APAI2025 - LAB10  
End-to-end deployment

*Authors: Davide Nadalini, Luca Bompani, Francesco Conti*

*Contacts: d.nadalini@unibo.it, luca.bompani5@unibo.it*

***Links:*** [***GitHub Link (code)***](https://github.com/EEESlab/APAI25-LAB10-End-to-End-Deployment.git)

**Summary**

1. Subject(s):
   * Automated tools for deploying a neural network on a PULP microcontroller
2. Programming Language: python C
3. Lab duration: 3h
4. Assignment:
   * Time for delivery: 1 week
   * **Submission deadline:**  Jan 2, 2026 at 15:30

# **How to deliver the assignment**

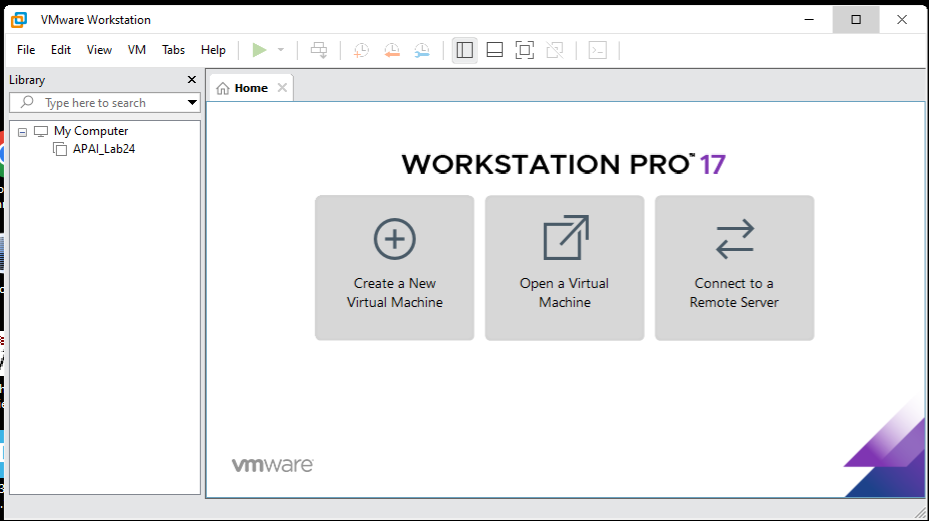
You will deliver ONLY THIS TEXT FILE, no code

* Download this file.
* Fill in the required results.
* Export to pdf format.
* Rename the file to: LAB<number\_of\_the\_lesson>\_APAI\_<your\_name>.pdf
* Use Virtuale platform to load ONLY your .pdf file

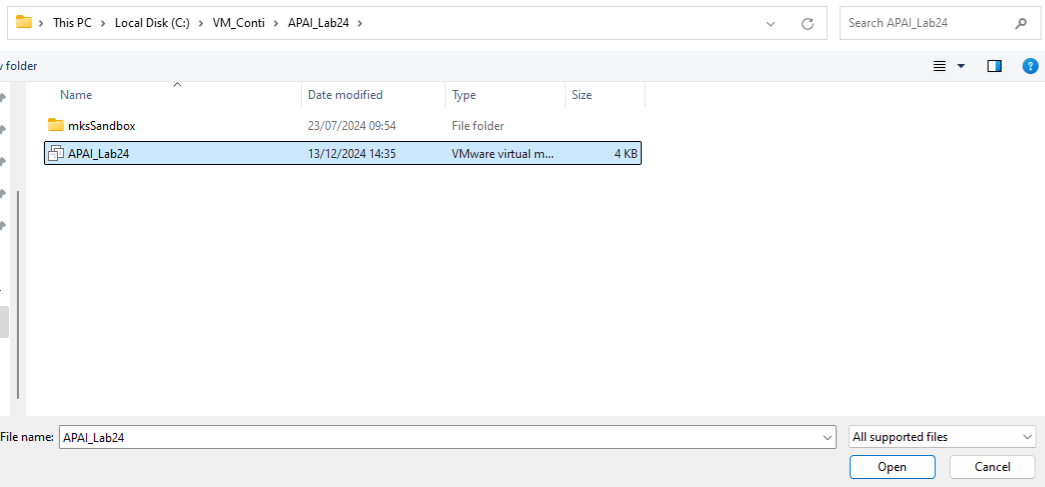
# **Part 1 (GAP9 and NNTool)**

# **Setup**

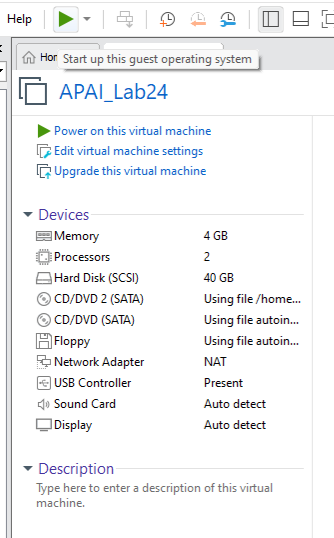
1. Open the LAB1 PCs (no VM available at home)
2. Open VMWare Workstation



1. Open the VM from C:/VM\_APAI/APAI\_Lab24/



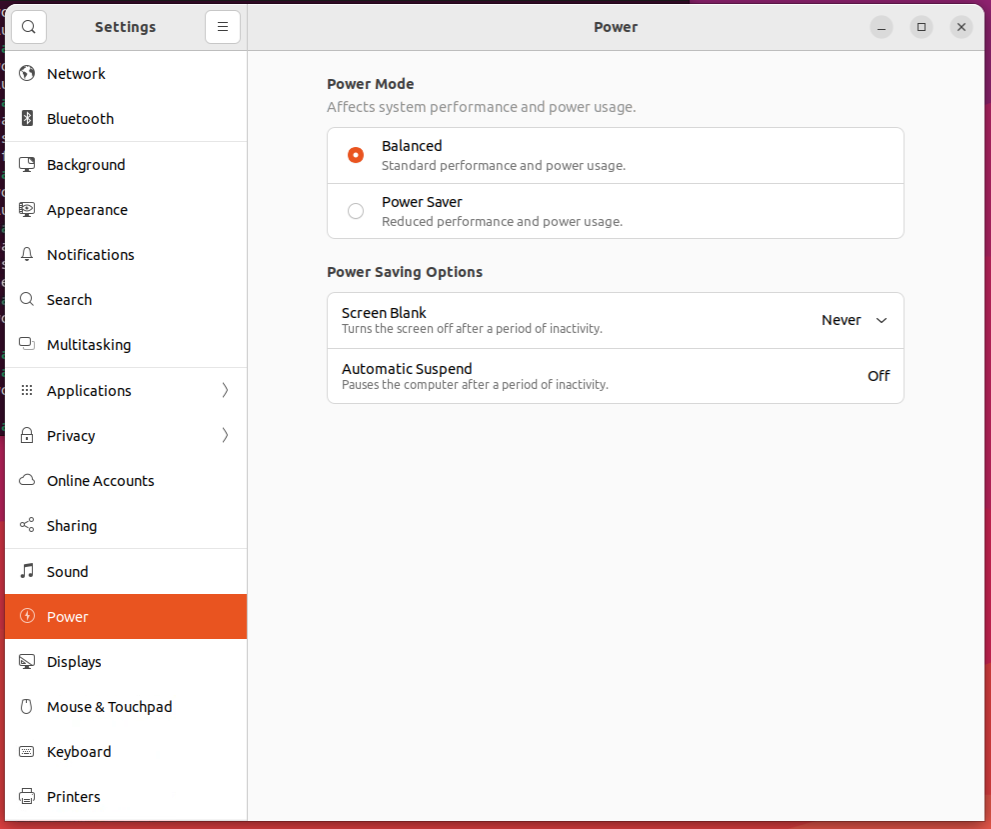
1. Start the VM by clicking on the green arrow



1. Connect the board via USB port and enable the USB in the VM with VMWare (VM menu)



1. Inside the VM, disable screen lock in Settings to avoid the screen to lock



1. Open a Terminal and setup the GAP-SDK

apai@apailab:-$ source gap\_sdk.sh

1. Select the correct device by entering “1”

apai@apailab:-$ source gap\_sdk.sh

Select the target:

1 - GAP\_EVK\_AUDIO

1

The target board you have chosen is : gap9\_evk, GAP9\_V2

1. Test the GAP9 board:

* cd helloworld/
* gap init
* gap build
* gap run

1. The board is correctly running if you see the Hello World from the cores of GAP9!

# **How to run the code**

1. **After setting up GAP9, position yourself in this lab’s directory in the same terminal**

|  |
| --- |
| $ git clone https://github.com/EEESlab/APAI25-LAB10-End-To-End-Deployment.git  $ cd APAI25-LAB10-End-To-End-Deployment |

1. **Run the python code**

|  |
| --- |
| $ python nntool\_generate\_model.py |

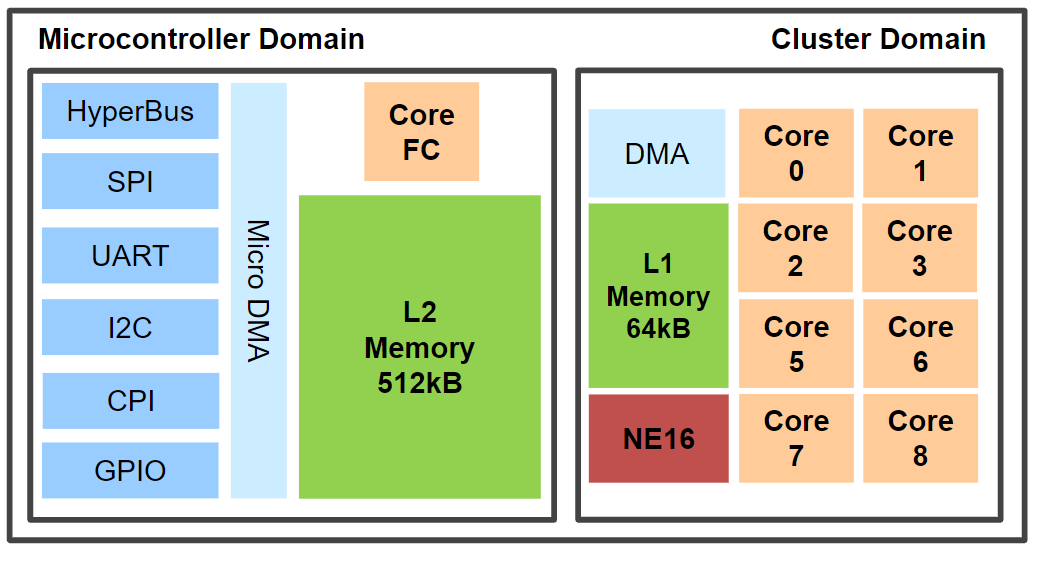
It will generate and compile the code for running the neural network and then run it.

## **LAB: THEORETICAL PART**

Here are the [lecture](https://virtuale.unibo.it/pluginfile.php/1489883/mod_resource/content/0/FConti_APAI_04_DigitalAccelerators2.pdf) slides on NE16

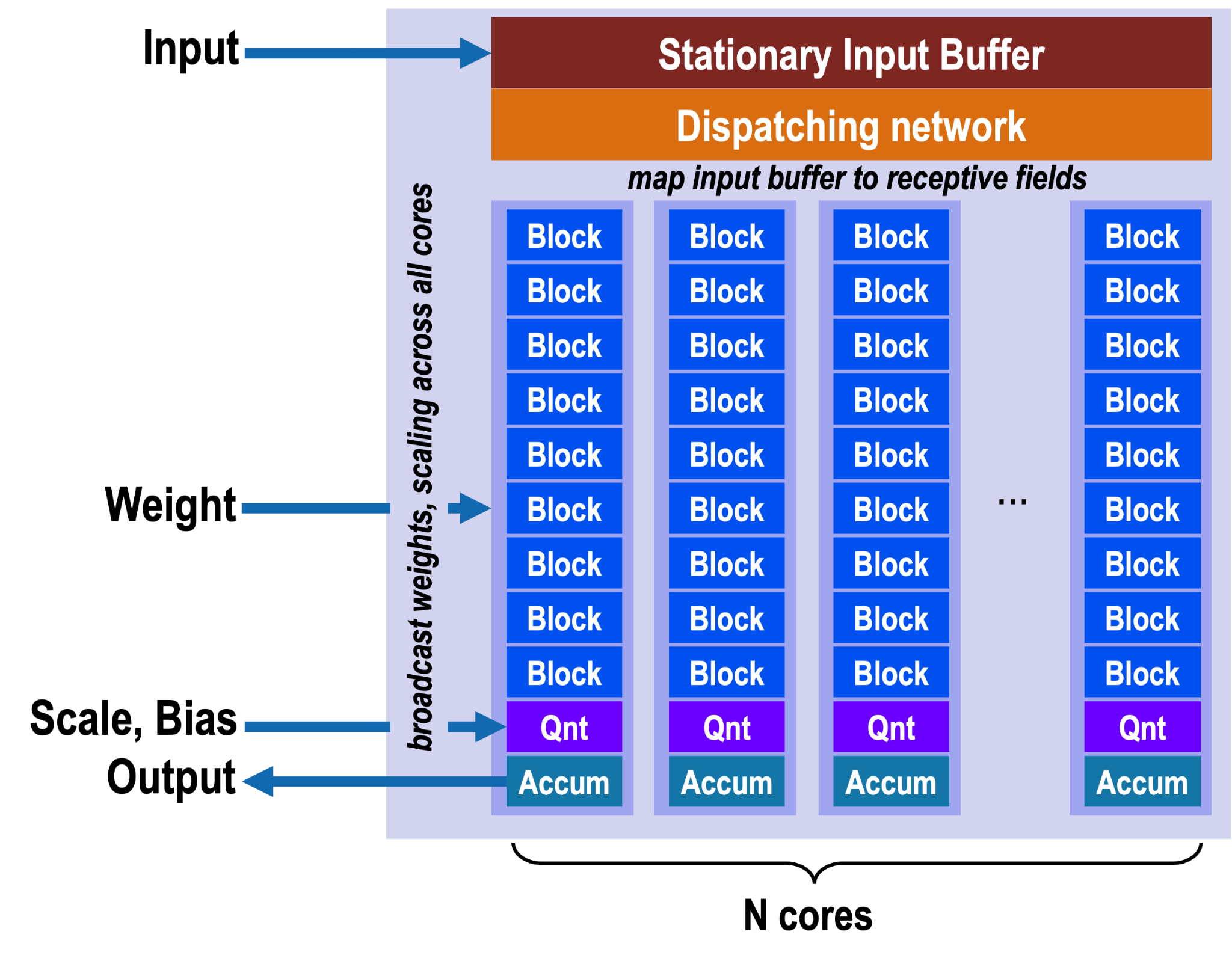
### Recap: architecture overview

* Fabric Controller (FC)
* 8-cores cluster (CL)
* NE16

****

*Figure 3. PULP Architecture*

NE16 is a flexible weight precision Neural Engine.



*Figure 4. NE16*

A **core** is responsible for computing 1 pixel of output data.

* Each Core contains 9 blocks.
* A block contains 16 1-bit × 8-bit multipliers that are used to multiply 1-bit of the weights with 8-bits of the input.
* The results of all the multipliers from all the blocks in one core are then summed up into one value that gets stored in the accumulator.
* The accumulated results can be optionally normalized and requantized to 8 bits.

**In our case:** the specific configuration of NE16 contains **9 cores**,each one having **9 blocks,**  that work on **ICP=16** input channels and produce **OCP=32** output channels.

**NE16’s peak performance**

|  |  |  |
| --- | --- | --- |
|  | **3x3 mode** | **1x1 mode** |
|  | **Perf** (8b MAC/cycle) | **Perf** (8b MAC/cycle) |
| **NE16** | 162 | 144 |

## **LAB: Hands-on**

Each of the following tasks consists of profiling the network execution to understand how the performance varies. We use the GAP9 board to do all the measurements.

# **Exercise A1: theoretical peak performance**

In this exercise, we will compute the theoretical peak performance of the 8-core PULP cluster in terms of computational capabilities in *MAC/cycle*.

You must count the number of *8b MAC* operations that can be done with 8-cores having 32-bit registers and 8-bit operands (SIMD operations enabled).

**[ADD YOUR RESULT HERE]**

|  |  |
| --- | --- |
|  | **Perf** (8b MAC/cycle) |
| **8-core cluster** |  |

# **Exercise A2: Network structure**

You can open the Tflite model of the network (which is in ./models/SSD\_tin\_can\_bottle.tflite) and see its structure, go to <https://netron.app>and drag and drop the *.tflite model* (tflite is an alternative to ONNX format) of the SSD. Take a screenshot and report here the first five layers of the network.

**[IMAGE HERE]**

Calculate also the number of parameters and of mac of the first and second layer

**[FILL THE TABLE HERE]**

|  |  |  |
| --- | --- | --- |
| Layer | MACs | Params |
|
| **Conv2d**  (Conv3x3) |  |  |
| **DepthwiseConv2d**  (Conv1x1) |  |  |

# **Exercise A3: Running the network 8-core cluster vs NE16**

We want to profile the performance of the Mobilenetv2-*SSD* by running it with both the 8 cores cluster and NE16*:*

Try running the network by simply launching the nntool\_generate\_model.py

|  |
| --- |
| $ python nntool\_generate\_model.py |

Did it work ? No! Obviously there is a problem. Take a screenshot and **report it here** (should start with “Allocation failed”).

**[IMAGE HERE]**

As we can see, the error is telling us that there is not enough memory for our network.

Let's look at the nntool\_generate model. We see that l2\_memory is set to 1.5 MB: this means that the whole memory is occupied by the network, but where would I load my code?

Go inside the “Execute\_on\_target/build/build/” folder (this is the full relative path from the folder you are now) and run the following command:

|  |
| --- |
| $ size ssd\_Bottle\_Tincan |

Take a screenshot and report it here:

**[IMAGE HERE]**

This command tells us the executable size to be run on our device in the dec field.

What it is ? **[ADD YOUR RESULT HERE]**

To be on the safe size, let’s try to set the l2\_memory field to 1.3 MB and then run the model.

We verify the correctness of the model by analyzing its output. After the execution, the python script also shows the output of the network (an image of a bottle with its corresponding bounding box).

Paste it here:

**[IMAGE HERE]**

Now, run the code both by using the NE16 or the Cluster (can by done changing the value of USE\_NE16 in the python code and rerunning the code).

Note:

* latency in [ms] is calculated as cycles / core frequency.
* Consider a Frequency of 370MHz.

**[FILL THE FOLLOWING TABLES]**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 8-core cluster | | | |
| Layer | MACs | Cycles | MAC/cycles | Latency [ms] |
|
| **S63\_DW\_CONV\_2D** |  |  |  |  |
| **S76\_expr\_3** |  |  |  |  |
| **S105\_CONV\_2D**  **(1x1)** |  |  |  |  |
| **S189\_CONV\_2D**  **(3x3)** |  |  |  |  |
| **Full CNN** |  |  |  |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | NE16 | | | |
| Layer | MACs | Cycles | MAC/cycles | Latency [ms] |
|
| **S63\_DW\_CONV\_2D** |  |  |  |  |
| **S76\_expr\_3** |  |  |  |  |
| **S105\_CONV\_2D**  **(1x1)** |  |  |  |  |
| **S188\_CONV\_2D**  **(3x3)** |  |  |  |  |
| **Full CNN** |  |  |  |  |

**Questions:**

1. Which layer has the highest MAC/Cycles when executing with the 8-core cluster? **[answer]**
2. Which layer has the highest MAC/Cycles when executing with NE16? **[answer]**

# **Exercise A4: Peak Performance Percentage**

Calculate the Peak Performance Percentage (PPP) for both the 8-cores cluster and the NE16.

The PPP is calculated as:

PPP = \frac{measured\  perf}{peak \ perf}
%e51c1fbd-40ac-4f85-98f3-af7dfefbdcc9

**[FILL THE TABLES]**

|  |  |  |  |
| --- | --- | --- | --- |
|  | 8-core cluster | | |
| Network | Measured  MAC/cycles | Peak Perf. (MAC/cycles) | PPP (%) |
|
| **S76\_expr\_3** |  |  |  |
| **S105\_CONV\_2D**  **(1x1)** |  |  |  |
| **S189\_CONV\_2D**  **(3x3)** |  |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
|  | NE16 | | |
| Network | Measured  MAC/cycles | Peak Perf. (MAC/cycles) | PPP (%) |
|
| **S76\_expr\_3** |  |  |  |
| **S105\_CONV\_2D**  **(1x1)** |  |  |  |
| **S188\_CONV\_2D**  **(3x3)** |  |  |  |

**NE16 hints:**

* remember that NE16 has different peak performance depending on the convolution type (1x1 or 3x3)
* Also remember that some operations cannot be performed on the NE16

**Questions:**

1. Which layer has the highest PPP when executing with the 8-core cluster? **[answer]**
2. Which layer has the highest PPP when executing with NE16? **[answer]**

# **Exercise A5: compare execution on the 8-core cluster vs. NE16**

Compare the performance (MAC/cycles) when executing the network with the 8-cores cluster vs. NE16

* Performance improvement: MAC/cycles [NE16] / MAC/cycles [8-cores]

**[FILL THE TABLE]**

|  |  |  |  |
| --- | --- | --- | --- |
|  | 8-core cluster | NE16 |  |
| Network | MAC/cycles | MAC/cycles | Performance improvement |
|
| **S63\_DW\_CONV\_2D** |  |  |  |
| **S76\_expr\_3** |  |  |  |
| **S105\_CONV\_2D**  **(1x1)** |  |  |  |
| **S188\_CONV\_2D**  **(3x3)** |  |  |  |
| **Full CNN** |  |  |  |

**Questions:**

1. Which layer gets the largest improvement? Why? **[answer]**

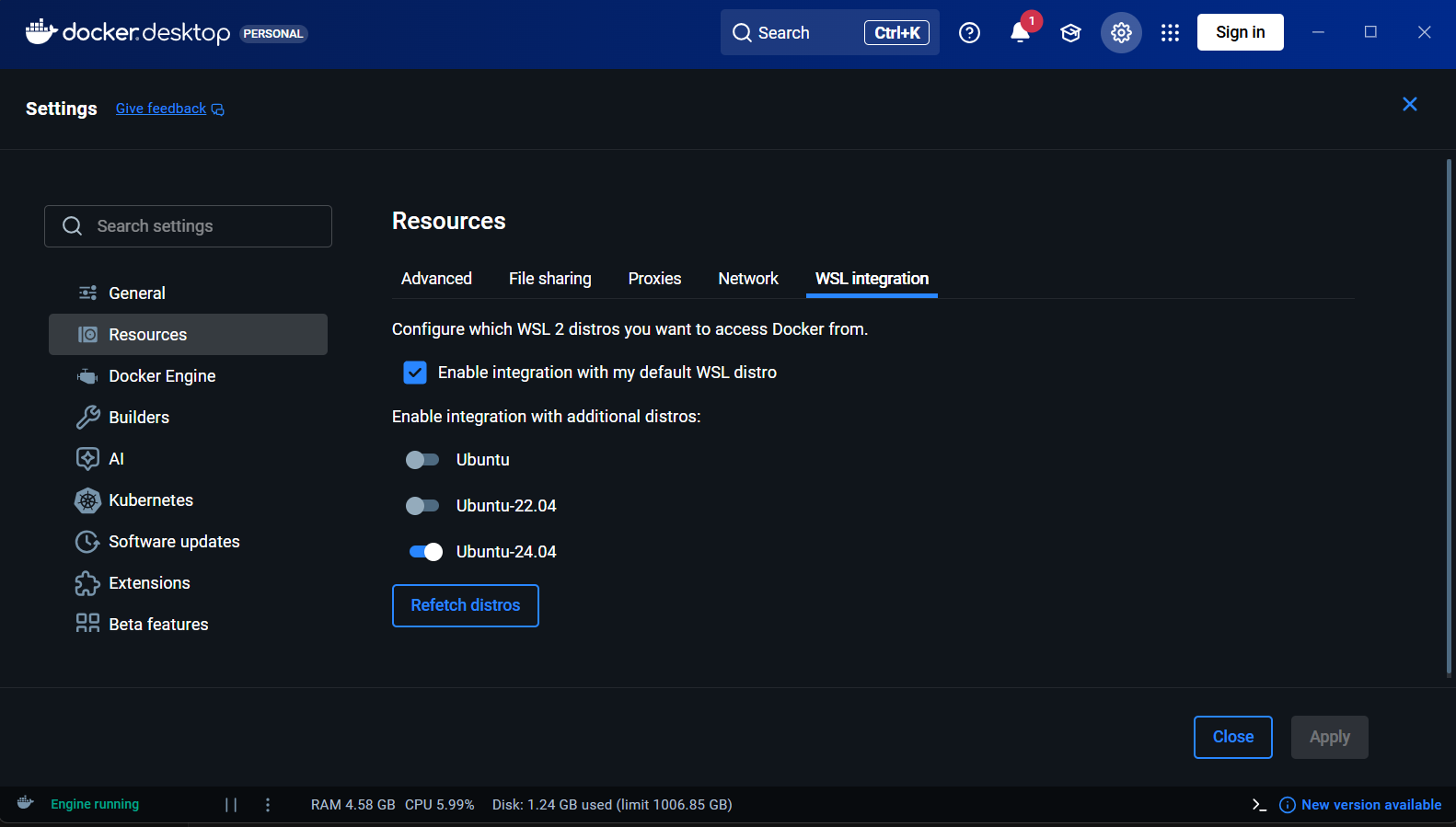
# **Part 2 (Neural Network Deployment with Deeploy)**

We thank Victor J.B. Jung (ETH Zurich) for this tutorial.

# **Setup**

This setup requires a different Docker image with respect to the one used in the previous labs. To this aim:

1. Open your WSL2 machine (Ubuntu24) that you already installed for the previous labs.   
   **It MAY be necessary (i.e., check first if it works as-is) to change settings on docker desktop (settings (the gear) -> resources -> wsl integration) as it defaults to OFF for all the WSL distros installed.**



1. Clone and enter this repository with:

$ git clone https://github.com/EEESlab/APAI25-LAB10-End-to-End-Deployment.git

$ cd APAI25-LAB10-End-To-End-Deployment

1. Open this folder in VSCode with:

$ code .

1. As the folder opens, click on "Re-open in container" to install and run the Docker image reserved to this lab.
2. Set up Deeploy with:

$ git submodule update --init --recursive

$ cd Part2-Deeploy && pip install -e . && cd .

# **Launching basic Deeploy tests**

Deeploy generates the code for DNN deployment from an ONNX file. From the:

APAI25-LAB10-End-To-End-Deployment/Part2-Deeploy/DeeployTest/

folder, you can use the testRunner to compile ONNXs and execute the output code using the appropriate simulators. To validate your installation, you can run a simple Add node on each platform:

$ python testRunner\_generic.py -t Tests/Adder

$ python testRunner\_cortexm.py -t Tests/Adder

$python testRunner\_mempool.py -t Tests/Adder

$python testRunner\_snitch.py -t Tests/Adder/

$ python testRunner\_siracusa.py -t Tests/Adder --cores=8

**At the end of every test, you will see a green line of prompt if the test succeeded, or a red one if failed. Other errors might appear, but if a green prompt is there, the test is passed anyway.** Once all these basic tests are passed, we can jump into the basics of Deeploy.

# **Deeploy basics**

Deeploy is a compiler that transforms static computational graph (represented with the [ONNX format](https://onnx.ai/onnx/operators/)) into bare-metal and optimized [C](https://www.c-language.org/). More specifically, it generates an application that can be deployed on the desired platform.

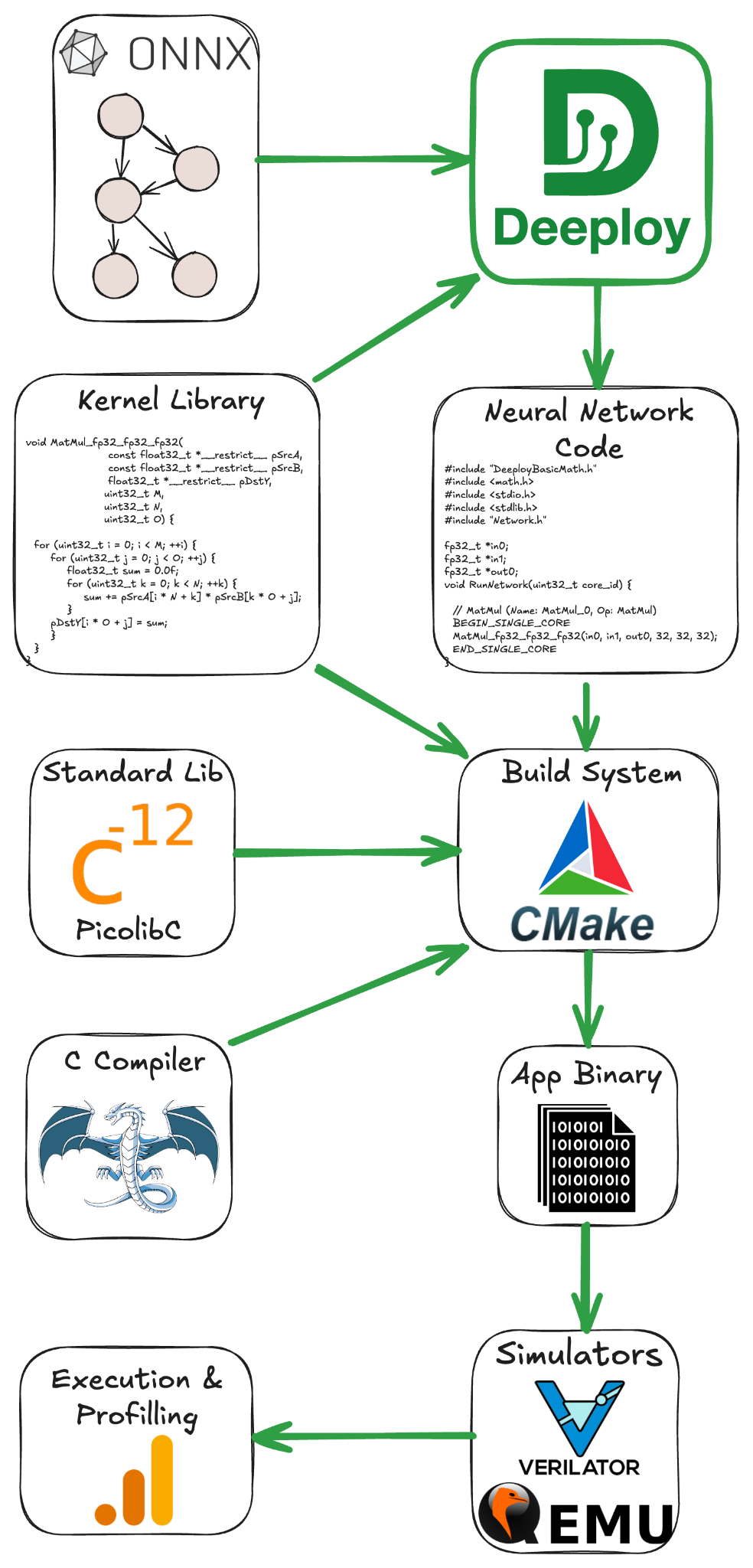
Hence, Deeploy's inputs are:

* An ONNX file describing your neural network.
* Input tensors.
* Expected output tensors generated with your favorite framework (ONNXRuntime or Torch, for instance).

Deeploy is shipped with a comprehensive testing framework conveniently named DeeployTest. This testing framework contains Test Runners for end-to-end testing of your network on a given platform. More specifically, a Test Runner compiles a given ONNX file, builds the project, feeds the inputs into the compiled neural network, and compares the output with the golden values to ensure correctness.

If you followed this tutorial correctly, you already used Test Runners (e.g., testRunner\_siracusa.py) to validate the Deeploy installation! We will dive into the details of the Test Runners Command-Line Interface (CLI) very soon, but first, let's look at the tools and libraries used downstream in Deeploy.

The figure below gives an overview of the deployment stack. As you can see, there are several steps to take before actually running the application. For the build system (*e.g.,* the tool to organize compilation and linking), we use [CMake](https://cmake.org/). The default C compiler shipped with Deeploy is [LLVM 15](https://llvm.org/), but it supports GCC, given that you provide a local installation. To generate the Application Binary, we link the Network Code with the necessary Kernel Libraries and a Standard C Library (here [Picolibc](https://github.com/picolibc/picolibc)). Then, we feed this Application Binary to the appropriate simulator; from there, you can verify the correctness and benchmark the application.



You can visualize the ONNX graphs using [Netron](https://netron.app/). Either use the web interface or install one of the VSCode extensions (e.g., [vscode-netron](https://marketplace.visualstudio.com/items?itemName=vincent-templier.vscode-netron)).

# **Target platform: Siracusa**

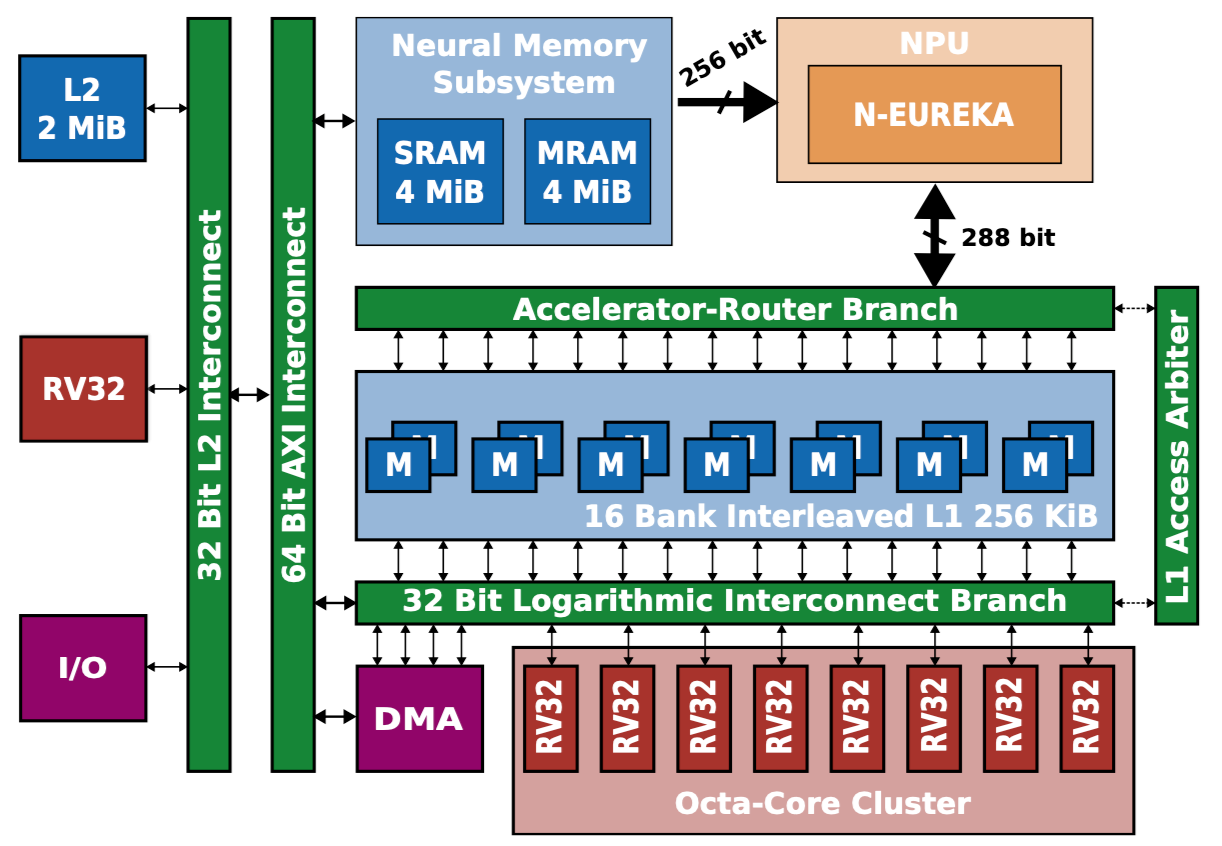
Let's also quickly review the Siracusa platform to understand what kind of hardware we are going to deploy on. Below is the high-level block diagram of Siracusa, compute-wise we will mainly use:

* The cluster of RV32 cores, they are modified to be great at crunching numbers. They feature [SIMD](about:blank), hardware loops (see the [RI5CY user manual](https://www.pulp-platform.org/docs/ri5cy_user_manual.pdf), p17), and the [XPULP](https://pulp-platform.org/docs/hipeac/acaces2021/04_PULP_Accelerators.pdf) ISA extensions.
* The [NEUREKA](https://github.com/pulp-platform/neureka) NPU, an accelerator targeting integer convolutions.

In terms of memories, we have:

* L3: An off-chip RAM (not shown on the block diagram) of 16MB capacity. The L3 has its own DMA that can transfer data to L2.
* Neural Memory Subsystem (NMS): An SRAM/MRAM-based *Weight Memory* to store constants with a direct link to the NPU.
* L2: An on-chip SRAM-based L2 memory of 2MB.
* L1: A TCDM memory of size 256KB.

The on-chip DMA indicated on the block diagram can transfer data between the Weight Memory, the L2, and the L1.



Now that you understand the hardware and the kind of workload we want to execute. Let's deploy using various optimizations to study their impact. The first parameter we can play with is the number of cores from the RV32 cluster to use.

# **Exercise B1 - Visualize the ONNX**

The ONNX graphs are in DeeployTest/Tests/<TestName>/network.onnx. The networks are increasing in complexity, Adder is a single node network for unit testing, while MobileNetv2 is a simple sequential network mostly made of convolutions. Finally, the Transformer network showcases a typical transformer block used in Encoder and Decoder networks. If you want to peek at a complex network, you can visualize microLlama/microLlama128.  
  
**Netron might cause problems because it is not updated. In case you have issues, this can be fixed by running on the terminal:  
pip install netron==8.7.8**

✅ Task: Visualize the ONNX graph of the Adder, MobileNetv2 (first block), and Transformer (one of the blocks):

**[PASTE IMAGES HERE]**

# **Exercise B2 - See Deeploys’ graph optimizations**

Now that we understand Deeploy's input, let's check the output-generated code! The generated code is located in the following directory: DeeployTest/TEST\_<PlatformName>/Tests, and the Network.c file is the interesting one.

The generated code is trivial for the Adder graph; we simply use the template for the Add node of the Generic platform. You can find the template declaration in Deeploy/Targets/Generic/Templates/AddTemplate.py.

Now, if you want to look at something a bit more complex, run python testRunner\_generic.py -t ./Tests/miniMobileNetv2 (from DeeployTest) and look at the generated code. There are two interesting points you can notice:

* We hoist the constants at the top of the file.
* In the RunNetwork function, we sequentially have node templates to execute the operands and malloc/free to manage the memory. You can open the ONNX graph of miniMobileNetv2 on the side to try to match the nodes of the graph with their generated code.

Deeploy applies passes on the ONNX graph to transform its topology and optimize its execution. Let's visualize the effect of the passes used in the Siracusa Platform. First, let's execute our miniMobileNetv2 on Siracusa with python testRunner\_siracusa.py -t ./Tests/miniMobileNetv2. You can find the original ONNX graph at DeeployTest/Tests/miniMobileNetv2/network.onnx, and the transformed ONNX graph at DeeployTest/TEST\_SIRACUSA/Tests/miniMobileNetv2/deeployStates/backend\_post\_binding.onnx. Open both ONNX graphs side by side to compare them.

You can notice the effect of two passes on the graph:

* One pass fuses the Conv and RequantShift nodes. This is a common technique named [Operator Fusion](https://medium.com/data-science/how-pytorch-2-0-accelerates-deep-learning-with-operator-fusion-and-cpu-gpu-code-generation-35132a85bd26) and used in many DNN compilers.
* Another pass is adding a Transpose node before the RequantizedConv in order to align the tensor layout from CHW to HWC (where C = Channels, H = Height, and W = Width). The HWC tensor layout is required to use optimized Convolution kernels (to learn more, check out [this blog post](https://www.intel.com/content/www/us/en/developer/articles/technical/pytorch-vision-models-with-channels-last-on-cpu.html)).

✅ Task: Visualize the effect of passes on the ONNX graph for the Siracusa platform.

**[PASTE IMAGES HERE]**

Now that you understand the basics of Deeploy, let's jump into the optimized deployment of a small language model on the Siracusa SoC.

# **Micro Llama on Siracusa - Transformer brief summary**

In this section, we will study the optimization of the deployment of a small language model. To fully understand this section, you need some basic understanding of Transformer's architecture and Language Model inference mode. If you need a refresher on Transformer's architecture, check out the *Transformer Basics* section of [Lilian Weng's blog post](https://lilianweng.github.io/posts/2023-01-27-the-transformer-family-v2/#transformer-basics).

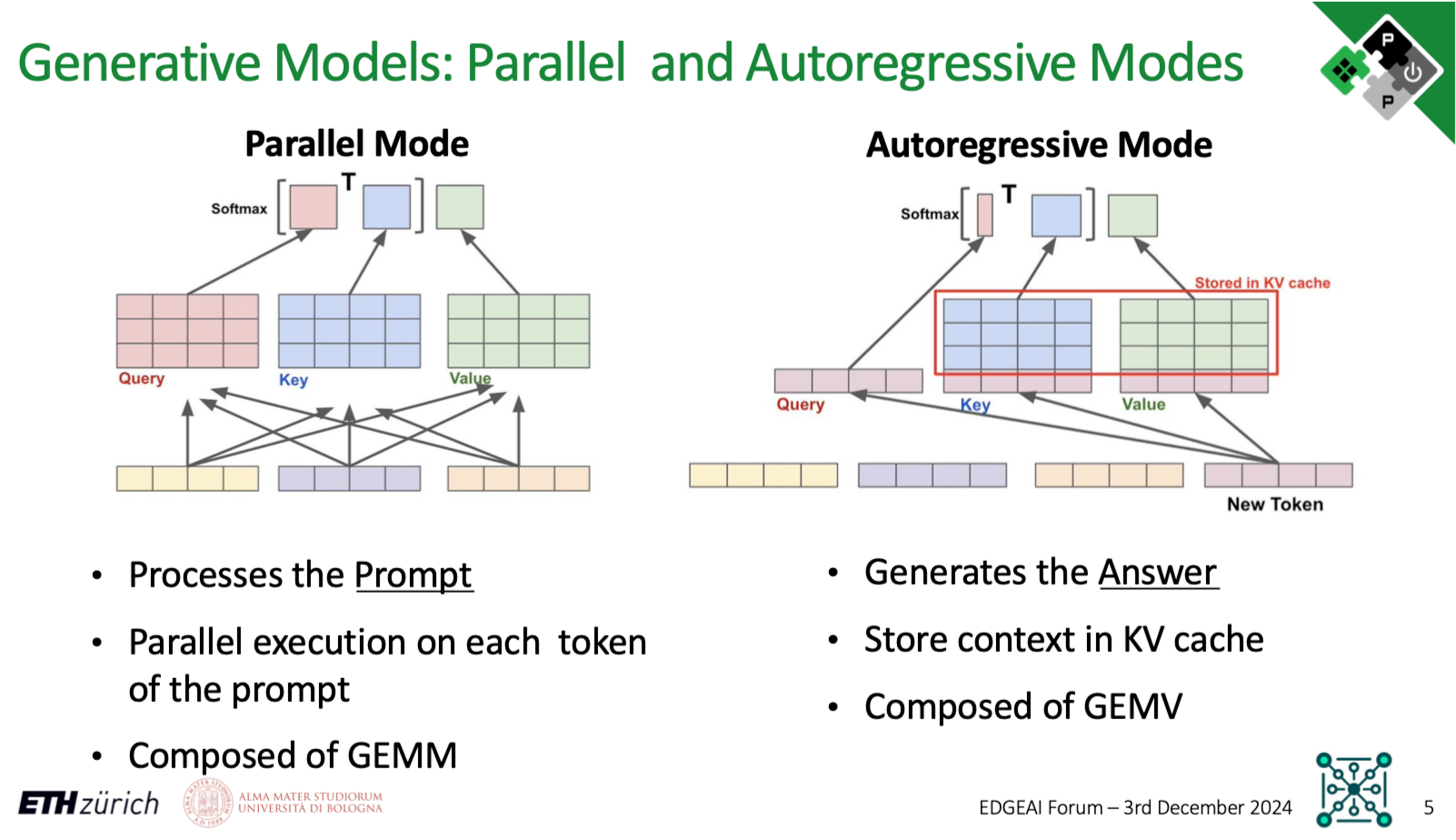
Now, Language Models have two inference modes:

* The **Parallel Mode** (AKA *Prefill Mode*) is used to process the tokens of the prompts in parallel and generate the KV cache of the prompt and the first token of the Language Model's "reply". This mode contains mostly GEMMs.
* The **Autoregressive Mode** generates the rest of the Language Model's reply. It uses the KV cache from the previous step, generates a new KV cache entry, and predicts the next token. This mode contains mostly GEMVs.

To summarize, to generate a Language Model reply of *N* tokens, there is:

* One **Parallel Mode** inference to process the prompt and generate the first token.
* *N*−1 **Autoregressive Mode** inferences to generate the rest of the tokens.

The slide below visually represents the **Parallel Mode** and **Autoregressive Mode**.



# **Exercise B3 - Profiling Micro Llama on Siracusa**

# Now that you understand the hardware and the kind of workload we want to execute. Let's deploy using various optimizations to study their impact. The first parameter we can play with is the number of cores from the RV32 cluster to use.

✅ Task: Measure and compare the runtime of the microLlama128 model using 1 and 8 cores. Compute the speedup ratio; why is it not 8? **[YOUR ANSWER HERE]**

Hint: python testRunner\_siracusa.py --help will list and explain the available flags.

**[FILL THE TABLE]**

|  |  |  |
| --- | --- | --- |
| Number of Cores | Cycles | Speedup |
| 1 |  | 1x |
| 8 |  |  |

# **Exercise B4 - Tiling in Deeploy**

It's due time to talk about data movement now! We use all 8 cluster cores, which is great, but where do these cores fetch the data from? By default, when using testRunner\_siracusa.py, all data is in L2; there is no tiling, and cores read and write data directly to/from L2. As the L2 memory is "further away" from the cluster, load/store takes several cycles, which is non-optimal.

What we really want is to use the L1 memory, which provides 1 cycle latency load/store! But as the capacity is relatively small (256KB), we need to **tile our layers**. Tiling operands for an accelerator featuring only scratchpad memories is not trivial (unlike in architectures with data caches). For each layer, the compiler has to decide on tile size, a tiling schedule, a buffering strategy (single buffer, double buffer, etc...), and a memory allocation strategy. Then, the compiler must generate the code to configure and launch each transfer and place barriers accordingly to maximize concurrency.

The good news is that Deeploy can already do that! So, let's generate and run some tiled code to see the impact of tiling on the runtime.

✅ Task: Run microLlama64\_parallel on Siracusa with different L1 configurations. Find one "bad" and one "good" configuration, and explain why. To do so, launch the command:

*python testRunner\_tiled\_siracusa.py -t Tests/microLlama/microLlama64\_parallel --cores=8 --l1 <YOUR\_VALUE> --defaultMemLevel=L2`*

**[FILL THE TABLE]**

|  |  |  |
| --- | --- | --- |
| Case | L1 Allocated Memory | Cycles |
| Bad |  |  |
| Good |  |  |

***EXTRA: Assessing DMA Overheads***

To measure the effect of some optimizations in more detail, you can use the --profileTiling flag. This flag will enable a code transformation that will insert print displaying the runtime of several critical code sections. For instance, profiling an *Integer Layer Normalization* layer from L2 with two tiles will return the print the following:

[INTEGER\_RMSNORM L2][SB][0 ops][Tile 0] Input DMA took 489 cycles

[INTEGER\_RMSNORM L2][SB][0 ops][Tile 0] Kernel took 43305 cycles

[INTEGER\_RMSNORM L2][SB][0 ops][Tile 0] Output DMA took 534 cycles

[INTEGER\_RMSNORM L2][SB][0 ops][Tile 1] Input DMA took 82 cycles

[INTEGER\_RMSNORM L2][SB][0 ops][Tile 1] Kernel took 3254 cycles

[INTEGER\_RMSNORM L2][SB][0 ops][Tile 1] Output DMA took 49 cycles

With this profiling trace, you can clearly measure the overhead of DMA transfers. When the profiling is turned ON, the total runtime of the application will encompass the prints.

# **Exercise B5 - Using the on-chip Neureka NPU accelerator and the NMS**

To use the NPU, you can use the testRunner\_tiled\_siracusa\_w\_neureka.py. The Linear layers will automatically be executed by the NPU. To enable the NMS, use the --neureka-wmem flag. When the NMS is enabled, the constant tensors used by the accelerator will be placed in the Weight Memory.

✅ Task: Execute Micro Llama in parallel and autoregressive mode using the NPU, derive the speedup at the model level and at the layer level compared to execution without NPU. To do so, run:

python testRunner\_tiled\_siracusa\_w\_neureka.py -t Tests/microLlama/microLlama64\_parallel --cores=8 --l1 64000 --defaultMemLevel=L2

**[FILL THE TABLES]**

|  |  |  |
| --- | --- | --- |
| Parallel mode | Cycles | Speedup |
| Tiled |  | 1x |
| Neureka NPU |  |  |

|  |  |  |
| --- | --- | --- |
| Autoregressive mode | Cycles | Speedup |
| Tiled |  | 1x |
| Neureka NPU |  |  |

✅ Task: Why does the NPU bring more speedup in parallel mode than in autoregressive mode?

**[ANSWER HERE]**

# **Et voilà, this is the end of the tutorial. Thank you for following it until the end. If you are interested in learning more about Deeploy, see the section below!!**

# **VERY NICE EXTRA FOR HACKERS: Other Deeploy tutorials**

This tutorial introduced the basics of how to use Deeploy from the user perspective.

For more in depth tutorials on the Deeploy environment, please visit the [unitn-2025-dl](https://github.com/pulp-unibo/unitn-2025-dl) tutorials (they are PhD-level course material)!

In particular, if you want to hack Deeploy and introduce new custom operators, follow the [deeploy-newop-tutorial](https://github.com/pulp-unibo/deeploy-newop-tutorial/tree/dec652f454d0c50ca3816e695fdd142f67d651ea)!