A MODULAR FRAMEWORK FOR EFFICIENT SOUND RECOGNITION USING A SMARTPHONE

Matthias Mielke, Lars Weber*†*, Rainer Brück**

* University of Siegen, Microsystems Engineering Group, Siegen, Germany
† WEBER-SOFTWARE, Betzdorf, Germany

ABSTRACT

The identification of sounds is an important tool in ubiquitous and context aware applications. Today's smartphones are capable of performing even computational intensive tasks, like digital signal processing and pattern recognition. In this contribution an implementation scheme and a framework for sound recognition for smartphones are presented. A basic sound recognition flow consists of preprocessing, feature extraction, feature selection, classification, and action trigger. A flow is not hard coded but described in a JSON file and build dynamically by the framework. The framework itself is implemented in Java for the Android operating system. But specific algorithms can be realized in Java, C(++), and Renderscript for execution on the CPU, or in Filterscript for execution on a GPU. An example flow is presented and benchmark results are shown for Java-, C-, and Filterscript-implementations of Mel Frequency Cepstral Coefficients (MFCC). Recommendations for technology selection are made.

Index Terms— sound recognition, smartphone, GPU computation

1. INTRODUCTION

In the last years, a new class of computer has emerged: the smartphone. A smartphone provides sufficient resources even for demanding computational tasks, like digital signal processing [1] or sound recognition [3]. In combination with low energy consumption it is a good platform for various applications in ubiquitous and context aware computing. The analysis of the acoustic environment can be used in a variety of applications, among others assistive technology for people experiencing hearing impairment [4].

When surveying research in sound recognition it becomes obvious that different approaches and algorithms are used to identify sounds. Until now no "standard" approach exists that is capable of accurately recognize every type of sound. In reverse, when very accurate sound recognition is needed the algorithms must be tailored to the specific sound. With ubiquitous computing in mind this observation leads to the following two questions:

- How can different sound recognition flows be described and build flexibly (i.e. without hard-coding each flow)?
- What technologies allow efficient implementation of sound recognition on a smartphone?

A framework for easy and flexible implementation of sound recognition is presented in this contribution. Together with benchmark result of different implementation technologies it provides a good tool for implementing efficient sound recognition flows.

In the next section a small overview of sound recognition frameworks is given, followed by a description of the framework in section 3. In section 4 the extraction of Mel Frequency Cepstral Coefficients (MFCC) is explained which was used for the benchmark. The benchmarks results of different MFCC implementations and its impact on battery run time are presented in section 5. A conclusion closes the paper.

2. RELATED WORK

For sensing and analysing sound data, different approaches were already introduced in literature. An approach for a flexible framework was presented with the Modular Audio Recognition Framework (MARF) [5]. It is capable of analysing a continuous audio data stream, that is processed by a pipeline of algorithms consisting of the three steps preprocessing, feature extraction, and classification. The framework provides a plug-in API for easy addition of new algorithms. An Android port of the framework used for speaker recognition is introduced in [2], but no information about the execution time or the power consumption, both crucial parameters for continuous sound analysis, is stated. In addition it only offers the execution of one single pipeline of processing steps.

Another approach not specifically designed for sound recognition is introduced with the Funf Open Sensing Framework [6]. The framework samples, saves, and processes data from different sensors available in Smartphones, among others the microphone. But the sensor data is not sampled continuously rather at predefined intervals. Further more the sampled data is not evaluated on the smartphone but saved for later evaluation or send to a server for evaluation. The configuration of the framework is done using a JSON file.

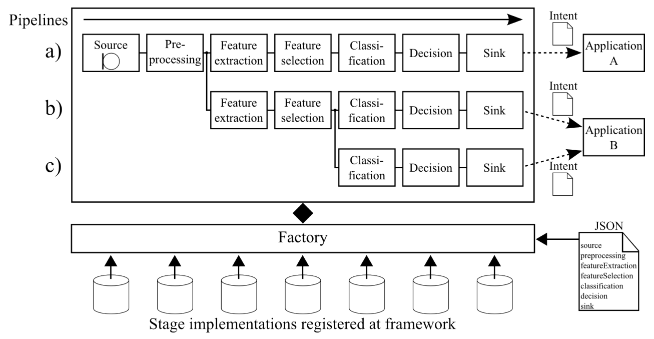

Fig. 1. Example configuration of different sound recognition flows running in the framework.

3. THE FRAMEWORK

The framework (named Android Audio Analysis (libAAA)) which is described here combines continuous sound recognition with the easy configuration using a file. It provides three basic functionalities: 1. a definition of basic processing stages and interfaces for each step, 2. the construction of a complete sound recognition flow defined by a configuration, and 3. the means to start and control the execution of the constructed flow.

A sound recognition flow in the framework consists of basic processing stages (see section 3.1). The concrete algorithms in the processing stages are not implemented in the framework directly. In fact the framework defines interfaces for the algorithms of the different stages. A new algorithm just needs to implement the appropriate interface and register at the framework. The construction of sound recognition flows is described in a configuration file (see section 3.2).

After a sound recognition flow is build from the configuration it is started in a background service, which manages the lifecycle of the flow.

3.1. Basic sound recognition flow

An application can instruct the framework to perform an audio classification task with a specific flow, by sending an intent with the description of the flow in the form of a JSON file (see section 3.2).

Each sound recognition flow consists of basic processing stages that are processed sequentially. The results of one stage is the input for the next stage. They form a pipeline (a complete pipeline is shown in Figure 1 a)). The seven stages are:

Source: the source stage provides access to the audio data. It sets the audio source, the sampling rate, and the sample depth. Framesize and overlapping are also set in this stage. Only one source may exist in the sound recognition flow but this source may be the root for a tree structure of different pipelines. The frames are passed to the preprocessing stage.

Preprocessing: in the preprocessing stage, transformations can be applied to the data from the source stage. Examples are windowing functions or low/high pass filter. If more than one transformation should be applied to the data preprocessing stages can be combined sequentially.

Feature extraction: in this stage the features necessary for the classifier are extracted. Different feature extraction algorithms can be run in parallel to form N features. The algorithms can extract features from a single frame or from a number of frames in parallel. In this way it is possible to utilize the GPU for feature extraction (for more details see section 4).

Feature selection: a subset of the features can be chosen for later classification in the feature selection stage. E.g. a Principal Component Analysis can be used to eliminate correlated features. Only one feature selection stage can be used in each pipeline.

Classification: here the actual classification is implemented. The framework itself only supports one classification stage per pipeline. Even though it is possible to build complex multi layered classifiers by instantiating and connecting different classifiers in the stage implementation.

Decision: on the basis of the classification results the decision stage determines which class(es) is(are) detected. E.g. Postprocessing and prioritization can be done in this stage.

Sink: the sink triggers an action depending on the result of the decision stage. An action may be e.g. the start of another application by sending an intent (Figure 1) or showing a note on the smartphone's screen.

Joint stages of different pipelines can be used together to minimize redundancy and processor utilization. An example is shown in Figure 1. Pipeline a) and b) share the source and the preprocessing stage, whose result is used by both. Additionally pipeline c) uses the same stages as b) until the classification. A fork can be introduced between stages resulting in a tree structure.

3.2. Configuration of a recognition flow

The construction of a pipeline is realized using a factory pattern. The stage implementations register with the factories to make themselves known, so that they become available for pipeline construction.

The actual pipeline is described using a JSON file (Javascript Object Notation). JSON is a text-based data format and can be used for storage, and interchange of data. The data in a JSON file are organized as attribute-value pairs. The file contains a textual representation of a pipeline. It consists of identifiers of the stages and the information which implementation(s) should be used for the stage.

An excerpt of a flow configuration is shown in Listing 1. In the excerpt the feature extraction (”featureExtraction”), feature selection (”featureSelection”), and the classification (”classification”) of a sound recognition pipeline are defined. Of particular importance is the keyword ”implementation” which is used to denote the actual implemen-

Listing 1. Excerpt of a flow configuration in JSON

```
                          ⋮
"featureExtraction": [
  {
    "elements": [
      {
        "featureCount": 13,
        "implementation": "de.uni_siegen.mse.aaa.↙
            featureextraction.MFCCFilterScriptMultiple↙
            ",
        "lift": 22.0,
        "melBandCount": 20
      }
    ],
  "featureSelection": [
    {
      "classification": [
        {
          "implementation": "de.uni_siegen.mse.aaa.↙
              NeuronalNetworkClassification",
          "serializedNeuronalNet": "eNq1WHls[...]Ckh0=",

                          ⋮
```

tation used for the stage. In the excerpt an MFCC implementation "MFCCFilterScriptMultiple" from the package "de.uni_siegen.mse.aaa.featureextraction" is used for the feature extraction. Additional keys are used to parametrize the implementation. E.g. key "featureCount" is used to set that 13 MFCCs have to be extracted. The results are passed through an empty feature selection stage to the classification. The implementation "NeuralNetworkClassification" from the package "de.uni_siegen.mse.aaa" realizes a Neural Network classifier. The actual configuration of the Neural Network is serialized and saved in the Base64-coded string after "serializedNeuronalNet". Binary data can be included in the flow configuration file through the Base64 coding.

The actual construction of a flow is done by a factory pattern. For each stage a factory exists at which the stage implementations must register so that they can be used to construct a sound recognition flow (see Figure 1).

The necessity to implement the stages' interfaces introduces an overhead compared to hard wiring a sound recognition flow. This overhead was measured using a Samsung Galaxy Nexus (two core CPU, 1 GB RAM) with Android 4.3. The processor's frequency was fixed at 350 MHz and a simple flow consisting of MFCC extraction and classification with a Neural Network was used. The flow was compared to the hard wired flow of an earlier experiment [3]. The measurement showed that the overhead introduced by the framework is 3.1 %. CPU utilization was 16 % during the measurement.

4. MFCC EXTRACTION USING GPU

4.1. MFCC extraction

MFCC are often successfully applied in speech and sound recognition. The extraction starts with a frame x with length N. At first the frame is processed with a preemphasis filter given by (1) to pronounce the high frequencies.

$$x'[n] = x[n] - 0.97 \cdot x[n-1],\ 1 \leq n < N. \tag{1}$$

The resulting vector x' is windowed with a Hamming Window $w[n]$ (3) and transformed into the frequency domain using a Discrete Fourier Transform (2).

$$X[m] = \sum_{n=0}^{N-1} w[n]x'[n]e^{-i2\pi nm/N},\ 0 \leq m < N. \tag{2}$$

$$w[n] = 0.54 - 0.46 \cdot cos(2\pi n/N). \tag{3}$$

The spectrum $X[m]$ is warped into the Mel scale by calculating the magnitude and the application of triangular Mel spaced filter bank H (see [7, p. 314ff.] for more details).

$$S[m] = log\left[\sum_{k=0}^{M-1} |X(k)|^2 H(k)\right],\ 0 \leq m < M. \tag{4}$$

At last a Discrete Cosine Transform (DCT) (5) is applied to the filter bank energies $S[m]$.

$$c[n] = \sum_{m=0}^{M-1} S[m]cos(\pi n(m+1/2)/M),\ 0 \leq m < M \tag{5}$$

The MFCC extraction is implemented in different technologies: Java, C, and Filterscript. Because Java and C are standard technologies and available for a multitude of platforms the implementations will not be explained here in detail. Available on the Android platform only are Renderscript and Filterscript. A Renderscript/Filterscript implementation is platform independent. It is compiled to machine code on the target device just prior to execution, resulting in highly optimized code for the specific target. Filterscript, a subset of Renderscript, can be used to implement algorithms for the Graphics Processing Unit (GPU).

4.2. MFCC on GPU

A Filtercript implementation consists of a kernel that is called for each element in an allocation, i.e. an array of objects of one type. The runtime manages allocation to processors, and scheduling and synchronization of the kernels. No synchronisation is possible inside a kernel. To achieve synchronization, an algorithm has to be split into different sequential kernels. Between two kernels the runtime performs the synchronization.

Two GPU implementations of the MFCC algorithm are implemented using Filterscript. Because of the little impact on the overall performance framing and windowing are not done in Filterscript but in the Java context. The *FilterScriptMultiple* implementation extracts MFCC from multiple frames in parallel (data parallelism), so that each processing core has a separate frame to work on. Each kernel implements the whole algorithm from preprocessing to DCT. Since

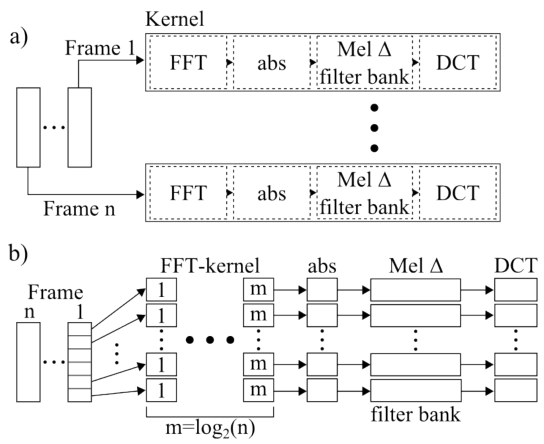

Fig. 2. Overview of the dataflow of the two GPU implementations. a) FilterScriptMultiple, b) FilterScriptSpawn.

each kernel implements the whole MFCC algorithm it is not necessary to synchronize data or execution between the GPU cores. The concept is illustrated in Figure 2 a). A drawback that comes with this implementation is that multiple frames must be buffered before they are processed all at once. This creates an additional processing delay.

The second implementation is the *FilterScriptSpawn* that processes a single frame with multiple processing cores in parallel (task parallelism). Each calculation step is implemented in an own kernel, that is executed for each element in the allocation. The elements of the allocation are processed in parallel. In total five different kernels are involved in that implementation: the four kernels depicted in Figure 2 b) and one kernel that controls the data flow and calls the kernels in the correct sequence.

The FFT is realized with a single kernel that is called for each stage of the FFT's butterfly structure. This results in $m = log_2(n)$ sequential executions of the kernel for an n-point FFT. A dedicated kernel is then used to calculate the absolute value of the FFT's results. The filter bank is realized in a dedicated kernel that is called once for each filter bank channel. Finally the DCT is performed in the last kernel.

The matrix H that describes the triangular filter bank and the DCT matrix are precomputed in Java once and passed into Filterscript context.

5. BENCHMARK

In an earlier experiment with a hard wired sound recognition flow the MFCC extraction was responsible for the largest fraction of the calculation cycle [3]. Based on this observation MFCC extraction was chosen for benchmarking the implementation technologies.

The benchmark was run on a LG Nexus 5 smartphone which is equipped with a quad core processor Qualcomm Snapdragon 800, 2 GB of main memory and an Adreno 330 GPU. Operating system is Android 4.4. To eliminate the influences of dynamic frequency scaling by the Linux Govenor and thermal management, the processor's frequency was fixed to 960 MHz (the highest frequency without throttling the processor because of heat).

Another source of influence on execution time of Java code is the Just in Time (JIT) compilation supported by Android's Dalvik VM. For a fast start up time the program code is just interpreted or compiled without optimization. Later the runtime recognizes often used program sections and optimizes them, which takes some time but improves performance significantly. To exclude the execution speed before JIT optimization the first 10 % of extractions are ignored.

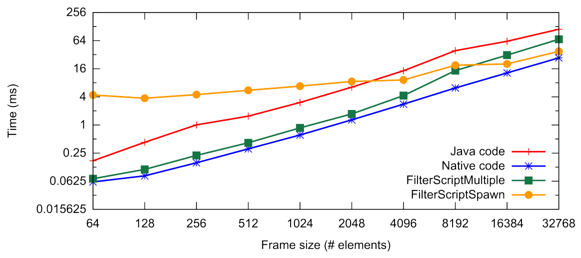

Fig. 3. Execution time of MFCC calculation for a single frame.

5.1. Extraction time

The implementation technology has direct influence on the performance of the MFCC. To get an inside into that the extraction time of MFCC implemented in Java, native Code (C) and Filterscript was measured and compared. The input for the measurement was provided by a sound file that contains a recording of a siren of an emergency vehicle in Germany. The recording has a duration of six seconds, sample rate of 16 kHz and depth of 16 bit. 13 MFCC were extracted for different frame sizes ranging from 64 samples (4 ms) to 32768 samples (2048 ms). For each technology and frame size the mean of the extraction times of 480 frames was calculated. For the FilterScriptMultiple implementation the number of frames was multiplied by the number of processing cores in the GPU (32) and the measured time was also divided by that number. In that way the number of frames to be processed by each core was equal to the other implementations so that the overhead of the measurement itself was still the same. The extraction times depending on frame size are plotted in Figure 3. Native code written in C showed the best performance for all frame sizes. It provides the shortest extraction time. FilterScriptMultiple performs nearly as good as the native code. Surprisingly the FilterScriptSpawn implementation performs rather poor for small frame sizes; even worse than the Java implementation. The Java implementation is in average five times slower than the native implementation.

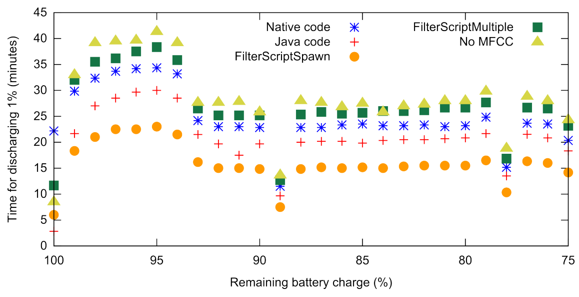

Fig. 4. Energy consumption of the different MFCC implementations. Each data point indicates the runtime that was achieved with 1 % of battery charge.

5.2. Measurement of energy consumption

But not only the extraction time is of interest for evaluating the implementations. The power consumption is especially important for mobile applications and was measured in a second experiment for each technology. Since the power consumption of the Nexus 5 cannot be measured directly (battery not removeable) the time for discharging the battery to 75 % of it's capacity (8.5 Wh) was measured for each implementation. The smartphone was set to flight mode and all other applications were closed using the Android task manager to minimize the impact on the runtime measurement. In this test the microphone's audio data was sampled with 16 kHz frequency and 16 bit depth. 512 samples long frames (32 ms) were processed to extract 13 MFCC. The base energy consumption was assessed using an empty flow. In this flow the data was just captured but not processed and only passed from one stage to the next.

Figure 4 shows the discharge times for the different implementations. The x-axis shows the battery charge and the y-axis the time it takes to spend one percent of the battery charge. As expected the empty flow (green triangle) achieves the longest runtime, followed by FilterScriptMultiple, the native and Java implementation. The FilterScriptSpawn has the highest energy consumption. Table 1 shows the average time for discharging (t_d) and the energy consumption per frame. Even though the calculation time of MFCC for one single frame is longer using the FilterScriptMultiple than Native code (see 5.1) its power consumption is less than half of the native code.

Implementation	**t_d (min)**		**Energy**
	25 %	**1 %**	**per frame (μJ)**
Native code	640.3	25.61	853
Java Code	535.2	21.41	2105
FilterScriptSpawn	413.8	16.53	4340
FilterScriptMultiple	696.7	27.87	338
No MFCC	739.3	29.57	—

Table 1. Power consumption of different technologies.

6. CONCLUSION

The contribution of this paper is twofold. First, a modular framework for implementation of sound recognition flows is presented. Through the definition of a pipeline and introduction of interfaces for each pipeline stage, the framework can be easily tailored to different needs. New algorithms in different technologies can be easily added without recompilation of the framework. The framework introduces a small overhead (3.1 %) compared to a hard-coded sound recognition flow. Second, by choosing an appropriate implementation technology the power consumption of a pipeline can be optimized. If a large amount of data needs to be processed a GPU implementation should be considered. For the MFCC the FilterScriptMultiple has lowest power consumption even though the native implementation needs less time to extract MFCC. A drawback of the FilterScriptMultiple implementation is that it processes multiple frames at once that introduces delay. Providing the best trade-off between efficiency and power consumption a native implementation should be chosen if extraction time and power consumption are an issue.

REFERENCES

[1] A. J. Bianchi and M. Queiroz, "On the performance of real-time dsp on android devices," in *Proc. of the 9th Sound and Music Computing Conf*, 2012, pp. 113–120.

[2] Y. Chen, E. Heimark and D. Gligoroski, "Personal Threshold in a Small Scale Text-Dependent Speaker Recognition," in *Proc. of the Intl Symp on Biometrics and Security Technologies*, 2013, pp. 162–170.

[3] M. Mielke and R. Brück, "Smartphone application for automatic classification of environmental sound," in *Proc. of the 20th Intl Conf Mixed Design of Integrated Circuits and Systems*, 2013, pp. 512–515.

[4] M. Mielke, A. Grünewald, and R. Brück, "An assistive technology for hearing-impaired persons: Analysis, requirements and architecture," in *Proc. of the 35th Annual Intl Conf of the IEEE Engineering in Medicine and Biology Society*, 2013.

[5] Serguei A. Mokhov, "Choosing best algorithm combinations for speech processing tasks in machine learning using marf," in *Advances in Artificial Intelligence*, vol. 5032 of *LNCS*, pp. 216–221. Springer, 2008.

[6] N. Aharony, W. Pan, C. Ip, I. Khayal, and A. Pentland, "Social fmri: Investigating and shaping social mechanisms in the real world," *Pervasive Mob. Comput.*, vol. 7, no. 6, pp. 643–659, 2011.

[7] X. Huang, A. Acero, and H.-W. Hon, *Spoken Language Processing: A Guide to Theory, Algorithm and System*, Prentice Hall, 2001.