Pipeline for recording datasets and running neural networks on the Bela embedded hardware platform

Teresa Pelinski*
Centre for Digital Music
Queen Mary University of London, UK
t.pelinskiramos@qmul.ac.uk

Rodrigo Diaz*
Centre for Digital Music
Queen Mary University of London, UK
r.diazfernandez@qmul.ac.uk

Adán L. Benito Temprano
Centre for Digital Music
Queen Mary University of London, UK
a.benitotemprano@qmul.ac.uk

Andrew McPherson
Dyson School of Design Engineering
Imperial College London, UK
andrew.mcpherson@imperial.ac.uk

ABSTRACT

Deploying deep learning models on embedded devices is an arduous task: oftentimes, there exist no platform-specific instructions, and compilation times can be considerably large due to the limited computational resources available on-device. Moreover, many music-making applications demand real-time inference. Embedded hardware platforms for audio, such as Bela, offer an entry point for beginners into physical audio computing; however, the need for cross-compilation environments and low-level software development tools for deploying embedded deep learning models imposes high entry barriers on non-expert users.

We present a pipeline for deploying neural networks in the Bela embedded hardware platform. In our pipeline, we include a tool to record a multichannel dataset of sensor signals. Additionally, we provide a dockerised cross-compilation environment for faster compilation. With this pipeline, we aim to provide a template for programmers and makers to prototype and experiment with neural networks for real-time embedded musical applications.

Author Keywords

Embedded AI, Real-time, Deep Learning, Pipeline, Bela

CCS Concepts

• **Computer systems organization** → **Embedded software;** • **Computing methodologies** → *Artificial intelligence;* • **Applied computing** → *Sound and music computing;*

*Equal contribution

NIME'23, 31 May — 2 June, 2023, Mexico City, Mexico.

1. INTRODUCTION

There are many instruments in the NIME community based on embedded systems such as single-board computers (e.g. Bela, Raspberry Pi) or microcontrollers (e.g. Teensy, Arduino) [31, 1, 37, 21]. Plenty of these platforms provide open-source APIs and IDEs that abstract, among others, the complexities of compilation or interfacing with peripherals, which along with supportive online learning communities [15], well-documented code bases and tutorials, make them a valuable teaching resource for music and audio programming courses [18].

One of the most attractive features of these devices to instrument designers is their self-containedness. Embedded hardware-based instruments can work off-the-shelf and are less susceptible to issues caused by system updates, which may result in software and hardware compatibility problems when using general-purpose computers [25]. In general, they require less maintenance than laptop-based instruments [1], which, in turn, is a desirable feature in terms of the instrument's longevity [24]. Their small size also allows for integrating them into the instrument's body.

Many of these platforms provide APIs that simplify their usage. These APIs are typically written in C++, but many have incorporated other audio programming languages. In this context, Morreale et al. [25] introduce the notion of "pluggable communities", which refers to how disparate communities establish a connection when users are empowered to map their knowledge into a new domain. Incidentally, there is a growing interest in deep learning techniques in NIME [16, 32, 2, 26]. However, deploying neural models into embedded platforms is an arduous task: oftentimes, no platform-specific instructions exist, and the large building times (due to the limited computational resources) complicate debugging. Frequently, the APIs provided by these platforms do not integrate deep learning inference engines, and the programmer needs lower-level software development skills to compile them. In addition, the deep learning models need to be very computationally efficient to run in real-time. While there exist instances of porting deep learning frameworks into embedded systems in the context of NIME [17, 8, 30], considerable effort is still needed to make these accessible for non-expert makers.

This paper presents a pipeline to run neural networks

in real-time in the Bela[1] hardware platform [20], a platform extensively used in NIME [37, 23, 25, 12]. Given the stringent real-time requirements of musical instruments and interfaces, we have chosen Bela due to its low input-output latency [19]. As part of our pipeline, we include a tool to record a multichannel dataset of sensor signals. Additionally, we provide a dockerised cross-compilation environment for faster compilation. With this pipeline, we aim to streamline the prototyping and experimentation with simple deep learning models for real-time embedded audio applications, and to contribute towards bridging the deep learning for audio and the embedded hardware communities in NIME.

2. BACKGROUND

The practice of running neural networks in embedded devices is referred to by different names across various domains, such as *edge AI* in networked devices and cloud applications, *AIoT* in the context of artificial intelligence integrated into everyday devices, and *tinyML* for devices that operate on a few milliwatts of power.

In this paper, we continue with the terminology adopted on the NIME 2022 workshop *Embedded AI for NIME: Challenges and Opportunities* [26] and use the term "embedded AI" with some nuances: first, we focus on embedded platforms that have low-power and low-resourced CPUs, but that allow for real-time sensor or audio signal processing; and secondly, we concentrate on deep learning[2] rather than other techniques that fall under the umbrella of artificial intelligence.

2.1 Embedded Inference

Inferencing with deep learning models for real-time audio is a computationally demanding task, even for conventional computers. For instance, to generate audio at 44.1kHz, the model should be able to generate at least 44100 samples every second. Since they are typically based on matrix operations, neural networks are usually trained and run in Graphics Processing Units (GPUs). There exist embedded computers with integrated GPUs[3]. However, the communication overhead of interfacing between CPU and GPU can be challenging for real-time audio applications, as well as the balance between meeting the sampling rate and staying within limits for latency and jitter [28]. Recent advancements have been made in this direction in the context of real-time audio effects [29]. Non-GPU alternatives for deep learning exist: Kiefer [14] explores the usage of field-programmable gate arrays (FPGAs), which can run large parallel processes at very high frequencies, though they are complex to manipulate when compared to commercial embedded computers such as Bela or Raspberry Pi [14].

A perhaps more straightforward alternative is to directly run the networks on the CPU. Mittal et al. [22] survey the field of deep learning in CPUs, and find that CPUs can outperform GPUs for large models and batch sizes due to their greater available memory [36]. The memory advantage is also beneficial for networks where the number of computations rises with sequence lengths but parallelisation is complex due to sequential dependency (e.g. RNNs) [38]. Furthermore, in embedded systems, the CPU can be a better suited choice for inference than the GPU since continuous inference in the GPU can lead to a high energy consumption [38]. There exist hardware accelerators that can be attached to embedded computers to speed up training and inference[4], however, the software must be able to leverage these optimisations [13].

An alternative approach to hardware optimisation is to reduce the network size through compression techniques such as pruning, quantisation or knowledge distillation, among other strategies. Pruning techniques reduce the model's size by discarding a substantial amount of weights in a neural network without significantly decreasing its accuracy. Alternatively, quantisation strategies quantise the weights and activations of a network to a lower-precision datatype. Lastly, in knowledge distillation approaches, a small student model mimics a larger teacher model. In the case of pruning, the ratio of pruned parameters to the number of parameters originally present in the network (sparsity) can be used as a proxy to estimate the performance of a network in a target platform. However, calculating the exact time a model will need to run, which is relevant when prototyping for real-time applications, is complex since many other factors influence the efficiency of a model, such as the enabled hardware optimisations, memory management and latency inherent to certain data operations (e.g. FFTs).

Finally, hard real-time systems are usually programmed with compiled languages such as C or C++, which favour deterministic code. Some deep learning frameworks provide C++ distributions (e.g. Libtorch[5] for PyTorch and TFLite[6] for TensorFlow), however, they tend to rely on resizable data structures that can allocate memory dynamically (e.g. C++'s `std::vector.resize()`), which makes them potentially inappropriate for real-time implementations [7]. Stefani et al. [30] run a comparison of audio classification performance of the TFlite, Libtorch, ONNX Runtime[7] and RTNeural [7] inference engines. The authors find that these frameworks can be used safely for hard real-time applications. Alternatively, the IREE (Intermediate Representation Execution Environment) is a compiler and runtime stack based on the MLIR (Multi-Level Intermediate Representation) compiler infrastructure [34], which converts the models into an intermediate representation that allows optimising the model for the target platform hardware. A recent Google Summer of Code project [27] developed support for compiling, running and benchmarking IREE projects on Bela.

2.2 Existing Tools

Whilst these technical contributions represent significant steps toward embedding deep learning models for musical applications, there is still a need for tools that streamline these achievements to facilitate experimentation and prototyping. In their Cognitive Dimensions of Notations Framework, Blackwell and Green [4] include the "viscos-

[1] `https://bela.io/`

[2] Performing deep learning inference in a resource-constrained device might seem counter intuitive: the "deep" in "deep learning" implies that the model has a considerable number of layers and parameters, and consequently, that it will need significant computational resources to run. A more accurate term would be "shallow" learning, however, we use the term deep learning due to its extended adoption.

[3] Such as the Nvidia Jetson Nano (`https://developer.nvidia.com/embedded/jetson-modules`) or the Coral Dev Board Mini (`https://coral.ai/products/dev-board-mini/`).

[4] Such as Google Coral (`https://coral.ai/products/accelerator`) or the Intel Neural Compute Stick (`https://www.intel.com/content/www/us/en/developer/articles/tool/neural-compute-stick.html`)

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

[6] `https://www.tensorflow.org/lite`

[7] `https://onnxruntime.ai/`

ity" dimension, which refers to the resistance of a system to change. In this sense, the process of compiling inference engines and prototyping with deep learning models for embedded platforms is viscous, since it often takes a considerable amount of attempts and it involves a variety of programming languages, frameworks and devices. Although Blackwell and Green use the term viscosity to refer to the number of actions needed to accomplish a goal, here we might extend it to the time it takes for a system to change. A pipeline for compiling and running deep learning models on Bela reduces the viscosity of the task by providing a streamlined set of steps and templates, including a cross-compilation environment that reduces the compilation times.

The pipeline is intended for non-expert programmers with sufficient skills to follow a tutorial involving interacting with the target platform trough the CLI and coding in C++ and python. It aims to encourage prototyping and experimentation through code rather than interfaces. There exist a few laptop- and interface-based tools (typically in Pd and Max/MSP, or as VST plugins) that allow applying deep learning and machine learning models in real-time to input audio or sensor signals, such as FluCoMa [33], the nn_tilde[8] and torchplugins[9] Max/MSP and Pd externals, or the Neutone VST plugin[10]; yet to the authors' knowledge, none specifically offers an embedded implementation. However, there are a couple of examples of audio-based embedded models: the real-time neural audio synthesis model RAVE [6], that has been embedded[11] into a Raspberry Pi and an Nvidia Jetson Nano, and the Neurorack [8], a Eurorack module running a neural source-filter model, also embedded on the Nvidia Jetson Nano.

Meanwhile, music-making deep learning models involving the performer's body and gesture have received much less attention, although the 2022 workshop *Embodied Perspectives on Musical AI* held at the University of Oslo in Norway [9] manifested its increasing relevance. There exist many laptop- and interface-based machine learning toolkits for gesture classification and mapping, such as GIMLeT (Max/MSP) [35], ml.lib (Max/MSP, Pd) [5], XMM (C++, python and Max/MSP) [11] or MnM (Max/MSP) [3], however, there are only a few embedded approaches related to deep learning and gesture in the context of NIME, such as Martin et al.'s work [17], where a recurrent neural network runs in a Raspberry Pi to predict the performer's control gestures. Whilst some of these projects provide very detailed instructions for deploying the embedded models, we aim to provide a more general and model-agnostic pipeline for practitioners to apply to their own projects.

3. PIPELINE

We present a pipeline to record a dataset of signals, export a light model trained on those signals, and run the model in real-time on Bela. An overview of the pipeline is given in Figure 1. It should be noted the pipeline relies on a host machine for dataset processing, training and exporting a light model, and cross-compiling the inference code. The pipeline's code and instructions for each step are available in Github:

`https://github.com/pelinski/bela-dl-pipeline`

[8] `https://github.com/acids-ircam/nn_tilde`
[9] `https://github.com/rodrigodzf/torchplugins`
[10] `https://neutone.space/`
[11] `https://youtu.be/jAIRf4nGgYI`

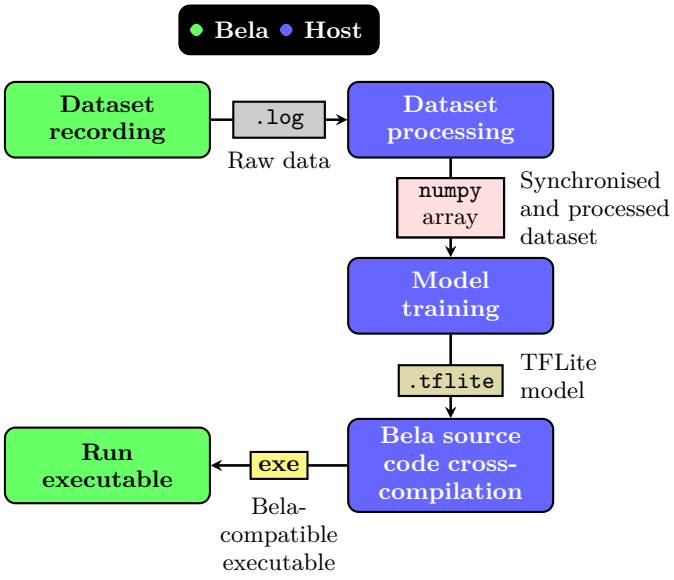

Figure 1: Pipeline for recording datasets and running neural networks on Bela. Green and blue nodes indicate, respectively, that the code runs in Bela or in the host machine.

In the pipeline's first stage, a dataset of sensor signals can be recorded in a single or in multiple Bela boards. The raw data files are then transferred into the host machine for the data processing step, where the signals recorded on different boards are aligned sample-wise, and the dataset is converted into a numpy[12] array, which can be loaded into deep learning python frameworks such as Tensorflow or PyTorch. After the model is trained on the dataset, it must be exported as a `.tflite` file, since the Bela inference code is based on the TFLite C++ library. Finally, the inference code is cross-compiled and transferred to Bela, where it can be executed. Below, we describe each step in detail.

3.1 Recording and processing datasets

In the first stage of the pipeline, a dataset from analog (e.g. piezo sensors, microphone signals) or digital (e.g. distance sensors, rotary encoders) inputs is recorded on Bela. Our code, based on the `BelaParallelComm` library, allows recording datasets simultaneously on various Bela boards, which enables capturing more channels than those available on a single Bela board (i.e. 8 analog and 16 digital inputs). As illustrated in Figure 2, one of the Bela boards acts as a transmitter (TX), whilst the other boards act as receivers (RXs). Every number of frames, the TX Bela sends a digital bit to the RXs Bela boards. Besides logging the sensor signals' values, all Bela boards log the frame at which the bit was sent (in the case of the TX), or received (in the case of the RXs).

In the second stage of the pipeline, these log files are transferred into the host machine, where the `DataSyncer` library synchronises the signals at sample level and returns a single numpy matrix in which every row corresponds to a sensor channel and every column corresponds to a timestep. This matrix can later be loaded into the user's preferred python deep learning framework. If the dataset is recorded only on one Bela, the `DataSyncer` library will simply convert it into a numpy matrix.

Recording datasets on multiple Bela boards is valuable

[12] `https://numpy.org/`

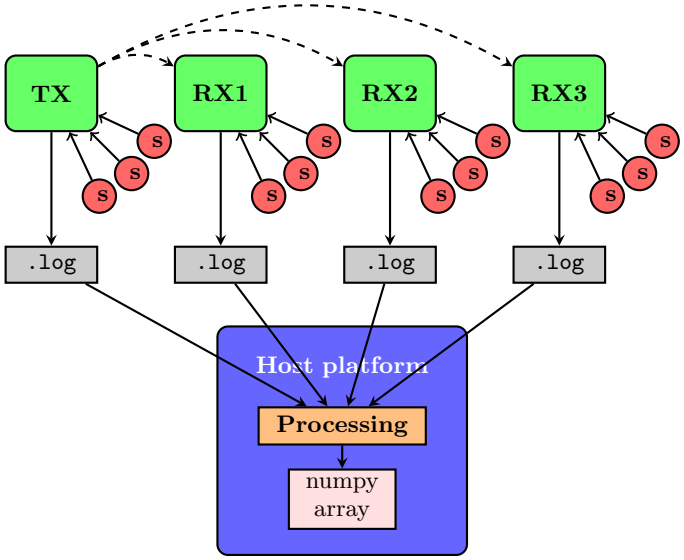

Figure 2: Example of recording a dataset on four different Bela boards, with three sensors (red circles) connected to each. The dashed line represents the clock signal (a digital bit) sent from the transmitter Bela (TX) to the receiver Bela boards (RX1, RX2, RX3). The generated log files are processed in the host platform.

for analysis tasks (e.g. to analyse the timing variations of various performers simultaneously playing the same piece on a piezo-based instrument) or if the Bela boards are used as an interface to stream sensor data to a model running in the host machine. Inferencing with a deep learning model on Bela with inputs simultaneously proceeding from various boards is a complex process involving real-time communication within Bela boards, which would require sufficient bandwidth to enable the transmission of multichannel data to the Bela board executing the model, or alternatively, distributing the network across boards.

3.2 Training and exporting a model

Given the limited computational capabilities of Bela, in the third stage of the pipeline, the model training is carried out on the laptop or, alternatively, on a computing cluster. In the provided example code, we use PyTorch, a widely extended python deep-learning framework. However, Stefani et al. [30] find that the TFLite inference engine is faster than PyTorch's C++ inference engine. For this reason, we wrote the model in PyTorch and then exported it to Tensorflow Lite using the `TinyNeuralNetwork`[13] library. The process of converting PyTorch models to the TFLite format may not always be straightforward, particularly when PyTorch primitives lack a direct mapping to an equivalent TFLite primitive. In such cases, it may be necessary to prototype the model without using these primitives (i.e. explicitly writing the model equations). Alternatively, the model may be prototyped using Keras o Tensorflow, since these frameworks can natively export a TFLite-compatible model.

In the provided example, we train an LSTM network that receives two sensor signals (S1 and S2) and predicts the future values for the sensor S1 signal. This is illustrated in Figure 3. A model that predicts the future behaviour of sensor signals has interesting options for performance, such

[13] `https://github.com/alibaba/TinyNeuralNetwork`

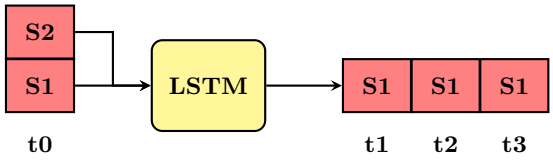

Figure 3: Example model. The model receives a window of the sensor channels S1 and S2 and predicts the following three windows of sensor S1.

as instruments that speculate about what the performer might do, or instruments that sonify the prediction error. In our setup, sensor S1 was an accelerometer attached to a pendulum, and sensor S2 was a piezo sensor attached to a drumstick. The dataset consists of recordings of the sensor signals when hitting the pendulum (S1) with the drumstick (S2). A window of 32 samples of the two sensor channels is passed into the LSTM, which predicts the next three windows of the S1 signal (in total, 96 samples). The model, which runs in real-time in Bela, had 7096 parameters and a size of 1MB when exported as `.tflite`.

3.3 Cross-compiling

Building a complex library or program can take considerable time on an embedded device, which makes prototyping and debugging tedious. Cross-compilation reduces building times by compiling the program in a host platform (e.g. a laptop) with greater computational resources. To facilitate cross-compilation for Bela, in the fourth stage of the pipeline, we provide a dockerised container to encapsulate the cross-compiler. The workflow is illustrated in Figure 4. This enables compiling Bela code on any host that can run Docker[14] (i.e. Linux, Windows, MacOS). Docker is a tool to package software and its dependencies in a container, which allows running that software across platforms without needing to install OS-specific dependencies each time. Inside the Docker container, we use CMake[15] for cross-compiling. CMake is an open-source tool commonly used for building C++ projects, in which the instructions for compilation are passed in a `CMakeLists.txt` file. CMake also allows cross-compiling code by using a "Toolchain", a file that describes the target platform. The toolchain file for Bela is included in the provided Docker container.

To cross-compile Bela code using the Tensorflow library, the library must be previously built inside the container. Once the Tensorflow library has been compiled, its location can be passed to the compiler using CMake. In our repository, we provide the pre-compiled libraries for Bela (and instructions to cross-compile it), as well as example code and the `CMakeLists.txt` file needed to compile it.

3.4 Inference in real-time and multi-threaded processing

In this section, we discuss the real-time coding practices that should be followed in the inference code for the deep learning models to run in real-time (fifth stage of the pipeline). These practices apply to any real-time system, but they are particularly relevant here since neural networks' inference is a computationally expensive operation. Template code for running inference in Bela is included in the provided repository. It should be noted that these prac-

[14] `https://www.docker.com/`
[15] `https://cmake.org/`

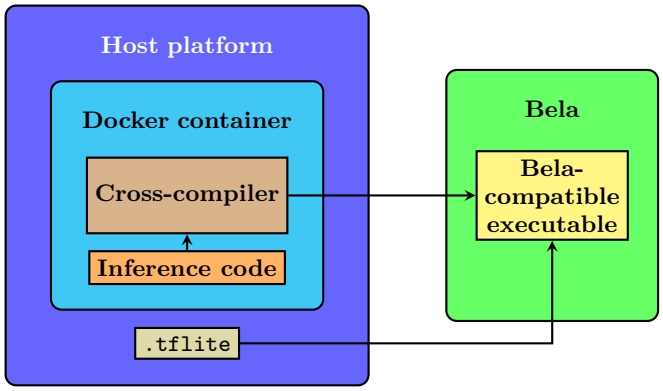

Figure 4: Cross-compilation for Bela. The inference source code is compiled in a Docker container in the host machine (e.g. a laptop). Both the Bela-compatible executable and the `.tflite` model are then transferred to the Bela.

tices will enable the model to run in real-time only if the model is light enough to run using the device's CPU. For instance, Esling et al. [10] evaluate the theoretical embeddability of deep learning models in terms of compression and complexity according to three metrics: floating point operations, model disk size and number of read-write operations. Further work is needed to determine these parameters' thresholds on Bela.

3.4.1 *Pre-allocating memory*

Allocating memory is a non-deterministic process, which means that its duration can not be known in advance. In order to guarantee our code can run in real-time, we need to ensure that every part of the code has a bounded execution time, and that it will meet our real-time deadlines. Therefore, memory needs to be allocated before the audio processing starts. In the Bela API, memory should be allocated in the function `setup()`, which runs at the beginning of a program's execution and before the audio processing starts.

3.4.2 *Multi-threading to avoid underruns*

The audio callback (in the Bela API, the `render()` function) is a function in the audio thread that is called for each block of samples and does the audio processing. In Figure 5, this is indicated by the red boxes. For simplicity, we assume that a block of samples processing always takes the same amount of time (a fourth of a block) and that the inference task only takes a block and a fourth[16]. If the inference task is called from the audio thread it will not be able to finish on a block's time, which will cause an underrun. Increasing the block size would ensure that the inference computation finishes on time, however, large block sizes add significant inherent latency to the system.

Tasks that are executed occasionally (i.e. not in every block, for example, after filling an input buffer) and that are computationally expensive, such as the inference task, should be called from an auxiliary thread with a lower priority than the audio thread. This is shown in Figure 5: in orange, the inference task is called when the input buffer is filled with two audio blocks (i.e. at the end of block 2). However, it does not start running immediately at the beginning of block 3, since the audio callback has higher priority. Once the audio callback task is finished, the inference task starts executing. When block 4 starts, the inference task is put to sleep (indicated with a gray bar), and the CPU executes the audio callback instead. When the audio callback task is finished, the CPU is free again to complete the inference computation. At the end of block 4, the input buffer has been filled again (with samples from blocks 3 and 4), and the process repeats.

[16]In reality, the inference task, due to its computational complexity, would probably take longer (e.g. 16 audio blocks).

4. CONCLUSION

Many embedded platforms have lowered entry barriers to real-time audio programming by abstracting complex tasks through APIs and IDEs. However, deploying deep learning models involves an ecosystem of tools that demands lower-level software development skills, such as building a program using a custom CMake recipe. This paper presents a pipeline for recording datasets and deploying neural models in the Bela embedded hardware platform. In contrast to existing embedded AI projects that target a specific deep learning task or application, this pipeline serves as a template for programmers and makers to prototype and experiment with deep neural networks for real-time musical applications. With this pipeline, we aim to reduce the complexity (or viscosity [4]) of this process and contribute towards bridging the deep learning for audio and embedded hardware communities in NIME.

As part of the pipeline, we provide tools for recording multichannel datasets (by connecting multiple Bela boards) and a dockerised cross-compilation environment. These tools are of interest beyond their role in the pipeline: the dataset recording tool allows capturing datasets from multichannel sensor arrays for later analysis, and the cross-compiling environment significantly reduces the compilation times.

Finally, the models will only run in real-time if their computational complexity is low enough to run on the Bela CPU. Future work will focus on including network compression and embeddability diagnosing tools into the pipeline.

5. ACKNOWLEDGMENTS

This work was supported by the EPSRC UKRI Centre for Doctoral Training in Artificial Intelligence and Music (EP/S022694/1) and the Royal Academy of Engineering under the Research Chairs and Senior Research Fellowships scheme. The authors would like to thank Giulio Moro and Franco Caspe for their technical support, Lia Mice for the mini Chaos Bells lending, and the anonymous reviewers for their valuable feedback.

6. ETHICAL STANDARDS

Adán L. Benito and Andrew McPherson are part of Augmented Instruments Ltd, the company that produces the Bela platform. Teresa Pelinski's PhD is also partly supported by the same company.

7. REFERENCES

[1] E. Berdahl. How to Make Embedded Acoustic Instruments. In *Proceedings of the International Conference on New Interfaces for Musical Expression*, pages 140–143, London, United Kingdom, June 2014. `https://zenodo.org/record/1178710`.

[2] M. Bergomi, A. Caillon, A. C. Romeu-Santos, N. Devis, C. Douwes, P. Esling, Mantovani, and

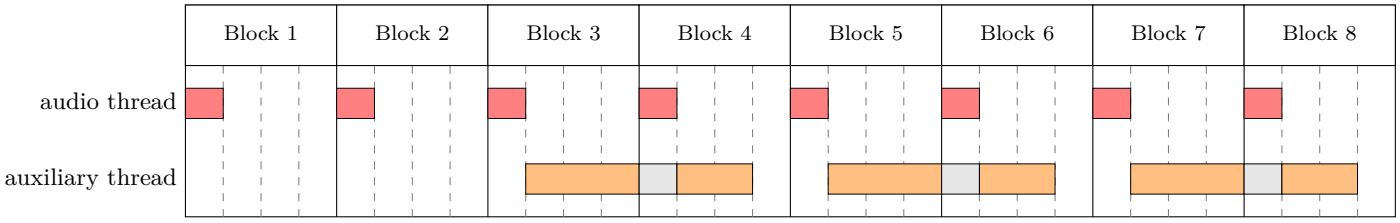

Figure 5: Multi-threading. The audio thread (highest priority) and the auxiliary thread are shown, respectively, in red and orange. Grey indicates that the thread is sleeping. In this example, the auxiliary thread waits for a buffer of two blocks of samples to be filled. For this reason, the auxiliary thread does not run in the first two blocks.

H. Scurto. Tools and Perspectives for Interaction with Neural Audio Synthesis. In *Proceedings of the International Conference on New Interfaces for Musical Expression*, Auckland, New Zealand, June 2022. https://nime.pubpub.org/pub/bo41qut9/release/1.

[3] F. Bevilacqua, R. Müller, and N. Schnell. MnM: A Max/MSP mapping toolbox. In *New Interfaces for Musical Expression*, page 85, 2005. https://hal.science/hal-01161330.

[4] A. F. Blackwell and T. Green. Notational Systems—The Cognitive Dimensions of Notations Framework. In J. M. Carroll, editor, *HCI Models, Theories, and Frameworks: Toward a Multidisciplinary Science*, pages 103–133. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.

[5] J. Bullock and A. Momeni. Ml.lib: Robust, Cross-platform, Open-source Machine Learning for Max and Pure Data. In *Proceedings of the International Conference on New Interfaces for Musical Expression*, Louisiana State University, Baton Rouge, LA, 2015.

[6] A. Caillon and P. Esling. RAVE: A variational autoencoder for fast and high-quality neural audio synthesis [Preprint]. http://arxiv.org/abs/2111.05011, Nov. 2021.

[7] J. Chowdhury. RTNeural: Fast Neural Inferencing for Real-Time Systems [Preprint]. http://arxiv.org/abs/2106.03037, June 2021.

[8] N. Devis and P. Esling. Neurorack: Deep audio learning in hardware synthesizers. In *EPFL PhD Seminar "Human Factors in Digital Humanities"*, Lausanne, Switzerland, 2021. https://infoscience.epfl.ch/record/291222?ln=en.

[9] Ç. Erdem, R. Simionato, S. Mojtaba Karbasi, and A. Refsum Jensenius. Embodied Perspectives on Musical AI (EmAI) - RITMO Centre for Interdisciplinary Studies in Rhythm, Time and Motion. https://www.uio.no/ritmo/english/news-and-events/events/workshops/2022/embodied-ai/index.html, 2022.

[10] P. Esling, N. Devis, A. Bitton, A. Caillon, A. Chemla–Romeu-Santos, and C. Douwes. Diet deep generative audio models with structured lottery. In *Proceedings of the 23rd International Conference on Digital Audio Effects (DAFx-20)*, Vienna, Austria, Sept. 2020. http://arxiv.org/abs/2007.16170.

[11] J. Françoise, N. Schnell, R. Borghesi, and F. Bevilacqua. Probabilistic Models for Designing Motion and Sound Relationships. In *Proceedings of the 2014 International Conference on New Interfaces for Musical Expression*, page 287, London, UK, June 2014. https://hal.science/hal-01061335.

[12] V. E. Gonzalez Sanchez, C. P. Martin, A. Zelechowska, K. A. V. Bjerkestrand, V. Johnson, and A. R. Jensenius. Bela-based augmented acoustic guitars for sonic microinteraction. In *Proceedings of the International Conference on New Interfaces for Musical Expression*, pages 324–327, Blacksburg, Virginia, USA, 2018. Virginia Tech. https://www.zenodo.org/record/1302599.

[13] J. Hanhirova, T. Kämäräinen, S. Seppälä, M. Siekkinen, V. Hirvisalo, and A. Ylä-Jääski. Latency and throughput characterization of convolutional neural networks for mobile computer vision. In *Proceedings of the 9th ACM Multimedia Systems Conference*, MMSys '18, pages 204–215, New York, NY, USA, June 2018. Association for Computing Machinery. https://doi.org/10.1145/3204949.3204975.

[14] C. Kiefer. Stochastic Optimisation of Lookup Table Networks, for Realtime Inference on Embedded Systems. In *Proceedings of the 2nd Joint Conference on AI Music Creativity*, page 10, online, 2021. https://zenodo.org/record/5137956.

[15] S. Kuznetsov and E. Paulos. Rise of the Expert Amateur: DIY Projects, Communities, and Cultures. In *Extending Boundaries - Proceedings of the 6th Nordic Conference on Human-Computer Interaction*, page 304, Reykjavik, Iceland, Dec. 2010. https://dl.acm.org/doi/10.1145/1868914.1868950.

[16] C. Martin, F. Morreale, B. Wallace, and H. Scurto. Workshop in Critical Perspectives on AI/ML in Musical Interfaces. In *International Conference on New Interfaces for Musical Expression*, Birmingham City University, Birmingham, UK, 2020. https://intelligence-in-musical-interfaces.github.io/about/.

[17] C. P. Martin, K. Glette, T. F. Nygaard, and J. Torresen. Understanding Musical Predictions With an Embodied Interface for Musical Machine Learning. *Frontiers in Artificial Intelligence*, 3, 2020. https://www.frontiersin.org/article/10.3389/frai.2020.00006.

[18] R. Masu and F. Morreale. Composing by Hacking: Technology Appropriation as a Pedagogical Tool for Electronic Music. In B. Stevens, editor, *Teaching Electronic Music: Cultural, Creative, and Analytical Perspectives*. Routledge, New York, 1st edition, June 2021.

[19] A. McPherson, R. Jack, and G. Moro. Action-Sound Latency: Are Our Tools Fast Enough? In *Proceedings of the International Conference on New Interfaces for Musical Expression*, pages 20–25, Brisbane, Australia,

2016. https://www.zenodo.org/record/3964611.

[20] A. McPherson and V. Zappi. An environment for submillisecond-latency audio and sensor processing on BeagleBone black. In *138th Audio Engineering Society Convention*, Warsaw, Poland, May 2015. http://www.aes.org/e-lib/browse.cfm?elib=17755.

[21] E. Meneses, J. Wang, S. Freire, and M. Wanderley. A Comparison of Open-Source Linux Frameworks for an Augmented Musical Instrument Implementation. In *Proceedings of the International Conference on New Interfaces for Musical Expression*, pages 222–227, Porto Alegre, Brazil, June 2019. https://zenodo.org/record/3672934.

[22] S. Mittal, P. Rajput, and S. Subramoney. A Survey of Deep Learning on CPUs: Opportunities and Co-Optimizations. *IEEE Transactions on Neural Networks and Learning Systems*, 33(10):5095–5115, Oct. 2022.

[23] G. Moro, A. Bin, R. Jack, C. Heinrichs, and A. Mcpherson. Making High-Performance Embedded Instruments with Bela and Pure Data. In *Proceedings of the 2016 International Conference on Live Interfaces*, University of Sussex, Brighton, UK, June 2016. https://users.sussex.ac.uk/~thm21/ICLI_proceedings/2016/Practical/Workshops/129_Bela.pdf.

[24] F. Morreale and A. McPherson. Design for Longevity: Ongoing Use of Instruments from NIME 2010-14. In *Proceedings of the International Conference on New Interfaces for Musical Expression*, pages 184–189, Blacksburg, Virginia, USA, June 2017. https://www.zenodo.org/record/1176218.

[25] F. Morreale, G. Moro, A. Chamberlain, S. Benford, and A. McPherson. Building a Maker Community Around an Open Hardware Platform. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, Denver, CO, USA, May 2017. https://dl.acm.org/doi/10.1145/3025453.3026056.

[26] 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. In *International Conference on New Interfaces for Musical Expression*, Auckland, New Zealand, 2022. https://nime.pubpub.org/pub/rwr2c3zs/release/1.

[27] E. Pierce. Machine Learning with Bela & IREE - Google Summer of Code. https://gist.github.com/ezrapierce000/af89fdab4dac376f21f2a836807c6c62, 2022.

[28] H. L. Renney, T. Mitchell, and B. Gaster. There and Back Again: The Practicality of GPU Accelerated Digital Audio. In *Proceedings of the International Conference on New Interfaces for Musical Expression*, pages 202–207, Birmingham, UK, June 2020. https://www.zenodo.org/record/4813320.

[29] T. Skare. GPGPU Patterns for Serial and Parallel Audio Effects. In *Proceedings of the 23rd International Conference on Digital Audio Effects (DAFx-20)*, pages 125–131, Vienna, Austria, Sept. 2020. https://www.dafx.de/paper-archive/2020/proceedings/papers/DAFx2020_paper_73.pdf.

[30] D. Stefani, S. Peroni, and L. Turchet. A Comparison of Deep Learning Inference Engines for Embedded Real-Time Audio Classification. In *Proceedings of the 25th International Conference on Digital Audio Effects (DAFx20in22)*, pages 256–263, Vienna, Austria, Sept. 2022. https://www.dafx.de/paper-archive/2022/papers/DAFx20in22_paper_16.pdf.

[31] J. Sullivan, J. Vanasse, C. Guastavino, and M. Wanderley. Reinventing the Noisebox: Designing Embedded Instruments for Active Musicians. In *Proceedings of the International Conference on New Interfaces for Musical Expression*, pages 5–10, Birmingham, UK, June 2020. https://zenodo.org/record/4813166.

[32] K. Tahiroğlu, M. Kastemaa, and O. Koli. AI-terity 2.0: An Autonomous NIME Featuring GANSpaceSynth Deep Learning Model. In *International Conference on New Interfaces for Musical Expression*, Shanghai, China, June 2021. https://nime.pubpub.org/pub/9zu49nu5/release/1.

[33] P. A. Tremblay, G. Roma, and O. Green. Enabling Programmatic Data Mining as Musicking: The Fluid Corpus Manipulation Toolkit. *Computer Music Journal*, 45(2):9–23, June 2021. https://doi.org/10.1162/comj_a_00600.

[34] N. Vasilache, O. Zinenko, A. J. C. Bik, M. Ravishankar, T. Raoux, A. Belyaev, M. Springer, T. Gysi, D. Caballero, S. Herhut, S. Laurenzo, and A. Cohen. Composable and Modular Code Generation in MLIR: A Structured and Retargetable Approach to Tensor Compiler Construction [Preprint]. http://arxiv.org/abs/2202.03293, Feb. 2022.

[35] F. G. Visi and A. Tanaka. Interactive Machine Learning of Musical Gesture. In E. R. Miranda, editor, *Handbook of Artificial Intelligence for Music: Foundations, Advanced Approaches, and Developments for Creativity*, page 994. Springer International Publishing, 2021.

[36] Y. E. Wang, G.-Y. Wei, and D. Brooks. Benchmarking TPU, GPU, and CPU Platforms for Deep Learning [Preprint]. http://arxiv.org/abs/1907.10701, Oct. 2019.

[37] E. Zayas-Garin, J. Harrison, R. Jack, and A. McPherson. DMI Apprenticeship: Sharing and Replicating Musical Artefacts. In *Proceedings of the International Conference on New Interfaces for Musical Expression*, NYU Shanghai, China, June 2021. https://nime.pubpub.org/pub/dmiapprenticeship/release/1.

[38] M. Zhang, S. Rajbhandari, W. Wang, and Y. He. DeepCPU: Serving RNN-based Deep Learning Models 10x Faster. In *Proceedings of the 2018 USENIX Annual Technical Conference*, 2018. https://www.usenix.org/system/files/conference/atc18/atc18-zhang-minjia.pdf.