Real-Time Embedded Deep Learning on Elk Audio OS

Domenico Stefani
Dept. of Information Engineering and Computer Science
University of Trento
Trento, Italy
domenico.stefani@unitn.it

Luca Turchet
Dept. of Information Engineering and Computer Science
University of Trento
Trento, Italy
luca.turchet@unitn.it

***Abstract*—Recent years have witnessed significant advancements in deep learning architectures for music, along with the availability of more powerful embedded computing platforms specific to low-latency audio processing tasks. These recent developments have opened promising avenues for new Smart Musical Instruments and audio devices that rely on the execution of deep learning models on small embedded computers. Despite these new opportunities, there is a lack of instructions on how to deploy neural networks to many promising embedded audio platforms, including the embedded real-time Elk Audio OS. In this paper, we introduce a procedure for deploying audio deep learning models on embedded systems utilizing the Elk Audio OS. The procedure covers the entire process, from creating a compatible code project to executing and diagnosing it on a Raspberry Pi. Moreover, we discuss different approaches for the real-time execution of deep learning inference on embedded devices and provide alternatives for handling larger neural network models. To facilitate implementation and support future updates, we provide an online repository with a detailed guide, code templates, functional examples, and precompiled library binaries for the TensorFlow Lite and ONNX Runtime inference engines. This work aims to bridge the gap between deep learning model development and real-world deployment on embedded systems, fostering the development of self-contained digital musical instruments and other audio devices equipped with real-time deep learning capabilities. By promoting the deployment of neural networks to embedded devices, we contribute to the development of Smart Musical Instruments that are capable of providing musicians and audiences with unprecedented services.**

***Index Terms*—Embedded Audio, Embedded Inference, Deep Learning Inference, Real-time Audio, Smart Musical Instruments**

I. Introduction

In recent years we have witnessed great advancements in Deep Learning (DL) architectures for audio [1] [2] and low-latency embedded computing platforms [3] [4] [5] [6]. DL has been successfully used to model audio effects [7], manipulate tone and timbre in new ways [2], and recognize in real-time high-level properties of a sound source such as expressive playing techniques [8] or beat tracking [9].

Similarly, the increase in computational power of commonly available embedded computers fostered the development of several embedded audio platforms such as Elk Audio OS [3], Bela [4], Prynth [10], Satellite CCRMA [11], and Axoloti[1]. However, developing DL models for audio and deploying them onto embedded platforms are two tasks that require two very different sets of skills with little overlap. In particular, DL requires the following skills:

- Some high-level coding language knowledge;
- Advanced knowledge of the mathematical and probability concepts behind layers, activations, and other operators;
- Advanced domain knowledge for the data and preprocessing (including preprocessing software and libraries).

Conversely, model deployment requires the following skills:

- Advanced knowledge of lower-level programming languages (often C++ and C);
- Advanced knowledge of compilation and cross-compilation procedures for the project and its dependencies;
- Familiarity with audio processing and feature extraction libraries for the target programming language. Alternatively, advanced knowledge of Digital Signal Processing (DSP) concepts and programming to implement the processing routines needed.

In this context, we believe that any effort in reducing the gap between development and deployment for audio and music DL can foster the creation of new self-contained Digital Musical Instruments (DMIs), such as Smart Musical Instruments (SMIs) [12], as well as AI-equipped audio devices. The focus on embedded platforms derives from the need to provide SMIs with artificial intelligence capabilities through computing devices that can be physically placed inside these instruments. SMIs are an emerging family of musical instruments that are a central component of the Internet of Musical Things (IoMusT), which is the extension of the Internet of Things (IoT) paradigm to the musical domain [13]. As IoMusT devices, SMIs are envisioned to be able to communicate and become part of a "network of interoperable devices" in order to share and receive musical content. In this context, the ability to execute DL inference inside SMIs provides a form of *embedded intelligence* [12] that can be harnessed for real-time audio processing, sensor data manipulation, and high-

[1] http://www.axoloti.com/

level audio property or feature extraction, which in turn can be shared with similar instruments in the network. In this interconnected scenario, embedded inference is particularly relevant as many real-time music applications do not tolerate the inherent latency that would be introduced by a cloud-based DL solution.

Among the embedded platforms mentioned above, Elk Audio OS [3] and Bela [4] are the most prominent and flexible systems available in the open-source scenario. Recently, a pipeline for deploying neural networks to Bela was developed and documented [14]. On the contrary, while Elk Audio OS proved to be a very capable platform for real-time DL [15] [8] [16], no documented deployment procedure exists for the platform. This hinders the creation of self-contained DMIs with intelligent features.

In this paper, we describe the steps required to deploy DL models to the Elk Audio OS on a Raspberry Pi, and we provide an online repository[2] with a detailed guide, code templates and precompiled dependencies to use either the TensorFlow Lite or ONNX Runtime inference engines. We selected such engines because, in our previous study, they ranked as the best-performing and easiest-to-use inference engines on the platform [15]. Moreover, TensorFlow Models can be easily converted to the *Lite* format, while models trained in other frameworks like Pytorch can be converted to the *ONNX* format. We propose a procedure that uses the open-source framework JUCE to develop Virtual Studio Technology (VST) plugins that can be executed on a Raspberry Pi 4 with the Elk Audio OS. This enables developers and DL engineers to run real-time and offline audio models on a Single-Board Computer (SBC) that can be embedded into instruments and standalone devices. Moreover, a byproduct of learning the proposed procedure is that the reader will also have the tools to compile DL-equipped VST plugins for desktop and laptop computers.

The remainder of this paper is organized as follows. Section II reviews other works related to DL and embedded platforms for audio. In Section III we present the tools required to follow the guide, and the motivation behind these choices. Then, Section IV presents an overview of the deployment procedure to follow in order to create a JUCE project, cross-compile plugins for Elk Audio OS, install the OS, configure its Digital Audio Workstation (DAW) and troubleshoot code issues with real-time execution. Furthermore, Section V presents a few considerations on the different modes of inference, i.e., offline, audio-rate real-time, and other real-time approaches. Finally, we draw our conclusions in Section VI.

II. Related Works

The deployment of DL models to embedded devices has recently seen an increase in popularity and relevance. This was the product of multiple factors, which include the increase in computing power of embedded devices and SBC and the research successes in AI for music [1], with a particular focus on real-time approaches [2]. The effects of the former manifested earlier in the form of the development of several open-source audio platforms, such as Elk Audio OS [3], Bela [4], Prynth [10], and Satellite CCRMA [11]. Meneses *et al.* [5] presented a comparison between some of the aforementioned open-source platforms (i.e., Prynth, Bela, and a custom processing unit), resulting in a clear overview of the strengths and drawbacks of the different solutions, which yielded no clear winner. More recently, the Elk Audio OS was presented along with a comprehensive comparison with similar platforms (see [3]). Additionally, Vignati *et al.* [6] compared the performance of the Xenomai Cobalt kernel (used by both Elk Audio OS and Bela) with that of the more common Preempt RT kernel patch for Linux systems, which resulted in a better overall performance from the former under heavy loads.

While most of the aforementioned platforms were not originally devised to perform DL inference, recent efforts have successfully shown how integration is possible and can be made easier for DL developers [14]. Moreover, the increased interest in embedding DL inference for audio into devices and musical instruments has led to events that specifically focused on *embedded AI*, such as the NIME 2022 workshop *Embedded AI for NIME: Challenges and Opportunities* [17].

This paper draws inspiration from the recent work of Pelinski *et al.* [14], who provided a DL deployment pipeline for the Bela platform [18]. The pipeline includes a tool to record sensor reading datasets and a cross-compilation environment to ease the deployment of DL models to Bela. With the latter, the authors stated their intent to promote fast prototyping and experimentation with neural networks for embedded real-time musical applications. The pipeline proposed by the authors is based on the TensorFlow Lite inference engine, and the authors provide a Docker container to cross-compile Bela-compatible DL programs from a host computer. Various efforts by other researchers have been made in similar directions (e.g., Flucoma-Bela[3]). Nevertheless, the work by Pelinski *et al.* relates more closely to this paper, as it presents a clear and stepwise deployment process without assuming deep knowledge of low-level programming concepts and it offers a prepackaged cross-compilation tool to ease the deployment itself.

The main distinction of our work lies in its focus on a radically different platform: the real-time Elk Audio OS and the Raspberry Pi 4. While the combination of Elk Audio OS and Raspberry Pi 4 has been used for DL deployment before [15] [8] [16], the deployment process has not been documented yet. Moreover, the Raspberry Pi offers more powerful hardware than the latest Bela, which can enable more intense tasks such as AI-based audio processing. For this reason, we provide code examples that execute inference at audio rate on the input signal (Section III-D) instead of the sensor data processing example provided by Pelinski *et al.* . Moreover, we provide both clean templates and code examples

[2]https://github.com/CIMIL/elk-audio-AI-tutorial

[3]https://github.com/jarmitage/flucoma-bela

for both TensorFlow Lite and ONNX Runtime, to ease the deployment process for a wider range of DL frameworks.

III. TOOLS

A. JUCE and VST

JUCE is a cross-platform framework for audio plugins and applications. It is a C++ framework with a dual license (i.e., GPLv3 open-source and commercial). JUCE embeds the VST3 SDK and Elk Audio provides support and instructions on how to build a JUCE plugin for their OS. The procedure reported in this paper, and in more detail in the project repository, was tested with JUCE 6 (version 6.0.7).

B. Elk Audio OS

Elk Audio OS [3] is an embedded operating system optimized for low-latency audio processing on embedded hardware. Currently, Elk offers a disk image and a cross-compilation SDK for the Raspberry Pi 4 SBC as open-source, and more hardware platforms[4] under a commercial license. Moreover, up to Elk Audio OS version 0.7.2 Raspberry Pi 3 was supported. The instructions in this paper refer to version 0.11.0. Potential differences in the deployment process for future versions of the Elk Audio OS will be documented in the project's repository (Section III-D).

C. Choice of Inference Engine

Inference of DL models is the process of running an input through the network and executing all the computations required to produce an output prediction. DL models are generally trained and tested in powerful server machines or PCs, using high-level programming languages (e.g., Python) and DL frameworks (e.g., PyTorch, TensorFlow). Training through backpropagation is a particularly compute-intensive task that often requires specialized acceleration hardware and drivers (e.g., CUDA GPUs or TPUs). On the contrary, inference is considerably less computationally expensive and can be optimized for deployment.

In recent years, DL framework companies and developers have focused on in-device inference for edge and mobile computing. In IoT, edge computing has the great advantage of performing relevant computations closer to where input data is gathered. Even more so, embedded in-device inference can be advantageous in terms of action-to-reaction latency, since inference computations can be performed right at the place where the inputs are gathered through sensors, which is devoid of the latency of communication with one or more cloud servers. While in-device computation can have marginal advantages in some IoT, it is an indisputable requirement of music performance tools and many IoMusT systems, even when the learning task allows for slightly more lenient time constraints than audio-rate deadlines [8] (see Section V). For this reason, several C and C++ libraries known as inference engines were made available along with most DL frameworks, to allow for efficient and quick inference, especially for resource-constrained embedded devices.

Our previous work [15] compared the performance and suitability of four of these engines (i.e., TensorFlow Lite, ONNX Runtime, Torch+Torchscript, and RT Neural) for audio tasks on a Raspberry Pi 4 running Elk Audio OS. While exact execution time can depend on a specific model, ONNX Runtime and TensorFlow Lite have been shown to be very quick, well-documented, and easy to use. For the code templates in the repository that follows this paper, we decided to include separately both TensorFlow Lite and ONNX Runtime because TensorFlow users will find it extremely easy to export their model for the former, while models from most frameworks can also be converted to the ONNX including PyTorch[5] too. Support for ONNX Runtime is particularly relevant as a large part of research on black-box audio effect emulation is currently carried out using Pytorch [7] [19] [20] and PyTorch-to-TensorFlow model-conversion is not a straightforward process.

For TensorFlow developers, we suggest choosing the TensorFlow Lite template code, while PyTorch and other developers should convert their models to ONNX and use ONNX Runtime.

D. Project's Repository

This paper defines a procedure to successfully deploy DL models to embedded devices running Elk Audio OS and perform inference. While this paper reports an overview of the deployment procedure, we provide a *detailed guide*, clean source code *templates*, working *examples*, and *inference engine binaries* in order to reduce the effort required for deployment. This substantial addition to the written part of this paper is contained in the `elk-audio-AI-tutorial` repository on the GitHub page of the *Creative, Intelligent & Multisensory Interactions Laboratory* (CIMIL): `https://github.com/CIMIL/elk-audio-AI-tutorial/`. The guide in the project's repository goes into more detail on the deployment process, and it will be kept up to date, addressing potential changes in the new version of Elk Audio OS or inference engines.

IV. DEPLOYMENT PROCEDURE

This section presents an overview of the procedure to deploy a DL model to a Raspberry Pi running the Elk Audio OS. This procedure can also be followed for deployment to a VST plugin (ignoring cross-compilation and device-specific steps) for any platform, including Windows, MacOS, and Linux. All the code and library binaries required to follow the guide are provided in the project repository (see Section III-D), and they will allow readers to skip parts of the guide for easier deployment (e.g., library compilation can be skipped if using the provided binaries). Further updates, improved instructions, and additional inference engines will be added to the detailed guide in the repository.

Instructions assume the use of a Unix-based OS (e.g., Ubuntu Linux), but they also can be followed using Windows

[4]https://elk-audio.github.io/elk-docs/html/intro/supported_hw.html

[5]https://pytorch.org/docs/stable/onnx.html

with the Windows Subsystem for Linux (WSL) or a Linux virtual machine. As mentioned previously, the instructions in this paper refer to version 0.11.0 of Elk Audio OS, but the project repository will be updated with new versions of the OS. Figure 1 shows an overview of the entire deployment process.

The next sections will describe the following steps:

1) Creation of a JUCE project for DL deployment on Elk Audio OS (Section IV-A);
2) Cross-compilation of a plugin and its dependencies (Section IV-B);
3) Installation and communication with Elk Audio OS (Section IV-C);
4) Configuration of Elk's DAW Sushi (Section IV-D);
5) Troubleshooting (Section IV-E);

A. Project creation

JUCE plugin projects can be created using the Projucer app, which is provided with any JUCE distribution. The Projucer handles the configuration of the project, its export formats, and its build systems along with their different configurations (i.e., *exporters*). Moreover, it takes care to create the build files for each exporter whenever the project is saved in the Projucer app. Additionally, it can execute a user-defined command every time a project is saved (post-export Shell command). It is also possible to use CMake to set up JUCE projects, but this will not be covered by the guide.

The following two alternatives can be used to obtain a JUCE project to compile VST plugins that will be compatible with Elk Audio OS:

- Use of one of the provided templates;
- Manual project creation.

1) Templates: The project repository (see Section III-D) contains template projects for ONNX Runtime and TensorFlow Lite. These include the precompiled dependencies and a project configuration (`.jucer` file). The project configuration file ensures that the relative inference library is correctly linked, headers are included and it creates a cross-compilation script for Elk Audio OS.

The user should open the `.jucer` file with the Projucer and save the project to create the build folder structure. After each change of configuration (e.g. renaming the project), the `.jucer` file should be opened with the Projucer app and saved. Users are expected to modify the template code that loads an inference model and executes inference depending on their needs and according to the documentation of each engine[6].

Notably, even if Elk Audio OS only handles headless plugins (i.e., without graphical user interface (GUI)) for optimized latency, it is not necessary to remove any GUI code from the plugin editor, as the graphic routines will simply not be called by the Sushi DAW.

2) Manual Creation: The steps needed for manually creating a JUCE project for a VST plugin that is compatible with Elk Audio OS are the following:

1) Creation of a JUCE project for an audio plugin;
2) Editing of project settings for compatibility with Elk Audio OS;
3) Addition of external library binaries and headers (e.g. TensorFlow Lite or ONNX Runtime);
4) Creation of a Linux exporter;
5) Creation of a cross-compilation script

The details of manual project creation steps are discussed in the up-to-date guide in the project repository.

B. Cross-compilation for Elk Audio OS

Deploying any plugin to Elk Audio OS (especially for Raspberry Pi and resource-constrained devices) will generally involve cross-compilation. Cross-compilation is a technique by which source code is compiled with a cross-compiler on a **host** computer, resulting in a binary file that is executable on a **target** computer with an architecture different from that of the host. This enables the compilation of binary executables for embedded devices from a different machine architecture which can take advantage of more powerful hardware. While native compilation on the Raspberry Pi is possible, it is discouraged as anything other than the last resort due to the constrained resources available, thus long computation times (i.e., tens of hours for most engines). For example, we successfully performed cross-compilation with an x86-64 Linux computer as the **host**, while the target is an ARM 64bit SBC (aarch64 architecture), running the Linux-based real-time Elk Audio OS. To cross-compile any program for a Raspberry Pi with Elk Audio OS, the Elk-PI SDK corresponding to the OS version of choice should be installed and used[7].

It is not necessary to follow the next step (i.e., *Dependencies Compilation*) to compile a project that depends only on the inference library of choice, as the pre-compiled binaries are included in the project repository (Section III-D).

1) Dependencies Compilation: To cross-compile a plugin, any eternal dependency needs to be compiled too. In the case of a simple plugin that integrates DL inference, the sole direct dependency will be the inference engine library (e.g., TensorFlow Lite or ONNX Runtime). Additionally, each direct dependency may have sub-dependencies that might need to be included for compilation (See the TensorFlow Lite template in the project's repository).

Cross-compilation of the dependencies is complex and not always possible. It can be hard to set up cross-compilation, especially for libraries that were not prepared for it, that are not well documented, or that use less known build systems. Libraries that rely on CMake[8] can be integrated reasonably well with Elk's toolchain. The main steps for cross-compilation of these libraries are the following:

1) Downloading and installing the Elk-PI SDK;

[6]https://www.tensorflow.org/lite/guide/inference#load_and_run_a_model_in_c, https://onnxruntime.ai/docs/get-started/with-cpp.html

[7]https://github.com/elk-audio/elkpi-sdk/releases

[8]https://cmake.org/

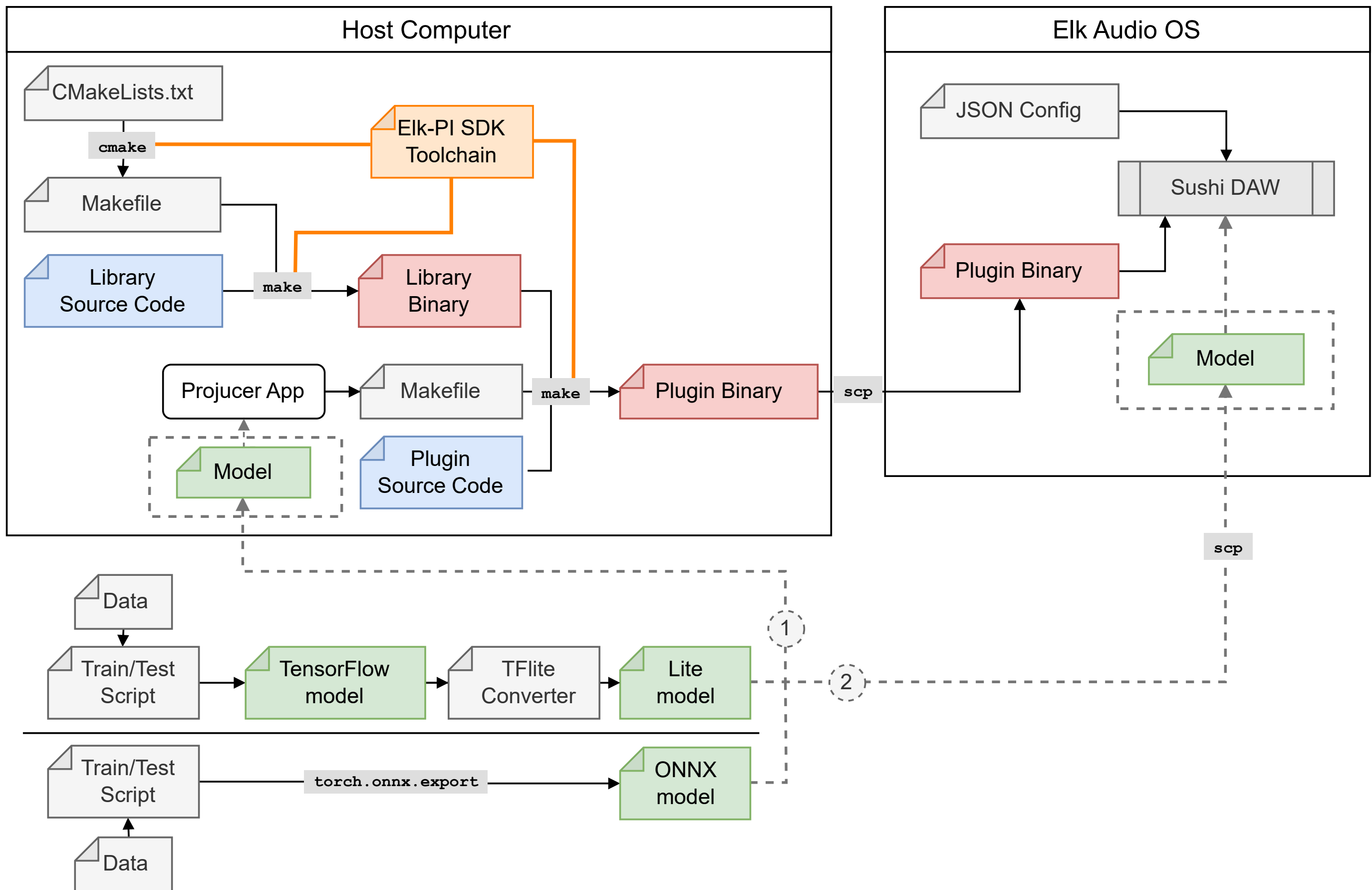

Fig. 1. Diagram depicting the process of deploying a DL model to an embedded device running Elk Audio OS. Plugin and dependency compilation happens on a Host computer (top left), where the Elk-PI toolchain allows to cross-compile the source code. Library binaries are linked during plugin compilation, which produces the binary file for a VSTplugin that can execute on the target device (right). The compiled plugin has to be moved to the embedded computer running Elk Audio OS, where Elk's DAW Sushi can be configured to load it into a new track and process audio in real-time. The lower part represents the training, testing, and model export phases required for either TensorFlow or frameworks that export to the ONNX format, such as Pytorch. Dashed arrows labeled as 1 and 2 represent two distinct options for model integration: in option 1 the DL model gets integrated as JUCE BinaryData into the plugin's binary, while for option 2 the model can be simply copied to the device. For the latter, the plugin code must load the model from a path relative to the target's folder structure.

2) Downloading the library source code for the desired version;
3) Creating a `build` folder;
4) Resetting the `LD_LIBRARY_PATH` variable and sourcing the Elk-PI SDK;
5) Executing CMake from the build directory: `cmake path/to/CMakeLists/dir/` ;
6) Compiling with `make` .

A working example can be found in the repository for this project (see Section III-D) for the TensorFlow Lite template. The example shows how the actual compilation procedure can deviate from the ideal process, due to peculiar characteristics of some sub-dependencies or bugs. In particular, for TensorFlow Lite 2.11.0 it is necessary to change the version of the dependency `FlatBuffers` from 2.0.6 to 2.0.8 and overwrite the `CMAKE_SYSTEM_PROCESSOR` variable. This build variable is set to `cortexa72` by Elk toolchain, but the `Abseil` dependency requires `aarch64` to avoid incorrect library linking. Both corrections were informed by comments in the Issues section of the Abseil and TensorFlow GitHub repositories, which were consulted by searching for specific error messages produced by failed compilation runs. A similar informed trial and error procedure can be followed for other libraries.

When cross-compilation cannot be set up, compilation can be performed natively on the Raspberry Pi, at the expense of high completion time. Users should follow the build instructions of the specific library for any Linux system. This may require compiling other sub-dependencies separately, in case they are not included in Elk Audio OS, or automatically fetched during compilation setup. This was the case for ONNX Runtime, whose compilation required a few tens of hours on the board, and for which the compiled binary is provided in the repository of this project.

Whenever a dependency compiles to a dynamic library (i.e. `*.so` files), the binary is needed both at linking time (on the host computer) and at execution time (on the board),

when the library is loaded dynamically. This means that the compiled `.so` binary for any dependency must be copied to the board, and placed in one of the system library paths (Run `echo $LD_LIBRARY_PATH` on the device to find dynamic loading paths). Alternatively, if placed in any other directory, the folder containing the binary should be appended after each reboot, either manually or automatically[9], to the `LD_LIBRARY_PATH` system variable as follows:

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/libpath/
```

On the contrary, static libraries (i.e., `*.a` files) are automatically included in the plugin binary and do not need to be copied to the board.

2) Plugins Compilation: Once the inference engine and other libraries required are compiled and properly added to the compilation exporter (see Projucer configuration in the project repository, Section III-D), the plugin can be compiled multiple times without re-compiling dependencies.

Assuming a correct Projucer setup, the steps for plugin compilations are the following:

1) Saving the project from the Projucer app, to create the build structure;
2) Opening the terminal in the `/build/linux-aarch64` folder;
3) Resetting `LD_LIBRARY_PATH`;
4) Sourcing the Elk-PI SDK;
5) Compiling with `make`, specifying `JUCE_HEADLESS_PLUGIN_CLIENT=1`;
6) For VST3 plugins, renaming the `PluginName.vst3/Contents/arm64-linux` folder to `aarch64-linux`.

Additionally, before the `make` command, the user can specify additional optimization flags such as the following:

```
export CXXFLAGS="-O3 -pipe -ffast-math -feliminate-
    unused-debug-types -funroll-loops"
```

The following is a simplified compilation script for any plugin project.

```
unset LD_LIBRARY_PATH

source /opt/elk/0.11.0/environment-setup-cortexa72-
    elk-linux

export CXXFLAGS="-O3 -pipe -ffast-math -feliminate-
    unused-debug-types -funroll-loops"

AR=aarch64-elk-linux-ar make \
-j$(nproc) CONFIG=Release\
CFLAGS="-DJUCE_HEADLESS_PLUGIN_CLIENT=1 -Wno-psabi"\
TARGET_ARCH="-mcpu=cortex-a72 -mtune=cortex-a72"
```

C. Elk Audio OS on the Raspberry

Elk Audio OS v0.11.0 is available as open source for the Raspberry Pi 4 board[10], while older versions support also the Raspberry Pi 3. More SBCs are supported under a commercial license. The OS image should be downloaded from the GitHub repository and flashed to a good-quality SD card. The Raspberry Pi must be coupled with an audio *"hat"* board supported by the OS, such as the HiFiBerry DAC+ ADC and HiFiBerry DAC+ ADC Pro boards.

Once an audio hat board is connected to the Raspberry Pi and the OS SD is inserted, the board should be powered on and the user can access the terminal either using a monitor connected via HDMI or with a remote Secure SHell (SSH) connection. Elk Audio OS does not have a GUI and requires the user to control it via terminal, or via network (e.g., using the Google Remote Procedure Calls (gRPC) or Open Sound Control (OSC) protocols). For remote access to the terminal, the board can be connected via an ethernet cable to either a network router or a computer directly. After the first successful access, the terminal can be used to connect the board to a Wi-Fi network if desired. The default hostname for Elk-Pi boards is `elk-pi.local` and it can be used to identify the board in a local network. To ensure that the board is connected, use the `ping` command as follows and wait for a positive reply:

```
ping elk-pi.local
```

The `arp -a` command on a Linux terminal can be useful to find the board IP address if the hostname is not reachable. Then, the SSH protocol can be used to access the terminal remotely. The `ssh` command is available on Linux, MacOS, and the latest Windows 10 and 11 terminals (PowerShell). For previous versions of Windows or PowerShell, SSH clients such as Putty[11] can be used to replace remote terminal and copy functions. The board can be accessed via the terminal with the following command:

```
ssh mind@elk-pi.local
```

The default password is `elk`.

Once a connection is made, files such as the compiled plugin, configuration files, and dynamic libraries can be copied to the board using `scp` [12] from the host computer:

```
scp -r /path/to/PluginName.vst3 mind@elk-pi.local:~/
scp libonnxruntime.so mind@elk-pi.local:~/
```

D. DAW configuration: Sushi

Once a VSTplugin is copied to the board, it can be hosted via Elk Audio OS's DAW Sushi. Similarly to other DAWs, Sushi allows the creation of multiple tracks, where each can have one or more audio and MIDI inputs and outputs. Each track can have a chain of plugins, which are loaded as a dynamic library at runtime. However, differently from most other DAWs, the GUI code of the hosted plugins is neither shown nor called by Sushi, and the audio processing callback is executed on a hard real-time Xenomai thread for low latency processing.

[9] The export line can be added to the `~/.bashrc` or `/etc/profile` text file for automatic execution.

[10] https://github.com/elk-audio/elk-pi/releases

[11] https://www.putty.org/

[12] https://linux.die.net/man/1/scp

For these reasons, executing the plugin prepared in the previous steps requires the configuration and execution of Sushi. Configurations are prepared in the form of JSON files. The following is a configuration file that will prompt Sushi to create a mono track with a single audio input and output, and to load the VST3 plugin "PluginName":

```
{
  "host_config":{ "samplerate":48000 },
  "tracks":[
    {
      "name":"main",
      "mode":"mono",
      "inputs":[
        {
          "engine_channel":1,
          "track_channel":0
        }
      ],
      "outputs":[
        {
          "engine_channel":1,
          "track_channel":1
        }
      ],
      "plugins":[
        {
          "uid":"PluginName",
          "path":"/path/to/vst/PluginName.vst3",
          "name":"arbitrary_plugin_name",
          "type":"vst3x"
        }
      ]
    }
  ],
  "midi":{
    "cc_mappings":[]
  }
}
```

Alternatively, the configuration for a stereo track should use the following `mode`, `inputs`, and `outputs` field values:

```
"mode":"stereo",
"inputs":[ {
    "engine_bus":0,
    "track_bus":0
  } ],
"outputs":[ {
    "engine_bus":0,
    "track_bus":0
  } ]
```

More configuration file examples are provided in the project repository (Section III-D).

Once a configuration file is prepared, Sushi must be executed via the terminal by providing the audio driver type and the configuration file path:

```
sushi -r -c "/path/to/config.json"
```

where the `-r` option specifies to use Elk's RASPA low-latency front-end. Users can add more options such as `-multicore-processing=2` to allow Sushi to use more cores. Moreover, "`&`" can be added at the end of the command to start Sushi in the background and keep using the terminal. The background execution can later be stopped with `pkill sushi`. If Sushi fails to start, errors in the configuration or plugin can be identified in the log file `/tmp/sushi.log`. The next section will provide a brief overview of diagnostic tools.

E. Diagnostic tools

By default, Sushi logs events and errors at run-time to the file `/tmp/sushi.log`. The logging level can be changed using the `-l` flag. The log can show errors such as having a plugin uid that does not match the actual VST3 unique-id or having an incorrect configuration format. Additionally, the `-timing-statistics` flag will prompt Sushi to log the fraction of CPU time available used for processing. This is particularly relevant for DL models meant to be executed in real-time, with one or more inference operations per audio block, as this will show if the plugin is reaching the allotted time budget for each call or even surpassing it. The next section will present a brief overview of different execution modes, including a few considerations on how to deal with real-time tradeoffs and large models.

Notably, Elk Audio OS provides tools to diagnose real-time execution issues since any part of the code that is meant to execute in real-time must respect specific real-time safe programming rules [21]. These rules include not allocating memory dynamically from the audio thread, not using locking mechanisms for concurrent memory access, or not waiting on lower-priority threads (e.g., querying system timers). In general, these rules can be summarized as "do not perform (on the audio thread) operations that have unbounded or unknown completion time". The inference libraries included were tested [15] and deemed real-time safe, but user code that is added to a plugin should respect the same rules.

Being Elk Audio OS a system based on a dual kernel, with audio processing running on Xenomai real-time threads, it can be straightforward to verify whether user code is violating real-time safety and troubleshoot the problem. In fact, in Elk Audio OS, non-safe operations in the audio callback will result in a **mode switch**, i.e., the system will give control back to the regular Linux kernel to handle the unsafe operations and switch back to the Xenomai kernel. Up to version 0.11.0 of Elk Audio OS, mode switches can be monitored by looking regularly at the `MSW` column in the `/proc/xenomai/sched/stat` file. The following command will update its output every three seconds:

```
watch -n 3 cat /proc/xenomai/sched/stat
```

Version 1.0.0, which was released while this procedure was being finalized, requires using the `evl ps -s` command instead and looking at the `ISW` counter.

The number of mode switches must remain stable after the plugin startup, with one or two mode switches being allowed at startup if not audible as artifacts. In the case of repeated mode switches, their origin can be traced back to the original source code using the gdb debugger as follows:

1) Running the GNU Debugger `gdb` on the sushi executable for the current block size (default 64):

```
gdb sushi_b64
```

2) Setting `gdb` to stop whenever the `SIGXCPU` signal is sent by the program:

```
catch signal SIGXCPU
```

3) Running Sushi with the debug-mode-sw flag:

```
r -r --debug-mode-sw -c config.json
```

Finally, the Elk Audio forum[13] can be a useful source to find support and solve similar problems.

V. Considerations on real-time inference

Audio-related applications of machine learning and DL can be separated into real-time and non-real-time (or offline) cases [22]. This distinction is used especially for audio tasks, and the terms must not be confused with the online and offline terms used to refer to single-sample and batch learning in DL research.

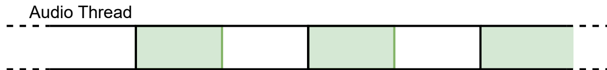

Fig. 2. Representation of repeated calls to the audio processing routine on the real-time thread. Black vertical lines represent the boundaries of the time-budget slots available to process each incoming audio buffer. In this case, green boxes represent computations on the input audio buffers, which are safely performed within the time budget available (i.e., before the current output buffer is consumed and the next input buffer is read).

In particular, real-time audio analysis and processing algorithms must be designed to continuously process audio data and produce a result (i.e., audio or other data) before pre-defined problem-specific deadlines. An example is a *real-time audio effect*: in this case, audio must be processed faster than it is consumed as output, to avoid incurring in buffer underruns [6]. Since digital audio is, in most cases, buffered into short audio blocks and processed at a set rate, this means that an input buffer of X samples should be processed and copied to the output in less than $\frac{X}{samplerate}$ seconds. Figure 2 depicts a situation in which computations are safely performed within the allotted time budget for each call to the audio processing routine. This is the case of the examples provided in the project's repository, where inference of a small neural network that models audio saturation is executed for each sample in the audio buffer. The provided examples run with the default of 64 samples per block and a *samplerate* of 48 kHz, and report using about 15% (on the Raspberry Pi 4) of the 1.33 ms available for each processing routine call (i.e., $\frac{64}{48{,}000}$).

However, the execution time of a model depends on the CPU used (or acceleration hardware like GPUs, if available) and the optimizations performed by the inference engine (see [15]). For this reason, a model that executes well within the allotted time budget on a laptop or desktop computer could struggle on a resource-constrained device such as the Raspberry Pi. Figure 3 depicts such a situation, where a set of computations fails to complete before the output buffers need to be converted to analog, and the new input buffer is read. This results in audible artifacts appearing in the output signal due to the corrupted buffers.

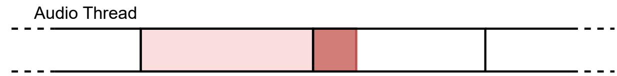

Fig. 3. Representation of a set of computations performed on the real-time audio thread, which fail to safely complete within the allotted time budget. The darker area represents the overlap with the subsequent call to the audio processing routine. If this happens, audible artifacts are produced in the output signal.

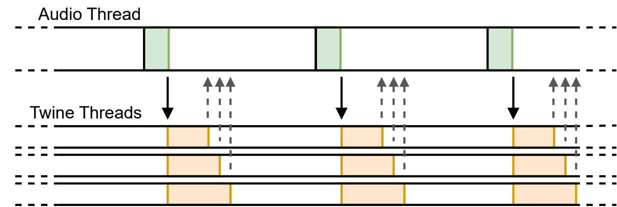

Fig. 4. Representation of an audio system where a costly series of operations has to be conducted at the rate of the audio callback, but it **can** be broken up into separate and independent tasks. In this case, different independent sets of computations can be performed in parallel on separate threads. Elk provides the TWINE library for real-time threads and worker pools[15].

In the case of a DL model that **must execute for each audio routine call** (e.g., effect modeling), most of the solutions will involve finding a tradeoff between latency and quality of the results. Some of these potential solutions, with which a DL developer is likely to often cross, are the following:

1) **Increasing the time budget**. This can be achieved by increasing the audio-block size or reducing the sample rate. These choices will inevitably increase the latency between input and output, but the time available for processing is increased. While increasing audio-block size means having more samples to process for each call, it means also reducing function call overhead, and processors can be more efficient by performing more operations at once.
2) **Optimizing on the model**. A first possibility is to train smaller models for the same task, which could mean having to find a compromise between size and "result quality" (e.g., error or accuracy). Transfer learning and knowledge distillation are techniques that can help retain a satisfactory quality of the results while using small models. Alternatively, some of the possibilities that do not affect model design are *quantization* and *pruning*. Quantization involves reducing the resolution of the

[13] https://forum.elk.audio/

[15] https://github.com/elk-audio/twine

network weights (e.g., from 32-bit float values to 8-bit integers). Pruning consists of progressively setting weights of the network to zero, which means, in an inference engine that supports sparse execution, skipping some of the multiplications during inference. All the alternatives above may reduce accuracy or increase test error in the results.

3) **Parallel execution**. Finally, in the seldom situations where the set of computations that take longer than the time budget can be broken down into *more manageable* and *independent* sets, these can be assigned to multiple real-time threads. In the quad-core (4 cores) Cortex-A72 processor of the Raspberry Pi4, more than one thread can be executed at the same time. However, this is only possible if the tasks to run in parallel are completely independent of the results of each other. This situation is depicted in Figure 4.

However, not all real-time audio analysis or classification systems need to run entirely for each call of the audio processing routine. This is the case with event-based systems, where deep signal analysis only needs to be performed when an event happens. An example can be a note-based real-time guitar technique classifier [8] where the prediction model is only executed upon onset detection. In the very likely case that events are expected to happen less frequently than processing routine calls (e.g., less frequently than one every 1.33ms for 64 sample blocks and 48 kHz sample rate), the only part of the system that needs to execute for each block is the detection stage (e.g., fast detection of onsets with DSP methods). In this case, the more in-depth analysis, which can involve deep inference, can be executed solely when triggered by detection, and take more time for completion, depending on the minimum inter-event time allowed. In this situation, however, it is necessary to offload the classification to a high-priority thread outside of the hard-real time kernel, to allow inference to take longer than a block without affecting the audio output. If needed, results can be moved to the real-time thread after inference has been completed (see Figure 5). In this case, it is of extreme importance to move input data and results without using locking data structures or unsafe operations. This is the case of the expressive guitar technique classifier presented in [8].

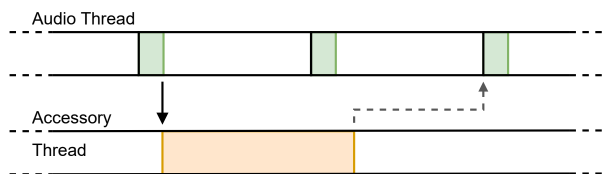

Fig. 5. Representation of a digital audio system with a more relaxed real-time constraint than those of Fig. 2 (i.e., not audio-rate deadlines) where some costly computations must be performed less frequently than calls to the audio callback and can be conducted on a separate thread. This can be the case of event-based DL inference [15], where signal analysis is performed only when events are detected. In these cases, event detection is a less costly operation performed for each audio block, while the heavier analysis is allowed to take more than the time budget for a single audio callback call.

VI. Conclusions

In this paper, we presented a procedure for deploying real-time DL inference on an embedded computer with Elk Audio OS. The procedure covers the steps from the creation of a compatible code project to the execution and diagnostic of a VSTplugin on a Raspberry Pi. Furthermore, we discussed different approaches to the real-time execution of deep learning inference on embedded devices and presented the alternatives that can be followed for larger neural network models. Along with this paper, we provided an online repository with a detailed and up-to-date guide, code *templates*, working *examples*, and *library binaries* for the two inference engines supported. The code repository serves the purpose of helping the reader to deploy their models but also to provide updates in case the process is subject to change with future versions of Elk Audio OS and the inference engines. This work enables developers and machine learning engineers to start executing inference of audio deep learning models on small embedded computers. A limitation of this study is that library cross-compilation can follow a very different process for other libraries, while this study focused mostly on providing a procedure for the TensorFlow Lite and ONNX Runtime libraries. This excludes some of the processing libraries that a developer could need for music information retrieval or applications based on audio spectrograms. Lastly, some of the details of the deployment procedure could become obsolete with updates to Elk Audio OS and inference engines. For this reason, this paper presented a general overview of the procedure, while a more detailed and updated guide is available in the project's repository. Furthermore, we believe that the procedure outlines a coherent process that will remain similar at its core. Finally, the accompanying code repository will be a useful resource as systems and libraries progress and change. The authors hope that the content of this study can contribute to the development of SMIs and associated services for musicians.

References

[1] J. Engel, L. H. Hantrakul, C. Gu, and A. Roberts, "DDSP: Differentiable digital signal processing," in *International Conference on Learning Representations*, 2020.

[2] A. Caillon and P. Esling, "RAVE: A variational autoencoder for fast and high-quality neural audio synthesis," *CoRR*, vol. abs/2111.05011, 2021.

[3] L. Turchet and C. Fischione, "Elk Audio OS: an open source operating system for the Internet of Musical Things," *ACM Transactions on the Internet of Things*, vol. 2, no. 2, pp. 1–18, 2021.

[4] A. McPherson and V. Zappi, "An environment for submillisecond-latency audio and sensor processing on beaglebone black," in *Audio Engineering Society Convention 138*. Audio Engineering Society, 2015.

[5] E. Meneses, J. Wang, S. Freire, and M. M. Wanderley, "A comparison of open-source linux frameworks for an augmented musical instrument implementation," in *Proceedings of the Conference on New Interfaces for Musical Expression*, 2019, pp. 222–227.

[6] L. Vignati, S. Zambon, and L. Turchet, "A comparison of real-time Linux-based architectures for embedded musical applications," *Journal of the Audio Engineering Society*, vol. 70, no. 1/2, pp. 83–93, 2022.

[7] A. Wright, E.-P. Damskägg, V. Välimäki *et al.*, "Real-time black-box modelling with recurrent neural networks," in *22nd international conference on digital audio effects (DAFx-19)*, 2019, pp. 1–8.

[8] D. Stefani and L. Turchet, "On the Challenges of Embedded Real-Time Music Information Retrieval," in *Proceedings of the 25-th Int. Conf. on Digital Audio Effects (DAFx20in22)*, vol. 3, Sept. 2022, pp. 177–184.

[9] S. Böck and M. Schedl, "Enhanced beat tracking with context-aware neural networks," in *Proc. 14th Int. Conf. on Digital Audio Effects (DAFx-11)*, 2011, pp. 135–139.

[10] I. Franco and M. Wanderley, "Prynth: A framework for self-contained digital music instruments," in *International Symposium on Computer Music Multidisciplinary Research*. Springer, 2016, pp. 357–370.

[11] E. Berdahl and W. Ju, "Satellite CCRMA: A musical interaction and sound synthesis platform," in *Proceedings of the Conference on New Interfaces for Musical Expression*, 2011, pp. 173–178.

[12] L. Turchet, "Smart Musical Instruments: vision, design principles, and future directions," *IEEE Access*, vol. 7, pp. 8944–8963, 2019.

[13] L. Turchet, C. Fischione, G. Essl, D. Keller, and M. Barthet, "Internet of Musical Things: Vision and Challenges," *IEEE Access*, vol. 6, pp. 61 994–62 017, 2018.

[14] T. Pelinski, R. Diaz, A. L. B. Temprano, and A. McPherson, "Pipeline for recording datasets and running neural networks on the bela embedded hardware platform," in *Proceedings of the International Conference on New Interfaces for Musical Expression*, 2023.

[15] D. Stefani, S. Peroni, and L. Turchet, "A comparison of deep learning inference engines for embedded real-time audio classification," in *Proceedings of the Digital Audio Effects Conference*, 2022.

[16] K. Bloemer, "GuitarML-NeuralPi," https://github.com/GuitarML/NeuralPi, 2021.

[17] T. Pelinski, V. Shepardson, S. Symons, F. S. Caspe, A. L. Benito Temprano, J. Armitage, C. Kiefer, R. Fiebrink, T. Magnusson, and A. McPherson, "Embedded AI for NIME: Challenges and Opportunities," *International Conference on New Interfaces for Musical Expression*, jun 22 2022.

[18] A. McPherson and V. Zappi, "An environment for Submillisecond-Latency audio and sensor processing on BeagleBone black," in *Audio Engineering Society Convention 138*. Audio Engineering Society, 2015.

[19] M. Comunità, C. J. Steinmetz, H. Phan, and J. D. Reiss, "Modelling black-box audio effects with time-varying feature modulation," in *ICASSP 2023 - 2023 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2023, pp. 1–5.

[20] A. Wright and V. Välimäki, "Neural modeling of phaser and flanging effects," *Journal of the Audio Engineering Society*, vol. 69, no. 7/8, pp. 517–529, 2021.

[21] R. Bencina, "Interfacing real-time audio and file i/o," in *Proc. of the Australasian Computer Music Conference (ACMC)*, 2014, pp. 21–28.

[22] W. Brent, "A perceptually based onset detector for real-time and offline audio parsing," in *Proceedings of the 2011 International Computer Music Conference, ICMC 2011, Huddersfield, UK, July 31 - August 5, 2011*. Michigan Publishing, 2011.