# A Benchmark Framework

The benchmark framework coordinates execution of the benchmark's behavioral model among the various components in the system as described in the following sections.

## A.1 Framework Hardware

In its most basic form, the framework consists of a host PC, a binary runner application GUI, the DUT, and a thin shim of firmware on the DUT that adheres to the framework communication protocol.

The framework for this particular benchmark provides two hardware configurations: one for measuring latency and accuracy, and the other for measuring energy (see Figure 3). The former mode is a simple connection between the host PC and the DUT through a serial port. The latter configuration expands the framework to include an electrical-isolation proxy (IO Manager), and an energy monitor (EMON), which supplies and measures energy consumption. Both configurations share the same process for initializing the DUT, loading the input test data, triggering the inference, and collecting results. The only difference is the energy configuration must initialize and manage the IO Manager and EMON hardware as well.

Two framework configurations are provided because energy configuration is a more complex setup, and energy scores may not be desired.

In the performance configuration, the Host PC talks directly to the DUT. In energy configuration, the IO Manager passes commands from the host software to the DUT through a serial port proxy. This proxy maintains a resilient serial connection state regardless of power cycling, and level shifters provide voltage domain conversion because the IO Manager GPIOs run at 5 Volts, whereas the DUT GPIO may vary. This configuration electrically isolates the DUT. For this framework version, the IO Manager is deployed as an Arduino UNO with its own custom firmware.

The energy monitor supplies and measures energy. The runner contains three EMON drivers for the following hardware: the STMicroelectronics LPM01A, the Jetperch Joulescope JS110, and the Keysight N6705. Regardless of the EMON used, it must supply one channel for the DUT and the other for the level shifters, the power used by level-shifters is not included in the total energy score because it is a cost associated with the framework and not the DUT. Only the power delivered to the core is measured. Furthermore, only one power supply is allowed to power the core; no additional energy supplies, such as batteries, supercapacitors, or energy harvesters may be used to defeat the measurement. Some energy monitors provide settings to increase the sample frequency or change the voltage. The runner GUI exposes these options to the user, so that the user can measure energy at a lower voltage. Note that the tradeoff between performance and energy can be dictated by the voltage, as higher voltage is often required to run at higher frequencies, and on-board SMPS conversion efficiency may vary considerably. This is one of the key insights of the benchmark: performance vs. energy tradeoffs.

The configurable sampling frequency is the rate at which data is returned from the EMON, which is much lower than the actual ADC sampling rate for all three devices. Since these internal sampling rates are typically well-within the time constant of the decoupling DUT's power delivery decoupling capacitance, the rate has no impact on the score.

## A.2 Framework Firmware

To provide consistency between the benchmark framework and the DUT implementation, a simple API is defined in C code that runs on the DUT. The DUT must provide two basic functions to interface to the host runner: a UART interface for receiving commands and returning status, and a timestamp function. In performance mode, the timestamp function is a local timer with a resolution of at least one millisecond (1 kHz); in energy mode, the timestamp function generates a falling edge GPIO signal which is used to trigger an external timer and synchronize data collection with the energy monitor. The API C code also provides functions to load data into a local buffer, translate and copy that data to the input tensor, trigger an inference, and read back the predictive results.

All of this functionality is partitioned into boilerplate code that does not change, or internally implemented, and code that has to be ported to the particular platform, called submitter implemented.

The submitter implemented code is the API that connects to the particular SDK, which may consist of optimized MCU libraries, or interface to hardware accelerators.

The API implemented by the submitter requires:

1. A timestamp function which prints a timestamp with a minimum resolution of one millisecond, or generates an open-drain, GPIO toggle, depending on whether the DUT is configured latency or energy measurement.
2. UART Tx and Rx functionality, for communicating with the framework software.
3. A function for loading the input tensor with data sent down by the framework runner UI.
4. A function for performing a single inference.
5. A function for printing the prediction results.

The first two API functions provide external triggering and generic communication support; the latter three implement the minimum requirements to perform inference. Once these functions have been implemented, the framework software can successfully detect and communicate with the DUT.

The internally implemented firmware connects the API function calls in such a way that the framework software can send down an input dataset and instruct the firmware to perform a given number of iterations. The results are then extracted from the DUT (or the energy monitor) by the framework software.

### A.3 Framework Software

The benchmark framework includes a Host PC GUI application called the runner. The runner allows the user to perform some basic setup and configuration, and then executes a pre-defined command script which activates the different hardware components required to execute the benchmark.

The runner is needed for three main reasons:

1. To provide a consistent benchmark interface to the user.
2. To standardize the execution of the benchmark, including measurements
3. To facilitate downloading the large number of input files required for accuracy measurements, since the typical target platform has less than a megabyte of flash memory.

The runner also provides visualization of the energy data, and feedback that can be useful for debugging framework connectivity.

The runner GUI, for the energy configuration, is shown in Figure 4.

The runner detects and initiates a handshake with detected hardware. After initialization, the test is started by clicking a button on the GUI, and a series of asynchronous commands are issued. These commands use the DUT API firmware to load input data and perform measurements. After the test completes, the runner collects the scores from the DUT and presents them to the user. The input stimuli files are located in a directory on the host PC, and are fed into the DUT depending on the measurement.

The measurement procedure is as follows:

1. Latency – Perform the following five times: download an input stimulus, load the input tensor (converting the data as needed), run inference for a minimum of 10 seconds and 10 iterations, measure the inferences per second (IPS); report the median IPS of the five runs as the score.
2. Accuracy – Perform a single inference on the entire set of validation inputs, and collect the output tensor probabilities. The number of inputs vary, depending on the model. From the individual results, compute the Top-1 percent and the AUC. Each model has a minimum accuracy that must be met for the score to be valid.
3. Energy – Identical to latency, but in addition to measuring the number of inferences per second, measure the total energy used in the timing window and compute micro-Joules per inference. The same method of taking the median of five measurements is used.

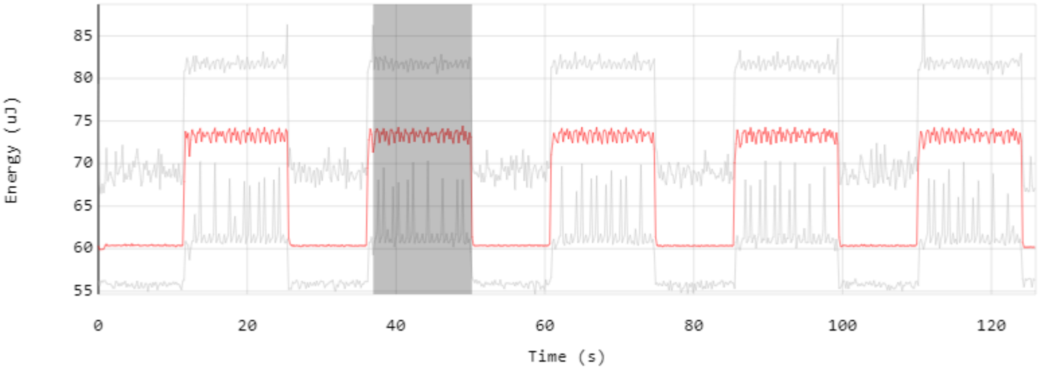

Figure 6: Energy Viewer

Results are stored in a folder and can be reloaded again. An energy viewer allows the user to examine the energy trace (Figure 6)