.tudelft.nl)
(thisislujy@163.com)
Title
Split Inference on Networked Microcontrollers
MSc Presentation Date
Friday 28^{th} June, 2024

Graduation Committee

Qing Wang	Delft University of Technology
Jie Yang	Delft University of Technology

Abstract

With the rapid development of Artificial Intelligence (AI), the size and complexity of models are increasing rapidly. The limited memory and computing power of microcontroller units (MCUs) pose significant challenges for running AI applications on them. This thesis presents a method to run deep learning models on MCUs using a distributed approach.

First, we identified memory size as the primary constraint for implementing deep learning models in MCUs. To address this issue, our method involves splitting the models into smaller weight fragments distributed across multiple networked MCUs. A coordinator MCU manages the overall process, including neuron mapping and data relaying. We demonstrated that our approach reduces peak RAM usage during inference compared to existing methods.

For optimization, we employed layer fusion and quantization to reduce model size while preserving accuracy. We also introduced a rating system to assign capability scores to MCUs for efficient task allocation, explained through mathematical equations.

Our simulation phase validated the effectiveness of our method, showing successful distributed inference in MCUs and providing insights for real-world applications. Implementation on a network of MCUs confirmed the practical applicability and efficiency of our approach.

In conclusion, this thesis presents a feasible and efficient distributed inference method for networked MCUs, addressing resource limitations and enabling practical AI applications on constrained MCU platforms.

"Be yourself, everyone else is already taken" – Oscar Wilde

Preface

The motivation for this thesis project comes from my interest of Artificial Intelligence (AI) as well as the passion of my supervisor (Dr. Qing Wang). With the advancement of AI, deploying it on edge devices has become an interesting area of exploration. As a master student studying embedded systems and who is interested in AI, what is more perfect than doing a thesis on the deployment of AI on microcontrollers?

After two years of hard work, I have finally reached the last phase of my studies at TU Delft, and I can still remember the first time I arrived in the Netherlands with a heart filled with excitement. Looking back, during these two years, there are happy moments, there are sad moments, there are days when I stay up late fighting for a deadline, there are days that I walk casually in the summer breeze, and these memories will always be a precious part of my life.

I want to take this time to thank some people during this period. I want to first thank my parents who are always there for me, being so supportive, I will not be who I am today without you. I would like to thank my supervisor, Dr. Qing Wang, for being so helpful during the thesis project, and I would also like to give a special thank to Dr. Jie Yang for being a part of the thesis committee. I want to thank my friends who are there for me to avoid the loneliness of studying abroad. I also thank all members of the embedded system group for providing insightful feedback for my thesis project.

Junyu Lu

Delft, The Netherlands
20th June 2024

Contents

Chapter 1

Introduction

Artificial Intelligence, often abbreviated as AI, stands at the forefront of technological innovation, reshaping the landscape of industries and revolutionizing the way we interact with machines. At its core, AI refers to the development of computer systems capable of performing tasks that typically require human intelligence. These tasks encompass a wide range of functionalities, from understanding natural language and recognizing patterns to making autonomous decisions and learning from experience.

Today, AI manifests itself in various forms, including virtual assistants like Siri and Alexa, recommendation systems that power personalized content delivery, autonomous vehicles that navigate complex environments, and algorithms that drive financial trading decisions. Its applications span industries such as healthcare, finance, transportation, manufacturing, and entertainment, catalyzing innovation and efficiency while presenting new opportunities and challenges.

Running AI on edge devices such as microcontroller units (MCUs) presents significant challenges due to their limited storage and computational resources. MCUs often have very restricted amounts of RAM and storage, sometimes only a few kilobytes to a few megabytes. This severely limits the size and complexity of AI models that can be deployed, as large models with millions of parameters require substantial memory and storage capacity that MCUs simply do not have.

In addition, MCUs generally have lower processing speeds and fewer computational cores compared to more powerful devices. They are designed for low-power operations and cannot handle the intensive computations required by complex AI models. This limitation can result in higher latency, making it difficult for MCUs to perform the necessary computations efficiently.

Given these constraints, efficiently deploying AI models in MCUs becomes a highly interesting topic of exploration. One promising approach to address these challenges is distributed inference, where the computational load is shared among multiple devices. In this setup, MCUs can perform lightweight tasks, while more intensive computations are distributed across other MCUs in the network. This approach leverages the combined computational power of multiple devices, potentially overcoming the limitations of individual MCUs and enhancing the overall efficiency and scalability of AI deployment in resource-constrained environments. Exploring and refining distributed inference techniques is crucial to making AI feasible in MCUs, balancing performance with the inherent resource constraints of these devices.

In this thesis, we explore a possible way to use distributed inference on multiple MCUs to handle complex inference tasks, effectively distributing the computational load across several devices to overcome individual limitations in memory and processing power, thus enhancing the overall capability and efficiency of AI deployment in constrained environments.

1.1 Problem statement

The constrained computation and storage capabilities inherent in edge devices, such as MCUs, present a formidable hurdle to the effective deployment of deep learning models. With off-the-shelf MCUs typically offering RAM sizes ranging from a mere few KBs to tens of MBs, and lacking floating-point arithmetic units in many cases, the challenges are multifaceted. Conventional techniques such as quantization and weights pruning have been used to adapt deep learning models for edge device deployment. However, these methods often fail when it comes to the stringent resource limitations of a single MCU. Novel frameworks such as TF-Lite, TensorRT, and OpenVino provide efficient runtime systems for edge devices such as mobile phones; however, they have not yet reached a suitable size for MCUs [11]. Emerging methods like MCUNetv2 [10] succeeded in significantly reducing peak memory usage, enabling high-resolution input inference on MCUs [11], at the cost of computation overhead and long inference time and is not applicable for models larger than MCU's flash memory size.

Currently, there are various methods that make use of distributed computing for deep learning models on edge devices. Nonetheless, the majority of these methods are intended for devices with considerably more resources than MCUs, such as smartphones. This thesis aims to bridge the gap by focusing on the deployment of such complex models on MCUs.

1.2 Research goal

The main aim of this master's thesis is to develop an innovative workflow that enables distributed inference on resource-constrained MCUs, with a focus on enhancing the feasibility and speed of the inference.

1.3 Research challenges

To achieve the research goal, there are multiple challenges need to be addressed.

1. **Finding a suitable distributed algorithm.** The primary challenge is to identify or develop a distributed algorithm that can effectively manage inference tasks across multiple MCUs.
   (a) Efficiency: The algorithm must be optimized for limited processing power and memory available in microcontroller units.
   (b) Scalability: It should scale up as the number of MCUs increases.
2. **Ensuring optimized latency and communication overhead.** It is important to reduce the overall communication cost as much as possible.

Communication protocols need to be efficient to avoid excessive overhead, which could slow down the system.

3. **Balancing workloads across MCUs.** To prevent any single MCU from becoming a bottleneck, workloads must be distributed according to the ability of each MCU to handle tasks.

1.4 Contributions

This thesis provides three main contributions.

1. Conducted an in-depth analysis of the computational and memory limitations inherent in contemporary MCUs. This analysis specifically highlighted the significant memory constraints that MCUs face, such as limited RAM and flash memory storage capacities. These memory limitations pose substantial challenges for the implementation of machine learning algorithms, which often require large amounts of memory to store models and process data. Despite these challenges, the analysis also identified promising opportunities through the adoption of distributed inference techniques. By distributing the inference tasks across multiple MCUs, it is possible to mitigate the memory constraints of individual devices. This approach allows for more complex models to be utilized without overburdening a single MCU's memory capacity. In addition, advances in communication protocols and efficient partitioning strategies are essential to optimize the performance and reliability of distributed inference systems. Using these strategies, it is possible to enhance the capabilities of MCUs, enabling more sophisticated and resource-efficient machine learning applications in IoT and embedded systems.

2. Developed a novel distributed inference workflow specifically designed for deploying CNNs on MCUs. This workflow addresses significant computational and memory constraints of MCUs by partitioning CNN models into smaller, more manageable patches that can be processed across multiple MCUs. By distributing the inference tasks, the workflow effectively mitigates the limitations of individual MCUs, enabling the deployment of more complex and resource-intensive CNN models than would otherwise be impossible. In addition, this approach incorporates efficient partitioning strategies to ensure flexible and reliable operation. This innovative solution opens up new possibilities for implementing sophisticated machine learning applications in resource-constrained environments, improving the capabilities of IoT devices and embedded systems.

3. Provided an in-depth analysis of the distributed inference process and proposed a mechanism for rating each MCU to distribute the workloads based on their respective computation and communication resources.

1.5 Structure

The structure of the thesis is arranged as follows. Chapter 2 discusses some of the related works of this thesis and provides a background survey of the

current computation and memory resources of MCUs, comparing them with the requirements of contemporary deep learning models. Chapter 3 describes some of the preliminary steps taken to facilitate the implementation of the thesis project. Chapter 4 details the method for conducting distributed inference on MCUs, Chapter 5 provides an analysis of the workflow optimization strategy, and Chpater 6 presents simulation results, while Chapter 7 presents the testbed implementation on Teensy 4.1 MCU boards along with the results obtained.

Chapter 2

Related work & background

This chapter first discusses relevant research in the field of Tiny Machine Learning (TinyML), and then provides an overview of the computation and memory resources available in current off-the-shelf MCUs. We then analyze the challenges associated with the deployment of deep learning models on these devices.

Recently, there has been considerable research aimed at reducing the size of deep learning models for deployment on edge devices, these methods can be referred to as *Tiny Machine Learning*. Different methods have been proposed in TinyML, and these methods can be summarized into two categories, Algorithm solutions and System solutions [11].

2.1 Algorithm solutions

Quantization: Quantization refers to the process of reducing the precision or range of numerical values in data. It involves approximating continuous values with a smaller set of discrete values, typically to improve efficiency in storage, computation, or transmission while minimizing loss of information. In the field of TinyML, considerable works are proposed to quantize model for running on IoT devices and MCUs [3][6][20][23]. Most models are quantized to 8-bit from 32-bit, resulting in reducing the size to one fourth, but normally there is going to be an accuracy loss if not fine tuned. However, there are works that focuses on retaining the model accuracy while reducing the model size, for example, in [23], yuejiao etal. proposed to use mixed precision quantization together with reinforcement learning to find the optimal quantization strategy for target hardware resources while also keep the accuracy of the model nearly unchanged. Po-Yuan etal. [3] proposed to use multi-scale dynamic quantization scheme to divides the quantization ranges into different regions, and each region is assigned different quantization levels and quantization parameters, and kept the accuracy drop of the quantized model within 1 percent.

When applying models to MCUs, they must be quantized to very low bit widths because of limited memory resources. This leads to a significant drop in accuracy, making it impractical to rely solely on quantization for the deployment of complex deep learning models in MCUs.

Weights Pruning: Weights pruning involves removing redundant or less important connections (weights) from neural network models. This process

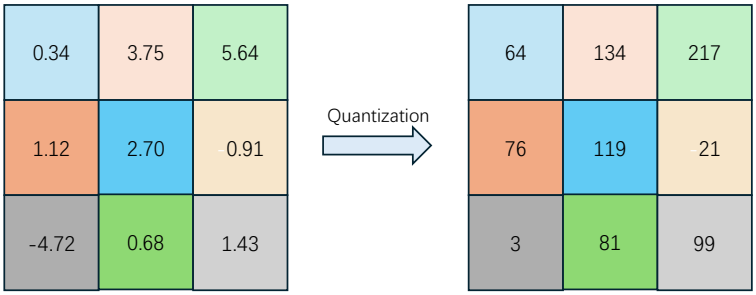


Figure 2.1: **An example of using quantization to compress the model**

reduces model size, speeds up computation, and can lead to more efficient inference. Techniques include setting small weights to zero or removing connections based on their magnitude or importance in the network [1][2][13][25]. While weights pruning can effectively reduce the model size, it can also bring about the drop in accuracy and training overhead. Several studies have focused on enhancing the model's performance during the pruning process, for example, in [13], a novel approach combining Patterned Weight Searching (PWS) and Kernel Equivalent Splitting (KES) are proposed to improve the model accuracy while also achieved model compression and general platform compatibility during pruning. However, using weights pruning alone won't be sufficient to run complex deep learning models on MCUs due to the relatively large size after pruning.

Knowledge distillation: Knowledge distillation is a technique where a large, complex model (teacher) transfers its knowledge to a smaller, simpler model (student). The student learns to mimic the behavior of the teacher by minimizing the difference between their predictions. This process enables compact models to achieve performance close to the larger ones [19][21][22][24]. However, knowledge distillation depends heavily on the teacher model, and sometimes, the student model does not generalize well to unseen data resulting in drop in accuracy, knowledge distillation is a promising field for IoT devices, but when talking about MCUs, the limited memory resources make it very challenging to find a student model that will perform well.

Neural architecture search Neural Architecture Search (NAS) automates the design of neural network architectures. It explores a search space for pos-

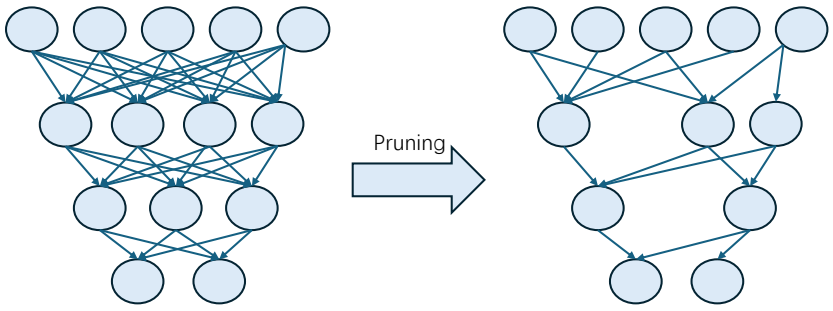


Figure 2.2: **An illustration of performing weights pruning**

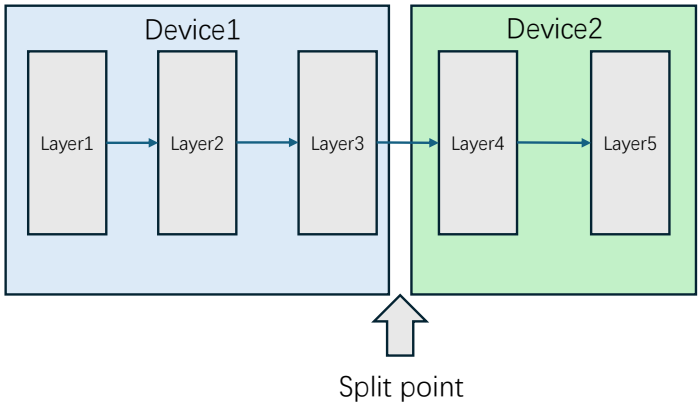


Figure 2.3: **An illustration of the coarse-grained split strategy**

sible architectures, evaluates their performance on a task, and selects the best-performing ones. NAS aims to discover architectures that are highly effective for specific tasks, improving model performance and efficiency [12][17]. There are various methods aimed at performing NAS, but for MCUs, the search space for it is relatively small, making finding a suitable model architecture for various tasks a very challenging task.

2.2 System solutions

For system solutions of TinyML, a lot of inference frameworks were proposed to enable MCU inference, for example, TF-Lite proposed by Google is a light-weight framework for deploying machine learning models on mobile and embedded devices efficiently. CMIS-NN, proposed by Arm, is a collection of different deep learning kernels that aim to minimize the memory footprint of the neural network. Although these framework enables the inference on MCUs, they limited the size and complexity of the models that can be deployed on MCUs.

In my thesis, I investigated the feasibility of employing distributed inference to address this issue. Some similar research has also been conducted that explores distributed inference on edge devices. For example, in [14], the authors distributed a model on Android smartphones by dividing the convolution layers according to the computing capacity of each smartphone. Similarly, [5] proposed distributing convolution layers across different GPUs according to 2D space dimensions. However, these methods primarily focus on accelerating inference speed and are tailored for devices with larger storage capacities than MCUs, making them unsuitable for MCU implementation.

Some studies consider a coarse-grained split strategy at the level of individual layers, as shown in Figure 2.3. In [7], the authors explored a scenario where there are both servers and edge devices, proposing an adaptive split point for layers to accommodate the gains of the wireless channel varying over time and reduce the latency of inference. Similarly, [8] proposed server-device collaboration with an optimal strategy derived through an online change point detection algorithm and model right-sizing using early exit. However, these approaches are not suitable for MCUs because of the coarse-grained layer allocation, which is infeasible for highly resource-constrained MCUs.

More recently, Li et al. [9] tackled the high computational cost of generating high-resolution images with diffusion models using parallel computation across multiple GPUs. They split the diffusion models into patches while optimizing communication costs. However, this approach is specific to diffusion models and designed for GPUs, rendering it inapplicable for MCUs.

2.3 Survey of MCUs

We collected data from 28 MCUs that were the top sellers online. The result is shown in Table 2.1. The survey was conducted by me alone, and the survey method involves browsing the official websites of MCU manufacturers, technical forums, and MCU datasheets to gather relevant data, such as RAM size, flash memory size, frequency, presence of floating-point units, which are then recorded for analysis. Furthermore, to ensure the accuracy and reliability of the survey result, cross-referencing with multiple sources and verifying data consistency were essential steps undertaken during the data collection process.

From Table 2.1, we can see that the RAM and flash memory sizes of various MCUs range from a few kilobytes to tens of megabytes. Similarly, the operating frequencies range from tens of millions to hundreds of millions of hertz. Notably, some of the MCUs lack floating-point units.

These properties of different MCUs make the deployment of deep learning model challenging. To start with, low operation frequency and lack of floating-

Table 2.1: **The list of surveyed MCUs**

MCU name	RAM size (KB)	Flash memory (KB)	FPU	Frequency (max.)
Teensy4.1	1024	8192	Yes	600MHz
Teensy4.0	1024	2048	Yes	600MHz
STM32L031K6	8	32	No	32MHz
STM32F407VG	192	1024	Yes	168MHz
STM32F103C8T6	20	64	Yes	72MHz
STM32F303CBT6	32	128	Yes	72MHz
STM32L476RG	128	1024	Yes	80MHz
STM32H743ZI	1024	2048	Yes	400MHz
ATmega328P	2	32	No	20MHz
ESP32	520	4096	Yes	240MHz
PIC32MX250F128B	32	128	No	50MHz
STM8S103F3P6	1	8	No	16MHz
LPC1768	64	512	No	100MHz
MSP430G2553	0.5	16	No	16MHz
SAM D21E18A	32	256	No	48MHz
PSoC 6 BLE	288	1024	Yes	150MHz
RX65N-2MB	640	2048	Yes	120MHz
STM32F767ZIT6	512	2048	Yes	216MHz
ATSAMD51J19A	192	256	Yes	120MHz
RZ/A1H	10240	10240	Yes	400MHz
STM32L152RET6	80	512	No	32MHz
TMS320F28335	68	512	Yes	150MHz
MAX32660	96	256	Yes	96MHz
ATSAML21E18B	32	256	No	48MHz
LPC2148	32	512	No	60MHz
STM32F401RE	96	512	Yes	84MHz
ATSAM4S16B	128	1024	Yes	120MHz

point units make the calculation of deep learning models (which are almost always trained with floating-point numbers on high-performance GPUs) very slow; second, the small RAM size limits the peak RAM usage for different deep learning models, which makes the deployment of complex deep learning models almost impossible; lastly, the limited flash memory size (storage size) limits the total size of the deep learning model that can be deployed.

However, one advantage of MCUs is that they are not expensive and can be purchased easily. Moreover, with the rapid development in the field of Internet of Things, MCUs can now be networked using various communication techniques, such as LoRA, Bluetooth, Zigbee, etc. Combining these two advantages, we can tackle the challenges mentioned above by exploring the potential of distributed computing among networked MCUs. By making use of their low cost and easy accessibility, together with their ability to communicate through various protocols, we can create a network of interconnected MCUs capable of collectively performing deep learning inferences.

This distributed computing approach can help alleviate the computational and memory constraints of individual MCUs by distributing the workloads across multiple MCUs. Each MCU can contribute its computational resources and memory to collectively execute deep learning models, effectively combining resources to overcome the limitations of individual devices.

Chapter 3

Preliminaries

This chapter explains the initial steps taken to implement the project. First, we discuss how we store the weights and information for the CNN model. Then, we introduce the framework we used for implementation and the setup of the software testbed used in this project.

3.1 The reinterpretation of pre-trained models

This project requires that we be able to manipulate individual neurons in deep learning models. Traditional frameworks like PyTorch and TensorFlow only let us operate at the layer level, not at the individual neuron level. Most deep learning models today are built with these frameworks, which means that we cannot easily make the fine-grained changes we need. Therefore, we need to find a new way to represent these pre-trained models that allows for more detailed and flexible manipulation.

For this project, we consider the reinterpretation of Pytorch models. PyTorch is an open-source deep learning framework developed by Facebook's AI Research lab. It provides flexible and efficient tools for building and training neural networks, featuring dynamic computational graphs, strong GPU acceleration, and seamless integration with Python. PyTorch is widely used for research and production in the field of AI.

The workflow of reinterpreting the pre-trained model is shown in Figure 3.1, we can either train our own model written in Pytorch or use the pre-trained model provided by Pytorch community; the model's computation graph is then traced through different Pytorch modules by performing one forward propagation, during which hooks are inserted onto each individual module. During this process, the module type, the layer weights, and the layer information are saved. Lastly, these pieces of information are recorded in a JSON file.

The format for storing layer weights and layer information is shown in Table 3.1. Here we consider 4 kinds of layer; these layers are most common in a CNN, other layers like adaptive pooling can be stored using similar format.

After storing the weights data and relative information in the JSON file, the next step is to distribute them to each MCU; more details are given in Chapter 4.

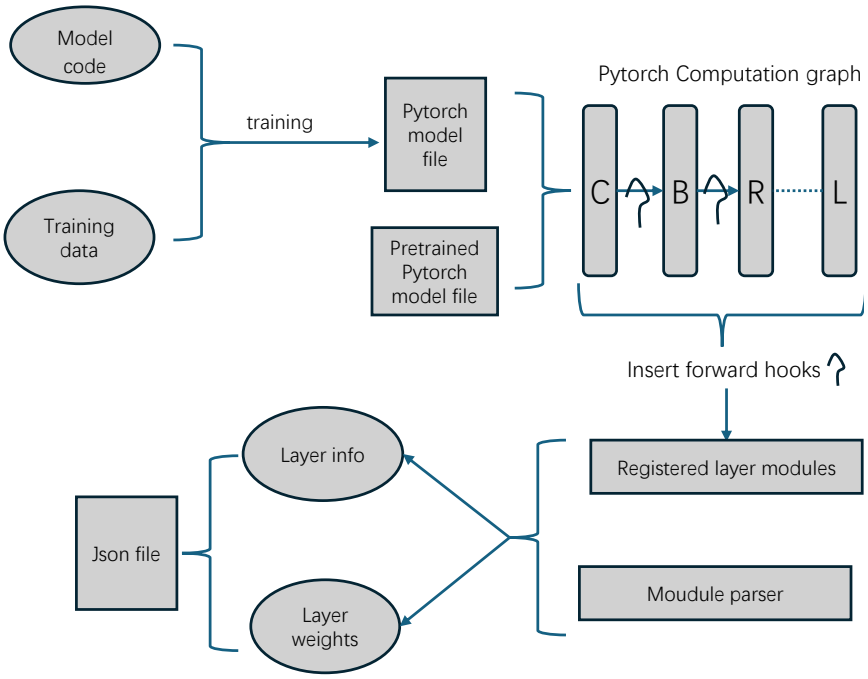


Figure 3.1: **An illustration of the process of interpreting the Pytorch model. In the figure 'C' stands for Convolutional modules, 'B' stands for Batchnorm modules, 'R' stands for Relu modules and 'L' stands for Linear modules**

3.2 Inference framework and testbed

As mentioned before, in order to manipulate the model on an individual neuron level, just storing the weights and information data is not enough, we also need the corresponding functions to manipulate the model, as a result, we decided to write a very simple inference framework for the project just for CNNs. To ensure that we have correctly implemented everything, we also constructed a testbed to ensure the correctness of each sub-module before fusing them together in the project.

3.2.1 Inference framework

The goal of building an inference framework is to allow us to have an easy and specific way to manipulate the model on the level of individual neuron. This framework should have all the necessary functionalities that we need to implement this project.

The framework is written in Rust language, the model is read from the JSON file acquired from Section 3.1 and transferred into several objects with the same Rust public *Layer* trait. This public trait allows for an easy and straightforward access of neurons and weights as well as showing their relations with each other in the layer. For a more detailed description of the implementation of this trait, please refer to the appendix of the thesis.

Table 3.1: **The format for storing the weights in JSON file**

Layer type	Convolution	Linear	Batchnorm	Relu
Weights and Information	Weights	weights	weights	input dimensions
	output per group	input batch size	bias	
	input per group	input channel number	mean	
	stride	output batch size	variance	
	kernel size	output channel number	input dimensions	
	input dimensions			
	output dimensions			
	bias			

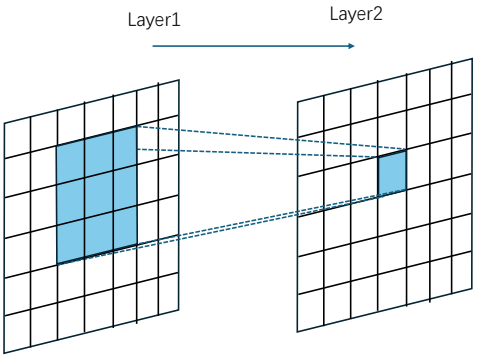


Figure 3.2: **Receptive field**

The most important functionality of this trait is to find the relative receptive field of each neuron in the convolutional layer, as the receptive field plays an important role in deciding which input neurons are used to calculate the output, as indicated in Figure 3.2.

The specific function implemented for this trait can trace back the receptive field of the specific neuron of the output feature map, producing an indicator of the input neurons used for calculating this output neuron, enabling the fine-grained handling of each individual neuron. This method is used to implement the distribution method in Chapter 4.

3.2.2 Software testbed construction

To make sure that we have correctly implemented everything, we built a testbed to test each individual component of the framework before combining them. The test bed is also written in the Rust language, making use of the functionalities

of the unit tests in Rust language. The procedure of testing each individual module follows a systematic approach.

- **Component Isolation:** Each component of the framework is isolated to ensure it functions independently. This isolation is achieved by mocking dependencies and using Rust's std::test module.
- **Test Case Development:** For each isolated component, specific test cases are developed to cover various scenarios, including edge cases. These test cases are designed to validate the component's functionality, performance, and error handling.
- **Execution and Verification:** The test bed executes the test cases and verifies the outputs against the expected results produced by Python scripts. Any discrepancies are logged for further analysis.
- **Integration Testing:** Once individual components are validated, integration tests are performed to ensure that the components work seamlessly when combined. This step involves creating comprehensive test scenarios that reflect real-world usage of the framework.

This meticulous testing procedure helps in identifying and resolving issues at an early stage, ensuring the reliability and robustness of the framework. Using Rust's powerful testing capabilities, we achieve a high level of confidence in the correctness and performance of our implementation.

Chapter 4

Distribution method

This chapter describes the overall workflow used to coordinate the distributed inference process and the method used to split the models into different MCUs.

4.1 Overall workflow

In order to perform distributed inference in a swarm of networked MCUs, the first step is to find a suitable workflow for the MCUs to collaborate. The workflow should be subject to the following conditions:

1. The peak RAM usage in each MCU must not exceed its RAM size.
2. The data stored in each MCU must not exceed its storage limit (flash memory size).
3. The workloads distribution among MCUs should be balanced to prevent overloading certain devices while underutilizing others.
4. The workflow should be adaptable and scalable to accommodate changes in the network topology or the addition/removal of MCUs.

Moreover, the overall workflow must also consider the fact that in the network, we might have different MCUs with different computational resources, memory resources, and communication delay.

4.1.1 The workflow

For the distributed approach to work, we need to devise a strategy for the collaboration of the MCUs. Considering the fact that each MCU only handles a fraction of the model's outputs, it is inevitable that these outputs will have overlapping areas during the computing of convolutional layer. To address this issue, we propose to assign 2 roles for MCUs, namely, **Coordinator** and **Worker**.

1. **Coordinator**: In charge of 1) sending inputs to the specific workers according to precalculated mapping; 2) storing intermediate results for residual connections.
2. **Worker:** In charge of doing calculations using the stored model weight fragment and sending the result to the coordinator.

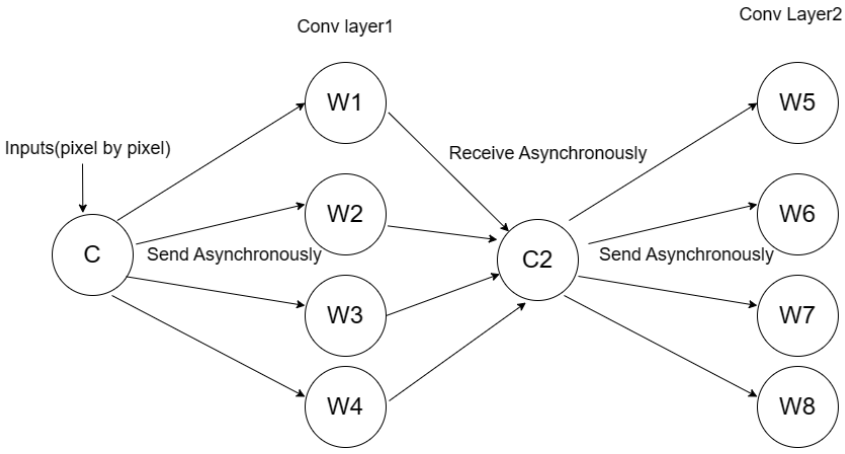


Figure 4.1: **The overall workflow**

After assigning these two roles, there is only one thing left to do: we need to decide how workers and the coordinator communicate with each other. Considering that coordinators are only in charge of relaying the data to different workers and storing data, it is possible to use a small number of coordinators to manage many workers. In addition, to make the system more robust, different workers should work independently as long as they have received enough data from the coordinator.

With these pieces of information in mind, we propose our method for the overall workflow as illustrated in Figure 4.1. Using two convolutional layers as an example, the first coordinator receives the input pixel and sends the pixel to the mapped worker according to the precalculated mapping, the worker then performs the calculation and sends the result to the next coordinator. The sending and receiving can happen asynchronously if optimized, meaning worker 1 can be sending results to coordinator 2 while coordinator 1 is sending pixel to worker 2.

It is worth mentioning that the role of coordinator can be played by the worker with large memory and computation resources; in this case, the Coordinator/Worker MCU can use a mechanism like preemption to handle the relaying of the data while processing its own workloads.

4.1.2 The rating mechanism

In a network of MCUs, it is highly possible that the network is consisted of MCUs with different specifications, network bandwidth and network latency. The purpose of proposing the rating mechanism is to find the relative workloads that each MCU should handle. MCUs with high ratings should be allocated with more workloads. As a result, the rating should be based on each MCU's computational and memory resources as well as communication delay in the network. A more detailed description of how we calculate the rating for each MCU will be given in Chapter 5.

Computation: The idea is that MCUs with higher processing power should handle more workloads to speed up the overall workflow. Considering a swarm

of MCUs, the rating for each MCU should be calculated based on their computation capability. The computational capability of MCUs is often characterized by their operation frequency, although other factors such as architecture, instruction set, and peripherals also play significant roles, in this thesis, we only considered the frequency factor and the presence of Floating-Point Units (FPU).

Communication: When talking about communication, we are talking about the time spent on passing on calculation results from one MCU to the others. There are two factors to consider for communication, the first one is the bandwidth of the network and the second one is the communication delay of each MCU.

Storage: When talking about storage size, we mean the flash memory size which is used for storing code and files and the RAM size which is used for allocating variable during run time. Unlike computation and communication, the storage limitation is a **hard** limitation, meaning exceeding the storage limitation of MCU will result in serious consequences, so we need to pay special attention to this requirement.

4.2 The splitting algorithm

This section discusses the algorithm used for splitting the model onto different workers and the algorithm used to calculate the mapping in each coordinator used to coordinate the workflow.

4.2.1 Split the model

To split the model onto different worker MCUs according to their ratings obtained in previous section, we proposed two algorithms corresponding to the convolutional layer and the linear layer.

For the convolutional layer, we have Algorithm 1 to distribute the output feature map into individual patches and allocate each patch to the corresponding MCU.

What we do basically is first calculate how many output neurons-which are represented by n in this algorithm-should be distributed to this worker based on its rating, then we iterate through the output positions to find which kernel we are now using, which is represented by $c1$, and distribute this weight kernel to worker if not distributed, or add 1 to the count of this weight kernel if we have already distributed this kernel. By doing the split this way, we ensure the flexibility of the split, which means that the split for MCU can vary a lot, some MCUs may be in charge of only tens of output positions, we can use this method to optimize the workloads distribution easily. An illustration of this distribution algorithm is shown in Figure 4.2.

Regarding the linear layer, we maintain the same R and N for the ratings and numbers. However, the weight matrix now has the shape h, w, and W represents the weight matrix to be distributed. The process is quite similar to that of convolutional layers, with the key difference being that we distribute the columns of the weight matrix to the workers. Figure 4.3 illustrates this split.

Algorithm 1 Weight split for Workers convolutional layer

Require: c, h, w: output channels, width, height of this layer
R: ratings for MCUs, MCU: set of MCUs
N: number of worker MCUs for this layer
W: weight kernels to be distributed with length c

$s \leftarrow 0$
for r **in** $0..N$ **do**
  $n \leftarrow \frac{R[r]}{\sum_{i=1}^{N} R} \times c \times h \times w$
  $i \leftarrow s$
  while $i - s < n$ **do**
    $c_1 \leftarrow \frac{i}{w \times h}$
    if $W[c_1]$ not distributed to $MCU[r]$ **then**
      distribute $W[c_1]$ to $MCU[r]$
    else
      count of $W[c_1]$ **in** $MCU[r]$+=1
    end if
    $i \leftarrow i + 1$
  end while
  $s \leftarrow i$
end for

Algorithm 2 Weight split for Workers Linear layer

Require: h, w: width, height of the weight matrix
R: ratings for MCUs, MCU: set of MCUs
N: number of worker MCUs for this layer
W: weights to be distributed with size $h \times w$

$s \leftarrow 0$
for r **in** $0..N$ **do**
  $n \leftarrow \frac{R[r]}{\sum_{i=1}^{N} R} \times h$
  $i \leftarrow s$
  while $i - s < n$ **do**
    distribute $W[i]$ to $MCU[r]$
    $i \leftarrow i + 1$
  end while
  $s \leftarrow i$
end for

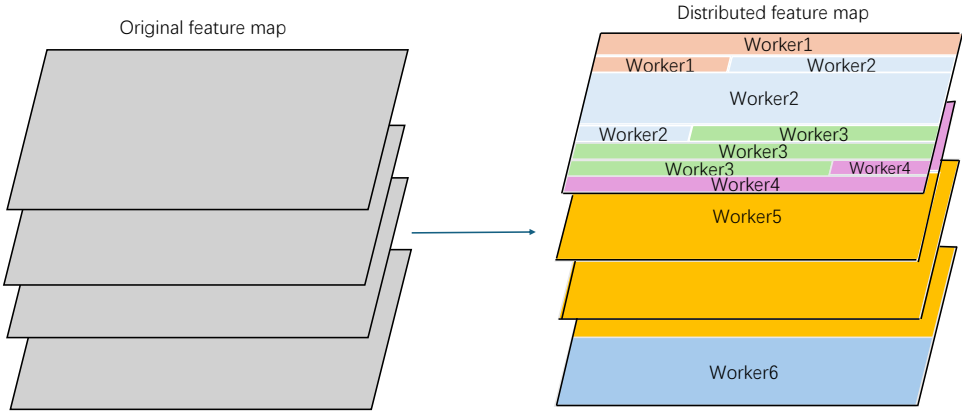


Figure 4.2: **Distribution of Convolutional layers**

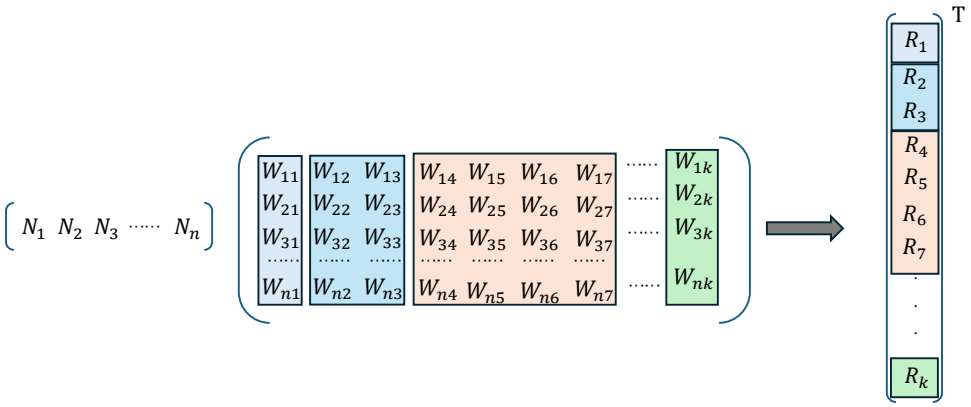


Figure 4.3: **Distribution of Linear Layers**

4.2.2 Get the mapping

The coordinator's task is to map each input and the calculation results from the source or previous layer's workers to the next layer's workers. The calculation results from the previous layer serve as the inputs for the current layer. We understand how these results are split and how to distribute the workloads for the next layer using Algorithm 1 and Algorithm 2. With this knowledge, we propose Algorithm 3.

In this context, the subscript 0 indicates the previous layer, while 1 indicates the current layer. We start by initializing two mappings: the raw mapping and the real mapping, denoted as $Raw\_Mapping$ and $Mapping$ in Algorithm 3. The raw mapping assigns each input position to a worker in the current layer, whereas the real mapping assigns the raw mapping to a worker in the previous layer. Lines 3 to 18 describe the process of obtaining the raw mapping, and lines 19 to 31 explain how to derive the real mapping.

For raw mapping, we intend to find the relative mapping of output positions of this layer, similar to Algorithm 1, we iterate through the output of this layer, and use the ratings of MCUs to find out how many output positions that each

Algorithm 3 Input mapping for coordinator convolution layers

Require: C_0, H_0, W_0: output channels, width, height of previous layer
C_1, H_1, W_1: output channels, width, height of this layer
R_0: ratings for previous layer MCUs,
R_1: ratings for this layer MCUs,
N_0: number of worker MCUs for previous layer
N_1: number of worker MCUs for this layer

Initialization: $Raw\_Mapping \leftarrow [C_0 \times H_0 \times W_0]$
Initialization: $Mapping \leftarrow [N_0]$
$s \leftarrow 0$
for r **in** $0..N_1$ **do**
    $n \leftarrow \frac{R_1[r]}{\sum_{i=1}^{N_1} R_1} \times C_1 \times H_1 \times W_1$
    $i \leftarrow s$
    while $i - s < n$ **do**
        $c_2 \leftarrow \frac{i}{W_1 \times H_1}$
        $h_2 \leftarrow \frac{\mathbf{mod}(i, W_1 \times H_1)}{W_1}$
        $w_2 \leftarrow \mathbf{mod}(\mathbf{mod}(i, W_1 \times H_1), W_1)$
        $positions \leftarrow get\_input(c_2, h_2, w_2)$
        for *position* **in** *positions* **do**
            bitwise or$(Raw\_Mapping_{[position]}, (1 << r))$
        end for
        $i \leftarrow i + 1$
    end while
    $s \leftarrow i$
end for
$s \leftarrow 0$
for r **in** $0..N_0$ **do**
    $n \leftarrow \frac{R_0[r]}{\sum_{i=1}^{N_0} R_0} \times C_0 \times H_0 \times W_0$
    $i \leftarrow s$
    while $i - s < n$ **do**
        $c_3 \leftarrow \frac{i}{W_0 \times H_0}$
        $h_3 \leftarrow \frac{\mathbf{mod}(i, W_0 \times H_0)}{W_0}$
        $w_3 \leftarrow \mathbf{mod}(\mathbf{mod}(i, W_0 \times H_0), W_0)$
        Record$(Mapping_{[r]}, Raw\_Mapping_{[c3, h3, w3]})$
        $i \leftarrow i + 1$
    end while
    $s \leftarrow i$
end for

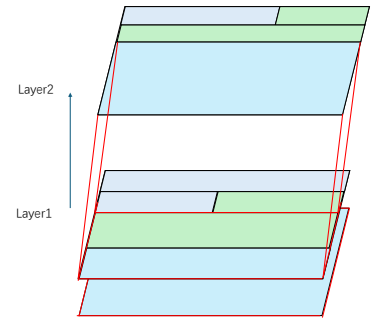


Figure 4.4: **Illustration of the mapping**

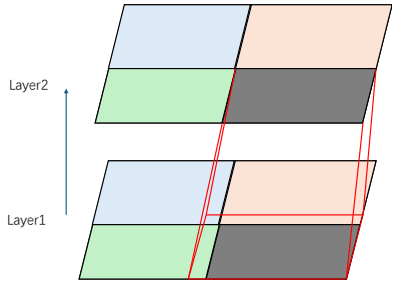


Figure 4.5: **Split strategy in COTS [5]**

MCU should handle, next, for all these output positions in each individual MCU, we use *get_int*() function to trace back the receptive field and find out which **input** positions are used to calculate these **output** positions, then, we mark these **input** positions by doing a *bitwiseor* operation with 1 left shifted by the *MCUid* to mark these positions as *claimed* by this individual MCU.

Note that Algorithm 3 only considers convolutional layers. Linear layers in CNNs often serve as a classification network, it is obvious that every single input position should be *claimed* by the linear layer's workers.

An illustration of this algorithm is shown in Figure 4.4, in the figure, to calculate the partition of the blue part in the second layer, we need the corresponding input neurons in the first layer as indicated by the red line, the coordinator is responsible for storing such knowledge and sending these input data to the corresponding worker according to the mapping.

4.3 Comparison of different split strategy

The split strategy for the models can play a huge role in reducing the peak RAM usage and deciding the communication protocol. In this section, we discuss several split strategies proposed in recent research and compare them with the split strategy proposed in this thesis.

In COTS paper [5], the author proposed to distribute the image arrays onto different GPUs to accelerate the workflow, the split scheme of this paper is shown in Figure 4.5 This split strategy split the feature map matrix into several sub-matrix and then each GPU is responsible for handling one sub-matrix. This

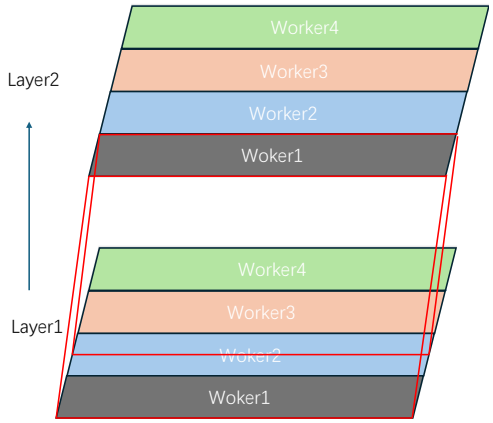


Figure 4.6: **Split strategy in MoDNN [14]**

split strategy is effective in paralleling the computation across several separate GPUs, however, this method requires the frequent communication among the workers, for example, to compute the grey part in the second layer of Figure 4.5, the GPU will need the parts indicated by the red line in the first layer, which overlaps with three other GPUs's workloads. What's more, as this split strategy is intended for GPUs, it fails to consider the memory limitation of each worker when deployed in an MCU based environment as each sub-matrix could exceeds each MCU's RAM size when the intermediate feature maps get larger.

In the MoDNN paper [14], the author proposed a way to partition the input matrix along its longer edge based on the computing capabilities of individual nodes for implementation on smartphones, as shown in Figure 4.6.

This split strategy further reduced the data transfer among different nodes, to be more specific, in Figure 4.6, to calculate the workloads of worker1, one only needs to transfer the overlapping calculation result in layer1 of worker2 as marked by the red line.

This method proves to be effective when the channels of the output feature map is small; however, it fails to consider the peak RAM usage during inference for each worker node which is essential for implementing the inference process on MCUs. There are 3 parameters that decide the peak RAM usage during the calculations, *1) input neurons, 2) weights data, and 3) output neurons.* This split strategy requires the neurons of all the channels in both input and output feature maps, which will become large when the channel number is large.

In Figure 4.7, when the output feature map's channels is large, to compute the result, the worker also needs all the kernels of the layer to produce the output feature map, this can take up significant amount of RAM, for example, in MobileNetV2, the feature map with the largest kernels has channels of 960, each kernel is of size 3×3, in total, the kernel data alone could take up around $960 \times 3 \times 3 \times 4 = 34560 Bytes$, around 34KB, which is more than most power-efficient MCUs RAM size and this is not considering the input and output neurons.

The split strategy proposed in this thesis has a much more flexible distribution method which can satisfy most off-the-shelf MCU's RAM size. For spatial convolution, where each feature map requires the calculation of the whole of the

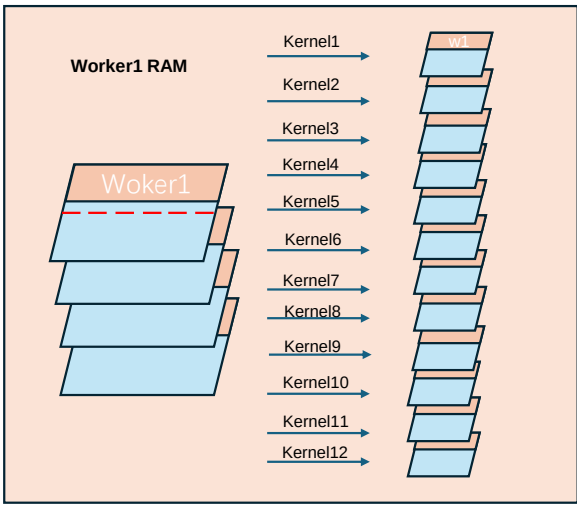


Figure 4.7: **RAM usage in worker MoDNN**

input feature maps, the usage of RAM is shown in Figure 4.8.

We can see from Figure 4.8 that the worker MCU with the least RAM can be allocated less workloads to fit its limited RAM size, as shown in the figure as worker1. It is worth mentioning that due to the nature of spatial convolution layers, MCUs allocated more workloads may require the full volume of the input feature map to perform its calculation, potentially increasing the peak RAM usage and communication cost in these MCUs, one possible solution of this is to split within each group of the convolution instead of the whole output volume, an illustration is shown in Figure 4.9. This method is very similar to the method proposed in MoDNN, the difference lies in the grouping of the feature maps and the fact that the split is done based on neurons instead of longer edges. However, when the input feature map and output feature map are within the same group(i.e., spatial convolution), this method still suffers from a large channel number. When the feature map and the output feature map are 1 to 1 (ie, depth-wise convolution), this method is exactly the same as the method proposed in Section 4.2.

To summarize, while our method proposed in Section 4.2 effectively reduces peak RAM usage in most cases compared to other approaches, it encounters challenges with spatial convolution on very large input feature maps. In such instances, the improved method described in Figure 4.9 can be utilized to address this issue.

Considering the fact that a large volume of input feature maps usually appears in the first few layers of the network, as shown in Figure 4.10, it is possible to combine the two methods proposed to further reduce the maximum RAM usage.

For most CNNs, the input is usually an RGB image with 3 channels and large spatial dimensions in width and height, for example, models trained on ImageNet have an input image size of $3 \times 224 \times 224$, while models trained on Cifar-10 and Cifar-100 have an input image size of $3 \times 32 \times 32$. The input image is then fed into the network, where the number of channels is increased and the spatial dimensions are reduced. Because the spatial dimensions of the feature

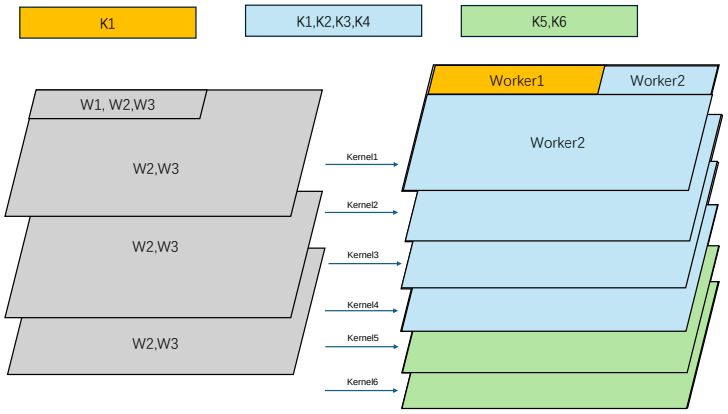


Figure 4.8: **RAM usage in each worker using our spilt strategy in spatial convolution**

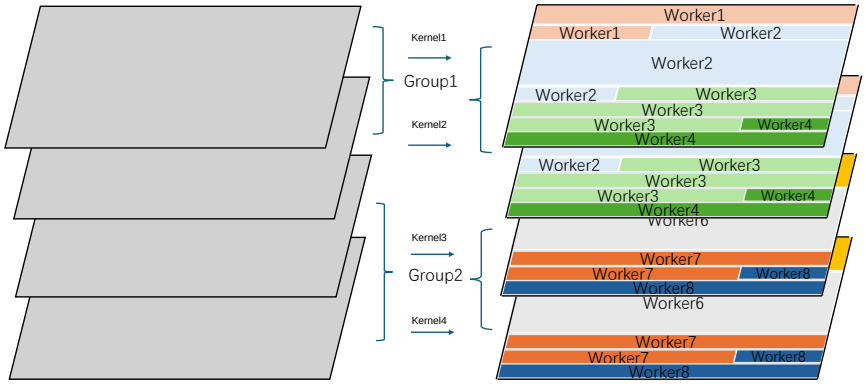


Figure 4.9: **A possible improved split strategy for reducing RAM usage on large input feature maps**

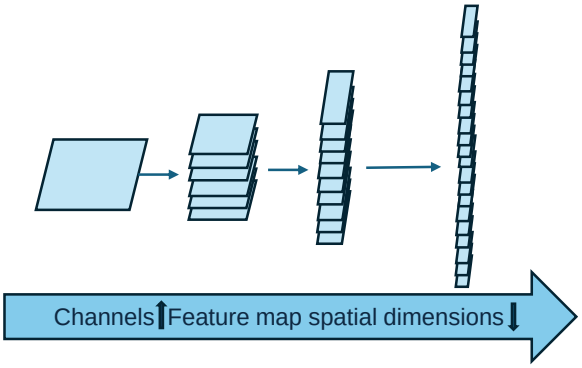


Figure 4.10: **An illustration of feature maps' dimensions in most CNNs**

map is of n^2, and the number of channels is of n, it is often seen that the largest feature maps occur in the first few layers of the network. So, if we use the improved split strategy for the first few layers and use the original split strategy for the other layers, it is possible to further reduce the memory footprint on each worker. However, adapting 2 split strategy may incur additional overhead such as workflow synchronization.

During our implementation and simulation in the following chapters, we did not use the improved split strategy, as the original strategy is sufficient to run the inference on our chosen MCU, the real implementation of the improved split strategy is left for future work.

Chapter 5

Optimization

In this chapter, we discuss some of the optimization methods we used to reduce the model's size and speedup the workflow while striving to maintain model accuracy, each optimization's effect being analyzed relation to the requirements of this thesis project.

5.1 Layer fusion

To balance the need to maintain model accuracy with the necessity of reducing the model's size, we implemented layer fusion algorithms, which involve the combination of certain layers in the neural network. Specifically, in this thesis, we focused on a commonly used fusion technique that integrates the convolution, batch normalization (BatchNorm), and rectified linear unit (ReLU) layers. Layer fusion is a strategy used to streamline neural networks by merging several operations into a single, more efficient operation. This process reduces the computational burden and memory footprint of the model without significantly compromising its performance. In the context of this thesis, the goal was to make the model compact enough to be suitable for deployment on resource-constrained MCUs while still retaining high accuracy. The fusion method adapted in this thesis involves combining these three types of layers into a single fused layer. This process typically includes the following steps:

1. **Combining Operations**: The convolution, batch normalization, and ReLU operations are merged into a single composite operation. This reduces the number of separate computations the model must perform, thereby streamlining its execution.

2. **Parameter Consolidation**: The parameters from the batch normalization layer (such as the scale and shift parameters) are incorporated into the convolution layer's filters, effectively transforming the convolution operation to include normalization.

3. **Activation Integration**:The ReLU activation function is applied directly to the output of the fused convolution and normalization operation, further simplifying the model's structure.

An illustration of how we implemented layer fusion in the thesis project can be found in [15]. Essentially, what we do is fold the BatchNorm layer's weights and biases into the weights and biases of convolutional layers.

Implementing layer fusion is a very important step for this thesis work; by fusing layers together, we not only reduced the size of the model to be distributed, but also reduced the total number of calculation operations to be performed by reducing the complexity of the model, thus increasing the model's performance on MCUs.

5.2 Quantization

Although the proposed method is theoretically capable of achieving full-precision (32-bit) inference across a network of MCUs, practical limitations necessitate the consideration of alternative approaches. For example, many ultra-low power MCUs are not equipped to handle floating-point arithmetic efficiently because of the absence of on-chip Floating-Point Units (FPUs) and their inherently low operating frequencies. Consequently, these constraints make it important to adopt a quantization strategy that converts the model to use integer arithmetic only.

The quantization process we followed is largely based on the method detailed in Google's paper on quantization techniques [6]. However, instead of inserting fake quantization checkpoints during the training phase, we used a Post-Training Quantization (PTQ) approach. This involved using a calibration set to better understand the distribution of activations during inference. To perform effective quantization, we collected a calibration set consisting of 800 images, each with dimensions of $224 \times 224 \times 3$, sourced from the Hugging Face platform. The calibration set serves a critical role in the quantization process by providing representative data that help in accurately mapping the model's activations from floating-point to integer values. By performing forward passes with this calibration set, we can gauge the approximate distribution of activations, ensuring that the quantized model maintains high accuracy. Quantizing the model to 8-bit integers brought several notable benefits:

1. **Size Reduction**: The quantized model is approximately one-fourth the size of the original 32-bit model. This substantial reduction in model size is crucial to fitting the model into the limited memory available on MCUs.

2. **Reduced Complexity**: Integer arithmetic is computationally less intensive compared to floating-point arithmetic. By converting the model to use integer operations, we significantly lower the computational complexity, which is essential for the limited processing capabilities of some ultra-low power MCUs.

3. **Enhanced Efficiency**: The reduction in model size and computational complexity results in faster inference times and possible lower energy consumption, which are critical for the deployment of deep learning models on resource-constrained devices.

5.3 Workflow optimization

In the context of a network of MCUs, each device can exhibit a variety of characteristics, such as different operating frequencies, communication bandwidths, communication delays, RAM sizes, and flash memory sizes. The primary goal of workflow optimization in such a heterogeneous environment is to accelerate the overall inference process as much as possible. For each worker, the time it spent in total can be divided into 2 parts:

1. **Calculation Time**: The time required to perform the necessary computations for the given workload.
2. **Communication Time**: The time required to receive input data and send the results back to the coordinator.

The total time t spent by an MCU on these activities can be expressed using Equation 5.1

$$t = \frac{W}{f} + (d + \frac{1}{B})f(W), \tag{5.1}$$

where W represents work load in $MillionCycles$, d is communication delay in $seconds/KB$ instead of $seconds$ because of the acknowledgement signal, more detail will be given in Chapter 7, f is frequency of the MCU in MHz, B is bandwidth in KB/s. $f(W)$ is a function that calculate how many workloads in KB need to be received from and sent to the coordinator for this MCU, $f(W)$ can be represented using Equation 5.2

$$f(W) = K_1 K_c W, \tag{5.2}$$

where K_1 is decided by the hardware specification and algorithm complexity used to calculate the results, it is a proportion factor in $KB/MillionCycles$ that scale the million cycles into KB of corresponding output neurons, it is a constant for a given hardware specification and algorithm complexity. K_c is another proportion factor I like to call **communication coefficient** which determines how much data is received from and sent to the coordinator proportional to the workload(output neurons), K_c is a variable determined by the network structure of CNN, how the CNN is distributed and how the workflow is constructed. It is a value greater then 0, a small K_c means that the MCU tends not to communicate with other MCUs, for example, when, there is only one MCU in the network, $K_c = 0$, meaning no communication happens and all calculation happens in only one MCU. When $0 < K_c < 1$, it means that the MCU only sends a portion of its calculation result to the coordinator and receives a small portion of input from the coordinator while keeping all of its calculation results as the inputs for next layer.

To calculate the workload W for each worker MCU, we need to find out how many workloads each MCU can handle per second. Take Equation 5.1 and make $t = 1s$, we have

$$W(\frac{1}{f} + (d + \frac{1}{B})K_1 K_c) = 1. \tag{5.3}$$

Move W to the left and the rest to the right and multiply both sides by K_1, we have

$$WK_1 = \frac{fK_1}{(d + \frac{1}{B})fK_1 K_c + 1}. \tag{5.4}$$

In Equation 5.4, the left hand WK_1 represents the number of output positions in KB that the worker can handle per second, and the right hand shows a way to calculate this workload.

To balance the workload for each MCU, it is obvious that each MCU's workload should be proportional to its' ability to process workload, thus, the rating R_i for each MCU_i is given by Equation 5.5.

$$R_i = \frac{f_i K_1}{(d_i + \frac{1}{B_i}) f_i K_1 K_{c_i} + 1}. \tag{5.5}$$

For each MCU, the difficult part is to find its communication coefficient K_{c_i}, normally we can calculate this value by running simulations post-distribution by finding out how much data is received and sent to the coordinator during the whole inference period; more detail will be given in Chapter 6 and Chapter 7.

Next, we need to consider the storage/memory constraint, as mentioned in Chapter 4, the storage/memory constraint is a **hard** constraint, so, we need to take special care of it.

After calculating each MCU's rating according to Equation 5.5, we can run Algorithm 1 and Algorithm 2 to split the model, because we split the model linearly, the size of weight fragment after spliting can be calculated as

$$S_i = \frac{R_i S_m}{\sum_{j=1}^{n} R_j}, \tag{5.6}$$

where S_i represents the weight fragment size in MCU_i, S_m is the total model size, R_i : the rating for MCU_i, n : number of MCUs in the network.

According to Equation 5.6, to fit S_i into each MCU_i's storage, we need to adjust the rating R_i while keeping the sum $\sum_{i=1}^{n} R_i$ unchanged. Furthermore, to avoid overloading certain MCUs, we should keep the adjusted rating ratio as close to the original ratio calculated using Equation 5.5 as possible. For each MCU that receives more weights than its storage size, we have

$$R_{io} = \frac{(S_i - S_{it}) \sum_{j=1}^{n} R_j}{S_m}, \tag{5.7}$$

where R_{io} represents the overflowed rating for MCU_i,S_{it} is the total storage size of MCU_i. We then distribute R_{io} evenly to the MCUs that still have room left for more weight data and repeat this process until the weight data are fit into all the MCUs.

For RAM, it is crucial to identify the peak RAM usage during inference. We will discuss this in more detail in Chapter 6.

Chapter 6

Simulation

6.1 Considered model: MobilenetV2

We chose MobileNetV2 as our CNN model to run simulations on, given its efficiency and suitability for mobile and embedded applications. MobileNetV2 is a state-of-the-art CNN architecture proposed by Google, specifically designed to perform well on mobile platforms like smartphones. This design focus makes it an ideal choice for applications where computational resources and power consumption are limited.

The implementation of MobileNetV2 we utilized is based on PyTorch, a widely-used deep learning framework that offers flexibility and ease of use. For our simulations, we used an input size of $224 \times 224 \times 3$, which means that the network receives images with dimensions of 224 pixels by 224 pixels and three color channels (RGB). This input size is standard for many image classification tasks and ensures that the model can capture sufficient detail while maintaining manageable computational requirements.

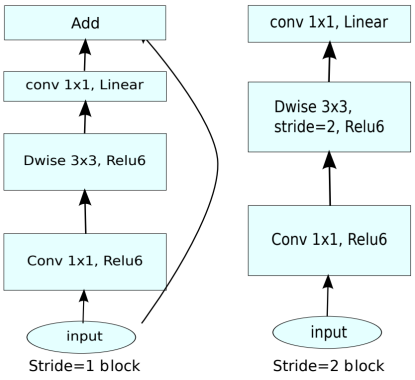
In terms of data representation, we used the original floating-point values with a precision of 32 bits just to simulate the inference without loss of accuracy.

MobileNetV2 is characterized by its computational efficiency, featuring a computational cost of approximately 300 million multiply-add operations. This relatively low cost is achieved through a series of optimizations in the network's architecture, including the use of depthwise separable convolutions, which significantly reduce the number of parameters and operations required. Specifically, MobileNetV2 employs around 3.4 million parameters, which is approximately 13 megabytes of storage. These parameters are organized into various layers and structures within the network, with a significant component being the bottleneck blocks. As illustrated in Figure 6.1, these bottleneck blocks play a crucial role in compressing the feature maps and subsequently expanding them, thus preserving important information while reducing the overall computational burden.

Overall, the design and implementation of MobileNetV2 make it an ideal choice for our simulation needs.

Input	Operator	t	c	n	s
$224^2 \times 3$	conv2d	-	32	1	2
$112^2 \times 32$	bottleneck	1	16	1	1
$112^2 \times 16$	bottleneck	6	24	2	2
$56^2 \times 24$	bottleneck	6	32	3	2
$28^2 \times 32$	bottleneck	6	64	4	2
$14^2 \times 64$	bottleneck	6	96	3	1
$14^2 \times 96$	bottleneck	6	160	3	2
$7^2 \times 160$	bottleneck	6	320	1	1
$7^2 \times 320$	conv2d 1x1	-	1280	1	1
$7^2 \times 1280$	avgpool 7x7	-	-	1	-
$1 \times 1 \times 1280$	conv2d 1x1	-	k	-	

((a)) **Network structure**



((b)) **The bottleneck blocks**

Figure 6.1: **Network structure of MobilenetV2 [18]**

6.2 Simulation setting

The simulation is done on my laptop which has a CPU of 11th Gen Intel(R) Core(TM) i7-11800H with 8 cores running at base frequency of 2.30GHz. The simulation only uses one coordinator with different size of workers, and the models are evenly distributed, meaning all the workers have the same rating. Each thread in PC represents an MCU, threads communicate through channels in Rust language. Figure 6.2 shows how the simulation works in case of 4 workers, it is worth mentioning that with only one coordinator, each worker MCU must wait for its turn to communicate with coordinator, thus causing more delay during the overall workflow.

To explain in more detail how the simulation works, let us look at Figure 6.2, in the first round, the coordinator thread reads the image on which to perform inference, then the coordinator gives these input pixels to the corresponding workers according to the mapping calculated using Algorithm 3. The workers, after receiving enough input from the coordinator, start performing calculations using the inputs gotten from the coordinator and the weight data stored in themselves. After the calculation is completed, the workers will send a signal to the coordinator to propose a potential transmission of results and wait for the permission of the coordinator. The coordinator receives this signal and sends permission to the proposing worker if the coordinator is not currently busy receiving results from another worker. The worker, after receiving permission, starts sending results to the coordinator, during the process, the coordinator receives the data, and then transfers the results to the corresponding MCUs according to the mapping. After the transmission is completed, the worker will send an end signal to indicate the end of the transmission, then the coordinator will tend to the next worker who has some data to send.

It is worth mentioning that during the process, there is no fear of data loss due to the nature of the simulation and the Rust language, so the coordinator can transfer the data as soon as possible to the next swarm of MCUs; however, in real settings, data loss is an important factor to take note of, which we will discuss further in Chapter 7.

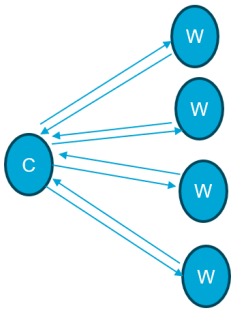


Figure 6.2: **Topology of the devices in the simulation**

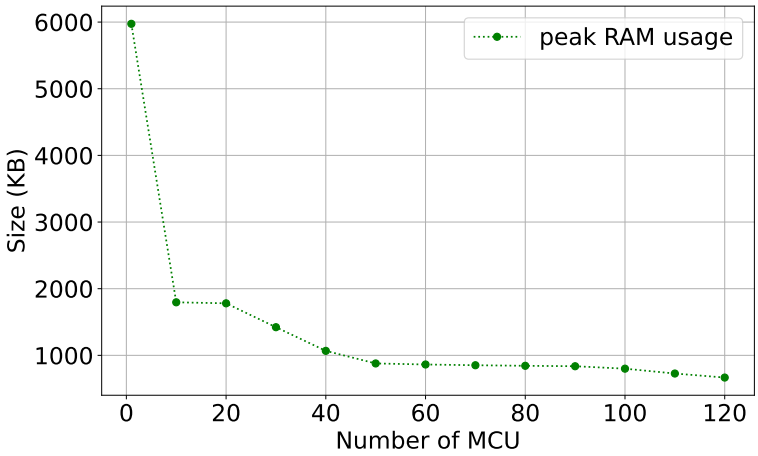


Figure 6.3: **Peak RAM usage in each MCU vs. number of MCUs**

6.3 Preliminary results

The purpose of running the simulation is to collect some data on how the number of MCUs affect the peak RAM usage, and the total inference time. So, from now on, we will present these results and some analysis of both. We will also discuss how we can acquire K_c through some detailed analysis.

Peak RAM usage: During the inference, the RAM usage in each worker is consisted of mainly 3 parts:

1. The input neuron data which is used to calculate the output
2. The weight data which is used together with the input neuron data to calculate the output
3. The output neuron data for storing the calculation result

So in order to find the peak RAM usage and which layer has the peak RAM usage, we inserted check points in each layer to find out the RAM usage, the peak RAM usage is shown in Figure 6.3.

We can see from Figure 6.3, it is evident that the peak RAM usage significantly drops for the first 10 MCUs, but slowly decreases after 10 MCUs. After

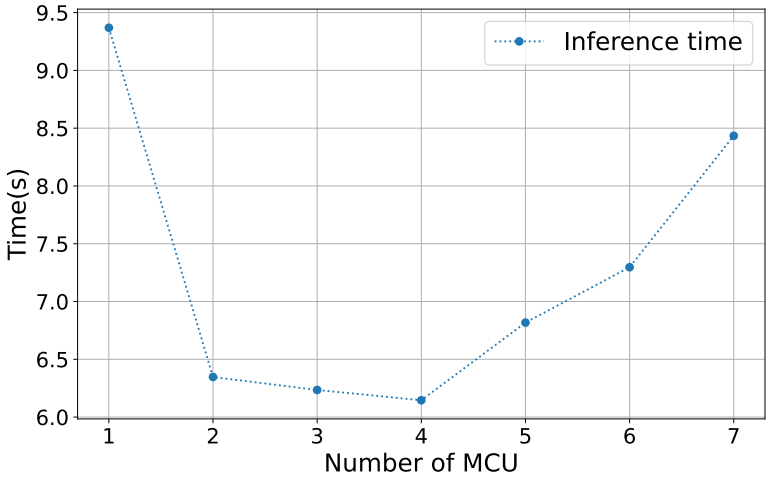


Figure 6.4: **Inference time**

examining the layers, we identified that the bottleneck layer during this phase was the spatial convolution layer at the beginning of the model. As mentioned in Section 4.3, in spatial convolution, a filter is applied to the entire depth of the input volume, generating feature maps that capture spatial patterns across all channels. Essentially, each MCU requires the entire input volume to produce a single feature map. However, due to the nature of Algorithm 1,when evenly distributed, only when the number of MCU exceeds the number of output feature maps will a single feature map be split into different MCUs, reducing the input size and RAM usage. One way to deal with this issue is to modify Algorithm 1 to distribute within each group of feature maps instead of the entire output positions as described in Section 4.3. See Figure 4.9 for a more detailed explanation. However, given that post-quantization, a RAM size of approximately 512KB is adequate for a network comprising about 3 to 10 MCUs, and considering that most modern MCUs can accommodate 512KB, we opted to retain the unmodified Algorithm 1 during real implementation in Chapter 7. The implementation of the modified Algorithm 1 is left for future work.

Inference time: The inference time denotes the duration required for the networked MCUs to classify a single input image. This duration comprises both the communication time among the MCUs and the time spent on calculations. As mentioned in Section 6.2, the simulation is conducted on my laptop equipped with an 8-core CPU. To ensure that the threads operate in parallel rather than concurrently, we only gather the data from 1 worker MCU to 7 worker MCUs. The result is shown in Figure 6.4.

It is clear from Figure 6.4 that for the first 4 MCUs, the inference time decreases with the number of MCUs, but after 4 MCUs the time increases instead. As mentioned at the beginning of Section 6.2, with more MCUs, more communication among MCUs is needed, the one and only coordinator gets too busy handling all the communication, as a result, workers spent more time waiting, causing delay in the workflow. However, even without such delays, as the number of MCUs increases, it becomes inevitable that the time spent on communication becomes dominant rather than the time spent on calculation.

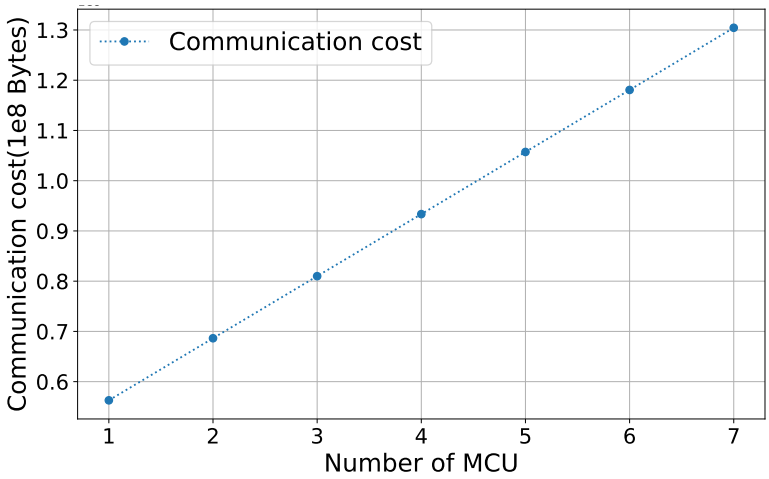


Figure 6.5: **Total communication cost**

To further analyze the impact of increasing number of MCUs on the communication cost, we also investigated how much data in bytes is transferred between the workers and the coordinator in total by counting how much data is received and sent by the coordinator. The figure is shown in Figure 6.5.

It is clear from Figure 6.5 that the size of the bytes to be transferred is linearly related to the number of MCUs. Which can be expressed in math equation as

$$C_t \simeq an + b, \tag{6.1}$$

where C_t is the total communication cost and n is the number of worker MCUs in the network,a and b is the relative slope and intercept of the function, from Figure 6.5 we can see that $a \approx 12060$ KB and $b \approx 42900$ KB.

From Algorithm 1 and Algorithm 2 we can see that when evenly distributed the workload on each MCU should be inversely proportional to the number of MCUs, in other words, the workload and the number of MCUs should have the following relation

$$W_i \simeq \frac{W_t}{n}, \tag{6.2}$$

where W_t is the total workload for MobilenetV2 model in KBs. To further prove this point, we collect the data of the workload size in KBs for the first worker thread. Figure 6.6 shows the workload on this MCU versus the number of MCU, it is clear that the workload for the MCU respects the above-mentioned relation. Since we distribute the workload evenly, we expect the same for other MCUs.

We can gather from Figures 6.4, 6.5 and 6.6 that under the simulation setting described in Section 6.2, the time spent on calculation is dominant for the first four MCUs while the time spent on communication becomes dominant after four MCUs causing the U-shaped curve observed in Figure 6.4.

Communication Coefficient K_c: As mentioned in Section 5.3, the communication coefficient can be calculated by running simulation to find how much data is transferred between worker MCUs and the coordinators and how many workloads are distributed to the worker MCUs. Just so happens that we have already covered the workload and communication cost in Figures 6.6 and 6.5.

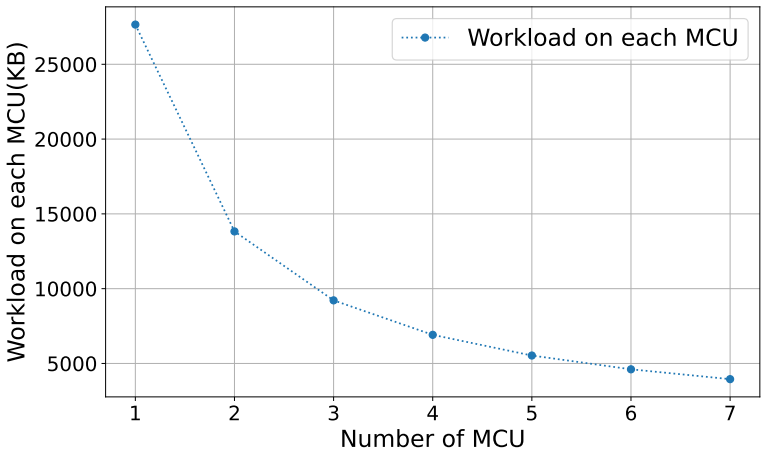


Figure 6.6: **Workload in worker MCU0**

We will use this as a chance to demonstrate how to calculate the communication coefficient of MCU_i.

Considering that we are now evenly distributing the model, each MCU should have basically the same communication cost and workload. As a result, the communication coefficient K_c for each MCU can be calculated using Equation 6.3.

$$K_{c_i} \simeq \frac{C_T}{NW_i}, \tag{6.3}$$

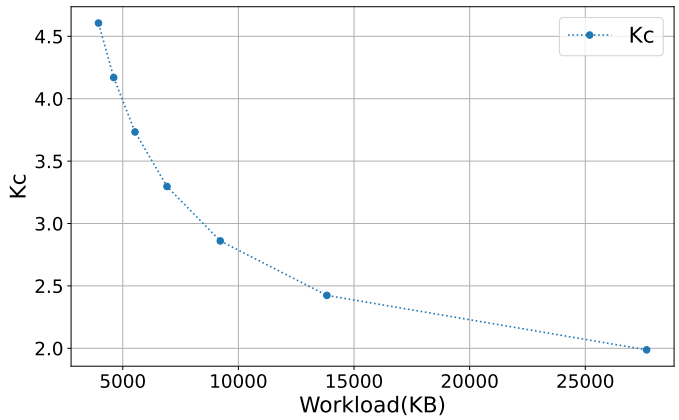
where K_{c_i} is the communication coefficient of MCU_i, C_T is the total communication cost, N is the number of worker MCUs in the network and W_i represents the workloads distributed to MCU_i.

Using Equation 6.3, we calculate K_c of the MCU and draw its relationship versus the number of MCUs and the size of workloads as shown in Figure 6.7. We can see that each MCU's K_c grows linearly with the number of MCUs same as the communication cost and decreases with workload. This means that the experiment results agree with our intuition that with more MCUs, the time spent on communication on each MCU becomes dominant rather than the time spent on doing calculations. The main reason for this is the network structure of MobilenetV2, as mentioned before, the structure of MobilenetV2 consists of spatial convolution layers, which require all input channels to produce a single feature map, as a result, during the inference process, despite the reduction in workload per MCU, the MCU still requires the same amount of input data from the coordinator to calculate this workload, as would with a higher workload, thus increasing K_c.

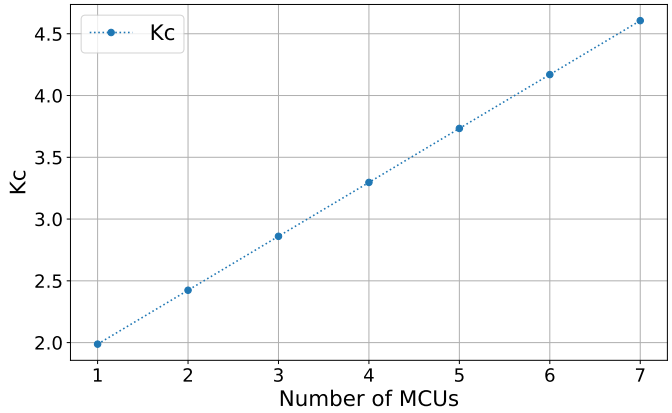
Figure 6.7(a) gives us a way to find the corresponding K_c, **when evenly distributed**, combining Equations 6.1 and 6.3 we have

$$Kc \simeq \frac{aN + b}{W_t}. \tag{6.4}$$

Equation 6.4 gives us a way to approximately calculate the communication coefficient K_c for MobilenetV2, to calculate K_c in the case of not even distribution, we need to find $\hat{N}$ to make the distribution as if it were evenly distributed, more details will be given in Chapter 7.



((a)) K_c **vs. workload**



((b)) K_c **vs. number of worker MCUs**

Figure 6.7: **Communication Coefficient** K_c

It is worth mentioning that Equation 6.4 is only intended for MobilenetV2, different models should have different C_t that may not follow the linear relation in Equation 6.1, in such cases a different approximation of Figure 6.5 should be adopted, thus influencing the equation to calculate K_c.

The preliminary result of the simulation gives us insights into some properties during implementation that we should consider and how the workload should be distributed.

Chapter 7

Implementation

This chapter describes the real implementation using the MobilenetV2 model on off-the-shelf MCUs and presents some important data collected during implementation.

7.1 Hardware

7.1.1 MCU selection

When choosing which MCU to use for our project, different aspects are considered, among which the most important factors are RAM and flash memory size. For implementation, due to limited budget, we only consider deploying the model on 3 MCUs, which means that for the model to work on these MCUs, according to Figure 6.3, also considering that the various libraries in the code may take up RAM size, taking quantization into account, the RAM size needs to be at least 512KB, and moreover, to make sure that we still have enough room for distributing larger chunks flexibly instead of only evenly, we decided that the RAM size needs to be at least 1MB. For flash memory size, the original MobilenetV2 is around 13MB and it is reduced to around 4MB after quantized to 8-bit representation, to make sure we have enough storage to hold the weight fragments, the flash memory size of each MCU needs to be at least 2MB.

STM32H743ZI is considered, it features an ARM cortex-M7 core running up to 480MHz with 1MB of RAM size and 2MB of flash memory size, it also features a double precision FPU, however, it does not support Ethernet communication, although it is possible to use other communication protocols such as Bluetooth or serial communication to achieve the same functionality, it would bring about unnecessary difficulties for implementation and potential inefficiencies into the system.

The chosen MCU for implementation is Teensy 4.1, it is an Arduino compatible MCU designed by PJRC (Paul J. Stoffregen Research & Consulting) based on the ARM Cortex-M7 architecture, which features a powerful 32-bit ARM Cortex-M7 processor running at 600 MHz, it comes with 8 MB of onboard flash memory, 1 MB of RAM memory which is divided into 2 sets of 512KB, half of this memory (RAM1) is accessed as tightly coupled memory for maximum performance. The other half (RAM2) is optimized for DMA access [16].

Figure 7.1: **Teensy4.1[16]**

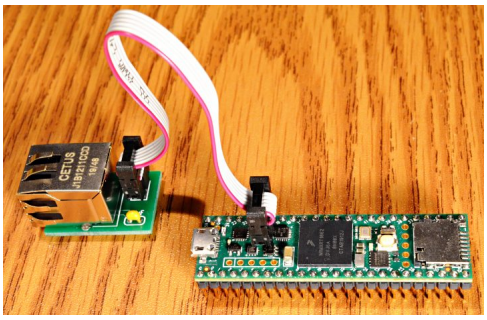


Figure 7.2: **Connecting teensy4.1 to Ethernet[16]**

It also supports Ethernet communication with a bandwidth of up to 100 Mbits/s, which is used in our project to construct the network for inference. To enable Ethernet functionality, we need to connect the Ethernet MagJack to the Ethernet port of Teensy4.1 as shown in Figure 7.2. Different MCUs are connected via Ethernet using the network switch.

7.1.2 Network switch

The network switch that we chose is Tplink TL-SG116. It is a 16-port gigabit Ethernet switch ideal for small to medium-sized networks. With plug-and-play setup, it offers fast data transfer speeds for seamless connectivity among devices. Each port is automatically adjusted for optimal performance, while energy saving features reduce power consumption without compromising efficiency.

7.1.3 The setup of the testbed

The connection of the system is shown in Figure 7.3. Three Teensy4.1 and one PC are connected together using a network switch, the PC is responsible for providing the power needed for the MCUs, as well as playing the role of coordinator and recording the relevant data during inference.

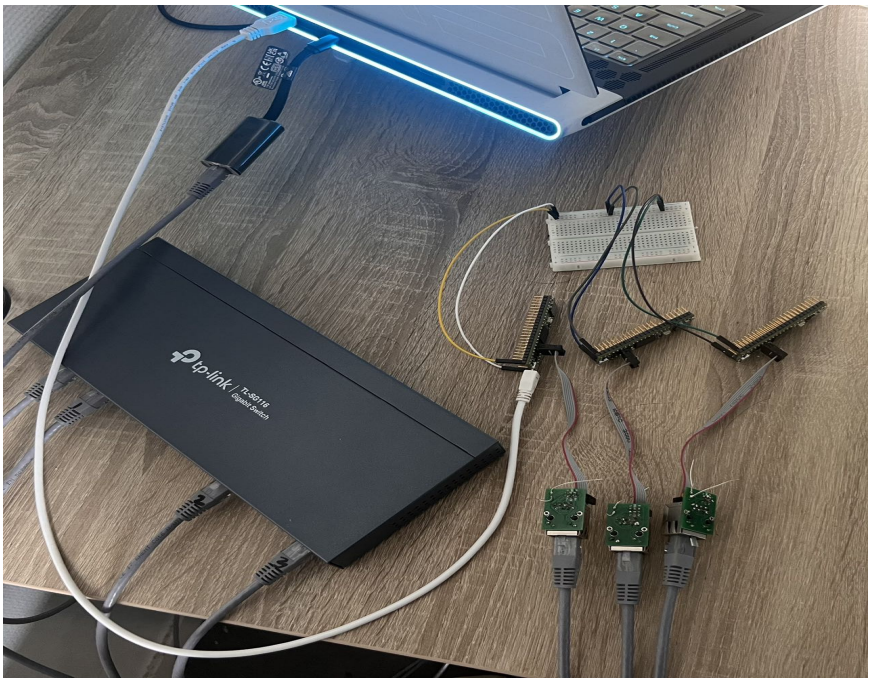


Figure 7.3: **The setup of the tested evaluation**

7.2 Implementation details

7.2.1 Software implementation

The code is written using Arduino Studio for simplicity. For deployment on the MCU, the specific parameters like frequency, network bandwidth, and network delay of different MCUs are given to the PC to calculate the relative ratings for each MCU,the model is then split according to its ratings on the PC into several JSON files, each JSON file is then read by the PC and written into each corresponding MCU's flash memory using LittleFS [4].

LittleFS, short for Little File System, is a file system created specifically for embedded devices. It is designed to be lightweight, efficient, and robust, making it well suited for resource-constrained environments typical of embedded systems. LittleFS offers features tailored to the needs of embedded applications, such as wear leveling for flash memory, power loss resilience, and efficient storage utilization. By prioritizing simplicity and reliability, LittleFS provides a practical solution for managing file storage in embedded devices, ensuring data integrity and system stability even under challenging operating conditions.

After downloading the weight fragment into each corresponding MCU, the next step is to perform the inference process. It is worth mentioning that due to the large memory size of Teensy4.1, it is possible for each MCU to act as both coordinator and worker at the same time, each MCU can send the calculation results directly to the next worker MCU as shown in Figure 7.5, however, doing so will require special synchronization techniques to find the relevant order of the data sequence from different MCUs, also, to avoid data loss during the transfer

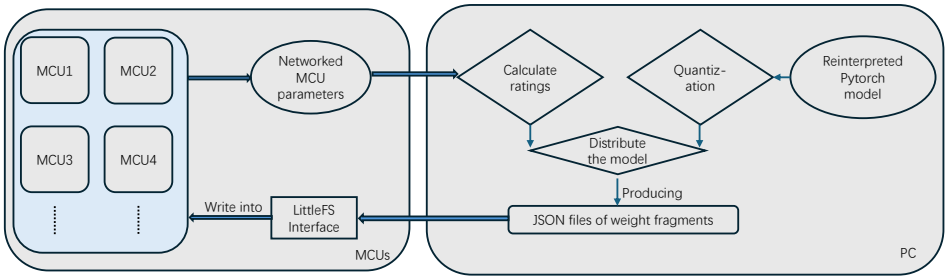


Figure 7.4: **Weight fragment download workflow**

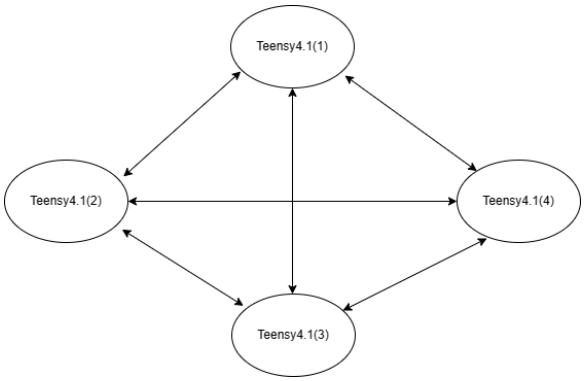


Figure 7.5: **A possible workflow with coordinator merged into worker**

process and to collect relevant data for analysis during the inference period, the calculation results are still first sent to the PC, then the PC will transfer the data to the corresponding MCU as shown in Figure 6.2. The implementation of the workflow in Figure 7.5 is left for future work.

For each workers, there are mainly 4 tasks to do, namely:

1. Read from flash memory the weights data.
2. Receives input neurons from the coordinator.
3. Do calculation using received input neurons and weights data.
4. Send the output neurons to the coordinator.

For these 4 tasks, we used the Round Robin scheduling policy to schedule them. A code flow chart is shown in Figure 7.6. We used the same communication protocol as described in the simulation, which is essential for producing the correct results but will introduce delay in the overall workflow as described in Section 6.2.

For communication between MCUs, we used the TCP protocol with PC being the server and MCU being the clients, the data is sent in packets instead of streaming because of the small buffering size for each MCU, in addition, to avoid message loss caused by bus traffic, each packet is received by MCU and PC with an acknowledgment signal. The packet format is shown in Figure 7.7.

Source ID is the worker or coordinator ID from whom the message is sent.

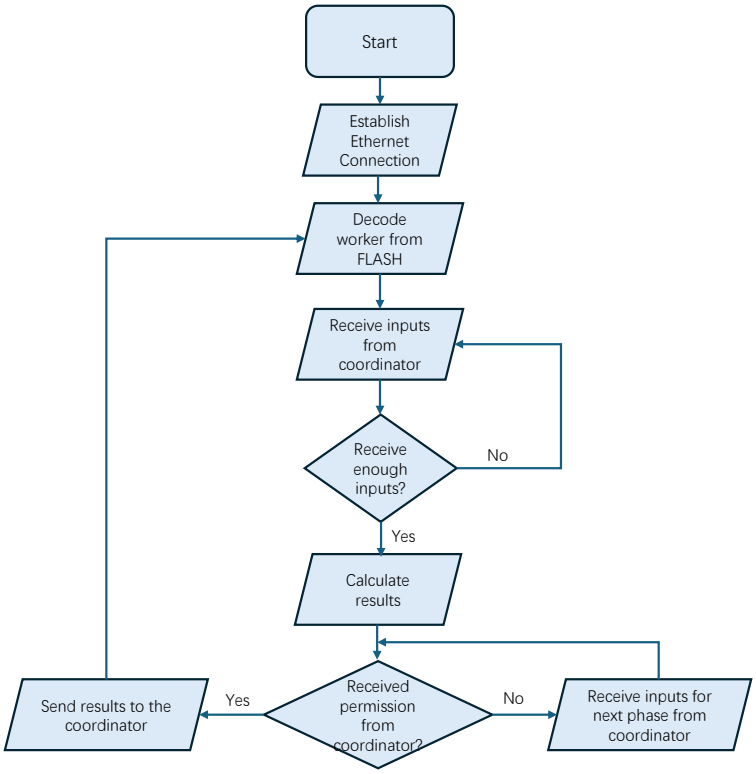


Figure 7.6: **Code flow chart on worker**

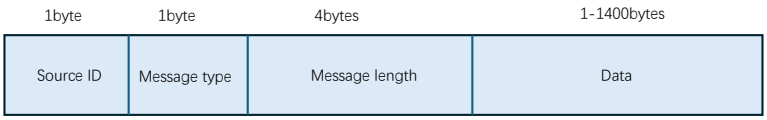


Figure 7.7: **Packet format**

Message type is the type of message that is sent, with the options shown in Table 7.1.

Message length is a 4 bytes value that indicates the message size of this packet, to avoid overloading the MCU, it's set to a maximum of 1400 bytes.

7.2.2 Determine K_1

To determine each MCU's rating, it is essential to first find relative K_1 for transferring workloads from cycles into KBs. In order to calculate K_1, we first need to remove the influence of communication, and we can achieve this by using dummy inputs instead of receiving real inputs from other MCUs. Using dummy inputs, we were able to collect the time t_c it cost to run one inference without communication. Then, we can use this time times the frequency $W_{cycles} = t_c \times f$ to calculate how many cycles it has spent on the calculation. Next, we can calculate K_1 using $K_1 = \frac{W_{KBs}}{W_{cycles}}$.

Table 7.1: **Message types**

Number	Message type
0-128	Data: Calculation result/ input. Number represents the ID this message is sent to
200	Permission granted from coordinator
199	MCU asks for permission from coordinator
198	MCU sent the last of its calculation results
197	acknowledgement signal
196	Signal the coordinator to perform adaptive pooling

Table 7.2: K_1 **under different settings**

Frequency (MHz)	Workload (KB)	Time (s)	K_1 (KB/MCycles)
600	510.29	6.41±0.3	0.133±0.007
450	510.29	7.52±0.3	0.150±0.006
150	510.29	16.11±0.3	0.211±0.004
600	421.50	5.51±0.3	0.127±0.007
450	421.50	6.21±0.3	0.151±0.007
150	421.50	13.80±0.3	0.204±0.005
600	730.39	7.40±0.1	0.165±0.002
450	730.39	9.06±0.2	0.179±0.004
150	730.39	21.34±0.5	0.228±0.006

To acquire K_1 using the aforementioned method, we run the inference on different frequencies and workloads, and the result we got is shown in Table 7.2. From Table 7.2, we can see that K_1 changes with different workloads and frequencies, but in an ideal scenario, K_1 should be constant, after some research, we discovered that reading from flash memory is consistently slow regardless of the core frequency, which leads to an overhead in the overall inference period.

The time spent on the calculation can be further split into 2 parts, the first part is the time spent on flash memory reading and writing, the frequency and workloads of the MCU do not greatly affect this part of time, the second part is the actual calculation time, this time should be proportional to the workload of each MCU and inversely proportional to the frequency of each MCU.

As a result, the time spent on the calculation can be further calculated using Equation 7.1

$$t_c = \frac{\hat{W}_{cycles}}{f} + C, \tag{7.1}$$

where C is a constant that represents the overhead time spent reading and writing flash memory. $\hat{W}_{cycles}$ is the actual cycles spent performing the calculations. Fitting Equation 7.1 into Table 7.2, we get $C \approx 3s$. As a result, the K_1 can be calculated using equation

$$K_1 = \frac{W_{KBs}}{\hat{W}_{cycles} + Cf}, \tag{7.2}$$

or

$$\frac{1}{K_1} = \frac{\hat{W}_{cycles} + Cf}{W_{KBs}}. \tag{7.3}$$

In Equation 7.3, $\frac{\hat{W}_{cycles}}{W_{KBs}}$ should be a constant value for any given workload when the algorithm used for the calculation and the hardware platform are fixed, from the data in the Tabel 7.2, and make $C \approx 3s$, we have $\frac{\hat{W}_{cycles}}{W_{KBs}} \approx 3.7$, also bring $C \approx 3$ to Equation 7.3. We have

$$K_1 = \frac{W_{KBs}}{3.7W_{KBs} + 3f}. \tag{7.4}$$

From Equation 7.4, we can see that to find K_1, we only need to find the relative workloads that this MCU is assigned; more details are given in Section 7.2.3.

7.2.3 Determine rating

From Chapter 5, we have the equation to calculate each MCU's rating, let us rewrite it in this section as Equation 7.5

$$R_i = \frac{f_i K_{1_i}}{(d_i + \frac{1}{B_i}) f_i K_{1_i} K_{c_i} + 1}. \tag{7.5}$$

From the equation, we see clearly that to find each worker's rating, we need to first calculate K_1 and K_c, however, according to Equation 7.4 and Equation 6.4, to calculate K_1, we need the workload of each MCU which in turn requires the rating of each MCU, to calculate K_c, we need the workload to be evenly distributed. This seems to be a dilemma here; however, the total workload W_t is fixed and the workload is distributed linearly, and we know the conditions that each rating should meet after distribution, by intuition, we can calculate each rating recursively using the **Successive approximation**.

Considering a network of n MCUs, each MCU's rating is denoted by R_i, frequency is denoted by f_i, communication delay is denoted by d_i, bandwidth is denoted by B_i. For K_1, for each worker's workload W_i(in KBs), we have

$$W_i = \frac{W_t R_i}{\sum_{j=0}^{n} R_j}. \tag{7.6}$$

Combining Equation 7.6 and Equation 7.4 we have

$$K_{1_i} = \frac{W_t R_i}{3.7 W_t R_i + 3 f_i \sum_{j=0}^{n} R_j}. \tag{7.7}$$

For K_c, we have previously discussed its calculation when evenly distributed in Chapter 6, and we draw its relations with workload in Figure 6.7(a). In Equation 6.4, we gave the relations between the number of MCUs and K_c when evenly distributed; now let us consider the distribution based on ratings.

In order for Equation 6.4 to hold, an evenly distributed distribution is needed; however, after distribution, each MCU's workload is completely independent of others. In other words, each MCU's can be seen as evenly distributed into $\hat{N}$ MCUs, $\hat{N}$ is given by

$$\hat{N}_i = \frac{\sum_{k=0}^{N} R_k}{R_i}. \tag{7.8}$$

In Chapter 6, we approximate the total communication cost C_t using Equation 6.1, to push things further, we can assume Equation 6.1 to be continuous, combine Equation 6.4 with Equation 7.8, we have

$$K_{c_i} = \frac{(a\hat{N}_i + b)}{W_t}. \tag{7.9}$$

Combine Equation 7.5, 7.7, 7.9 and 7.8, and let $a = \frac{12060}{4}, b = \frac{42900}{4}$ because we are using 8-bit data after quantization instead of the 32-bit original data, we get the final equation used in the successive approximation to get the rating of each MCU as

$$R_{i_{j+1}} = \frac{f_i W_t R_{i_j}}{(d_i + \frac{1}{B_i})(10725 R_{i_j} + 3015 \sum_{k=0}^{N} R_{k_j}) f_i + (3.7 W_t R_{i_j} + 3 f_i \sum_{k=0}^{N} R_{k_j})}. \tag{7.10}$$

We can use Equation 7.10 to calculate each MCU's rating recursively, to do so, we only need to specify a random non-zero initial rating and use this equation until the rating for each MCU stop changing. It is worth noting that Equation 7.10 does not guarantee to provide an optimal, but it guarantees to provide a relatively good distribution method because of all the approximation and the fact that Equation 7.8 and Equation 7.9 may not be continuous; however, a continuous approximation is a relatively good representation of the real scenario, thus producing suboptimal solutions.

Algorithm 4 Weight calculation for worker MCUs

Require: f, d, B: frequencies, communication delays, bandwidths of the MCUs
1: $ratings \leftarrow random\ values$
2: $thresholad \leftarrow 1 \times 10^{-3}$
3: **while dif**$(ratings_{j+1}, ratings_j) < threshold$ **do**
4: **for** i **in** $0..Number\ of\ MCUs$ **do**
5: $ratings_{i_j+1} \leftarrow$ **cacl_rating**$(ratings_{i_j}, d_i, B_i, f_i)$
6: **end for**
7: **end while**

Algorithm 4 showcases the way we used to calculate each MCU rating in more detail, in which **calc_rating()** is the function that performs the recursive calculation using Equation 7.10.

7.3 Testbed evaluation results

To determine how effective the rating scheme is, we tested the rating scheme by comparing it with other ways of distribution, like evenly distributed distribution and frequency only distribution (the workload is distributed solely based on the frequency of each MCU), and recorded the time it cost to run one inference on 3 MCUs, the result is shown in Table 7.3.

The test is done in a local network; each MCU is connected to the same network switch, thus sharing the same bandwidth. To ensure flexibility of the work distribution, the test is performed using an input image of size $112 \times 112 \times 3$, the relative parameters are adjusted accordingly. During the test, the communication delay is simulated using the *delay*() function in Arduino Studio, before each sending and receiving. We can see from Table 7.3 that when there is no delay in communication, the optimized rating scheme behaves just like a frequency-only distribution, which is expected, as mentioned before, Teensy4.1 supports a bandwidth of 10MB/s, which is super fast for this type of communication, when there is no communication delay, the time is mostly spent on doing calculation, it is justifiable to behave similar to a frequency-only distribution. When there is communication delay, we can see that the optimized rating scheme outperforms other distribution methods, proving the superiority of the optimization method.

However, it is worth mentioning that the overall workflow is still slowed down by the busy waiting delay as mentioned before in Chapter 6, however, this busy waiting is completely avoidable by using another communication scheme or using more coordinators, the implementation of this is left for future work.

Table 7.3: **Different distribution strategy using 3 worker MCUs**

Frequency(MHz)	delay(ms/KB)	evenly distributed time(s)	Frequency only time(s)	optimized time(s)
600	0			
600	0	9.80	9.80	9.80
600	0			
600	0			
150	0	20.10	12.40	12.52
450	0			
150	0			
396	0	22.30	13.43	13.37
528	0			
450	0			
396	0	11.44	10.75	10.61
528	0			
600	10			
150	0	32.81	33.01	31.50
450	5			
150	1			
396	4	29.10	20.90	21.33
528	1			
450	20			
396	7	54.73	54.2	47.41
528	13			
600	20			
396	5	53.08	54.83	44.45
150	10			
600	10			
600	20	49.18	49.18	41.95
600	5			

Chapter 8

Conclusion

In this thesis, we proposed a way to perform inference on multiple networked MCUs using a distributed approach to address resource limitation on individual MCUs.

First we analyzed the specification of some of the off-the-shelf MCUs, we found that the RAM size of the MCU poses the most significant issue for implementing deep learning models on them.

Then, we presented some of the preliminary steps taken to enable and facilitate the implementation of the thesis project on MCUs.

Next, we proposed our method for conducting distributed inference on networked MCUs. Initially, we outlined the overall workflow of our approach, introducing two roles for the MCUs and explaining the functionality of each role. We then detailed our technique for splitting models into weight fragments, to be stored in workers for both convolutional and linear layers, and presented the algorithm used for mapping the neurons, which is stored in the coordinator. Following this, we compared our splitting method with several existing methods in the literature, highlighting how our approach is superior in reducing peak RAM usage during inference. Finally, we suggest a potential improvement for our splitting method, although its implementation is reserved for future work.

The optimization part of the thesis focuses on compressing models and speeding up the entire inference process. We described two methods used in the thesis to reduce model size while maintaining accuracy as much as possible: layer fusion and quantization. Following this, we propose our method to optimize the workflow by assigning each MCU in the network a rating to represent its ability to handle tasks. We presented a detailed analysis using mathematical equations to explain how we assign these ratings to MCUs. In addition, we describe how to adjust these ratings to accommodate the limited storage size of the MCUs.

In the simulation part, we first describe the settings and implementation details of the simulation. We then presented some of the results obtained from these simulations. Following this, we explain how to use these results to calculate the communication coefficient introduced in the optimization part. The simulation shows that our method can achieve the desired goal of distributed inference on MCUs, and also gives us some insight into the real implementation on real MCUs.

In the implementation part, we first described the specification of the chosen hardware, then we delved into the specific of the implementation, including the

code used for downloading weight fragments and performing inference. In the end, a result table is presented that compares different workload distribution strategies.

In conclusion, this thesis presents a novel distributed inference method for networked MCUs, which addresses resource limitations by splitting models into weight fragments and optimizing workflows. Simulations validate our approach, demonstrating reduced peak RAM usage and efficient task handling. The implementation details confirm the feasibility of the method and its practical applicability for real-world MCU applications.

8.1 Future work

Throughout this thesis, we have already presented several directions for future work. In Chapter 4, we proposed a possible improvement of our splitting strategy to further reduce the peak RAM usage on large input feature maps; the implementation of it remains an interesting topic to be explored. Also, as mentioned in the thesis, during the simulation and implementation, the overall workflow is slowed down by the busy waiting time of the coordinator MCU, finding a better communication protocol to reduce the busy waiting time and speed up the inference process further is also a very interesting topic to be explored.

There are also some possible improvements for this project; for example, at the moment, it still takes considerable time to perform an inference on a $3 \times 224 \times 224$ image due to the unoptimized communication protocol mentioned above and the inefficiency of the algorithm used to perform the calculations. The inefficiency of the calculation results in a relatively small value of K_1 as shown in Chapter 7. Improving the efficiency of the calculation algorithm, possibly by redesigning or using on-chip accelerators, could significantly speed up the inference process.

This thesis focuses solely on the implementation of CNNs. Extending our methods to other types of model, such as transformers and diffusion models, presents a promising direction for future work.

Bibliography

[1] Mehmet Zahid Akpolat and Abdullah Bülbül. A global approach for goal-driven pruning of object recognition networks. In *2022 30th Signal Processing and Communications Applications Conference (SIU)*, pages 1–4, 2022.

[2] Chun-Chi Chang, Chung-Hsun Huang, and Yuan-Sun Chu. A hardware-friendly pruning approach by exploiting local statistical pruning and fine grain pruning techniques. In *2022 IEEE International Conference on Consumer Electronics-Asia (ICCE-Asia)*, pages 1–3, 2022.

[3] Po-Yuan Chen, Hung-Che Lin, and Jiun-In Guo. Multi-scale dynamic fixed-point quantization and training for deep neural networks. In *2023 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5, 2023.

[4] Christopher Haster. Little file system. `https://github.com/littlefs-project/littlefs`, 2024. Last accessed: May. 15, 2024.

[5] Adam Coates, Brody Huval, Tao Wang, David J. Wu, and Andrew Y. Ng. Deep learning with cots hpc systems. In *International Conference on Machine Learning*, Stanford,USA, June 2013. ACM.

[6] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 2704–2713, 2018.

[7] Jaeduk Lee, Hojung Lee, and Wan Choi. Wireless channel adaptive dnn split inference for resource-constrained edge devices. *IEEE Communications Letters*, 27(6):1520–1524, 2023.

[8] En Li, Liekang Zeng, Zhi Zhou, and Xu Chen. Edge ai: On-demand accelerating deep neural network inference via edge computing. *IEEE Transactions on Wireless Communications*, 19(1):447–457, 2020.

[9] Muyang Li, Tianle Cai, Jiaxin Cao, Qinsheng Zhang, Han Cai, Junjie Bai, Yangqing Jia, Kai Li, and Song Han. Distrifusion: Distributed parallel inference for high-resolution diffusion models. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2024.

[10] Ji Lin, Wei-Ming Chen, Han Cai, Chuang Gan, and Song Han. Mcunetv2: Memory-efficient patch-based inference for tiny deep learning, 2021.

[11] Ji Lin, Ligeng Zhu, Wei-Ming Chen, Wei-Chen Wang, and Song Han. Tiny machine learning: Progress and futures [feature]. *IEEE Circuits and Systems Magazine*, 23(3):8–34, 2023.

[12] Yang Liu and Jun Lu. Double loss block neural architecture search. In *2022 IEEE 10th Joint International Information Technology and Artificial Intelligence Conference (ITAIC)*, volume 10, pages 731–734, 2022.

[13] Yiding Liu, Yinglei Teng, and Tao Niu. Splittable pattern-specific weight pruning for deep neural networks. In *2023 IEEE International Conference on Multimedia and Expo (ICME)*, pages 1439–1444, 2023.

[14] Jiachen Mao, Xiang Chen, Kent W. Nixon, Christopher Krieger, and Yiran Chen. Modnn: Local distributed mobile computing system for deep neural network. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*, pages 1396–1401, 2017.

[15] Nenad Markus. Fusing batch normalization and convolution in runtime, 2018. Last accessed: Apr. 9, 2024.

[16] PJRC. handbook for teensy4.1 deveolopment board. `https://www.pjrc.com/store/teensy41.html`, 2024. Last accessed: May. 13, 2024.

[17] Jianyun Qiu, Yuntao Zhao, and Weigang Li. Neural architecture search method based on improved monte carlo tree search. In *2023 China Automation Congress (CAC)*, pages 9033–9037, 2023.

[18] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 4510–4520, 2018.

[19] Baitan Shao and Ying Chen. Reflective learning for online knowledge distillation. In *2022 IEEE 8th International Conference on Computer and Communications (ICCC)*, pages 2028–2032, 2022.

[20] Sujay Tadahal, Gopal Bhogar, Meena S M, Uday Kulkarni, Sunil V Gurlahosur, and Shashidhara B Vyakaranal. Post-training 4-bit quantization of deep neural networks. In *2022 3rd International Conference for Emerging Technology (INCET)*, pages 1–5, 2022.

[21] Fida Mohammad Thoker and Juergen Gall. Cross-modal knowledge distillation for action recognition. In *2019 IEEE International Conference on Image Processing (ICIP)*, pages 6–10, 2019.

[22] Zhigang Tu, Xiangjian Liu, and Xuan Xiao. A general dynamic knowledge distillation method for visual analytics. *IEEE Transactions on Image Processing*, 31:6517–6531, 2022.

[23] Yuejiao Wang, Zhong Ma, and Chaojie Yang. A new mixed precision quantization algorithm for neural networks based on reinforcement learning. In *2023 IEEE 6th International Conference on Pattern Recognition and Artificial Intelligence (PRAI)*, pages 1016–1020, 2023.

[24] Zihan Wang, Junwei Xie, Zhiping Yao, Xu Kuang, Qinquan Gao, and Tong Tong. Nffkd: A knowledge distillation method based on normalized feature fusion model. In *2022 IEEE 5th International Conference on Big Data and Artificial Intelligence (BDAI)*, pages 111–116, 2022.

[25] Zhongjin Zhao, Haijun Liu, Zhiwei Li, Chenxi Zhang, Tuo Ma, and Jie Peng. A pruning method with adaptive adjustment of channel pruning rate. In *2023 IEEE 4th International Conference on Pattern Recognition and Machine Learning (PRML)*, pages 618–623, 2023.

Appendix A

Computing algorithm

The calculation algorithm refers to the algorithm used to perform calculations using input and weight data for different layers, including convolutional layers and linear layers. In this thesis, the input of each worker is flattened into a one-dimensional vector for the calculation as shown in Figure A.1.

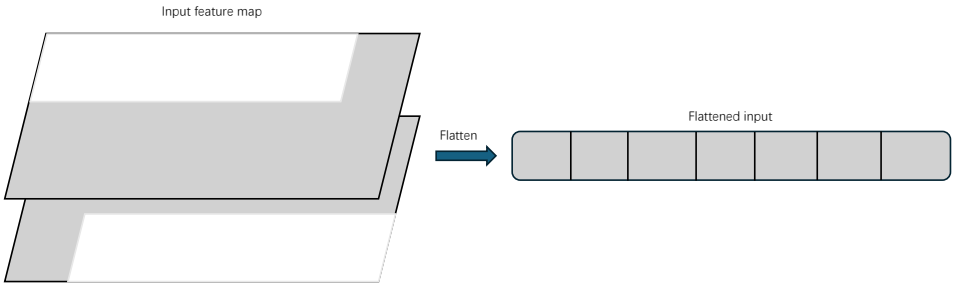


Figure A.1: **Flatten the input feature map**

Due to the flattened input, it is essential to determine which positions in the flattened vector to use for the calculation of a specific position in the output vector, as shown in Figure A.2.

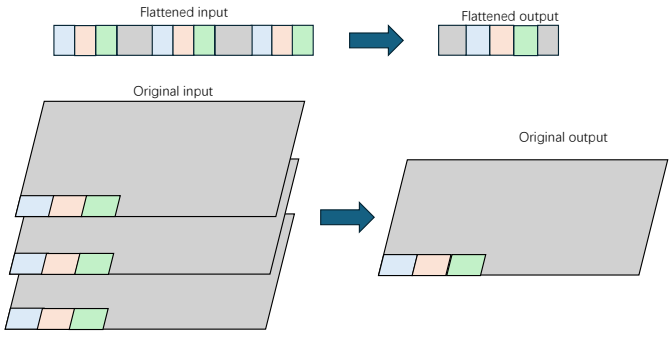


Figure A.2: **An illustration of the calculation after flattened**

In the implementation of this thesis, finding the positions in the input vector is done through a variety of adjustment factors, for the details of the implementation, please refer to the code repository.

Appendix B

Inference framework

The inference framework used in this project is written in Rust language, it is implemented using trait object, each object has a public *Layer* trait, this trait has the following methods.

1. $Identify()$ Identify the layer type of this object.
2. $get\_input(position)$ Trace back the receptive field to find the positions of the input neurons required to calculate the *postion* output.
3. $get\_output\_shape()$ Return the output dimension shape of this layer.
4. $get\_info()$ Return the information of this layer, for convolutional layers, it should return the kernel size, stride, input shape, output shape, input group, output group.
5. $get\_bias()$ Return the bias used to calculate the result.
6. $get\_all()$ Print all the relative information of this layer.
7. $print\_weight\_shape()$ Print the shape of the weight data for this layer.
8. $get\_weights\_from\_input(input\_positions, output\_channel)$ Sample weight data using the input positions and output channel.
9. $functional\_forward()$ Perform forward propagation if this is a functional layer,ie, ReLu,BatchNorm.

Note that the inference framework is only used for preliminary testing and validation of the distribution algorithm used in the thesis; it is not implemented on MCUs.

Appendix C

Code repository

The code repository for the thesis project can be found on my github page: *https* : *//github.com/Eagleggs/Split_learning_microcontrollers*