Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für Integrierte Systeme
Integrated Systems Laboratory

Department of Information Technology and Electrical Engineering

Machine Learning on Microcontrollers

227-0155-00L

Exercise 9

Keyword Spotting using a Parallel Ultra-Low Power MCU

Cristian Cioflan
Viviane Potocnik
Aishwarya Melatur

Tuesday 11th November, 2025

1 Introduction

Parallel Ultra-Low Power Platform (PULP) is a joint project between the Integrated Systems Laboratory (IIS) of ETH Zurich and the Energy-efficient Embedded Systems (EEES) group of the University of Bologna to develop an open, scalable Hardware and Software research platform to break the pJ/op barrier within a power envelope of a few mW.

The PULP platform is a multi-core platform achieving leading-edge energy-efficiency and featuring widely-tunable performance. PULP aims to satisfy the computational demands of IoT applications requiring flexible processing of data streams generated by multiple sensors, such as accelerometers, low-resolution cameras, microphone arrays, and vital signs monitors. As opposed to single-core MCUs, a parallel ultra-low-power programmable architecture allows meeting the computational requirements of these applications, without exceeding the power envelope of a few mW typical of miniaturized, battery-powered systems. Moreover, OpenMP, OpenCL, and OpenVX are supported on PULP, enabling agile application porting, development, performance tuning, and debugging.

PULP efficiently implements RISC-V cores with basic RISC-V instruction set (RV32IMC) or with custom instruction set extensions (XPULP).

2 GAP9

GAP9 is a PULP processor designed and manufactured by Greenwaves Technology[1]. The GAP9 comes with its own toolchain for deploying neural networks, called GAP *flow*[2]. Rather than using proprietary software, in this lab, we will use open-source tools developed to quantize (QuantLib[3]) and deploy (DORY[4]) neural networks on PULP platforms.

As depicted in Figure 1, the System-on-Chip (SoC) architecture of the GAP9 processor is ingeniously partitioned into two primary subsystems: the Fabric Controller (FC) and the Cluster. The Fabric Controller acts as the central control unit, featuring a single RI5CY core that implements the RV32IMCXPULP instruction set, optimized for efficiency in control tasks and peripheral management. This core is specifically designed to handle lightweight tasks, including system boot-up, peripheral management, and serving as an intermediary between external interfaces and the Cluster.

On the other hand, the Cluster constitutes the powerhouse of the GAP9 SoC, comprising eight RI5CY cores dedicated to parallel compute-intensive tasks. This multi-core Cluster is designed to accelerate the processing of data-intensive applications, such as machine learning inference, signal processing, and image processing tasks, through efficient parallel execution. Each core within the Cluster operates in concert with others, sharing a tightly integrated scratchpad memory that facilitates rapid data exchange and synchronization, thereby reducing the latency and energy consumption typically associated with such operations.

In addition to the computational capabilities provided by the RI5CY cores, the GAP9 SoC is further augmented with specialized hardware accelerators, which are pivotal in achieving high-performance and energy-efficient execution of specific tasks:

[1] https://greenwaves-technologies.com/gap9_processor
[2] https://greenwaves-technologies.com/gapflow
[3] https://github.com/pulp-platform/quantlib
[4] https://github.com/pulp-platform/dory

1. **Convolutional Neural Network (CNN) Accelerator** Integrated within the Cluster, this accelerator is tailor-made for the acceleration of convolutional neural network operations. It significantly enhances the SoC's ability to perform deep learning inference tasks by offloading the most computationally demanding operations from the RI5CY cores, thereby accelerating the throughput and reducing the power consumption of AI applications.
2. **Autonomous Input/Output (IO) Accelerators** To further optimize energy efficiency and reduce the workload on the central processing units, the GAP9 incorporates autonomous IO accelerators. These units manage data transfers between the memory and peripherals without CPU intervention, enabling continuous data flow for tasks like sensor data acquisition and digital signal processing, all while maintaining a low-power state.
3. **Advanced Memory Subsystem** Complementing the computational elements, the GAP9 features an advanced memory subsystem designed to support high bandwidth and low energy consumption. This subsystem includes an innovative hybrid memory architecture that intelligently manages data placement and access, optimizing the balance between performance and power efficiency.

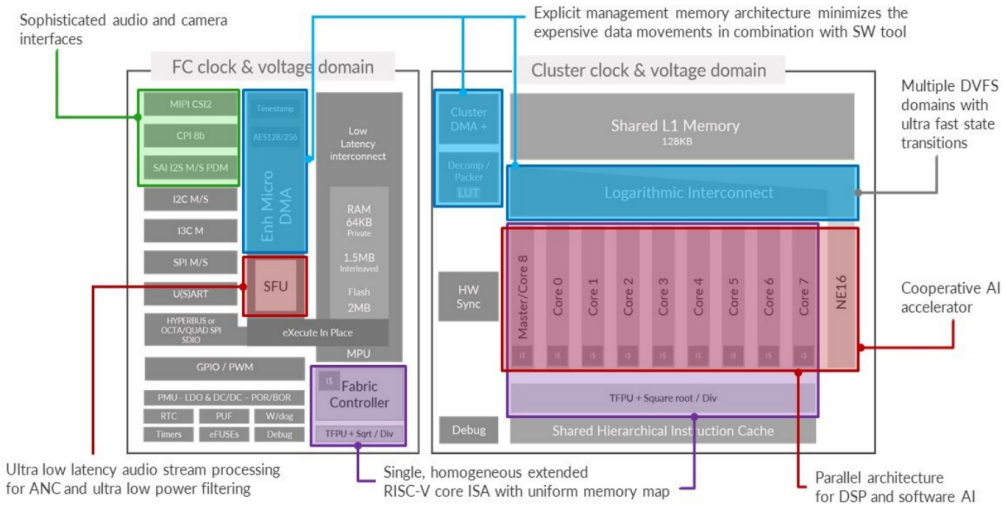

Figure 1: Block diagram of GAP9.

3 Notation

Student Task: Parts of the exercise that require you to complete a task will be explained in a shaded box like this.

Note: You find notes and remarks in boxes like this one.

4 Preparation

For this lab, you will need Python to quantize a deep neural network using PyTorch and QuantLib, DORY to generate the C code, and the GAP9 board on which we will run the inference.

To streamline your setup process and reduce the time needed for installation, we have provided you with a Virtual Machine that already contains the latest GAP_SDK installed. This VM was packaged using VirtualBox, which we recommend for optimal support, although other virtualization software may also work.

Virtual Machine Setup Instructions:

1. **Install VirtualBox**: If you haven't already installed a virtual machine program, start by installing VirtualBox.
2. **Open the Provided VM**: Locate and open the `.ova` file we provided to set up your Virtual Machine.
3. **Follow Setup Instructions**: Complete the setup by following the on-screen instructions. The password for the VM is *osboxes.org*. ccd

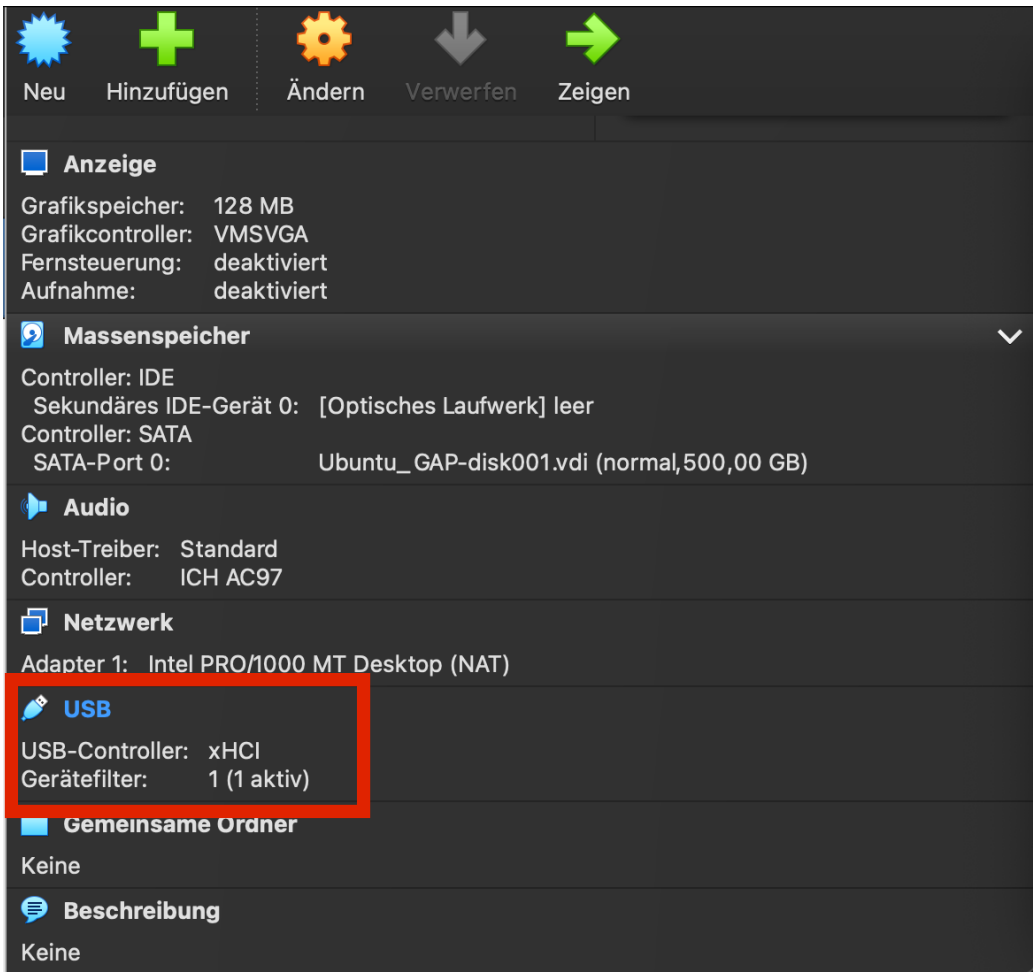

Figure 2: USB Setup in Virtual Box

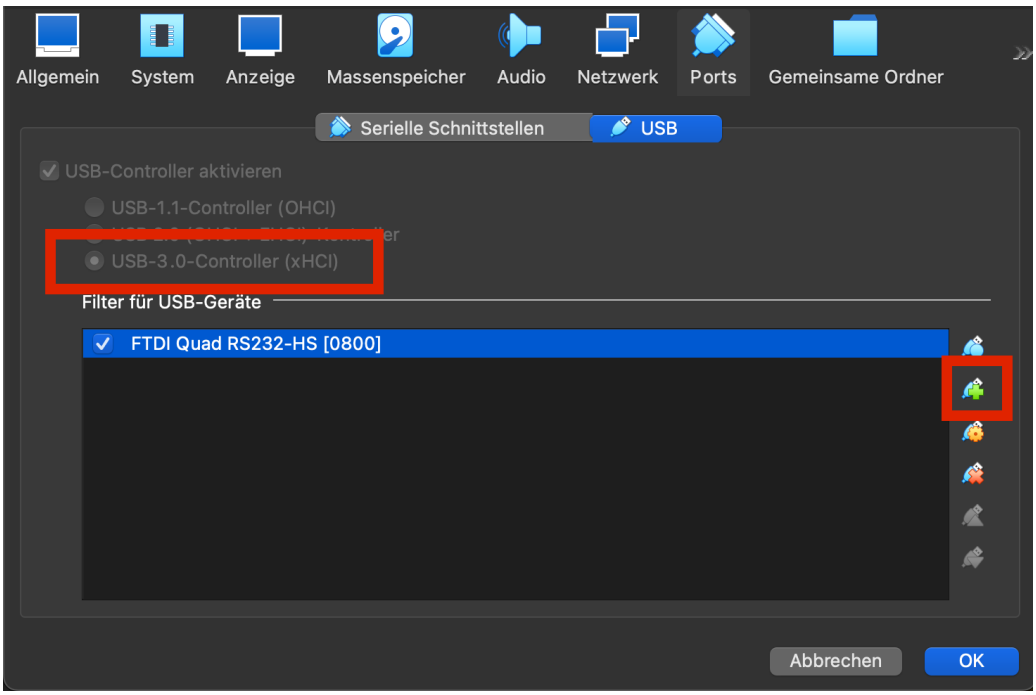

Figure 3: USB Setup in Virtual Box

Enabling USB Access in the Virtual Machine:

- **Install the VirtualBox Extension Pack**: This is essential to enhance the VM's capabilities, including adding USB2/USB3 support which is crucial for the lab activities. Download and install the Extension Pack from the VirtualBox website. Note: This pack is under a different, closed-source license.
- **Adjust VirtualBox Settings**:
  - **Start VirtualBox with Admin Rights**: Open a terminal and run `sudo virtualbox` to start VirtualBox with the necessary permissions to modify USB settings.
  - **Configure USB Settings**: In VirtualBox, open the USB configuration of the VM as shown in 2 and add a filter for your GAP9 Board, as shown in figure and change the controller to USB 3.0. This ensures the VM accesses the USB device before the host does.

During the lab, these settings will enable you to successfully flash the board by ensuring your VM has the appropriate USB access. Follow these steps carefully to ensure a smooth setup and efficient lab session.

Note: The VM is running on the hardware of your machine. It might be necessary to adapt the allocated resources to what your machine has to offer. If you have questions, consult your fellow TAs.

5 Workflow with GAP

We would first train the desired model in full precision (32-bit floats) using PyTorch; for today's exercise, we provide you with a pretrained network. Then, we use two frameworks to arrive at a deployed

network on GAP. The first, `QuantLib`, is for quantization - the second, `DORY`, is for deployment. These frameworks work independently from each other and use the ONNX (Open Neural Network eXchange) format to transfer the information on the network (i.e., topology, parameters) between each other.

5.1 QuantLib

`QuantLib` [5] is a library to train and deploy quantized neural networks (QNNs). It was developed on top of the PyTorch deep learning framework.

`QuantLib` is a component of `QuantLab` [6], which also includes organizing software to manage machine learning (ML) experiments (systems and manager packages, as well as the main.py façade script). Figure 4 shows the structure of the `QuantLib` library. It encompasses algorithms for quantizing full-precision networks and libraries for performing fake quantization on a full-precision network and for turning the fake quantized network into an integer quantized network.

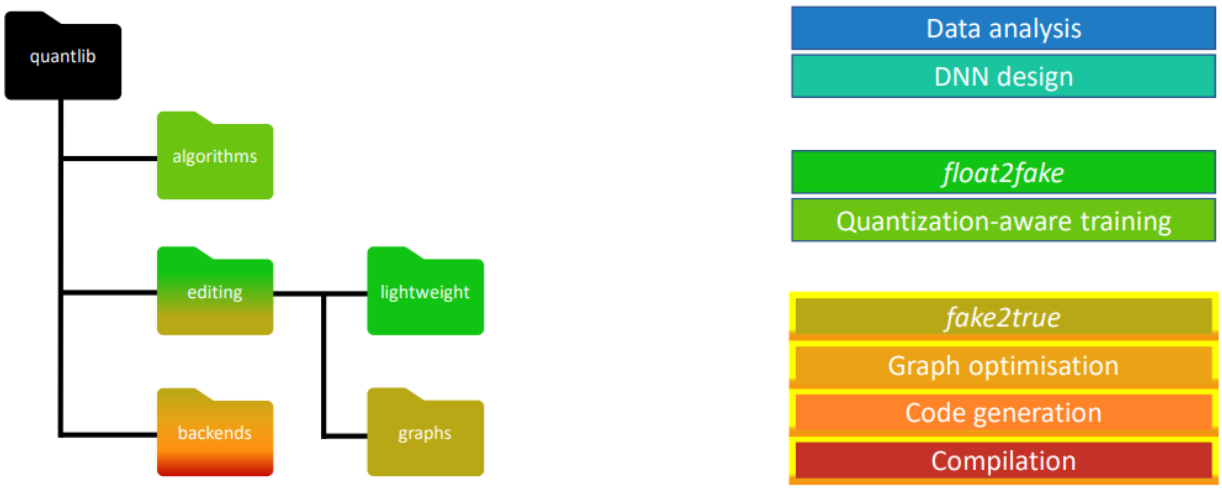

Figure 4: File tree of the `QuantLib` library

Why do we want to quantize neural networks?

DNN quantization is a powerful tool to shrink the model size (reducing the number of bits) while speeding up computation (thanks to integer-only operations).

5.2 DORY

DORY [7] is a tool for automatic deployment of DNNs on low-cost MCUs with typically less than 1MB on-chip SRAM memory. DORY abstracts tiling as a Constraint Programming (CP) problem: it maxi-

[5] https://github.com/pulp-platform/quantlib
[6] https://github.com/pulp-platform/quantlab
[7] `https://github.com/pulp-platform/dory`

mizes L1 memory utilization under the topological constraints imposed by each DNN layer. Then, it generates ANSI C code to orchestrate off- and on-chip transfers and computation phases

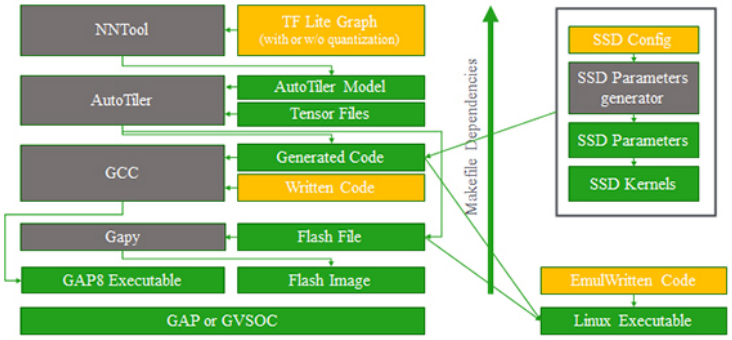

Figure 5: Workflow with GAP by GreenWaves Technologies.

6 Getting to know the GAPFlow

We will start with a simple `helloworld` example from GreenWaves' GAP-SDK. This will provide us with a practical starting point to understand the basics of the SDK and how to effectively utilize it for developing applications on GreenWaves platforms.

Student Task:

1. Once you open the VM and start a terminal, you need to type `GAP_SDK`. This will set up the virtual environment and paths to the GAP-SDK.
2. Navigate to the example `helloworld` project in the GAP-SDK (full path: `examples/gap9/basic/helloworld`).
3. Follow the steps of the `README`. Make sure your board is connected to your PC and run the `lsusb` command. You should see a device called `Future Technology Devices International`.
4. After running the `gap run` or alternatively, the `cmake --build build --target run` command, you should see the `Hello, World!` string from the compute units of the GAP9 board. If this is not the case, ask an assistant for help.

Student Task:

1. For the rest of the lab, we will use a conda environment. We will install Miniconda, a miniature installation of Anaconda Distribution. The instructions are available online [a]. Start by downloading the installer:

   ```
   wget https://repo.anaconda.com/miniconda/Miniconda3-latest-Linux-x86_64.sh
   ```

2. Install Miniconda by running the following command:

   ```
   bash ~/Miniconda3-latest-Linux-x86_64.sh
   ```

3. Close and re-open your terminal window for the installation to fully take effect, or run

```
source ~/.bashrc
```

4. Download to your Virtual Machine the exercise sources, the dependencies, and the data used for calibration.
5. Create the conda environment from the provided `environment.yml` file, which lists all the required packages for this project.

```
conda env create -f environment.yml
```

[a] https://www.anaconda.com/docs/getting-started/miniconda/install#linux-terminal-installer

7 Keyword Spotting

Keyword Spotting (KWS) represents the task of processing an utterance and recognizing a keyword from a predefined set. KWS is also known as closed-vocabulary automated speech recognition. Applications relying on KWS, such as voice-activated virtual assistants or sound source localization, target extreme-edge embedded systems, while the sensors (i.e., microphones) are also located on tinyML platforms. It is thus natural to aim to also perform keyword spotting at the extreme edge on platforms such as the GAP9 MCU.

Consider a Depthwise-Separable Convolutional Neural Network (DS-CNN) that you pre-trained on a KWS dataset such as Google Speech Commands. In this tutorial, we will understand the main steps required to use the pre-trained network to process a 1-second input acquired on-board and to classify the utterance.

The first part of this exercise guides you into the QuantLib quantization flow, using a small pre-trained network and going through post-training per-layer quantization down to QuantLib's deployable formats, which organize operations so that they are an accurate representation of the behavior on integer-based hardware. We will see how all this works through QuantLib's four progressive stages: Full-Precision (FP), Fake-Quantized (FQ), Quantized Deployable (QD), and IntegerDeployable (ID).

QuantLib uses `float32` tensors to represent data at all four stages - including IntegerDeployable. This means that QuantLib code does not need special hardware support for integers to run on GPUs. It also means that QuantLib is not (and does not want to be) a runtime for quantized neural networks on embedded systems!

8 Quantize and export with QuantLib

We will now explain the basic steps of neural network quantization for hardware deployment, as mentioned earlier. Afterward, we will go through the steps from loading a pre-trained network to deployment on the actual hardware.

8.1 FullPrecision (FP) stage

Here we operate at what we call the FullPrecision (FP) stage: the regular PyTorch representation, which relies on real-valued tensors represented by `float32` in your CPU/GPU.

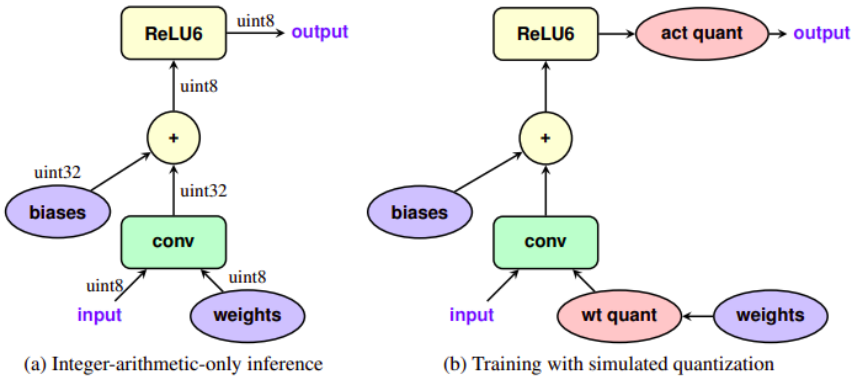

Figure 6: Fake Quantization Scheme.

8.2 Fake Quantization (FQ) stage

This representation is very similar to **FullPrecision**, as it still uses real-valued tensors for weights and activations. However, activation functions such as ReLU become quantization functions, imposing that the output is representable in a certain number of steps. Mathematically,

$$\mathbf{y} = \mathrm{ReLU}(\mathbf{x}) = \mathrm{clip\_}[0, \infty)(\mathbf{x}) \quad \longrightarrow \quad \mathbf{y} = \left\lfloor \frac{1}{\varepsilon} \cdot \mathrm{clip\_}[0, \alpha)(\mathbf{x}) \right\rfloor \cdot \varepsilon$$

Here, we introduce two changes to the ReLU. First, the clipping function is not only clipping at 0, but also at a maximum value α, which can be set to the maximum value of $\mathbf{y}$ in the *FullPrecision* stage (see later). Second, we introduce a **quantum** ε, which is the smallest real-valued amount that is representable in $\mathbf{y}$. With Q bits, $\varepsilon = \alpha/(2^Q - 1)$. Network weights are passed through a similar function before being used, with the main difference that they are not strictly non-negative: they are clipped between two values α and β.

At this stage, the network can still be trained / fine-tuned. Also, the numerical values of activations can differ a little from a real hardware implementation. This is because quantization of tensors is only actively imposed in the activation functions, but not in the other operations (i.e., in `ReLU`, not in `BatchNorm2d` and `Conv2d`).

Here, we first transform the model into a **FakeQuantized** version targeting a very relaxed 16-bit uniform quantization for weights and activations.

This FakeQuantization scheme follows Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference [8], as illustrated in Figure 6

Student Task: We have prepared scripts for you to perform the quantization and deployment of our network. Navigate to the `kws-on-gap9` directory for this task.

1. Start the VM, copy the exercise zip folder to the VM, and unzip the folder.
2. After unzipping the exercise materials and their subfolders, move the contents of the `dory` and `quantlib` folder in the respective subfolders inside the `kws-on-gap9` directory.
3. In the first step, we will run the `quantize.py` script. This script loads the pre-trained full-precision network, fake quantizes it, and updates the clipping bounds for integerization of

[8] https://arxiv.org/abs/1712.05877

the final quantized network. Run the following command: `python quantize.py --net DSCNN --fix_channels --word_align_channels --clip_inputs`.

4. What is the accuracy after fake quantizing the network? Do you observe a significant drop? And what about full quantization?

Note: When trying to run the quantize command in the VirtualBox VM on a Winwods 11 Machine, the script might throw the error "Illegal instruction (core dumped)".

This is connected to tensorflow being compiled to needing the AVX/AVX2 CPU instructions.

- Windows has a feature called *Hyper-V* which is mutually exclusive to running a VM that needs AVX/AVX2. [a]
- Windows 11, compared to Windows 10 and macOS has a new security setting which isolates the core memory. This also prevents pass-through of AVX/AVX2 instructions from the VM to the CPU.

Disabling Hyper-V and the security setting might help to run the code works without error.

[a] `https://learn.microsoft.com/en-us/troubleshoot/windows-client/application-management/virtualization-apps-not-work-with-hyper-v`

We obtain a quantized model saved in .onnx format, together with the per-layer activations. A configuration file, required for hardware deployment is additionally generated. You can find the files under `export/`.

9 Deployment and on-board inference

The second part of this tutorial guides you through the deployment process onto a strongly memory-restricted parallel processor platform, here GAP9. We employ DORY for this purpose, which takes the ONNX graph representation of the quantized neural network as an input, and afterward generates the C code, which can be built into an executable that will run on the hardware.

Student Task: Now, we will generate the C files that can be used to build the executable for the target hardware architecture. We will change the parameters of the code generation to see the impact of different units of the GAP9 board. In order to generate the C files, we will have to run the following command:

```
./dory_gen.sh gap_sdk 3 board 0 <target> generate export <num_cores>
```

1. We will first run the code generation with the GAP9 cluster and a single core. Adapt the command above for this configuration.

The parameters of the `dory_gen` represent, in order:

- the target SDK
- the highest memory available in the hierarchy (L2/L3)
- the target platform

- the MFCC computation (0 - using precomputed coefficients, 1 - online)
- the main computational unit. 0 - PULP GVSOC, 1 - GAP9 cluster (recommended), 2 - GAP9 NE16 accelerator (recommended)
- the destination directory for the generated code
- the source directory of the pre-trained network
- the number of cores to perform inference on

Student Task: After code generation, you need to flash the board. You can use the prepared `deploy.sh` script for this purpose. In the first step, we will use a `.wav` file as the input source.

1. How many cycles did the application on the board take when running it on a single core?
2. In the next step, increase the number of cores to 8. What speedup do you observe?
3. Finally, move to the `NE16` accelerator. What is the speedup compared to your previous run?
4. Now try the above hardware setup, but set the input to the `microphone`. Can the board correctly recognize what you said?

The application reads an input, computes the MFCCs, and then performs inference. The parameters of the `deploy` script represent, in order:

- the target platform (gvsoc/board). Make sure to select the desired platform in the ‘configmenu‘ as well. The other default options (e.g., Gapmod v1.0, EVK board 1.3) suffice.
- the input source (0 - microphone/1 - .wav stored in L3). For on-board data acquisition, make sure to place a jumper on J7 and a jumper on PIN 1-2 of CN9.
- the MFCC source (0 - online computation/1 - precomputed MFCCs stored in input.h).

$\mathcal{E}$ **Congratulations! You have reached the end of the exercise. If you are unsure of your results, discuss with an assistant.** $\mathcal{E}$