NEURAL AUDIO PROCESSING ON ANDROID PHONES

Jason Hoopes
College of Arts Media and Design
Northeastern University
Boston, MA (USA)
hoopes.j@northeastern.edu

Brook Chalmers
Khoury College of Computer Sciences
Northeastern University
Boston, MA (USA)
chalmers.b@northeastern.edu

Victor Zappi
College of Arts Media and Design
Northeastern University
Boston, MA (USA)
v.zappi@northeastern.edu

ABSTRACT

This study investigates the potential of real-time inference of neural audio effects on Android smartphones, marking an initial step towards bridging the gap in neural audio processing for mobile devices. Focusing exclusively on processing rather than synthesis, we explore the performance of three open-source neural models across five Android phones released between 2014 and 2022, showcasing varied capabilities due to their generational differences. Through comparative analysis utilizing two C++ inference engines (ONNX Runtime and RTNeural), we aim to evaluate the computational efficiency and timing performance of these models, considering the varying computational loads and the hardware specifics of each device. Our work contributes insights into the feasibility of implementing neural audio processing in real-time on mobile platforms, highlighting challenges and opportunities for future advancements in this rapidly evolving field.

1. INTRODUCTION

Neural audio is quickly becoming a staple in the field of digital signal processing (DSP), thanks to major advances in both audio processing and synthesis. This growth is largely driven by two key developments in machine learning: the introduction of new deep learning architectures that open up innovative approaches to DSP and the improvement of inference engines that make it possible to run neural models on a variety of devices. These engines are designed to take advantage of device hardware to speed up calculations, marking a significant step forward in smarter and more adaptive audio processing techniques.

The rise of machine learning applications has notably influenced the development of Android phones. Leading mobile manufacturing companies have integrated specific hardware and software features into their devices to support the real-time running of complex neural models, both for generative AI and for DSP applications. For example, Qualcomm has rolled out the Snapdragon 8 Gen 3 and X Elite chips, featuring a dedicated Hexagon Neural Processing Unit[1] (NPU). These chips are designed to handle trillions of operations per second, supporting a wide range of neural models right on the device; this includes everything from large language models to models for automatic speech recognition. Google's latest Tensor G3 chip[2] combines an ARM CPU with a GPU and NPU, enabling it to run models that are significantly more complex than those supported by earlier Pixel phones. Both Google and Samsung are incorporating their own AI models into their smartphones, enabling features like real-time speech processing and advanced image/video editing, showing a trend in using neural processing technology to improve user experiences with sophisticated DSP tasks. Moreover, Qualcomm and Google have developed software tools (Neural Processing SDK[3] and Neural Networks API[4], respectively) to make it easier to deploy neural models on their devices.

Despite the advances in hardware and software for neural processing on mobile devices, the literature reveals a noticeable gap in the application of neural models for musical purposes on Android platforms. Existing research often confines the application of audio models to speech detection, natural language processing and classification tasks (e.g., [1, 2]). Furthermore, a considerable portion of the literature predominantly focuses on video and image processing [3, 4, 5], with detailed benchmarks of neural models for computer vision [6], but scant attention to musical applications such as full-duplex processing (audio effects) and synthesis on Android devices.

In light of this observation, our work initiates the effort to bridge this gap by examining the potential for real-time inference of neural audio effects on Android phones. Our investigation involves testing three open-source models across five different devices using two C++ inference engines. This analysis aims to evaluate and compare the computational and temporal performance of these models, taking into account the hardware and software specifics of the devices involved.

2. RELATED WORK

Research into neural models has extended to their application on embedded audio boards, reflecting the growing interest in deploying advanced audio processing capabilities on compact and specialized devices. Pelinski et al. [7] offer an in-depth examination of the processes involved in deploying real-time neural models on the Bela embedded audio platform, highlighting the challenges of cross-compiling inference engines and providing templates to streamline this process. Similarly, Stefani et al. [8] turn their attention to the Raspberry Pi and Elk Audio OS, comparing the performance of four inference engines (Tensorflow Lite, TorchScript,

[1] https://www.qualcomm.com/products/mobile/snapdragon/smartphones/snapdragon-8-series-mobile-platforms

Copyright: © 2024 Jason Hoopes et al. This is an open-access article distributed under the terms of the Creative Commons Attribution 4.0 International License, which permits unrestricted use, distribution, adaptation, and reproduction in any medium, provided the original author and source are credited.

[2] https://blog.google/products/pixel/google-tensor-g3-pixel-8/

[3] https://developer.qualcomm.com/software/qualcomm-neural-processing-sdk

[4] https://developer.android.com/ndk/guides/neuralnetworks/

ONNX Runtime and RTNeural) across three variants of the same model, differing in complexity and the number of trainable parameters. Notably, both studies primarily focus on prediction and classification models, rather than delving into audio processing or synthesis.

In the broader landscape of neural audio models, the literature has seen significant advancements over the past few years. Pioneering models such as SampleRNN [9] and WaveNet [10] have set benchmarks for synthesizing raw audio waveforms with high fidelity, catering to both speech synthesis and musical audio generation. Subsequent developments, like GANsynth [11], have pushed the envelope further by optimizing for real-time performance, thus facilitating faster synthesis suitable for live applications. Moreover, innovations like RAVE (Realtime Audio Variational autoEncoder) [12] underscore the potential for efficient, high-quality audio synthesis on less powerful hardware, expanding the horizons for real-time audio generation technologies.

Of particular relevance to our work is the research focused on the neural modeling of audio effects. Damskagg et al. [13] delve into modeling audio distortion circuits with deep neural networks, examining three different guitar pedals. By adopting a black-box approach, they leverage a lightweight variant of the WaveNet architecture designed specifically to emulate the distinctive sonic characteristics of these devices, based on their input and output signals. Martínez-Ramírez et al. [14] introduce a novel deep learning model for simulating time-varying audio effects, such as chorus and tremolo, through the use of convolutional and recurrent neural networks. This model aims to offer a versatile solution capable of accommodating various audio modifications, encompassing both linear and nonlinear changes. Further expanding on this theme, the same authors [15] conduct a comprehensive study on the application of different deep neural network architectures for virtual analog modeling of nonlinear effects, including tube amplifiers, transistor-based limiters and the dynamic components of the Leslie speaker.

3. MATERIALS AND METHODS

3.1. Phones

Our experimental setup included five Android phones, chosen to span a range of release years from 2014 to 2022, thereby encapsulating a wide array of generational capabilities. The first phone (Phone 1) is an LG G2 Mini, released back in 2014 and equipped with the Qualcomm Snapdragon 400 chipset; this technology features a quad-core 1.2 GHz Cortex-A7 CPU. Phone 2 is a Zenfone 2 Laser released by Asus in 2016, powered by the Qualcomm Snapdragon 615 chipset; while still relatively low-tier, it offers a more performative octa-core CPU (quad-core 1.7 GHz Cortex-A53 and quad-core 1.0 GHz Cortex-A53). Phone 3 is a 2018 Xiaomi Mi 8 Lite, mounting the mid-range Qualcomm Snapdragon 660 chipset; this chipset features an octa-core CPU setup, with four Kryo 260 cores clocked at 2.2 GHz for performance tasks and four Kryo 260 cores clocked at 1.8 GHz for efficiency tasks. Phone 4 is a Samsung Galaxy S21+ from 2021, a powerful device running the Qualcomm Snapdragon 888 chipset, with an octa-core CPU that includes one Kryo 680 core clocked at up to 2.84 GHz, three Kryo 680 cores clocked at up to 2.42 GHz and four Kryo 680 cores clocked at up to 1.8 GHz for efficiency. The last and most recent phone (Phone 5) is a 2022 Asus Zenfone 9 powered by the Qualcomm Snapdragon 8+ Gen 1 chipset; this high-end processor features an octa-core CPU configuration, including one Cortex-X2 core running at up to 3.19 GHz for peak performance, three Cortex-A710 cores running at up to 2.75 GHz for balanced performance and four Cortex-A510 cores running at up to 1.80 GHz for efficiency. Table 1 summarizes the specifics of the five devices.

While all devices were tested in a full-duplex setting, utilizing the line-in/line-out via the headphone combo jack, it's noteworthy that the Xiaomi and Samsung models required a USB-C adapter due to the absence of a combo jack.

3.2. Models

Three open-source neural models, distinguished by their increasing complexity and computational load—measured in floating point operations (FLOPs) per sample—were selected for this study. These models were re-implemented in Google Colab notebooks[5], facilitating the export of weights as a JSON file for RTNeural and as an .onnx file, ensuring compatibility with our chosen inference engines (see the next subsections). The first neural network architecture under scrutiny (model A) is the Auto-Guitar Amp model proposed by Wright et al [16]; it is composed of a Long Short-Term Memory (LSTM) and a dense layer, set with an input size of 1 (current input sample), hidden size of 20 units and output size of 1 sample. Model B is the ED (encoder-decoder) model for emulating hardware compressors presented by Simionato and Fasciani [17]; the enconder is composed of two 1D convolutional layers and two dense layers, while the decoder comprises an LSTM and two dense layers. Our implementation matches the configuration recommended by the authors: the encoder uses as input the latest 32 samples and 4 conditioning parameters, while the decoder is set to 64 hidden units. This model works on a block basis, for it outputs a buffer of 16 processed samples per each call. Model C is GuitarLSTM, a variation on the architecture from [16] as proposed by Bloemer[6]; it features two 1D convolutional layers, an LSTM and an output dense layer, in our implementation set to an input size of 5 samples, 32 hidden units and output size of 1 sample.

Additionally, two "abstract" models designed in PyTorch were introduced to simulate varying computational load scenarios without emulating any specific audio effect (more details in Section 4.2). The first abstract model, referred to as the *baseline* model, is composed of two dense layers with an output size of 1 and a negligible number of FLOPs. The second model, called the *topline* model, is designed to assess performance under a large computational load[7] in a block-based scenario. It consists of a series of two dense layers with an output size of 16 and a number of FLOPs comparable to Model C, the largest of our test pool.

Further details on all models can be referenced in the cited papers and links. Table 2 lists the FLOPS per sample in each model, while Table 3 illustrates the FLOPs formulae we used for each type of layer employed.

3.3. Audio Environments

Our investigation utilized two distinct audio environments to assess the performance of the neural models under different conditions. The first environment was a bespoke Android audio app

[5] https://drive.google.com/drive/folders/161iVYccRiQ5IFHqqusx63qpHyngZwfRh?usp=sharing

[6] https://github.com/GuitarML/GuitarLSTM

[7] Large in the context of our pool of audio effect models.

Table 1: *Tested phones' details. The native buffer size of the embedded audio codec was obtained by inspecting Android's Audio Flinger details.*

Alias	Phone	Year	Qualcomm Chipset	Cores	Max Clock	Native Buffer
Phone 1	LG G2 Mini	2014	Snapdragon 400	4	1.2 GHz	240
Phone 2	Asus Zenfone 2 Laser	2016	Snapdragon 615	8	1.7 GHz	240
Phone 3	Xiaomi Mi 8 Lite	2018	Snapdragon 660	8	2.2 GHz	192
Phone 4	Samsung S21+	2021	Snapdragon 888	8	2.84 GHz	96
Phone 5	Asus Zenfone 9	2022	Snapdragon 8+ Gen 1	8	3.19 GHz	96

Table 2: *Output sizes and FLOPs per sample of each model.*

Alias	Model	Output Size	FLOPs/sample
Model A	Auto-Guitar Amp	1	3,400
Model B	ED	16	7,332
Model C	GuitarLSTM	1	20,416
Baseline	-	1	16
Topline	-	16	20,480

Table 3: *Formulae employed to calculate the FLOPs per each layer; i = input size, o = output size, u = hidden units, s = stride, k = kernel size, f = filters.*

Layer	FLOPs
Dense	$2 \times i \times o$
Sigmoid	$4 \times o$
LSTM	$8 \times (i + u) \times u$
1D Conv	$\lceil i/s \rceil \times k \times f \times 2$

project (app). The app was crafted with flexibility in mind and features an optimized full-duplex audio engine powered by the Oboe library. One of the key design principles was its ability to load and run code for various audio applications, including the neural models in question, through a straightforward C++ API. This design choice allowed for the effortless running and testing of different models by simply switching the C++ source files via CMake, without the need for recompilation in Java or Kotlin.

We also employed LDSP, a C++ mobile audio environment introduced by Zappi and Tapparo [18] (low-level). Unlike the app, LDSP offered a platform for evaluating the models' performance at a lower level, directly interfacing with the hardware without the mediation of the Android audio stack. This environment was particularly valuable for understanding how the models would perform in scenarios closer to the metal, where the overhead of the Android system is minimized. The C++ API designed for the app was compatible with LDSP's API, enabling us to test identical audio code across both environments seamlessly.

3.4. Inference Engines

RTNeural [19] and ONNX Runtime[8] were selected as our inference engines. This choice was influenced by their documented success in previous studies involving embedded Linux audio boards (e.g., [8] and the follow-up development of [20]), where they demonstrated efficient performance and robustness. Moreover, these two engines offer very distinct approaches to neural model inference. RTNeural specializes in efficient, real-time neural network inference for audio with a focus on speed and low overhead, but requires a complete redesign of models and supports a limited set of neural layers. As opposed, ONNX Runtime offers broad direct compatibility and flexibility for a very wide variety of model architectures across several platforms. Both engines were integrated into the app and low-level environment, with RTNeural compiled at build time and ONNX Runtime linked as a pre-built dynamic library, and set to default delegates for ARM CPUs.

To execute the models consistently in both the app and the low-level environment, we adopted two approaches: for RTNeural, we re-coded each model using its C++ API, enabling the loading of pre-trained model weights through JSON files. For ONNX Runtime, we exported the models from Python to the .onnx format, allowing them to be loaded and run at runtime.

3.5. Tests

We conducted two primary tests to evaluate the models under various configurations, i.e., different combinations of phones, environments and inference engines. The first test aimed to identify the smallest buffer size for each configuration that could operate without inducing underruns. In full-duplex applications, both an input and an output buffer are utilized, which may differ in size. Given that audio apps operate within the Android audio stack, they inherently involve two additional buffering layers—mixer (Audio Flinger) and audio HAL [18]. However, our focus was on the output buffer size, as it represents the primary computational bottleneck for model inference across all configurations. While the sizes of other buffers could affect round-trip latency, analyzing this aspect was outside the scope of our paper. We aligned our buffer size tests with integer multiples of each phone's native buffer size (Table 1); as described in [21], in Android apps, this enhances the efficiency of sample exchange with the audio driver/audio HAL by minimizing the number of bursts per each buffer. Should smaller buffer sizes prove viable, we further tested common powers of two (e.g., 128, 64). Notably, as the baseline and topline (i.e., abstract) models were not designed for training or meaningful processing and maintained randomly initialized weights, their outputs were disregarded during testing. Underruns were assessed based on a passthrough audio stream instead.

In the second test, we executed each configuration once for a duration of 10 seconds and recorded the inference times, calculating the mean and standard deviation values. All devices and applications were standardized to run at a 48 kHz sample rate, resulting in logs of 480,000 samples for each test run.

The full source code of the app environment, including the integration of inference engines, can be found here: `https://github.com/victorzappi/LDSPlite.git`. Two dedicated repositories host the app test projects using ONNX Runtime

[8] `https://github.com/microsoft/onnxruntime.git`

models[9] and RTNeural models[10].

The equivalent low-level codebase (LDSP) is accessible here: `https://github.com/victorzappi/LDSP.git`. LDSP test projects can be found here: ONNX Runtime models[11] and RTNeural models[12].

It is important to note that we were unable to root the Samsung S21+ and the Asus Zenfone 9 due to recent changes in the bootloader unlocking policies by both manufacturers. Rooting is essential for the operation of LDSP and, as such, we are unable to provide low-level measurements for these two devices within our current findings.

4. RESULTS

4.1. RTNeural

Table 4 collects the smallest buffer sizes that RTNeural was capable of supporting in real-time in each phone-model configuration. These results include values for both the app and the low-level environment.

It is noteworthy that, for the app, on Phone 1 and Phone 2 the choice of the buffer size was fixed to twice as the native value. These two devices run Android versions below 8.1 (SDK < 27) and hence the Oboe library employed in the app behaves as a wrapper for OpenSLES, which does not allow for the selection of arbitrary buffer sizes. As opposed, in the other more recent phones, Oboe can leverage the more advanced AAudio library that supports buffer size selection, but only down to the native buffer size.

The low-level environment does not impose any software limitation on the chosen parameters and allows for reaching the hardware limits of the devices. For example, the low-level buffer sizes reported for Phone 1 and Phone 3 coincide with the smallest sizes that these devices can support in a passthrough scenario, i.e., when no processing occurs.

Table 5 displays the mean and standard deviation of the inference times per sample, measured in each RTNeural configuration. As expected, all real-time configurations sport inference times below the sampling period, which is around 20.833 µs for the tests' sample rate. Although this is a good first indicator of the performance of each test, a couple of considerations are necessary to correctly interpret these data. First, the incidence of underruns depends on the collective inference times within each buffer, more than on the time mean across the full duration of the test. In other words, the actual time constraint is represented by the buffer period, defined as $buffersize/samplerate$. Hence, spikes in the inference times that are even way above the sampling period mark might not spoil real-time performance, as long as their occurrence is sparse enough. This explains why some time entries in the table where a large standard deviation may push the inference time quite close to the sampling period are still capable of running in real-time. Figure 1 illustrates the case of Phone 4-Model C. Conversely, Phone 1-Model A showcases a quite low mean and standard deviation measurements when running within the app, yet several underruns were reported; this was due to computational time spikes that were moderately high and limited in number, but appearing in short series within the same buffer. Second, the use of the buffer period as real-time time constraint is quite optimistic, for a portion of this time window must be allocated to sample data formatting (to comply with the audio format used by the driver/codec), data transfer (to/from the driver/codec) and—in the case of the app—extra mixing and buffering stages (see Section 3). This was reflected in our underruns/real-time measurements.

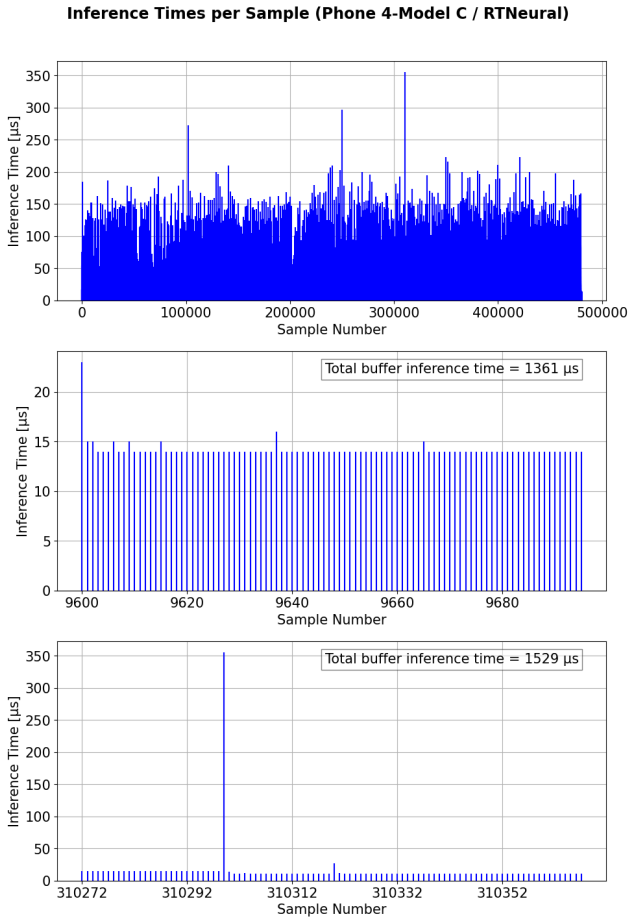

Figure 1: *Top: inference times for the full 480,000-sample run of the configuration Phone 4 - Model C. Middle and bottom: two close-ups on single 96-sample buffers, showing that very high inference time spikes are unlikely to occur consecutively in this configuration, with other calls within the same buffer around* 14 µs*, keeping the total inference time below the buffer period (*2000 µs*).*

4.2. ONNX Runtime

Table 6 shows the buffer sizes supported by ONNX Runtime for all models, while timing results for models A, B and C are showcased in Table 7 (top). The considerations made in the previous subsection about buffer size selection (in the app and the low-level environment) and the interpretation of the timing results still hold up here.

As noted in Table 7, in a few instances the app was not able to log the inference times due to crashes occurring before the conclu-

[9] `https://github.com/victorzappi/LDSPlite-ONNX.git`

[10] `https://github.com/victorzappi/LDSPlite-RTNeural.git`

[11] `https://github.com/victorzappi/LDSP-ONNX.git`

[12] `https://github.com/victorzappi/LDSP-RTNeural.git`

Table 4: *Smallest underrun-free buffer sizes with RTNeural; "-" means that the configuration could not run in real-time.*

	Model A		**Model B**		**Model C**	
	app	*low-level*	*app*	*low-level*	*app*	*low-level*
Phone 1	-	64	480	64	-	-
Phone 2	480	240	480	240	-	-
Phone 3	192	128	192	128	384	128
Phone 4	96	n/a	96	n/a	96	n/a
Phone 5	96	n/a	96	n/a	96	n/a

Table 5: *Mean and standard deviation of inference times (µs) per sample with RTNeural, calculated across 480,000 samples. Real-time configurations highlighed in green.*

	Model A		**Model B**		**Model C**	
	app	*low-level*	*app*	*low-level*	*app*	*low-level*
Phone 1	10.149 (2.320)	10.033 (0.745)	14.631 (0.999)	14.536 (0.352)	46.066 (7.636)	45.003 (5.033)
Phone 2	4.621 (1.302)	3.447 (1.423)	10.092 (2.840)	7.537 (0.981)	20.72 (2.326)	18.454 (2.714)
Phone 3	2.054 (1.331)	2.044 (1.022)	5.225 (0.977)	4.905 (0.674)	13.255 (2.726)	12.069 (2.144)
Phone 4	4.913 (4.449)	n/a	11.627 (2.325)	n/a	14.105 (5.821)	n/a
Phone 5	0.070 (0.377)	n/a	0.797 (0.347)	n/a	2.057 (0.652)	n/a

sion of the runs. This was observed for the heavy Model C on the three older phones, as well as for Model A when running on Phone 2. These crashes are likely due to inference times so large that they consistently exceeded the real-time constraints of each buffer period. Excessive delay could disrupt the synchronization between the input and output audio streams and the callback mechanism of the underlying audio driver, leading to the application's instability.

Preliminary data analysis underlined that, in general, ONNX Runtime struggles to run the audio effect models in real-time, within both the app and the low-level environment. However, upon closer inspection it is possible to note how Model B manages to reach real-time performance in several configurations and at reasonably large buffer sizes. The main difference between this model and Models A and C is its block-based nature, characterized by an output size of 16 samples per each inference. In other words, the model is called 16 times less often than Models A and B when processing the same continuous input stream. This detail, combined with the way larger inference times observed for Models A and C, suggests that the ONNX Runtime library suffers from a conspicuous call overhead, whose impact is more visible with smaller output sizes—due to a larger number of calls within each buffer period.

We decided to measure this overhead, by designing and testing the baseline model characterized by an output size of 1 and a negligible number of FLOPs (see Table 2). Likewise, we introduced the topline model to assess how ONNX Runtime performs under a large computational load, but in a more favorable block-based scenario. Inference time results for these two models are available in the bottom part of Table 7.

5. DISCUSSION

5.1. Cutting-edge and Legacy Mobile Neural Processing

In our evaluation, it's evident that all the smartphones tested are capable of executing the audio effect models in real-time. Consistent with expectations, the more recent devices particularly excel, often delivering inference times significantly below the real-time thresholds. This performance aligns with the rapid technological advancements in the mobile phone sector, where each year brings new chipsets designed for increasing speed, scalability and compatibility with contemporary computational demands, like mobile gaming and artificial intelligence applications.

Notably, some of the devices we tested, like Phones 4 and 5, are equipped with hardware accelerators specifically tailored for neural network inference. These include the Hexagon Tensor Accelerator, optimized for quantized models, and the Adreno GPU, which excels with floating-point models—both found in the Snapdragon Series 8 [5]. However, it's crucial to underline that our experiments exclusively utilized CPU delegates for both RTNeural and ONNX Runtime, hence bypassing these advanced accelerators. These delegates can though rely on CPU parallelization capabilities, including the utilization of ARM's NEON vectorized instruction set, to process the neural audio effects.

Surprisingly, our findings also indicate that even significantly older smartphones, like Phones 1 and 2[13], can effectively run neural audio effects in real-time. While the heaviest model proved too demanding for these older devices, they still managed to support the remaining models with commendable reliability. Though the number of viable configurations on these older phones is somewhat restricted and inference times are generally longer when compared to their newer counterparts, their performance remains robust. This suggests that, given the right conditions—such as the choice of an appropriate inference engine and audio environment tailored to the model's requirements—these older devices could find a new lease on life as dedicated neural audio processors, upcycled specifically for this task [18].

5.2. Beyond the Boundaries of Audio Apps

The comparison between the low-level environment and the app-based environment yields a notable distinction in performance, with the former consistently outperforming the latter. Inference times in the low-level setting are almost always substantially lower, a difference that becomes particularly pronounced on devices operating older Android versions, such as Phones 1 and 2. For instance, Phone 2 exhibits a reduction in inference times by more

[13] At the time of writing, these are 10 and 8 years old respectively!

Table 6: *Smallest underrun-free buffer sizes with ONNX Runtime; "-" means that the configuration could not run in real-time*

	Model A		**Model B**		**Model C**		**Baseline**		**Topline**	
	app	*low-level*	*app*	*low-level*	*app*	*low-level*	*app*	*low-level*	*app*	*low-level*
Phone 1	-	-	-	-	-	-	-	-	480	64
Phone 2	-	-	480	240	-	-	-	-	480	240
Phone 3	-	-	384	128	-	-	-	128	192	128
Phone 4	-	n/a	96	n/a	-	n/a	192	n/a	96	n/a
Phone 5	96	n/a	96	n/a	-	n/a	96	n/a	96	n/a

Table 7: *Mean and standard deviation of inference times (µs) per sample with ONNX Runtime, calculated across 480,000 samples. Real-time configurations highlighted in green;"(crash)" marks configurations that could not complete the log due to excessive computational load.*

	Model A		**Model B**		**Model C**	
	app	*low-level*	*app*	*low-level*	*app*	*low-level*
Phone 1	162.811 (14.187)	158.188 (11.084)	21.181 (1.597)	21.208 (1.343)	(crash)	699.073 (35.017)
Phone 2	(crash)	115.80 (17.198)	14.821 (2.060)	13.374 (2.024)	(crash)	352.449 (42.645)
Phone 3	71.438 (6.022)	72.873 (7.943)	8.630 (0.970)	9.242 (1.639)	(crash)	210.274 (17.788)
Phone 4	45.787 (205.695)	n/a	14.820 (3.331)	n/a	108.287 (289.971)	n/a
Phone 5	15.578 (3.190)	n/a	2.402 (2.051)	n/a	32.122 (23.438)	n/a

	Baseline		**Topline**	
	app	*low-level*	*app*	*low-level*
Phone 1	60.264 (11.343)	54.706 (11.987)	8.295 (3.477)	5.366 (0.697)
Phone 2	(crash)	27.041 (6.199)	4.607 (3.179)	2.848 (0.960)
Phone 3	30.257 (11.097)	17.898 (3.096)	4.752 (2.395)	1.924 (0.671)
Phone 4	16.420 (9.147)	n/a	4.791 (1.005)	n/a
Phone 5	4.085 (2.710)	n/a	0.376 (0.459)	n/a

than 50% in certain configurations, underscoring the efficiency of the low-level environment.

Even on newer devices, where the reduction in inference time might not be as drastic, the low-level environment demonstrates its value through more stable operation and the ability to utilize smaller buffer sizes. Phone 3 serves as a prime example of this benefit, where, despite not showing a significant decrease in inference time, the system stability and flexibility in buffer size selection are markedly improved.

This aligns with findings previously reported in the literature, where the superiority of low-level environments in terms of performance and flexibility has been noted [18]. The enhanced performance, combined with the greater range of buffer size options, not only facilitates the operation of more computationally intensive models but also enables the integration of neural processing within a broader spectrum of effects and control chains. However, it's important to acknowledge the limitations associated with low-level environments, such as LDSP, which necessitate root access (i.e., superuser privileges) and specific customizations to be fully operational.

Yet, beyond the specific settings required for its operation, LDSP does not fundamentally differ much from Oboe in its approach to handling audio data. Essentially, LDSP operates by bypassing certain software layers that become redundant in the context of running dedicated, high-priority audio applications, such as unnecessary mixer buffering. This streamlining contributes significantly to the performance enhancements observed in the low-level environment compared to the app-based approach. These findings suggest that with relatively minor architectural adjustments to the Android audio stack, it's possible to substantially enhance the audio, and particularly neural processing, performance across a wide spectrum of Android devices. This implies that the potential for dramatic performance improvements is not limited by the hardware capabilities of the devices but can be unlocked through optimized software design.

5.3. Choosing the Right Engine

Perhaps the most interesting insights arise from the comparative analysis of RTNeural and ONNX Runtime. Across all configurations, the three audio effect models we examined consistently performed better with RTNeural. This disparity is largely attributable to the call overhead associated with ONNX Runtime, which can be approximated on every phone by the inference times measured for the baseline model[14]. Except for the most advanced device (Phone 5), the mean inference time is perilously close to or exceeds the sampling period, rendering ONNX Runtime generally less suitable for models operating on a per-sample basis due to the significant overhead.

Interestingly, when models operate on a block basis (such as Model B), ONNX Runtime demonstrates improved outcomes, occasionally achieving real-time performance. Yet, even in these scenarios, RTNeural outperforms ONNX Runtime by a significant margin, making it the preferable option for implementation.

At first glance, these findings might prompt the dismissal of ONNX Runtime for real-time inference of neural audio models similar to those we examined. And this may seem a plausible ex-

[14] "With only 16 FLOPs, the baseline model's computational times are negligible, indicating that its run-time is primarily due to the call overhead.

tension of the results obtained by Stefani et al. in embedded classification tasks [8]. However, the performance of the topline model prompts a reassessment. Despite a computational complexity comparable to Model C, the most demanding among our audio effect models, the topline model runs in real-time across all devices and both environments. Remarkably, its inference times not only vastly undercut those of any other model evaluated with ONNX Runtime but are also on par with, or even superior to, the performance of the lightest model (Model A) under RTNeural. This efficiency is achieved despite the topline model's FLOPs per sample ratio being nearly six times higher.

The topline model's simple architecture—consisting of just two dense layers—might explain its unexpected efficiency, suggesting that even CPU delegates can effectively accelerate such configurations. Nonetheless, these findings hint that in embedded applications ONNX Runtime could be particularly well-suited for block-based models with significant computational demands, whereas RTNeural shines when processing single-sample data and lighter models.

6. CONCLUSION AND FUTURE WORK

This study embarked on an exploration of neural audio effects processing on Android smartphones, leveraging both RTNeural and ONNX Runtime across a variety of devices and environments. We demonstrated that all tested phones, including models up to 10 years old, are capable of running neural audio effects in real-time, with newer devices showing exceptionally low inference times. Our analysis underscored the superior performance of the low-level environment over traditional app-based environments, highlighting the potential for enhanced audio processing capabilities with minor architectural adjustments in the Android audio stack. The comparative study between RTNeural and ONNX Runtime revealed a significant performance edge for RTNeural, due to the lower call overhead. Yet, ONNX showed very encouraging results with larger models operating on a block basis.

The evolving landscape of neural audio and multimodal AI underscores the critical need for advanced neural audio processors. Responsive audio input/output is becoming increasingly vital and there is a pressing need for more open and flexible development frameworks on Android—i.e., lower level. While Google's AAudio represent a step in the right direction (see also [18]), our findings suggest that there is significant room for improvement. Enhancements in the coding and execution pipeline for audio apps could lead to substantial gains in performance and usability for newer devices, as well as democratize high-quality neural audio processing across a broader range of smartphones, even those that are not equipped with specialized hardware accelerators.

Looking ahead, we plan to delve into the potential of utilizing GPU and NPU acceleration for neural operations on newer phone models through specialized delegates, like NNAPI. This approach promises to unlock new capabilities in neural audio processing, by enabling the use of more advanced architectures that otherwise could not run in real-time. Our preliminary engagement with NNAPI has yielded mixed results and it was not included in this work. It necessitates a deeper analysis of various factors that could influence performance.

Moreover, our examination of ONNX Runtime will extend to larger and more complex models that carry out actual audio processing using advanced architectures. This exploration will also consider TensorFlow Lite, which has shown similar performance in related studies [8], to provide a comprehensive view of the best tools for neural audio effects processing.

Lastly, neural synthesis represents an exciting frontier for our future investigations. While many technical considerations overlap with those addressed in this study, the unique demands of synthesis models regarding architecture, computational load and real-time control mechanisms present a distinct research domain [12, 22]. Expanding our work into neural synthesis will further our understanding of the possibilities and challenges in advanced audio processing on mobile devices.

7. REFERENCES

[1] Abhishek Sehgal and Nasser Kehtarnavaz, "A convolutional neural network smartphone app for real-time voice activity detection," *IEEE access*, vol. 6, pp. 9017–9026, 2018.

[2] Francesco Mercaldo and Antonella Santone, "Audio signal processing for android malware detection and family identification," *Journal of Computer Virology and Hacking Techniques*, vol. 17, no. 2, pp. 139–152, 2021.

[3] Rateb mercaldo2021audio, Mohammed Shinoy, Mohamed Kharbeche, Khalifa Al-Khalifa, Moez Krichen, and Kamel Barkaoui, "Driver drowsiness detection model using convolutional neural networks techniques for android application," in *2020 IEEE International Conference on Informatics, IoT, and Enabling Technologies (ICIoT)*. IEEE, 2020, pp. 237–242.

[4] Andrey Ignatov, Anastasia Sycheva, Radu Timofte, Yu Tseng, Yu-Syuan Xu, Po-Hsiang Yu, Cheng-Ming Chiang, Hsien-Kai Kuo, Min-Hung Chen, Chia-Ming Cheng, et al., "Microisp: processing 32mp photos on mobile devices with deep learning," in *European Conference on Computer Vision*. Springer, 2022, pp. 729–746.

[5] Andrey Ignatov, Radu Timofte, Andrei Kulik, Seungsoo Yang, Ke Wang, Felix Baum, Max Wu, Lirong Xu, and Luc Van Gool, "Ai benchmark: All about deep learning on smartphones in 2019," in *2019 IEEE/CVF International Conference on Computer Vision Workshop (ICCVW)*. IEEE, 2019, pp. 3617–3635.

[6] Andrey Ignatov, Radu Timofte, William Chou, Ke Wang, Max Wu, Tim Hartley, and Luc Van Gool, "Ai benchmark: Running deep neural networks on android smartphones," in *Proceedings of the European Conference on Computer Vision (ECCV) Workshops*, 2018, pp. 0–0.

[7] Teresa Pelinski, Rodrigo Diaz, Adan L. Benito Temprano, and Andrew 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*, Miguel Ortiz and Adnan Marquez-Borbon, Eds., Mexico City, Mexico, May 2023, pp. 160–166.

[8] Domenico Stefani, Simone Peroni, Luca Turchet, et al., "A comparison of deep learning inference engines for embedded real-time audio classification," in *Proceedings of the International Conference on Digital Audio Effects, DAFx*. MDPI (Multidisciplinary Digital Publishing Institute), 2022, vol. 3, pp. 256–263.

[9] Soroush Mehri, Kundan Kumar, Ishaan Gulrajani, Rithesh Kumar, Shubham Jain, Jose Sotelo, Aaron Courville, and Yoshua Bengio, "Samplernn: An unconditional end-to-end neural audio generation model," *arXiv preprint arXiv:1612.07837*, 2016.

[10] Aäron van den Oord, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew W Senior, and Koray Kavukcuoglu, "Wavenet: A generative model for raw audio," *arXiv preprint arXiv:1609.03499*, 2016.

[11] Jesse Engel, Kumar Krishna Agrawal, Shuo Chen, Ishaan Gulrajani, Chris Donahue, and Adam Roberts, "Gansynth: Adversarial neural audio synthesis," *arXiv preprint arXiv:1902.08710*, 2019.

[12] Antoine Caillon and Philippe Esling, "Rave: A variational autoencoder for fast and high-quality neural audio synthesis," *arXiv preprint arXiv:2111.05011*, 2021.

[13] Eero-Pekka Damskägg, Lauri Juvela, Vesa Välimäki, et al., "Real-time modeling of audio distortion circuits with deep learning," in *Proc. Int. Sound and Music Computing Conf.(SMC-19), Malaga, Spain*, 2019, pp. 332–339.

[14] Marco A Martínez-Ramírez, Emmanouil Benetos, and Joshua D Reiss, "A general-purpose deep learning approach to model time-varying audio effects," *arXiv preprint arXiv:1905.06148*, 2019.

[15] Marco A Martínez-Ramírez, Emmanouil Benetos, and Joshua D Reiss, "Deep learning for black-box modeling of audio effects," *Applied Sciences*, vol. 10, no. 2, pp. 638, 2020.

[16] Alec Wright, Eero-Pekka Damskägg, Lauri Juvela, and Vesa Välimäki, "Real-time guitar amplifier emulation with deep learning," *Applied Sciences*, vol. 10, no. 3, pp. 766, 2020.

[17] Riccardo Simionato and Stefano Fasciani, "Fully conditioned and low-latency black-box modeling of analog compression," in *Proceedings of the International Conference on Digital Audio Effects*. DAFx Board, 2023.

[18] Victor Zappi and Carla Sophie Tapparo, "Upcycling android phones into embedded audio platforms," in *Proceedings of the International Conference on Digital Audio Effects*. DAFx Board, 2023.

[19] Jatin Chowdhury, "Rtneural: Fast neural inferencing for real-time systems," *arXiv preprint arXiv:2106.03037*, 2021.

[20] Teresa Pelinski, Rodrigo Diaz, Adán L Benito Temprano, and Andrew McPherson, "Pipeline for recording datasets and running neural networks on the bela embedded hardware platform," *arXiv preprint arXiv:2306.11389*, 2023.

[21] Alessio Balsini, Tommaso Cucinotta, Luca Abeni, Joel Fernandes, Phil Burk, Patrick Bellasi, and Morten Rasmussen, "Energy-efficient low-latency audio on android," *Journal of Systems and Software*, vol. 152, pp. 182–195, 2019.

[22] Kıvanç Tatar and Philippe Pasquier, "Musical agents: A typology and state of the art towards musical metacreation," *Journal of New Music Research*, vol. 48, no. 1, pp. 56–105, 2019.