VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

POROVNÁVÁNÍ DVOU AUDIO VZORŮ JAKO ANDROID APLIKACE

COMPARISON OF TWO AUDIO EXAMPLES AS AN ANDROID APPLICATION

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — **SULTAN ZHANTEMIROV**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. IGOR SZŐKE, Ph.D.**
SUPERVISOR

BRNO 2017

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav počítačové grafiky a multimédií Akademický rok 2016/2017

Zadání bakalářské práce

Řešitel: **Zhantemirov Sultan**
Obor: Informační technologie
Téma: **Porovnávání dvou audio vzorů jako Android aplikace**
Comparison of Two Audio Examples as an Android Application
Kategorie: Softwarové inženýrství

Pokyny:

1. Nastudujte základy implementace aplikací pro Android (SDK, NDK)
2. Nastudujte základy rozpoznávání řeči pomocí porovnávání dvou vzorů.
3. Implementuje knihovnu, která porovná dva vzory audia pomocí techniky DTW. Pokud to bude možné, využijte dostupných knihoven. Modely a referenční implementaci dostanete již připravené v C++ a Pythonu.
4. Tam kde to půjde, využijte low level implementace (NDK, Neon). Vyhodnoťte vypočetní náročnost jednotlivých bloků.
5. Vyhodnoťte celkovou úspěšnost a hlavně časovou a paměťovou náročnost. Navrhněte směry dalšího vývoje.
6. Vyrobte A2 plakátek a cca 30 vteřinové video prezentující výsledky vaší práce.

Literatura:

- Dle pokynů vedoucího

Pro udělení zápočtu za první semestr je požadováno:

- Body 1 a 2, a část bodů 3 a 4 ze zadání.

Podrobné závazné pokyny pro vypracování bakalářské práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Szőke Igor, Ing., Ph.D.**, UPGM FIT VUT
Datum zadání: 1. listopadu 2016
Datum odevzdání: 17. května 2017

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav počítačové grafiky a multimédií
612 66 Brno, Božetěchova 2

doc. Dr. Ing. Jan Černocký
vedoucí ústavu

Abstrakt

Tato bakalářská práce je zaměřená na implementaci Android aplikace pro porovnávání audio vzorků pomocí speciálních technik. Cílem výsledného programu je jednoduchá demonstrace algoritmu porovnávání a jeho urychlení. První část této práce se zabývá teoretickou analýzou a návrhem porovnávání, zatímco další jsou věnovány implementaci, urychlení algoritmu a jeho testování v rámci hotového demonstračního programu.

Abstract

This bachelor thesis deals with an implementation of an Android application for comparison of two audio examples by using special technics. The goal of the final program is a simple demonstration of comparison algorithm and it's acceleration. The first part of this thesis is concerned with theoretical analysis and suggestion for the comparison, while following parts review implementation, acceleration and testing of the algorithm in scope of final demo application.

Klíčová slova

Android, aplikace, audio, vzor, Java, porovnávání

Keywords

Android, application, audio, sample, comparison, Java

Citace

ZHANTEMIROV, Sultan. *Porovnávání dvou audio vzorů jako Android aplikace.* Brno, 2017. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Igor Szőke, Ph.D.

Porovnávání dvou audio vzorů jako Android aplikace

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením Ing. Igora Szőke, Ph.D. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Sultan Zhantemirov
17. května 2017

Poděkování

Rád bych poděkoval Ing. Igorovi Szőkemu, Ph.D., za vedení této bakalářské práce, odbornou pomoc a rady, které mi při její tvorbě významně pomohly.

Obsah

Kapitola 1

Úvod

Technika, která umožňuje rozpoznávání a porovnávání mluveného slova, se vyvíjí od poloviny minulého století. Od té doby se výrazně zlepšila a zlepšovat se samozřejmě ještě bude. Možnost zadávat textové informace pomocí hlasu nabídlo široké použití v různých oblastech spotřební elektroniky.

Cílem této bakalářské práce je porovnání dvou audio vzorů pomocí speciálních technik - samotné porovnávání audia by mělo probíhat při chodu Android aplikace. Její úlohou není nic jiného než nahrávání audio záznamů a zobrazení výsledné shody uživateli.

Ovšem rychlost, s jakou se provádí porovnání audia v nativním jazyce Java, nebude s ohledem na omezenou výkonnost a paměť současných mobilních zařízení postačující pro běžné použití. Právě proto je dalším cílem této práce urychlení algoritmu porovnání audia a následné srovnání rychlosti tohoto algoritmu napsaného v nativním jazyce a jazyce, který zajistí rychlejší porovnání. Rychlejší způsob by měl být aplikovatelný na běžných Android zařízeních.

Autora této bakalářské práce především zaujala možnost získání zkušenosti spojené s vývojem aplikací pro systém Android a optimalizací algoritmu porovnání. Výsledná aplikace by pak měla za úkol porovnávat vstupní audia za co možná nejkratší časový interval. Zadáním je najít takový způsob řešení implementace, který by umožnil dosáhnout předem stanoveného požadavku času a rychlosti, uplatnit tento způsob kde to půjde a vyhodnotit časovou a paměťovou náročnost ve srovnání s referenčním řešením.

Práce je rozdělena do čtyř kapitol. Kapitola 2 popisuje analýzu, která byla provedena nad požadavky, a specifikaci těchto požadavků; v kapitole 3 je popsána realizace jednotlivých kroků algoritmu porovnání; v kapitole 4 se popisuje návrh programu v prostředí systému Android; v kapitole 5 se podrobněji rozebírají prováděné pokusy o urychlení částí algoritmu, současně jsou hodnoceny i získané výsledky urychlení.

Kapitola 2

Analýza a specifikace požadavků

Jaké kroky zahrnuje porovnání audio vstupů a které detaily této operace stojí za zmínění? Tato otázka bude rozebrána během této kapitoly. Zanalyzujeme potřebné kroky a na základě této analýzy vytvoříme kostru pro specifikaci požadavků, které má výsledný demonstrační program splnit.

2.1 Hlavní cíl při porovnání audia

Úlohou této bakalářské práce je implementace algoritmu pro porovnání dvou audio vstupů. Aby bylo možné snadno používat demonstrační program na mobilních zařízeních, je potřeba tyto operace zrychlit.

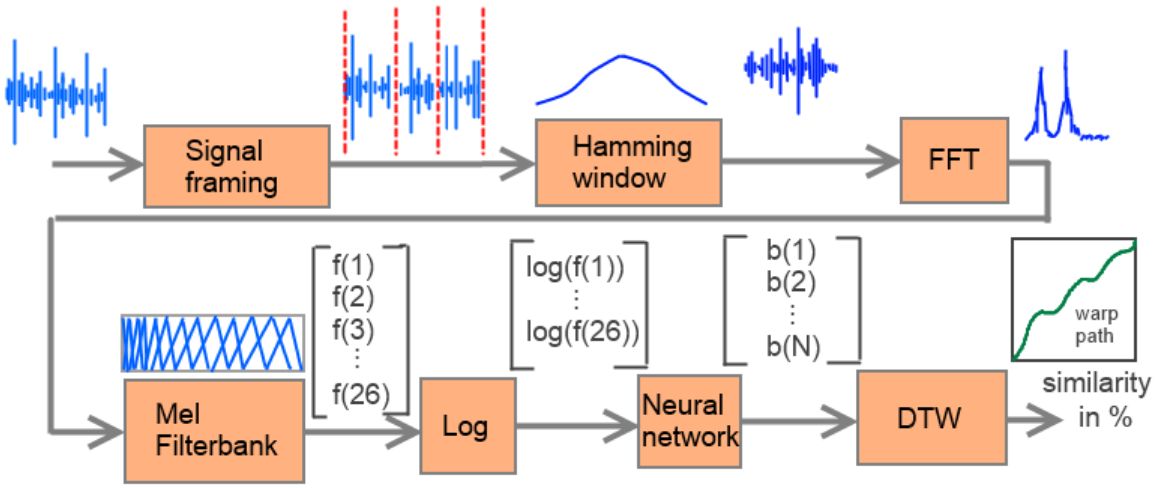
Máme velice omezený dostupný výkon a s tím i paměť pro potřebné výpočty. Proto se v dalších kapitolách budou rozebírat možnosti, jak zrychlit výpočet ne díky optimalizaci a přepsání částí algoritmu, ale díky zrychlenému provádění kódu na výpočetní jednotce mobilního zařízení.

2.2 Technika porovnání audio vstupů

Než začneme práci na implementaci hlavního algoritmu – Dynamic Time Warping[10] (česky dynamická časová transformace, dále jen DTW), o kterém bude podrobněji popsáno níže, je potřeba převést vstupní data do potřebného formátu. V této kapitole nebudeme uvádět přílišné detaily implementace – v abstraktnější formě popíšeme potřebné kroky a operace pro to, aby bylo možné získat rozumný výsledek po aplikaci DTW. Nejzasadnější kroky jsou na obrázku 2.1.

Porovnávají se audio vzorky ovšem až na samém konci cyklu porovnání. Pro začátek je potřeba nahrát audia, navzorkovat je na jednotlivé rámce, převést je do frekvenční oblasti pomocí Fourierovy transformace, udělat mezikroky oříznutí nepotřebných dat a také dostat amplitudu každého rámce. Dále je třeba získat filtr banky každého rámce, spočítat logaritmus těchto banek a přesměrovat výsledek na vstup neuronové sítě, která spočítá tzv. bottleneck příznaky[odkaz].

Řešení zadání této práce zahrnovalo i použití techniky DTW - jádrem tohoto algoritmus je matice (přesněji matice transformace a matice vzdálenosti). Z toho vyplývá, že pro účely této práce jsou nezbytné implementace rychlého násobení matic a matice a vektoru nejen v jazyce Java, tedy v původním jazyce pro operační systém Android, ale také v jiném jazyce, který zajistí rychlejší provádění kódu.



Obrázek 2.1: Obecné schéma postupu při porovnávání audio vstupů

Nad těmito rámci je již možné aplikovat techniku DTW, která spočítá celkovou shodu těchto dvou audio vstupů rámec po rámci nehledě na rozdíl v jejich celkovém počtu.

Výše uvedený popis jednotlivých kroků potřebných pro korektní porovnání dvou audií je dostatečný pro vytvoření celkové představy o tom, které operace se budou provádět nad vstupy. Aby bylo zřejmé: dále jsou uvedeny definice algoritmů a technik bez podrobných implementačních popisů, tedy pouze teoreticky. Implementační podrobnosti jsou pak v kapitole 3.

2.3 Vzorkování vstupního signálu

Než začneme porovnávat audio vstupy, musíme pochopit, jak budeme tyto vstupy porovnávat. Je potřeba rozdělit každý audio signál na diskrétní části. Nejlepší variantou, jak to udělat, je navzorkovat vstupní signál na jednotlivé rámce sestávající z množiny vzorků.

Ovšem porovnání každého rámce prvního signálu s patřičným rámcem druhého nebude příliš přesné v tom případě, mají-li tyto dva signály různou dobu trvání. Jinými slovy, při různé délce těchto signálů není každý prvek prvního signálu příslušen každému z prvků druhého signálu. Právě touto vyskytovací vlastností se zabývá DTW, podrobněji popsaný v podkapitole 3.7.

Analýza signálů pomocí Fourierovy tranformace dává srozumitelné výsledky pouze pro nehybné signály. Audio signál je nepřetržitě měnitelný – předpokládejme, že se na krátkých časových úsecích signál moc nemění (ovšem to je pouze předpoklad, je zřejmé, že i během krátkého časového úseku není signál úplně nehybný). Aby bylo možné aplikovat techniku analýzy ve spektrální oblasti, provádíme vzorkování původního signálu.

2.4 FFT

Fast Fourier Transform[3], česky rychlá Fourierova transformace, je algoritmem pro výpočet diskrétní Fourierovy transformace. Tato technika se dá využít od digitálního zpracování signálů až po řešení parciálních diferenciálních rovnic – v našem případě je optimální právě digitální zpracování. Níže je uveden vzor 2.1 pro obecnou FT:

$$X_k = \sum_{n=0}^{N-1} x_n e^{\frac{-i2\pi}{N}nk} \quad k = 0, ..., N-1 \tag{2.1}$$

Obecná a základní Fourierova transformace je charakterizována výpočetní složitostí $O(n_2)$ – to znamená, že se celkový počet kroků FT bude v nejhorším případě rovnat kvadrátu počtu vzorků (N násobení a N-1 sčítání). Na rozdíl od obecné transformace má její rychlejší varianta, tedy rychlá Fourierova transformace, výpočetní složitost $\mathcal{O}(N\log(N))$.

V rámci této práce byla použita nejpoužívanější varianta RFT – Cooley-Tukey algoritmus[12], který se liší od základní metody tím, že v každém kroku dělí transformace na dvě části. Tyto části můžou být zpracovávány paralelně, ovšem v naší práci, v níž budeme používat délku FT = 256 prvků, nebude mít paralelní zpracování takový efekt jako v případě větší délky.

Konečná implementace FFT je prováděna v souladu s Cooley-Tukey algoritmem a inspirovaná již existujícím řešením v jazyce Java bez možnosti paralelního zpracování.

2.5 Mel banky

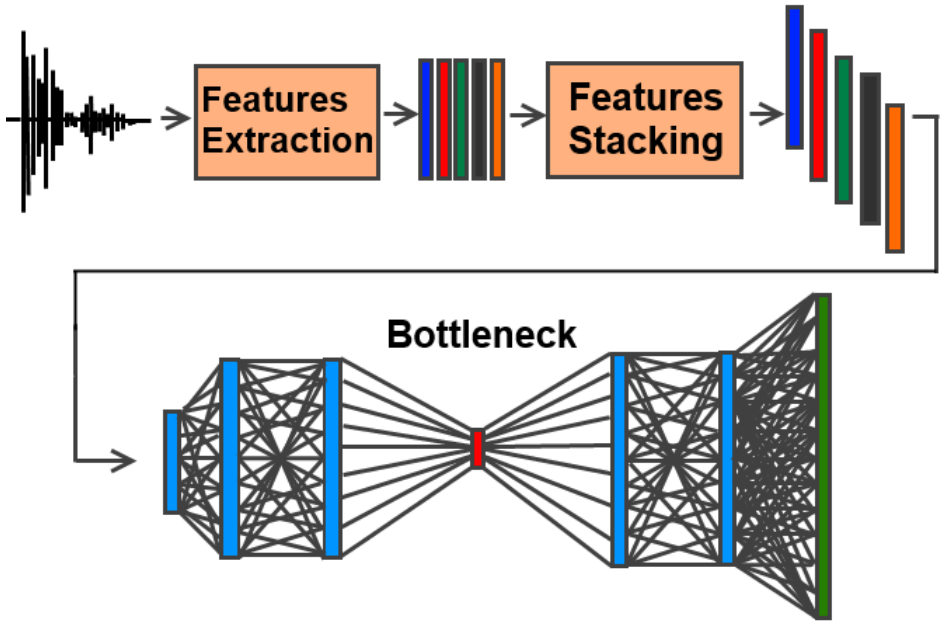
Mel filtr banky[4] jsou trojúhelníkovými filtry s délkou, která se rovná polovině délky FFT. Většina hodnot každého filtru obsahuje samé nuly, kromě té oblasti, na kterou je ten který filtr zaměřen.

Hlavní úlohou aplikování těchto filtru nad výkonovým spektrem rámce - získání energii spektra. Je nutné vynásobit každou banku výkonovým spektrem a následně složit koeficienty - jakmile je to uděláno, máme čísla indikující, kolik energie je v každé filtr bance. Obdobným způsobem funguje lidské ucho, které je schopné rozlišovat nižší frekvence lépe, vysoké - hůře.

2.6 Neuronová síť

Současně s Fast Fourier Transform byla využita obecná neuronová síť, zdrojové kódy, které byly poskytnuty vedoucím práce Ing. Szőkem, Ph.D.

Tato síť sestává ze tří vrstev různé dimenzionality, uprostřed kterých je tzv. bottleneck vrstva. Tato síť není nic jiného než matice, vektory a sigmoida. Výstupem neuronové sítě jsou bottleneck příznaky[11] (přesný počet vstupních a výstupních příznaků je uveden v kapitole 3.6). Tyto příznaky povolují zvýšit přesnost rozpoznávání řeči. Nad bottleneck příznaky se následně aplikuje algoritmus DTW (podrobněji je popsán v následující podkapitole).



Obrázek 2.2: Obecné schéma neuronové sítě s bottleneck vrstvou

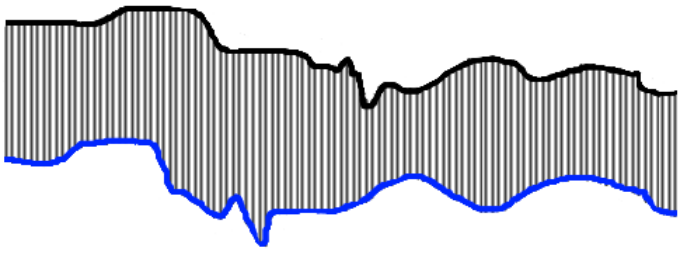
2.7 Dynamic Time Warping

Pro hledání optimální shody mezi dvěma časovými posloupnostmi je možné využít techniku DTW. Tento algoritmus byl poprvé použit pro účely rozpoznávání řeči, ale později našel uplatnění i v jiných oblastech.

V našem případě se časovými posloupnostmi rozumí vstupní signály – referenční a nahrávaný. Obecně je pro výpočet shody postačující tzv. Euklidova vzdálenost – jednoduché měření vzdálenosti mezi patřičnými prvky dvou posloupností.

Ovšem Euklidova metrika pro porovnání dvou audií není moc užitečná, protože tato audia můžou, ale nemusejí být vyrovnaná dle časové osy. Abychom byli schopni tyto zvukové posloupnosti porovnat mezi sebou, máme za úkol deformovat časovou osu jedné (nebo obojí) posloupnosti, aby bylo ve výsledku patrné lepší vyrovnávání.

Pro klasifikaci a určení shody mezi dvěma časovými řadami se obecně používá výpočet Euklidovy vzdálenost mezi prvky těchto řad – součet kvadrátu vzdálenosti mezi n-tým prvkem první posloupnosti a n-tým prvkem druhé posloupnosti. Výsledek výpočtu tohoto druhu vzdálenosti je na obrázku 2.3.



Obrázek 2.3: Porovnávání prvků pomocí Euklidovy vzdálenosti

Nicméně použití Euklidovy vzdálenosti má jeden zásadní nedostatek: v případě, jsou-li dvě časové řady stejné a jedna z nich je posunuta podle časové osy, považuje Euklidova metrika tyto řady za rozdílné.

Právě kvůli tomuto jevu se v oblasti rozpoznání řeči zavedlo použití techniky DTW. Tento algoritmus umí překonat výše popsaný nedostatek a poskytuje názorné měření vzdálenosti mezi časovými řadami nehledě na globální a lokální posuny dle časové osy.

Obrázek 2.4: Porovnávání prvků pomocí DTW

Z obrázku je vidět, že každému prvku jedné posloupnosti je přiřazen stejný nebo větší počet patřičných prvků druhé posloupnosti. Tato vlastnost je pro porovnání rámců dvou audio signálů ideální.

Tento algoritmus je jádrem techniky porovnání dvou zvukových posloupností. Má výše popsané výhody a podle předpokladu zabírá největší výpočetní čas celého porovnání audia. Dá se ovšem lehce přepsat do jakéhokoli běžného jazyka, včetně jazyka vhodného pro urychlení - tak se zkrátí čas potřebný na získání výsledku porovnání.

2.8 Typy prováděných operací

V této podkapitole se podrobněji rozebírá, které typy násobení a sčítání jsou používány v jednotlivých krocích. Zároveň se zde zjišťuje a uvádí, zda ze předpokladu bude mít aplikování nástrojů pro urychlení přínos.

Vzorkování neboli operace, při níž jsou vstupní signály rozděleny na jednotlivé vzorky, nebude dle předpokladu příliš výhodná, protože se při těchto krocích neprovádí žádné operace násobení, pouze kopírování určitých úseků vstupního pole hodnot. Navíc se neprovádí pouze operace vzorkování – po každém oříznutí vstupu se okamžitě přesouváme k dalším zásadním krokům: k Fourierově transformaci a aplikování Mel filtru.

Rychlá Fourierova transformace zahrnuje celou sadu operací násobení vektorů, obzvlášť při provedení násobení rámce časovým oknem Hamminga. V ostatních částech tohoto algoritmu se vypočítává jenom součet hodnot ve zpracovávaném vektoru.

Při aplikování Mel filtrů znovu dochází k tomu, že se jednotlivé hodnoty dvou vektorů násobí mezi sebou, ovšem v rozmezí <1, 26> (26 filtrů je standard – podrobněji popsáno v podkapitole 3.5). Pro tak malý rozměr vektorů není důvod používat rychlejší způsob, protože se i tento typ operace v nativním jazyce Java provádí za co nejkratší dobu. Detaily časové náročnosti jsou popsány v podkapitole 5.5.

Neuronová síť je vhodná pro provádění všech vnitřních operací rychlejším způsobem než v Javě – jsou tam jak násobení matic, což je ideální pro urychlení, tak násobení vektorů a matice vektorem - poslední dvě operace provádějí při průchodu vstupních dat vrstvami - vrstvami tedy jsou matice, vektory a sigmoidy.

Posledním krokem v porovnání audio signálů je DTW – algoritmus je ideální pro případ urychlení, v první řadě díky násobení a sčítání prvků vektorů při výpočtu distance mezi prvky matic.

2.9 Specifikace požadavků

V této podkapitole popisujeme zjištěné znalosti a požadavky z podkapitoly předcházející. Jak je patrné na základě dosavadních zjištění, hlavním požadavkem je rychlost. Přesněji, rychlost provádění algoritmu porovnání audio signálů a zároveň i to, jak dlouho daný proces probíhá.

Předpokládejme, že varianty těchto algoritmů jsou základní (neberou se v potaz varianty Fast DTW [5] a tak dále) – z toho vyplývá, že je nutné urychlit provádění napsaného kódu na zařízeních a zkrátit výsledný čas, potřebný na vykonání porovnání vstupů. Jednou z možností je použití speciálních vestavěných nástrojů a knihoven třetích stran pro lineární algebru.

Fourierova transformace by měla být implementována ve stylu *in-place*, to znamená, že by se nesměly vytvářet zbytečné objekty v paměti zařízení během provádění FT.

DTW by měl umět porovnávat nejen patřičné body jednotlivých vstupních posloupností (vektorů), ale také i podvektory těchto posloupností. Jinými slovy: mělo by se správně provádět porovnání koeficientů jednotlivých bodů obou posloupností mezi sebou.

Celkový požadavek na backend výsledné aplikace je rychle provádět algoritmus porovnání zvukových vstupů, tj. uživatel by neměl dlouho čekat na výsledky tohoto porovnání. Toho se dá docílit realizací časově nejnáročnějších kroků algoritmu způsobem rychlejším, než je referenční způsob v jazyce Java.

V následující kapitole jsou popsány některé z těchto knihoven, nejvýznamnější pro účely této práce je pak nástroj RenderScript[1].

[1] https://developer.android.com/guide/topics/renderscript/compute.html

Kapitola 3

Implementace

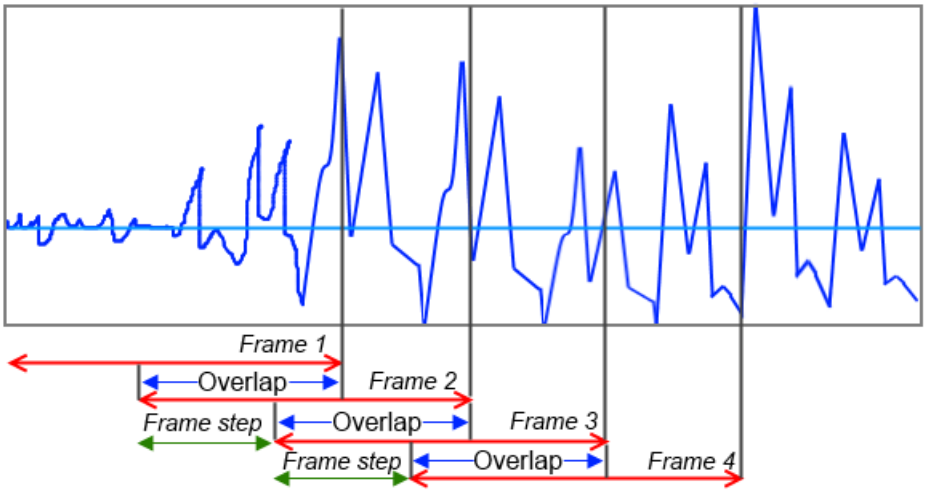
Tato kapitola se zabývá popisem použitých programátorských technik a navržených kroků v přechozí kapitole. Každá podkapitola má podrobnější popis a zdůvodnění jednotlivých kroků v celém algoritmu porovnání audia.

3.1 Vzorkování signálu

Signál se obvykle vzorkuje do rámců délky 20–40 ms[6]. Krok rámce je obvykle 10 ms, což vyhovuje celkovému pojetí této práce.

Tyto hodnoty byly získány jako kompromis mezi tím, že potřebujeme dělit vstupní signál na rámce velmi často, a tím, že každý ten rámec musí mít dostatečné frekvenční rozlišení (vstupní signál musí být stabilní).

Například když si vezmeme signál s frekvencí 8 kHz, bude se délka rámce rovnat (8000 * 0,025) 200 vzorkům, krok rámce = 80 vzorků. Obrázek 3.1 graficky popisuje tento příklad.



Obrázek 3.1: Příklad audia s frekvencí 8 kHz

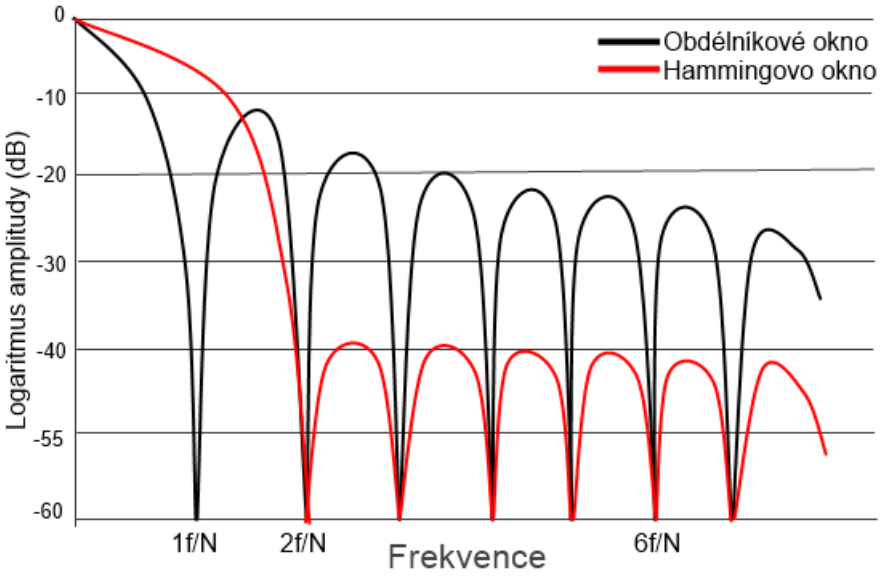
V případě kroku rámce 10 ms se první rámec začíná v 0. vzorku, další rámec se začíná v 80. vzorku a tak dále. V této práci byla za délku rámce považována hodnota v 25 ms, což je standardní při rozpoznávání řeči obecně a platí to i v našem případě.

3.2 Hammingovo časové okno

Pro omezení vlivu tzv. prosakování (leakage) energie ve výsledném spektru signálu je potřeba před samotným provedením FT vynásobit signál časovým oknem – v našem případě Hammingovým oknem.

Standardně (při obecném vzorkování signálu) je časovým oknem obdélníkové okno. Narozdíl od Hammingova, Hanningova a dalších nestandardních oken se obdélníkové okno charakterizuje největší úrovní prosakování.

Časové okno Hammingovo je modifikací okna Hanningova a má ve srovnání se standardním obdélníkovým oknem větší vliv na utlumení bočních laloků[8] – viz obrázek 3.2.



Obrázek 3.2: Srovnání obdélníkového a Hammingova oken

Hammingovo okno se definuje pomoci dvou koeficientů – α a β a má téměř konstantní potlačení v celém pásmu.

$$w(n) = \alpha - \beta\, cos(\frac{2\pi n}{N-1}) \tag{3.1}$$

Koeficient zesílení hlavního vrcholu v případě Hammingova okna se rovná 0,54. Z toho vyplývá, že druhý koeficient $\beta = 1 - 0{,}54 = 0{,}46$. Níže je uveden konečný vzorec 3.2.

$$w(n) = 0,54 - 0,46\, cos(\frac{2\pi n}{N-1}) \tag{3.2}$$

Po vynasobení rámce oknem Hamminga je již možné provádět transformaci do frekvenční domény - kroky FT jsou rozepsány v následující podkapitole.

3.3 Diskrétní Fourierova transformace

Hlavním důvodem k použití tohoto algoritmu v naší práci je převod diskrétního signálu z časové oblasti (domény) do frekvenční oblasti (domény).

Výstupem diskrétní transformace je frekvenční obsah (spektrum), jehož každá hodnota obsahuje dvě součásti – fázi a amplitudu (úhel a délka vektoru přiměřeně). Pro účely porovnání vzorků potřebujeme pouze amplitudu, fázi je možné vynechat.

Tedy po vynásobení rámce signálů Hammingovým časovým oknem se provádí Fourierova transformace, a současně s tím se vypočítá výkonové spektrum jako součet kvadrátu reálných a imaginárních složek.

Pro naše účely byla použita volně šířená knihovna pod autorstvím Damiena Di Fedea (licence GNU Library General Public) – třídy `FFT.java`[1] a `FourierTransform.java`[2] poskytují veškeré potřebné metody pro výpočet rychlé Fourierovy transformace.

3.4 Výkonové spektrum

Po aplikaci rychlé FT nad každým rámcem audia bylo také vypočítáno výkonové spektrum patřičného rámce, protože hlemýžďovitá část vnitřního ucha vibruje v různých bodech na základě frekvence vstupní zvukové vlny. Podle oblasti vnitřního ucha, která vibruje, informují různé nervy mozek o určitých frekvencích ve zvuku. Výkonové spektrum tím pádem vykonává podobnou práci – identifikuje, které frekvence jsou přítomné v daném rámci[9].

Pro výpočet výkonového spektra byl použit následující vzorec 3.3:

$$P(X) = \sum_{n=1}^{N} (X_{n_i} * X_{n_r}) \quad 1 \leq n \leq N \tag{3.3}$$

kde:

- P je výsledné výkonové spektrum
- X je výstupní spektrum FT
- N je počet prvků spektra FT
- X_{n_i} je imaginární složka prvku spektra FT
- X_{n_r} je reálná složka prvku spektra FT

3.5 Mel filtry

Po provedení operace popsané v předchozí podkapitole obsahuje výkonové spektrum stále většinu zbytečných informací pro další kroky porovnání. Lidské ucho například není schopné slyšet rozdíl mezi dvěma frekvencemi, které jsou ve spektru velmi blízko u sebe. Čím vyšší je frekvence, tím složitější je porovnávání lidským uchem.

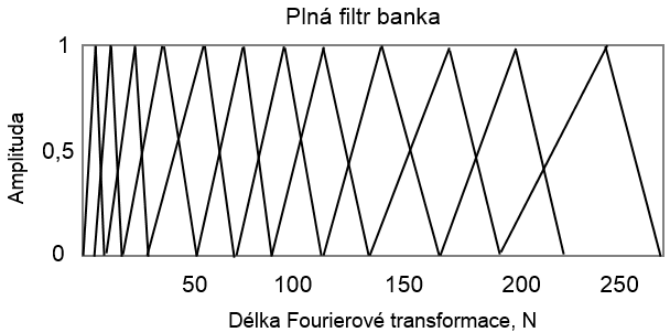
Právě z toho důvodu má smysl aplikování Mel filtr bank nad tímto spektrem. Obecně řečeno: určité skupiny binů ze spektra se skládají, aby se zjistilo, kolik je energie v dané frekvenční oblasti.

[1] https://www.acsu.buffalo.edu/ dmj32/Minim-2.2.0/src/ddf/minim/analysis/FFT.java
[2] https://www.acsu.buffalo.edu/ dmj32/Minim-2.2.0/src/ddf/minim/analysis/FourierTransform.java

Z toho plyne, že první Mel filtr je velmi úzký a indikuje, kolik je energie ve frekvenční oblasti 0 Herz. Čím větší je frekvence, tím širší je další Mel filtr. K určení toho, jak správně rozdělit frekvenční osu na filtr banky a jak široký má být každý filtr, slouží vzorec 3.4 konverze z frekvence do rozsahu Mel filtr banky. Standardní počet filtrů je 26.

$$M(f) = 1127 \ln (1 + f/700) \tag{3.4}$$

Po vynásobení každé filtr banky výkonovým spektrem a následném složení získaných koeficientů jsou na výstupu 26 čísel - dá se z nich zjistit, kolik energie se nacházejí v každé bance.



Obrázek 3.3: Příklad Mel filtr banek

Nad tímto součtem 26 čísel se vypočítá logaritmus, a to kvůli struktuře lidského ucha, které neslyší hlasitost v lineárním rozsahu. Právě logaritmická operace mění energii filtr bank na hodnotu, která více odpovídá tomu, co obvykle slyší lidské ucho. Po provedení operace logaritmu se také provádí normalizace energií filtr banek.

Výpočet Mel filtr banek

Máme například audio signál s frekvencí 8 kHz. Aby bylo možné získat filtr banky, je potřeba vybrat nejnižší a nejvyšší úrovně frekvence. Podle Nyquist teorémy je pro případ s frekvencí 8000 Hz vrchní úroveň omezena 4000 Hz, spodní může být 0 Hz. Ovšem v praxi se vybírá nenulová hodnota, v implementaci této práce se tato hodnota rovná 300 Hz dle rekomendací při zpracovávání signálu s frekvencí 8 kHz. Celkový počet filtrů obecně začíná od hodnoty 20, pro kompatibilitu s neuronovou sítí (podrobněji v podkapitole 2.6) byla vybrána hodnota 24.

Pomocí vzorce 3.4 byly nejnižší a nejvyšší frekvence překonvertovány[7] do 401.97 a 2146.07 Melu přiměřeně. Spočítáme tedy zbývající 24 Mel hodnoty, které budou lineárně rozdělené mezi hodnotami 401,97 a 2146,07. Ve výsledku se tyto hodnoty rovnají:

$$m(i) = 401,25; 469,04; 536,13; 603,21; ...; 1944,83; 2011,91; 2078,99; 2146,07;$$

Použijme vzorec 3.5 pro konverze znovu do Herzů:

$$f = 700(e^{\frac{m}{1127}} - 1) \tag{3.5}$$

$$f(i) = 300,00; 361,32; 426,42; 495,50; ....; 3231,40; 3472,51; 3728,41; 4000,00;$$

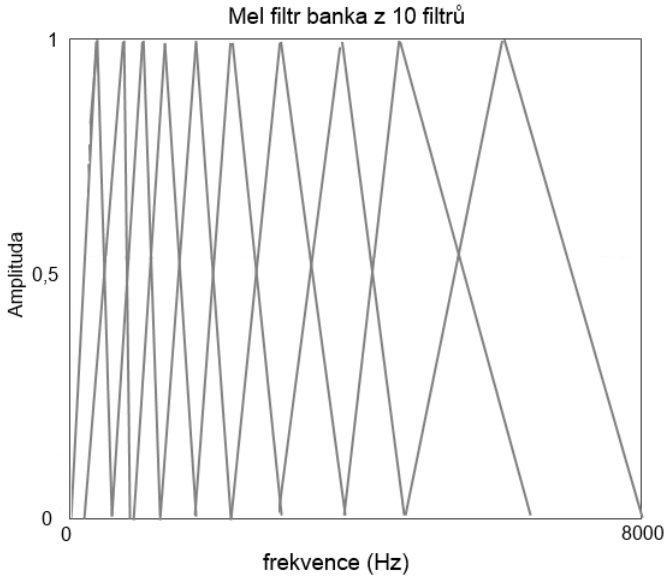
Je patrné, že první a poslední hodnota frekvence jsou právě ty, které byly definovány dříve. Dále je potřeba zaokrouhlit získané hodnoty frekvence na jednotlivé FFT biny, proto je potřeba znát délku FFT (256 prvků v případě 8 kHz) a frekvenci audia (8 kHz).

$$f(i) = floor((fftLength + 1) * f(i)/samplerate) \tag{3.6}$$

Po aplikaci vzorce 3.6 dostaneme následující hodnoty FFT binů:

$$f(i) = 12, 14, 16, 18, 21, 23, ..., 89, 96, 103, 111, 119, 128$$

Poslední hodnota filtr banky končí na 128, což odpovídá 8 kHz s FFT délky 256 prvků (bere se pouze jedná polovina hodnot, ta druhá je symmetrická). Dalším krokem je vytváření filtrů. První filtr začíná v prvním bodě, má vrchol ve druhém bodě a končí ve třetím. Druhý filtr začíná v druhém bodě, má vrchol ve třetím a končí v čtvrtém a tak podobně. Ukázková filtr banka na obrázku 3.4



Obrázek 3.4: Filtr banka v rozsahu frekvence

Tato filtr banka obsahuje 10 filtrů, začíná se na 0 Hz a končí se na 8000 Hz. V praxi zřejmě začíná první bod na 300 Hz.

3.6 Obecná neuronová síť

Jak bylo zmíněno výše, algoritmus a všechna potřebná data použité neuronové sítě byla poskytnuta vedoucím práce Ing. Szőkem, Ph.D. V této podkapitole se podrobněji zkoumá průchod vstupního signálu vrstvami této sítě.

Aby bylo možné zajistit generování bottleneck příznaků, je v první řadě potřeba načíst parametry neuronové sítě, přesněji jejích vrstev.

Obsahem vrstev použitých v programu neuronové sítě není nic jiného než matice, vektory a sigmoidy. Tyto hodnoty sítě je potřeba načíst, což provádí knihovna OpenCSV (podrobněji je popsána v podkapitole 4.3), která provede rozbor hodnot uložených v souboru. Veškeré hodnoty ze souboru budou přečtené do lokálních atributů.

Vstupní hodnoty jsou naskládány do vrstev sítě. Z těchto 24 vstupních hodnot Mel filtrů každého rámce signálu bottleneck vyprodukuje vrstva neuronové sítě bottleneck příznaky. Hlavní účel výsledných příznaků je vyšší přesnost jednotlivých slov ve vstupní řeči, která jsou reprezentována ve formě energií výkonového spektra.

Výsledné příznaky patřičného rámce se předávají pro porovnání do algoritmu DTW, který je podrobněji popsán v následující podkapitole.

3.7 Dynamic Time Warping

Obecný popis algoritmu a cíle jeho použití byly popsány v podkapitole 2.7. V této části bude podrobněji popsána implementační část.

Máme například dvě časové řady Q délky n a C délky m.

$$Q = q_1, q_2, ..., q_i, ..., q_n;$$

$$C = c_1, c_2, ..., c_i, ..., c_n;$$

Prvním krokem je sestavení matice vzdáleností d s rozměrem n krát m, ve které je prvek x_{ni} vzdáleností *d(qi, cj)* mezi body q_i a c_j.

Obvykle se používá Euklidova vzdálenost: $d(q_i, c_j) = (q_i - c_j)^2$, jenže pro účely této práce byla použita kosinová vzdálenost[3], protože obecná varianta tohoto algoritmu počítá vzdálenost mezi jednotlivými body – v našem případě je každý rámec signálu vektorem, obsahujícím vektor místo jednoho bodu. Pro tento případ je vhodnější použít kosinovou vzdálenost pro počítání nad vícedimenzionálními vektory (tj. matice).

Každý prvek d_{ij} matice vzdáleností d odpovídá vyrovnávání mezi body q_i a c_j. Dále je potřeba sestavit matici transformace/deformace D, jejíž každý prvek se počítá dle vzorce 3.7:

$$D_{ij} = d_{ij} + min(D_{i-1j}, D_{i-1j-1}, D_{ij-1}) \tag{3.7}$$

Jakmile je matice transformace D vyplněna, je finálním krokem sestavení optimální cesty transformace/deformace a výpočet vzdálenosti DTW (ceny cesty).

Optimální cesta transformace W je sada vedlejších prvků matice. Sada zadává shodu mezi Q a C. Cesta W minimalizuje celkovou vzdálenost mezi Q a C. K-tý prvek cesty W je určen vzorcem 3.8:

[3] https://en.wikipedia.org/wiki/Cosine_similarity#Definition

$$w_k = (i,j)_k,\ (w_k) = (q_i, c_j) = cosinedist \tag{3.8}$$

kde kosinová vzdálenost je určená vzorcem 3.9:

$$cosinedist(\vec{x},\vec{y}) = 1 - \frac{\sum_{i=1}^{N}(x_i * y_i)}{\sqrt{\sum_{i=1}^{N}(x_i^2) * \sum_{i=1}^{N}(y_i^2)}} \tag{3.9}$$

Tím pádem:

$$W = w_1, w_2, ..., w_k, ..., w_K;\ max(m,n) <= K < m+n,$$

kde

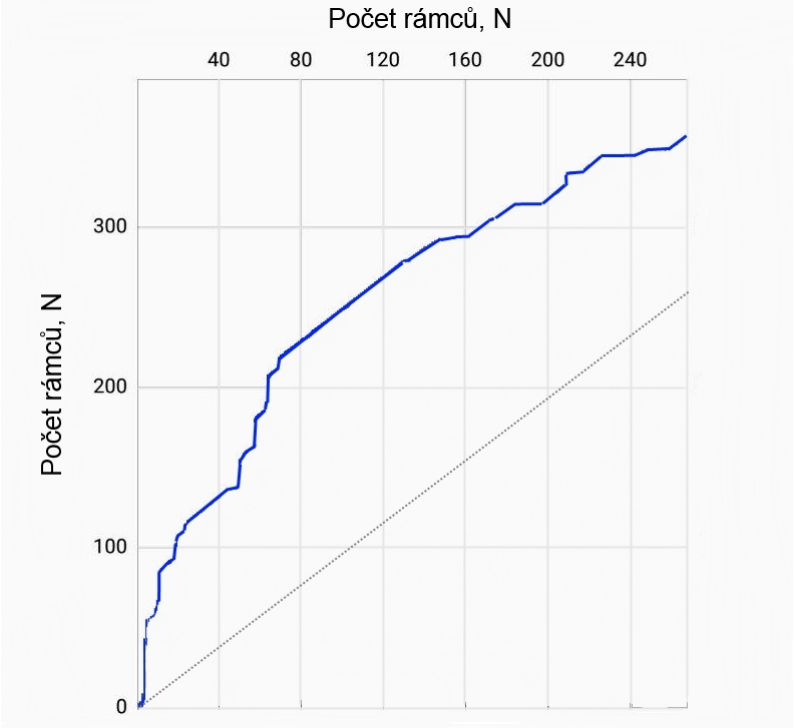
- K je délka cesty

Ze všech možných cest transformace nás zajímá právě ta, která minimalizuje vzdálenost DTW (cenu cesty). Tato vzdálenost mezi dvěma posloupnostmi se počítá na základě optimální cesty transformace pomocí vzorce 3.10:

$$DTW(Q,C) = min\left\{\frac{\sum_{k=1}^{K} d(w_k)}{K}\right\} \tag{3.10}$$

Je nezbytné dodat, že složitost klasického DTW algoritmu je kvadratická – $\mathcal{O}(nm)$, protože algoritmus musí projít každý prvek matice transformace.

Na obrázku 3.5 je znázorněná cesta deformace, na vstupu jsou dva signály různé délky trvání. Šedou čarou je zobrazená cesta deformace v případě úplné shody vstupů.



Obrázek 3.5: Cesta deformace po provedení DTW

Kapitola 4

Vývoj aplikace v systému Android

Tato kapitola se zabývá popisem aplikace potřebné pro provádění experimentů a následný sběr statistik a popisem výsledné demonstrační aplikace pro porovnání audio vstupů.

4.1 Program pro sběr statistik

První aplikace by měla sloužit pro sběr statistik na mobilním zařízení. Mobilní zařízení mohou být postavená na různých typech architektury procesoru, nicméně se musejí ovládat operačním systémem Google Android[1].

Pod statistikami se rozumí: model a architektura procesoru zařízení, na kterém se vykonává tato aplikace, a hlavně doba, po kterou trvalo provádění té nebo jiné operaci násobení.

Očekávané chování

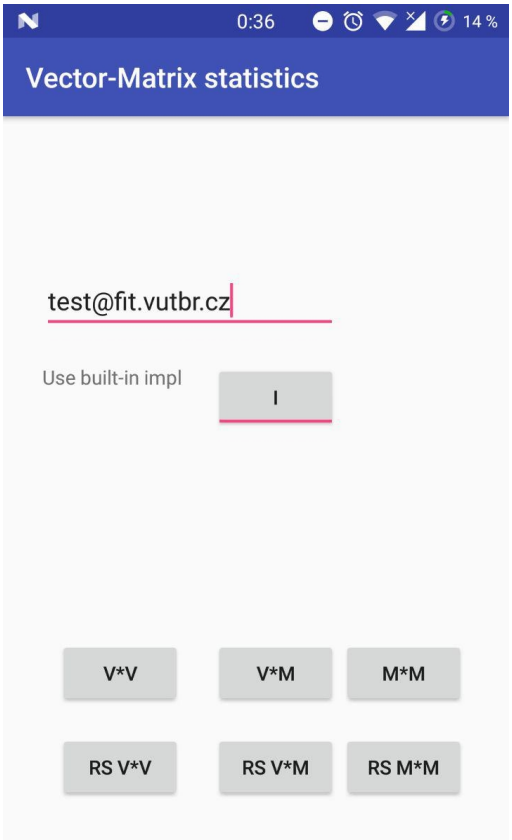
Uživatel zadá vstupní data (e-mail pro zaslání výsledků času potřebného pro výpočet) – v této fázi není od uživatele více údajů potřeba. Aplikace by také měla obsahovat různá tlačítka pro snadné označení jednotlivých algebraických operací (text na tlačítkách). Uživatel spustí stisknutím jednoho z těchto tlačítek zvolenou operaci.

Mezi těmito tlačítky je i tlačítko přepínání. Tento checkbox určuje typ používané Java metody – buď vestavěné v jazyce Java (například násobení matic použitím `for` cyklů), nebo specializované metody knihovny (JBLAS[2]). Jsou také dostupná tlačítka pro zahájení běhu algebraických operací v RenderScriptu. Na obrázku 4.1 je vidět výsledná podoba testovacího programu.

Po úspěšném provedení zvolené operace zašle aplikace na uvedený e-mail oznámení o době strávené touto operací (v milisekundách) a s názvem modelu procesoru zařízení. Pomocí tohoto testovacího programu lze určit časový rozdíl mezi vestavěnými funkcemi a funkcemi z knihoven. Současně bude poskytnuta informace o CPU zařízení, aby bylo možné posuzovat výkonnost dle typu použitého procesoru.

Aplikace je poskytovatelem všech nutných operací lineární algebry a byla použita na pěti různých zařízeních pro účely určení časového rozdílu mezi nativními a specializovanými metodami. Ve výsledku by měl tento program sloužit jako základ pro další vývoj demonstrační aplikace, blíže popsané v následující podkapitole.

[1] https://www.android.com
[2] http://jblas.org



Obrázek 4.1: Finální verze aplikace pro sběr statistik

4.2 Demonstrační program

V této části práce bude popsána očekávaná funkcionalita a vnější podoba konečného demonstračního programu.

Aplikace

Program implementovaný pro platformu Android by měl poskytovat uživatelské rozhraní pro nahrávání vstupu od uživatele ve formě audia, podporovat poslední aktuální verzi systému Android a interpretovat vstupní uživatelská data na výstupní ve formě procentuální shody audio vstupů.

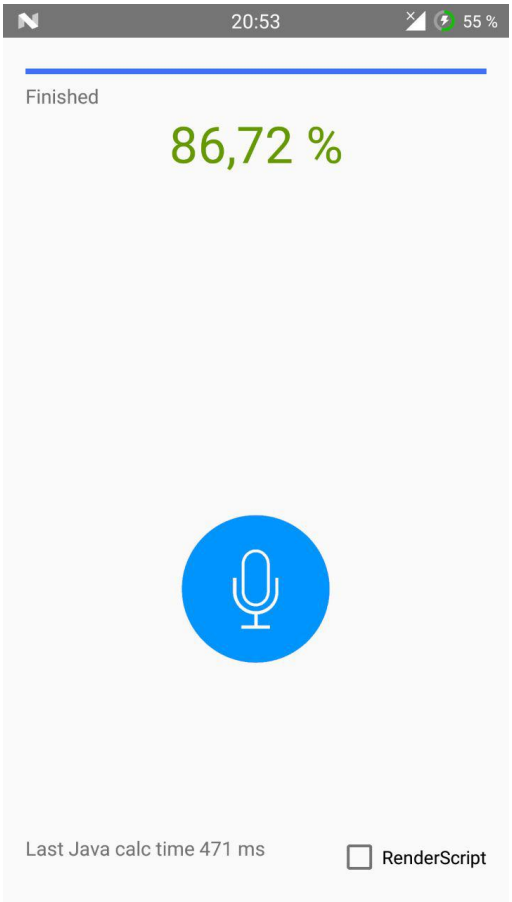
To znamená, že uživatel postupně nahraje dvě audia po sobě, více akcí od něj není potřeba. Posloupnost kroků pro porovnání nahraných audio vzorů se začne provádět okamžitě po ukončení nahrávání vstupů, tedy dvou audií. Tyto dva audio vzory můžou být libovolné, nicméně rozumné doby trvání.

Vnější podoba

Rozhraní testovací aplikace by mělo být docela jednoduché – hlavní okno jediné instance typu Activity s malým počtem grafických prvků. Samotné prvky by měly mít dostatečně velký rozměr. V souvislosti s předcházejícím požadavkem ohledně aktuální verze operač-

ního systému Android je použit Material Design[3] pro výslednou přehlednost a podporu potřebných grafických prvků. Po úspěšném provedení porovnání vstupů ukáže daná aplikace shodu, vyjádřenou procentuálně.

Jako další možnosti aplikace vidí autor této práce uvažovat o jediném hlavním okně programu, ve kterém bude pouze jedno tlačítko pro záznam a porovnání nahraných vzorů zvuku. Okno by také mělo obsahovat několik políček pro zobrazení informací o průběhu, stavu a výsledku prováděných operací s rozhraním aplikace. Na obrázku 4.2 je vidět výsledná podoba programu.



Obrázek 4.2: Demonstrační program pro porovnání audio vstupů

V případě verze Androidu 6 a novější po otevření hlavního okna programu by se měla objevit žádost o poskytnutí oprávnění uživatelem. Tato oprávnění jsou potřeba pro úspěšné nahrávání zvuku a uložení dočasných souborů.

Všechny případné výjimky a chyby by se měly uživateli ukázat, aby měl alespoň základní představu o chodu algoritmu. Tyto informace se zobrazují pomocí grafické komponenty Android Snackbar.

Pro úspěšnou realizaci algoritmu porovnání zvukových vstupů je nutné zajistit především nahrávání zvuku, transformaci FFT, Filter Banking a Dynamic Time Warping. Tyto a ostatní mezikroky byly popsány podrobněji v předchozí kapitole.

[3]https://developer.android.com/design/material/index.html

4.3 Použité knihovny

Kromě frameworku Android byly pro účely práce použity následující vestavěné knihovný a volně šířené knihovny třetích stran.

RenderScript support.v8

Pro podporu vykonávání kódu v prostředí RenderScript existuje několik verzí. Nejpopulárnější je právě balíček verze support.v8[4], jejíž výhodou je kompatibilita se zařízeními Android verze 2.3 (9. verze API) a vyšší/novější.

Druhou významnější verzí je modernější balíček android.renderscript, který je kompatibilní s verzí Android 3.0 (11. verze API) a vyšší. Má na rozdíl od support.v8 řadu nových možností především pro renderování 3D grafiky, což ale není v našem případě nijak přínosné.

Proto bylo rozhodnuto používat ve výsledné demonstrační aplikaci starší verzi pro vyšší kompatibilitu. Nevýhodou je ale nutnost přidání knihovny do složky libs, což vede k nárůstu rozměrů instalačního APK-souboru.

FFT

FFT je volně šířená knihovna pod autorstvím Damiena Di Fedea s licencí GNU Library General Public. Tento nástroj provádí rychlou Fourierovu transformaci metodou in-place. To znamená, že se původní hodnoty ve vstupech vyměnují nově spočítanými, nedochází k vytváření zbytečných hodnot. Na vstup je nutné dodat pole délky 2^n.

Práce s .wav

Pro učely nahrávání zvuku do .wav souborů a jejich správného dekodování byly použity Java třídy dekodování .wav balíčku audio-analysis[5] s licencí GNU Lesser GPL a třídy nahrávání a zápisu do .wav souboru balíčku simplesound[6] s licencí Apache License 2.0.

OpenCSV

OpenCSV[7] je otevřená jednoduchá knihovna, jejímž cílem je rozbor hodnot uložených v externích souborech. Každý z těchto souborů sestává z řádku, ve kterých jsou položky odděleny středníkem.

Formát CSV vyžaduje, aby byla každá položka oddělena čárkou, jenže pro účely této práce je vhodnější používat středník (čárka se používá pro definici hodnot vrstev s vektory, maticemi a sigmoidami).

OpenCSV má možnost definování vlastního symbolu pro oddělení uložených hodnot, funkci ukládání SQL tabulek a jednotlivých hodnot do .csv souboru, což ovšem není v rámci této práce potřeba.

[4] https://developer.android.com/reference/android/support/v8/renderscript/package-summary.html
[5] https://code.google.com/archive/p/audio-analysis/
[6] https://code.google.com/archive/p/simplesound/
[7] http://opencsv.sourceforge.net

Kapitola 5

Experimenty

Jednou z podmínek pro úspěšnou realizaci požadavků popsaných v předchozí kapitole je správná volba nástrojů a knihoven, které by umožnily zkrátit veškerý čas strávený výpočty. Samozřejmě, zvolit správnou variantu je možné pouze po provedení některých testů a experimentů. Krátký popis jednotlivých nástrojů a navazující experimenty jsou popsány níže.

5.1 BLAS

Byl proveden výzkum dostupných knihoven pro lineární algebru, resp. pro násobení matice vektorem a násobení matic mezi sebou. Tyto knihovny se opírají o BLAS[1] a LAPACK[2], napsané v jazyce C.

Basic Linear Algebra Subprograms jsou podprogramy[3], jejichž cílem je provedení základních (v závislosti na úrovni BLAS) operací s vektorem a maticí (skalární součin, násobení a tak dále).

Tyto knihovny pocházejí z roku 1979 a později byly přepsány z Fortranu do jazyka C. Je nezbytné konstatovat, že LAPACK je větší knihovnou založenou na BLAS, zatímco ATLAS je portabilní BLAS se samooptimalizací.

Pro jazyk Java existuje široká škála dostupných knihoven – například Netlib a JBLAS. Poslední z těchto knihoven byla použita v rámci testování rychlosti provádění výpočetních operací nad maticemi.

5.2 Knihovna Rendersript

Rendersript je součástí operačního systému Android. Hlavní výhodou je provádění výpočetně náročných operací na mnohojaderných CPU a GPU (verze Android 4.2 a novější). RenderScript je primárně zaměřen na paralelní zpracování, i když jsou sekvenční výpočty ve většině případů rychlejší ve srovnání s analogem v nativním jazyce Java.

To, že se kód napsaný v jazyce podobném jazyku C standardu C99 a vykonávaný pomocí RenderScriptu provádí[2] na více jádrech centrálního procesoru nebo na grafickém akcelerátoru, dovoluje nezabývat se plánováním jednotlivých částí sekvenčního algoritmu a více se zaměřit na vylepšení důležitých částí tohoto algoritmu.

[1] http://www.netlib.org/blas/
[2] http://www.netlib.org/lapack/
[3] https://en.wikipedia.org/wiki/Basic_Linear_Algebra_Subprograms

Toto rozhraní pro programování aplikací (API) je především určeno pro rendering 3D grafiky, ovšem dá se použít i při obecných výpočetně náročných operací. Pro vývojáře jsou tři nástroje: relativně jednoduché rozhraní pro 3D grafiku s akcelerací hardwaru, rozhraní pro modelování složitých výpočtů (obdobný architektuře CUDA) a jazyk standardu C99, který byl použit i v této práci.

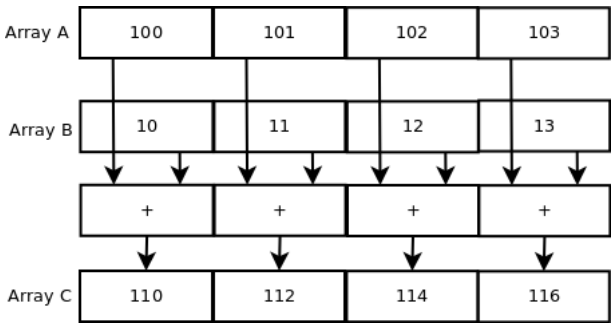
Samostatný kód algoritmu je uložen v souborech .rs, které se pak kompilují do meziformátu nezávislého na přístroji a rozmístí se v Java třídách. Pak existuje možnost volání funkcí přímo z Java kódu Android aplikace. Pro tyto účely je potřeba inicializovat tzv. RenderScript kontext, naalokovat paměť, nastavit si globální proměnné v .rs souboru a spustit provádění skriptu. Tyto kroky jsou detailně rozepsané dále.

Jakmile se voláním započalo provádění skriptu, kompiluje se tento skript do strojového kódu a současně se optimalizuje – není již třeba určovat konkrétní architekturu stroje. Samozřejmě že RenderScript není určen pro výměnu existujícího vysokoúrovňového rozhraní, proto je potřeba používat RS v případě, kdy obecné rozhraní nezajišťuje vysoký výkon.

Samotný kód se vykonává na centrálním nebo grafickém procesoru. Existuje-li příslušná přístrojová podpora, mohou se jednoduché skripty provádět na GPU. Pokročilejší skripty se provádějí na CPU (pokud je možné, pak na více jádrech). Všechno, co se vyžaduje od programátora, je zoptimalizovat algoritmus a správně rozdělit práce mezi několik jáder.

5.3 NEON

NEON[1] je rozšířením SIMD instrukcí (Single Instruction, Multiple Data). Podrobněji řečeno, NEON představuje skupinu instrukcí, které paralelně zpracovávají vektory, uložené v 64 a 128 bitových registrech.



Obrázek 5.1: Princip činnosti NEON v případě sečtení integerů. Převzato z http://www.fixstars.com/en/news/?p=125.

NEON je pouze obecný název pro danou implementaci, který se používá mimo specifikaci architektury. Tato instrukční sada se používá především pro zpracování obrazů a signálů, dekódování/kódování videa, syntézu.

Implementace NEON instrukce je použita ve všech ARM procesorech v sérii Cortex-A. Současné se můžou provádět do 16 operací. Existuje více způsobů, jak použít/napsat tyto instrukce.

- Použití vestavěných funkcí – poskytování rozhraní volání funkcí v jazyce C pro použití NEON operací.
- Assembler – ruční psaní určitých sekcí kódu.

- Vektorizace kompilátoru – možnosti vektorizace kompilátoru GCC pro automatické generování NEON kódu.
- Použití optimalizovaných knihoven – to jsou OpenMax DL[4], Eigen[5] a Ne10[6].

Detaily rozdílu mezi dvěma hlavními způsoby napsání NEON instrukcí (buď ručně pomocí asembleru, nebo za využití vestavěných funkcí) jsou snadnost čtení/psaní NEON kódu, kompatibilita a hlavně výkonnost kódu.

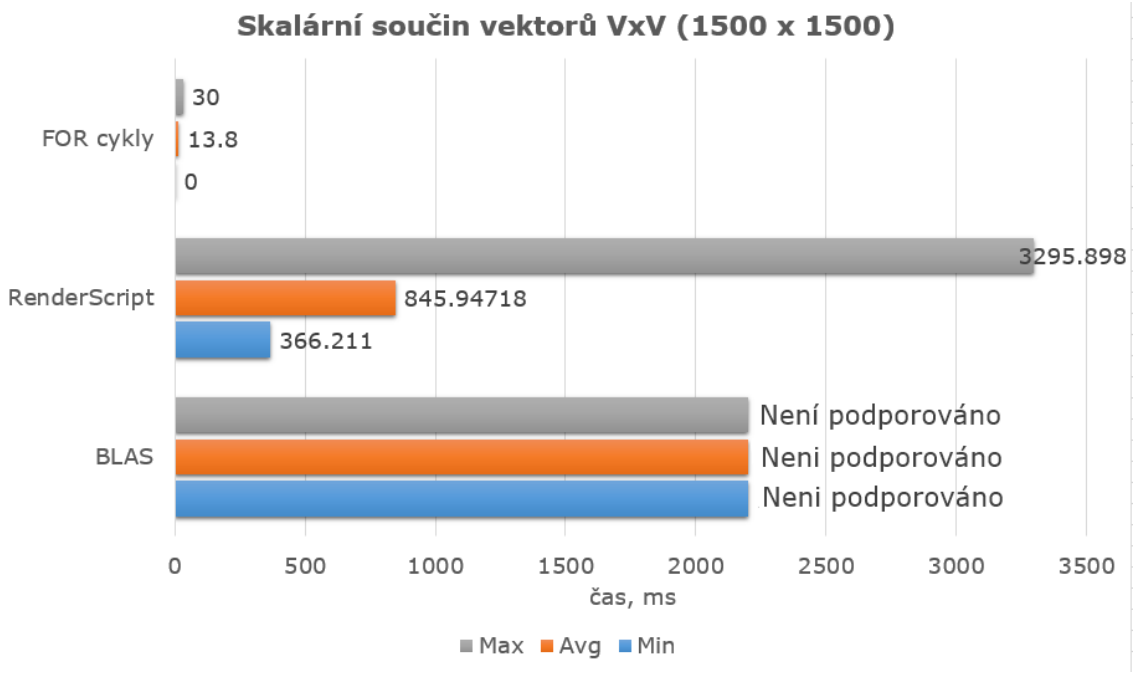
5.4 Srovnání rychlosti nástrojů

Po provedení experimentu a testu výše popsaných nástrojů je možné určit nejvhodnější variantu pro další použití v rámci aplikace.

Nejběžnějšími operacemi v krocích popsaných v kapitole 2.2 je samozřejmě operace lineární algebry, tedy násobení vektorů, vektoru a matice a také násobení matic.

Pro podrobnější zkoumání těchto tří hlavních operací byla zvolena forma experimentů. V první řadě bylo zjištěno, jak rychlé je násobení vektorů při použití BLAS a RenderScriptu.

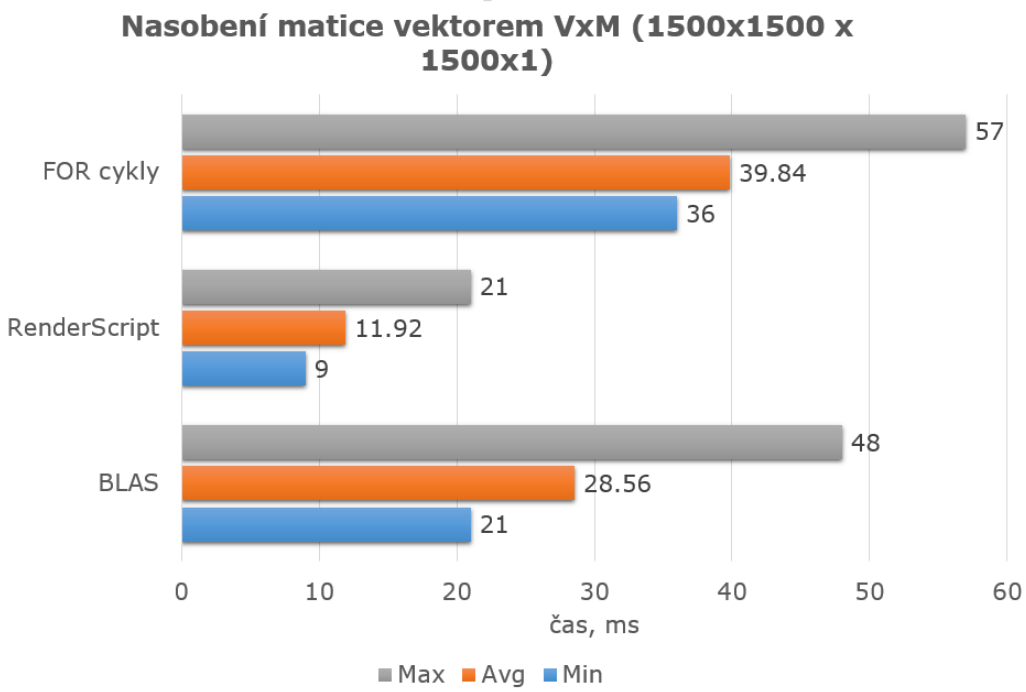
Velikosti vektorů se variovaly v prostoru <1, 1500> prvků, matice - <1500, 1500> prvků. Hodnoty těchto prvků se generovaly náhodně funkcí `rand()` vestavěné knihovny Math. Všechny následující testy a experimenty se prováděly na zařízení Samsung GT-i9505 (Galaxy S4) s čipem Snapdragon, počet iterací se rovnal 50. Výsledek prvního testu je znázorněn na obrázku 5.2. Relativně dlouhá výpočetní doba v případě RenderScriptu bude vysvětlena v následující podkapitole.



Obrázek 5.2: Násobení vektorů, každý o velikosti 1500 prvků

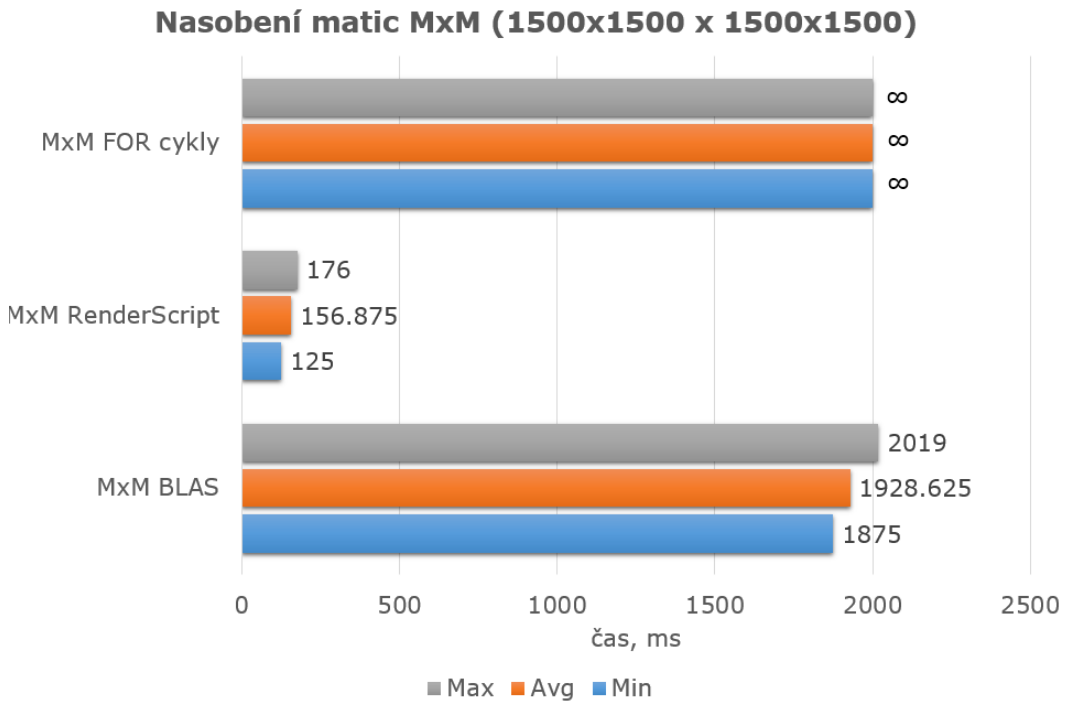
[4] https://www.khronos.org/openmax/dl
[5] http://eigen.tuxfamily.org/index.php
[6] https://projectne10.github.io/Ne10/

Následujícím objektem zkoumání byla operace násobení vektoru a matice. Hodnoty se také generovaly náhodným způsobem. Výsledky těchto experimentů je vidět na následujícím obrázku 5.3.



Obrázek 5.3: Násobení vektoru a matice

Poslední zkoumanou operací na obrázku 5.4 je operace násobení matic, která by měla dle samotného algoritmu násobení zabírat nejvíc výpočetního času. Hodnoty se definovaly náhodným způsobem.



Obrázek 5.4: Násobení matic, každá o velikosti 1500 krát 1500 prvků

Ze všech provedených experimentů vyplývá, že nejvhodnější variantou pro většinu případů užití je RenderScript. Mezi výhody tohoto nástroje patří relativně snadné použití a nízká hranice ovládnutí programátorem.

5.5 Alokace paměti a časová náročnost

Na grafu 5.2, který ukazuje srovnání rychlostí operace násobení v různých jazycích a nástrojích, je vidět, že v případě RenderScriptu bylo po ukončení operace stráveno větší množství času – 846 nanosekund oproti 14 nanosekundám v případě `for()` cyklu jazyku Java. Čím je způsoben daný efekt?

Jak bylo podrobněji uvedeno v podkapitole 5.2, potřebuje nástroj RenderScript pro úspěšný začátek vykonání kódu napsaném dle standardu C99 v první řadě provést alokaci určitého objemu paměti. Během provádění alokace je samozřejmostí jistá malá prodleva a s tím související ztráta času.

Ovšem alokace paměti nepředstavuje velkou část celkové prodlevy. Čas potřebný pro ukončení výpočtů definovaných v algoritmu se liší a záleží na jeho složitosti. Tady se dostáváme k nejzajímavější části procesů, provádějících se při volání funkcí v RenderScriptu. Jedná se o kopírování výsledků z, volně řečeno, paměti NDK[7] do paměti SDK[8].

Po úspěšném ukončení zavolané funkce v RenderScriptu je nutné zkopírovat data z vnitřní paměti RenderScriptu do paměti frameworku Android. Pro triviální malé datové struktury nezabírá tato operace, podle očekávání, velké množství času.

[7] https://developer.android.com/ndk/index.html
[8] https://developer.android.com/studio/index.html

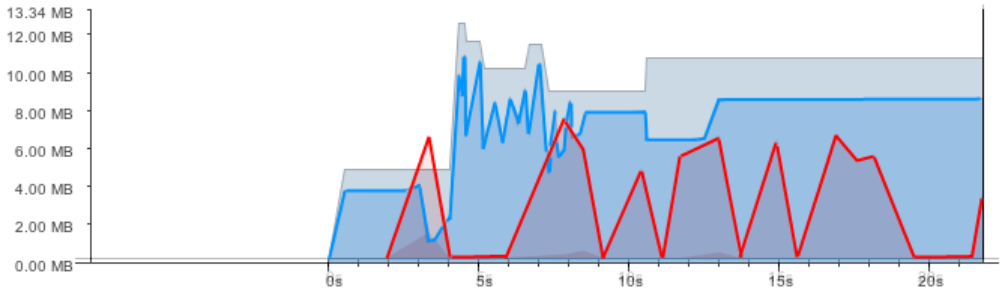
Právě výše uvedené kroky (alokace paměti, kopírování vstupních a výstupních dat) jsou při vykonání kódu příčinou větší časové spotřeby ve srovnání s voláním vestavěných Java metod.

Situace s výpočetním časem se ale může výrazně zhoršit v případě, je-li rozměr přenášených dat mezi dvěma různými pamětmi výrazně větší než výpočetní složitost algoritmu. Pouze v opačném případě bude mít využití výhod nástroje jako RenderScript patrný přínos.

Příkladem může být výpočetně náročný DTW nebo výpočet bottleneck příznaků při průchodu neuronové sítě, jinými slovy – při násobení dvou matic a matice vektorem. V případě násobení vektorů malých rozměrů strávíme více času kopírováním výsledků a alokováním potřebné paměti než samotným výpočtem.

Co se týče alokace paměti zařízení, během celého průběhu porovnávání nedocházelo k alokaci více než 10 MB – algoritmus Garbage collector jazyku Java stíhal uvolňovat nepotřebné objekty v alokované paměti. Největší nárůst alokace byl zaznamenán při načtení parametrů vrstev používané neuronové sítě, při výpočtu bottleneck příznaků nebo provádění DTW. Ovšem se počet alokovaných MB může lišit dle doby trvání vstupních audio vstupů.

Na obrázku 5.5 je znázorněna spotřeba paměti zařízení (modrá čára) a odpovídající zatížení procesoru (červená čára).



Obrázek 5.5: Zatížení CPU a RAM při průchodu celého cyklu chodu programu

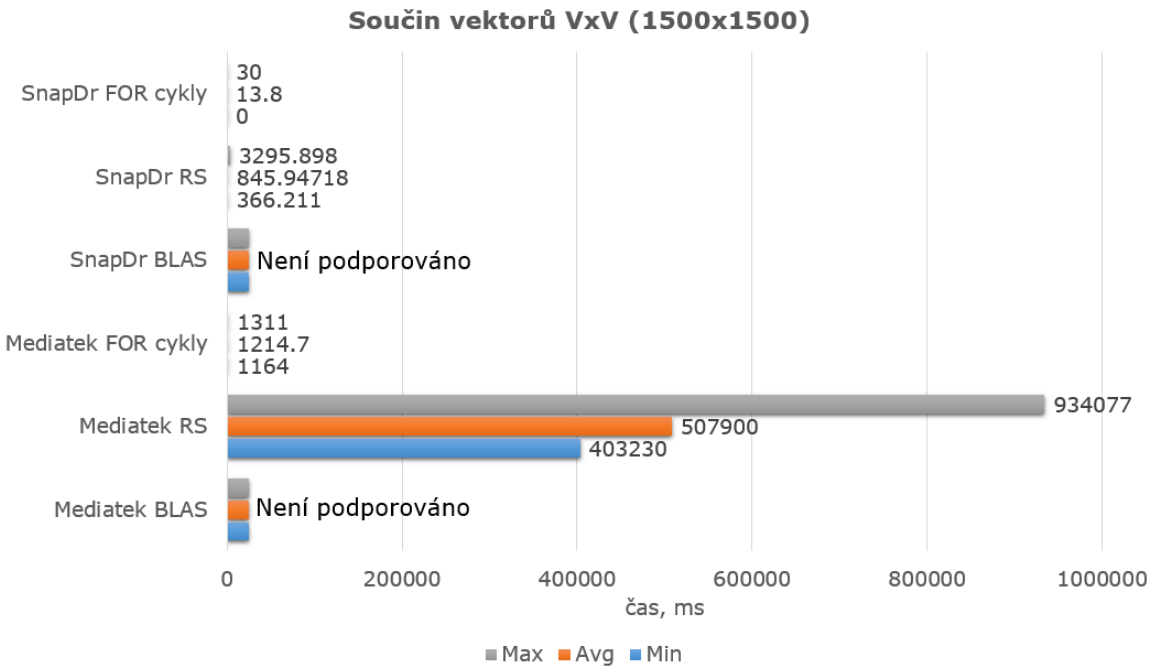
Je vidět, že s růstem vytíženosti CPU roste i množství alokovaných MB pamětí, ovšem nikdy nebyla zaregistrována nadměrná alokace paměti.

5.6 Vliv architektury procesoru na časový výsledek násobení

Bylo také potřeba zjistit, jak moc se změní výpočetní čas operací násobení a vyprodukuje-li se nějaký vedlejší účinek toho, že se bude kód vykonávat například na architektuře Mediatek.

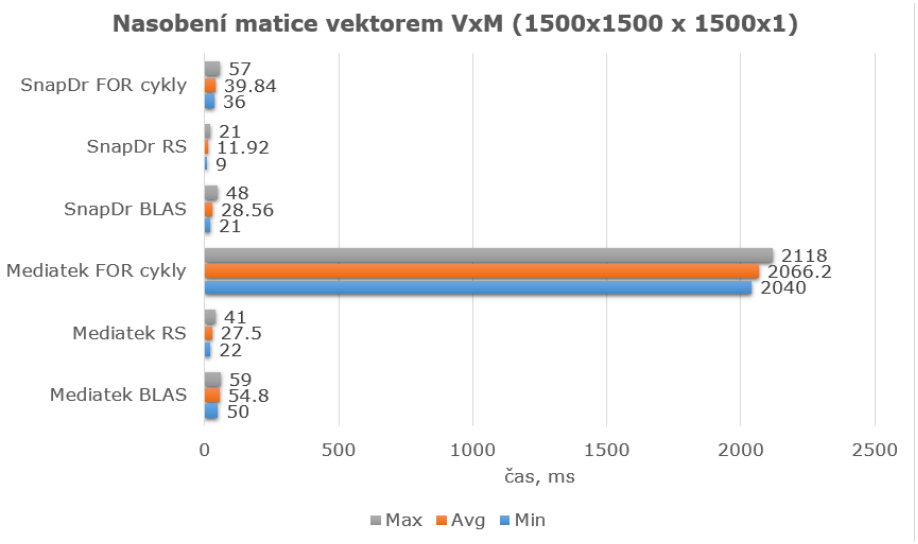
Proto byly v rámci experimentů změřeny dříve popsané operace násobení vektorů, vektoru a matice a dvou matic za použití mobilu Samsung GT-i9505 (Galaxy S4) s čipem Snapdragon a Lenovo K3+ s čipem Mediatek. Počet iterací se rovnal 50, hodnoty byly generovány náhodně pomocí funkce `rand()` vestavěné knihovny Math v Javě.

Z grafu na obrázku 5.6 vyplývá, že pomalejší zařízení K3 očividně potřebuje více času na vykonání potřebných operací. Čím je operace násobení jednodušší, tím je použití RenderScriptu méně vhodné – nemalý čas zabírá inicializace, volání potřebných funkcí ve vygenerovaném kódu a kopírování výsledků zpět do paměti Androidu.



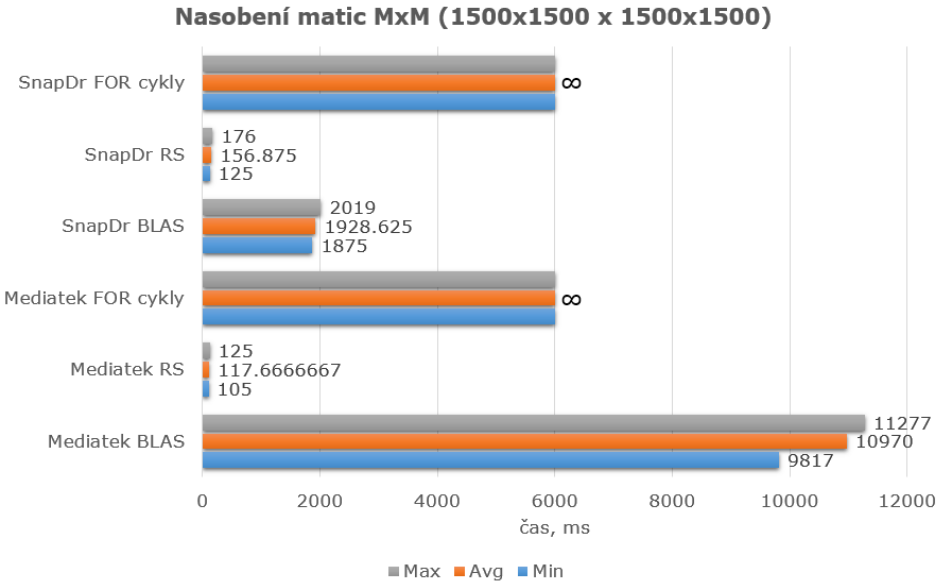
Obrázek 5.6: Násobení vektorů na architekturách Mediatek a Snapdragon. Knihovna BLAS nebyla brána v potaz kvůli nemožnosti provádění operace násobení vektorů potřebné délky instrukcemi NEON.

Na následujícím obrázku (5.7) je vidět, že se při zvětšení dimenzí násobených částí (matice místo vektoru) postupně zpomaluje násobení v nativním jazyce Java (vnořené cykly `for()` druhé úrovně místo první úrovně). RenderScript a BLAS naopak demonstrují výrazně kratší čas a v souladu s tím vyšší rychlost.



Obrázek 5.7: Násobení vektoru a matice na architekturách Mediatek a Snapdragon

Na posledním obrázku (5.8) u variant `for()` cyklu v Javě jsou časy rovné kladné nekonečnosti. Kvůli příliš dlouhé době, potřebné na provádění jedné iterace násobení, byla tato varianta z výsledného grafu vynechána. RenderScript byl dle předpokladu rychlejší v případě násobení velkého počtu hodnot, tedy matic.



Obrázek 5.8: Násobení dvou matic na architekturách Mediatek a Snapdragon

Závěrem je, že v případě architektur Snapdragon a Mediatek není patrný rozdíl v rychlosti RenderScriptu a BLAS, poměr času v případě těchto nástrojů prakticky vždy zůstával stejný. Důvodem patrného rozdílu v čase v případě Mediatek je nízká frekvence jáder čipu a relativně nižší výkon. Tím pádem RenderScript je znovu nejvhodnějším kandidátem.

5.7 Zrychlení jednotlivých kroků algoritmu

Na základě výsledků v minulé podkapitole bylo autorem práce rozhodnuto použít RenderScript v těch částech kódu, kde by bylo jeho použití přínosem ve zvýšení rychlosti. Knihovna BLAS nevyhovovala rychlostním požadavkům (ve srovnání s nástrojem RenderScript), a co se týče rozšíření NEON, byla by realizace většiny algoritmů v Asembleru daleko náročnější na čas, který by pak následně nepostačil na úspěšné vyřešení zadání této práce.

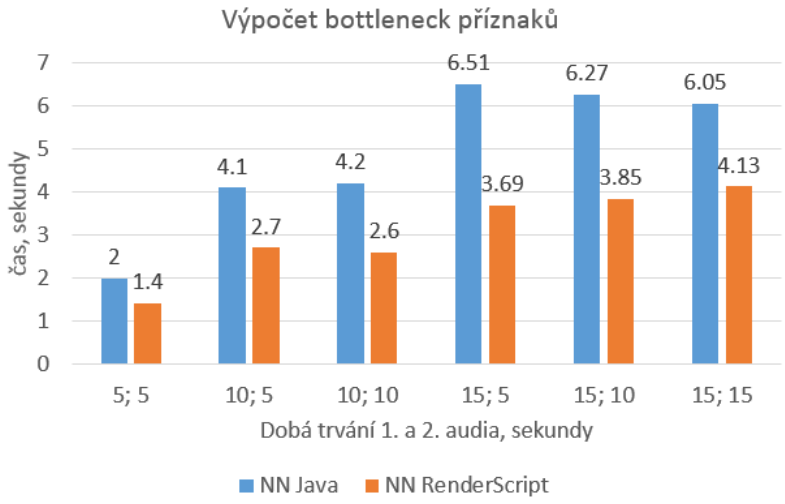
Níže jsou popsány činnosti, které byly provedeny nad jednotlivými částmi celého algoritmu porovnání. Tyto činnosti jsou v souladu s popisem v podkapitole 2.8.

Výpočet bottlneck příznaků

Máme vzorky stejného audio záznamu s frekvencí 8 kHz (formát 16 bit PCM[9]) a délkami záznamu 5, 10 a 15 vteřin. Z toho vyplývá, že po vzorkování o 25 ms sestává každý rámec z 200 vzorků, překryvání je o 80 vzorků. Celkový počet vzorků v případě délky 15 sekund je 120 000 (15 sekund * 8000 Hz), celkový počet rámce se tak rovná 1497 (120 000 - 200 / 80) rámcům.

Z předchozích výpočtů vyplývá, že je nutné provést výpočty bottleneck příznaků každého rámce 1497 krát. V nativním jazyce Java zabíralo výpočet příznaků 1497 rámců přibližně 6 vteřin. Cílem testování výpočtu příznaků audio vstupů různých délek bylo zjistit, jak moc se urychlí tato operace při použití RenderScriptu.

Na nasledujicím obrázku 5.9 jsou zobrazený výsledky tohoto experimentu, počet iterací se rovnal třem, výsledné hodnoty stráveného času se zaokrouhovaly.



Obrázek 5.9: Výpočet příznaku neuronovou sítí v RenderScriptu a nativní Javě

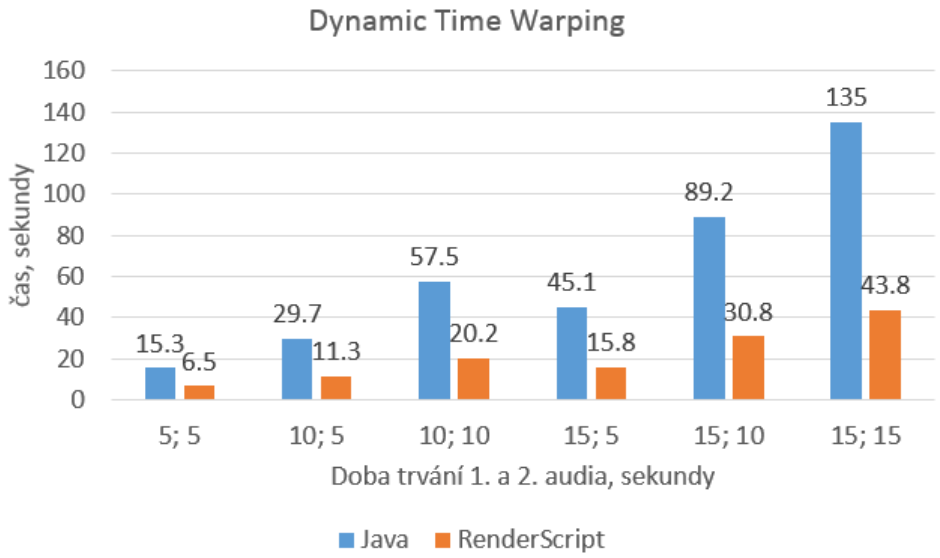
[9] http://soundfile.sapp.org/doc/WaveFormat/

Nakonec máme následující situaci: po přenosu algoritmu výpočtu bottleneck příznaků do RenderScript se rychlost výpočtu ve všech případech zkoumaných délek zrychlila 1,58 krát. Stojí za zmínění, že samotný výpočet příznaků zabírá prakticky veškerý čas (přesněji 2,8 z 3,06 sekund v průměru, což je 91,5 % celkového stráveného času), zúženým místem tedy proces kopírování výsledků zpět do paměti frameworku Android není. Posuneme se tedy k algoritmu DTW.

Dynamic Time Warping

Počet vstupních audio signálů se vždy rovná dvěma, a proto z toho plyne, že nemáme velký počet iterací – tím lepší je výsledná situace pro urychlení. Důvodem je alokace paměti (která je podrobněji rozepsaná v podkapitole 5.5). Stačí naalokovat místo v paměti pro dva vstupy, patřičné matice vzdálenosti a transformace a výsledek porovnání. Algoritmus dynamické transformace se moc nezměnil – byl přepsán v souladu se syntaxí jazyka standardu C99.

Předpokládejme, že znovu máme na vstupu příznaky Mel filtr banek výkonových spektrů rámců signálu o délkách 5, 10 a 15 vteřin. Ve výsledku těchto experimentů byl zjištěn průměrný rozdíl v rychlosti mezi RenderScriptem a Javou - tento rozdíl je znázorněn na obrázku 5.10:



Obrázek 5.10: Výpočet shody dvou audio signálu algoritmem DTW v RenderScriptu a Javě

Z výsledků je možné udělat závěr, že RenderScript byl v průměru 2,77 krát rychlejší ve srovnání s nativní implementací v Javě. Lze podle těchto výsledků posoudit, že takový rozdíl ve výpočetním čase je docela citelný.

5.8 Zhodnocení dosažených výsledků

Na základě získaných výsledků urychlení algoritmu porovnání audio signálů je možné zhodnotit výhodnost těchto technik urychlení. Z předchozí podkapitoly je jasně vyplývá, že jednoho z hlavních cílů této práce – urychlení algoritmu – bylo dosaženo.

Při pokusech zkrácení celého času potřebného na provádění porovnání audio vstupů byl kladen důraz na užitečnost přepsání jednotlivých částí algoritmu do rychlejší podoby.

Nežádoucím vedlejším efektem by mohlo být výrazné zpomalení těchto přepsaných částí algoritmu. Ve výsledku byla větší část fází porovnání viditelně zrychlena.

Kapitola 6

Závěr

Cílem této bakalářské práce byla jak realizace algoritmu pro porovnávání dvou audio vzorů, tak urychlení tohoto implementovaného porovnávání. Ukázková implementace algoritmu porovnávání byla provedena v jazyce Java, další otázkou byla správná volba nástroje pro urychlení operace porovnávání.

Především byla realizována analýza algoritmu porovnávání – jednotlivých potřebných kroků pro získání výsledné shody. Po této analýze byly navrženy požadavky vůči demonstrační aplikaci a ty byly následně splněny.

Po realizaci fází algoritmu porovnání bylo dalším krokem zvolení vhodného způsobu, jak tyto části zrychlit. Mezi možnými variantami figurovaly vestavěná knihovna Androidu RenderScript a knihovna JBLAS. Nejprve bylo nutné zjistit, která z těchto variant by měla větší přínos ve většině případů užití.

Výsledky syntetického testování ukázaly, že vestavěná knihovna Androidu je pro účely této práce vhodnější volbou. Následně byly časově naročné algoritmy přepsány do kódu standardu RenderScript. Po ukončení implementace bylo provedeno reálné testování na audio vzorcích - z tohoto testování bylo vidět podstatné urychlení jednotlivých částí porovnávání, přesněji výpočet příznaků pomocí neuronové sítě a výpočet shody pomocí DTW.

Zadání práce požadovalo, aby bylo implementován program pro porovnání dvou audio vzorů pomocí techniky DTW, realizována implementace na nízké úrovní (např. NDK) a zhodnoceny časová a paměťová náročnost.

Po realizaci všech těchto kroků autor by rád konstatoval, že záměr této práce - realizace porovnání audio vzorů a urychlení tohoto porovnání - byl tedy splněn. Práce autorovi dala především zkušenost ve vývoji aplikace pro systém Android, použití nástrojů NDK, a současně s tím i základní znalosti v oblasti porovnávání audio signálů.

I přesto, že je ve výsledku práce určitý rozdíl ve výpočetním čase, je zde dostatečný potenciál pro další urychlení (například NEON instrukce). Jako další možnosti pro vývoj autor této práce vidí použití rozsáhlejší neuronové sítě pro přesnější výpočet příznaků rámce audia.

Literatura

[1] *NEON*. [Online; navštíveno 15.05.2017].
URL https://developer.arm.com/technologies/neon

[2] *Renderscript Computation*. [Online; navštíveno 14.05.2017].
URL https://stuff.mit.edu/afs/sipb/project/android/docs/guide/topics/renderscript/compute.html

[3] Bakhurin, S.: *Fast Fourier Transform Introduction*. [Online; navštíveno 16.05.2017].
URL http://en.dsplib.org/content/fft_introduction.html

[4] Ben Shannon, K. P.: *A Comparative Study of Filter Bank Spacing for Speech Recognition*. [Online; navštíveno 15.05.2017].
URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.59.1426&rep=rep1&type=pdf

[5] Chan, S. S. . P.: *FastDTW: Toward Accurate Dynamic Time Warping in Linear Time and Space*. [Online; navštíveno 15.05.2017].
URL http://cs.fit.edu/~pkc/papers/tdm04.pdf

[6] Devasahayam, S.: *Signals and Systems in Biomedical Engineering: Signal Processing and Physiological Systems Modeling*. Springer Science and Business Media, 2012, ISBN 978-1-4613-6929-5.

[7] Fayek, H.: *Speech Processing for Machine Learning: Filter banks, Mel-Frequency Cepstral Coefficients (MFCCs) and What's In-Between*. [Online; navštíveno 11.05.2017].
URL http://haythamfayek.com/2016/04/21/speech-processing-for-machine-learning.html

[8] Li Tan, J. J.: *Fundamentals of Analog and Digital Signal Processing*. AuthorHouse, 2007, ISBN 978-1-4259-6031-5.

[9] Michael Norton, D. K.: *Fundamentals of Noise and Vibration Analysis for Engineers*. The press syndycate of the university of Cambridge, 2003, ISBN 978-0-5214-9913-2.

[10] Müller, M.: *In Information Retrieval for Music and Motion*. Springer, 2007, ISBN 978-3-540-74047-6.

[11] Pavel Matejka, T. N. S. H. M. O. G. J. M. B. Z., Le Zhang: *Neural Network Bottleneck Features for Language Identification*. [Online; navštíveno 12.05.2017].
URL http://www.fit.vutbr.cz/research/groups/speech/publi/2014/matejka_odyssey2014_299-304-35.pdf

[12] VanderPlas, J.: *Understanding the FFT Algorithm*. [Online; navštíveno 14.05.2017]. URL https://jakevdp.github.io/blog/2013/08/28/understanding-the-fft/#DFT-to-FFT:-Exploiting-Symmetry

Přílohy

Seznam příloh

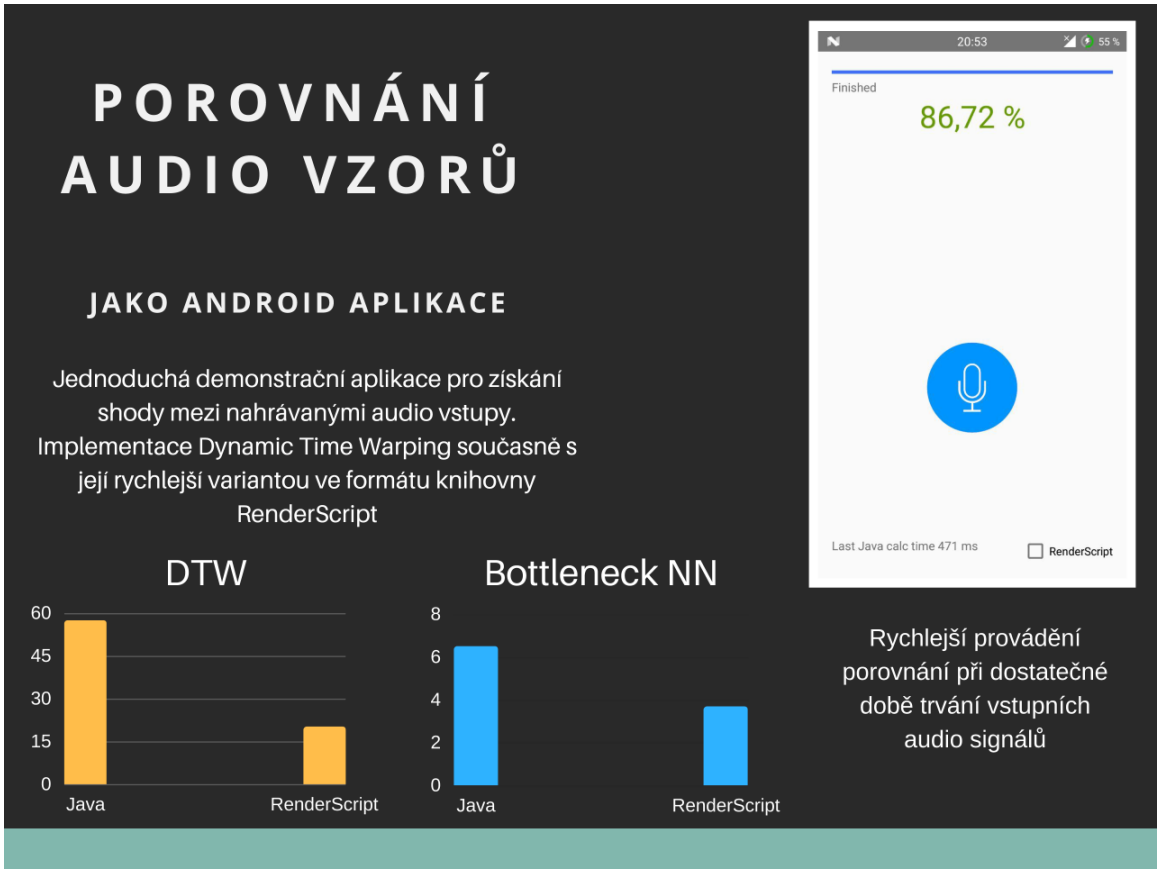
Příloha A

Obsah CD

- */src* - Zdrojové kódy aplikací
- */apk* - Instalační soubory aplikací
- */doc* - Zdrojové kódy pro technickou zprávu
- *doc.pdf* - Technická zpráva ve formátu pdf
- *poster.pdf* - A2 plakátek
- *video.mp4* - video prezentace bakalářské práce
- *README* - popis spuštění aplikací

Příloha B

Plakát



Obrázek B.1: Plakát bakalářské práce