VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

OPTIMALIZACE ROZPOZNÁVÁNÍ ŘEČI PRO MOBILNÍ ZAŘÍZENÍ

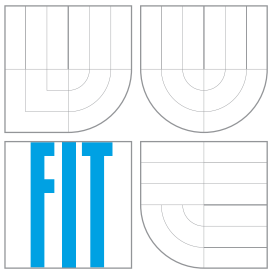
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. MARTIN TOMEC

BRNO 2010

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

OPTIMALIZACE ROZPOZNÁVÁNÍ ŘEČI PRO MOBILNÍ ZAŘÍZENÍ

OPTIMIZATION OF VOICE RECOGNITION FOR MOBILE DEVICES

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. MARTIN TOMEC

VEDOUCÍ PRÁCE
SUPERVISOR

Doc. Dr. Ing. PETR HANÁČEK

BRNO 2010

Abstrakt

Práce se zabývá optimalizací algoritmů pro detekci klíčových slov na procesorové architektuře ARM Cortex-A8. Nejprve je popsána tato architektura a zejména její jednotka NEON pro vektorové výpočty. Dále jsou stručně popsány algoritmy pro detekci klíčových slov a navržena jejich optimalizace pro danou architekturu. Jádro práce tvoří implementace těchto optimalizací a zhodnocení jejich vlivu na výkon.

Abstract

This work deals with optimization of keyword spotting algorithms on processor architecture ARM Cortex-A8. At first it describes this architecture and especially the NEON unit for vector computing. In addition it briefly describes keyword spotting algorithms and also there is proposed optimization of these algorithms for described architecture. Main part of this work is implementation of these optimizations and analysis of their impact on performance.

Klíčová slova

optimalizace, ARM, NEON, SIMD, detekce klíčových slov, rozpoznávání řeči, neuronová síť

Keywords

optimization, ARM, NEON, SIMD, keyword spotting, speech recognition, neural net

Citace

Martin Tomec: Optimalizace rozpoznávání řeči pro mobilní zařízení, diplomová práce, Brno, FIT VUT v Brně, 2010

Optimalizace rozpoznávání řeči pro mobilní zařízení

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením Doc. Dr. Ing. Petra Hanáčka. Další informace mi poskytl Ing. Igor Szöke. V závěrečné části dokumentu jsou uvedeny všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Martin Tomec
26. května 2010

Poděkování

Na tomto místě bych chtěl vyjádřit poděkování Doc. Dr. Ing. Petru Hanáčkovi za vedení a konzultace. Poděkování patří také Ing. Igoru Szökemu za odborné rady a čas věnovaný konzultacím.

Obsah

Kapitola 1

Úvod

Strojové rozpoznávání řeči prožívá v poslední době velký rozmach. Do praxe se postupně dostávají systémy, které v reálném čase přepisují plynulou řeč. Ale i v mobilních zařízeních se postupně prosazuje detekce klíčových slov. Doposud byla většinou implementovaná jako porovnávání s předem nahraným slovem, takže její použití bylo omezeno na několik málo klíčových slov. Zajímavou možností v mobilních zařízeních však může být obecné rozpoznávání slov, pouze na základě fonetického přepisu textové podoby slova. Díky automatickému fonetickému přepisu by pak bylo možné například vytvořit přepis telefonního seznamu a umožnit vyhledání kontaktu na základě vysloveného jména.

Právě na mobilní zařízení by se měla zaměřit tato práce. Díky neustálému zvyšování výpočetních možností těchto zařízení je dnes možné uvažovat o plnohodnotném detektoru klíčových slov, jenž by celý pracoval v mobilním zařízení. Případné další využití by mohlo být u vestavěných systémů, kde se nižší spotřeba zařízení využije ke zmenšení velikosti zařízení (není potřebné aktivní chlazení čipu).

Myšlenka implementace rozpoznávání řeči v mobilním zařízení už byla samozřejmě zpracována v několika pracích. Asi nejdále v této oblasti je projekt MobilDictate z Technické univerzity v Liberci. Já bych zde rád navázal na diplomovou práci Ing. Tomáše Cipra [3], ve které je navržen a implementován detektor klíčových slov pro operační systém Symbian. Tento detektor byl sice optimalizován pro mobilní zařízení, ale nebyl otestován na reálném hardware. Mimo to byl optimalizován pouze na úrovni algoritmů, nikoliv na úrovni strojového kódu pro cílovou architekturu.

Pro optimalizaci a otestování programu byl vybrán vývojový kit OMAP3530 s procesorem založeným na architektuře ARM Cortex-A8. Procesory založené na architektuře ARM dnes dominují na trhu s mobilními zařízeními zejména díky své nízké spotřebě. Tato konkrétní architektura patří k nejnovějším (z roku 2007) a bude tedy postupně nahrazovat své předchůdce. Proto by se měla stát standardem v oblasti mobilních zařízení. Novější architektury jsou navíc zpětně kompatibilní, takže kód optimalizovaný pro tuto architekturu bude využitelný i v budoucnu.

Ve druhé kapitole jsou stručně popsány algoritmy pro rozpoznávání řeči. Práce se nezaměřuje přímo na obecnou optimalizaci těchto algoritmů, ale spíše na jejich přizpůsobení pro vybranou architekturu. Tato kapitola tedy slouží spíše pro uvedení do problematiky a odkázání na literaturu, která se těmto algoritmům věnuje podrobněji.

Cílem práce tedy není obecná optimalizace algoritmů pro rozpoznávání řeči, ale optimalizace výsledného kódu pro konkrétní architekturu. Proto jsou ve třetí kapitole podrobně popsány její nejdůležitější vlastnosti. Popis se zaměřuje zejména na matematické koprocesory určené pro výpočty v plovoucí řádové čárce, které jsou klíčové pro algoritmy rozpozná-

vání řeči. Ostatní části architektury jsou zde popsány pouze okrajově a obecně. Dále jsou v této kapitole shrnuty dostupné překladače pro tuto platformu a podpora matematických koprocesorů v těchto překladačích. V závěru kapitoly je uvedeno i krátké výkonnostní srovnání možných přístupů k vektorovému koprocesoru. Ten je patrně nejdůležitějším rozšířením architektury pro výpočty v plovoucí řádové čárce.

Ve čtvrté kapitole jsou navrženy optimalizace těchto algoritmů. Pro efektivní optimalizaci je důležité určení výkonově kritických částí programu. Proto je součástí kapitoly zhodnocení výpočetní náročnosti a vlivu na výsledný výkon. V závěru kapitoly jsou shrnuty výsledky profilování implementovaného detektoru klíčových slov, včetně určení časově nejnáročnějších funkcí.

Pátá kapitola tvoří jádro práce. Je v ní postupně popsána implementace jednotlivých optimalizací, včetně zhodnocení jejich vlivu na výkon. Největší prostor je věnován neuronové síti, protože její vyhodnocení tvoří podstatnou část doby výpočtu.

V šesté kapitole jsou shrnuty výsledky a nastíněna další možná využití implementovaných optimalizací.

V rámci semestrálního projektu byly vybrány vhodné algoritmy a navržena jejich optimalizace pro architekturu ARM Cortex-A8. Z jeho výsledků tedy vychází druhá třetí a částečně i čtvrtá kapitola.

Kapitola 2

Rozpoznávání řeči

Samotné rozpoznávání řeči je poměrně obsáhlá vědecká disciplína, která zahrnuje oblasti od fonetiky a akustiky až po teoretickou informatiku. V následujících kapitolách jsou pouze shrnuty nejčastěji používané algoritmy pro detekci klíčových slov. Zároveň je zde nastíněna jejich výpočetní náročnost, která je pro mobilní zařízení klíčová. Popis algoritmů z větší části vychází z literatury [3] a [8].

Obecné zpracování řeči sestává z několika fází:

- Zpracování vstupního signálu
- Extrakce příznaků
- Klasifikace fonémů
- Dekódování

Pro každou jednotlivou část lze použít různé algoritmy, ale použití jiného algoritmu většinou znamená i větší úpravy sousedních částí.

2.1 Zpracování vstupního signálu

Základem pro jakékoliv zpracování řeči je její zaznamenání. Tato část tedy zahrnuje vzorkování a kvantizaci zvukového vstupu. To je většinou zajištěno specializovaným hardware a úkolem programátora je pouze nastavení vzorkovací frekvence. Pro zpracování řeči se nejčastěji používá vzorkovací frekvence 8 kHz, jelikož podle vzorkovacího teorému je dostatečná pro signály s frekvenčním rozsahem do 4 kHz (což je většina řeči).

V praxi je ale potřeba vstupní signál omezit filtrem s dolní propustí, nebo jej vzorkovat s vyšší vzorkovací frekvencí a následně aplikovat antialiasingový filtr. V opačném případě by se vyšší frekvence (které může obsahovat hluk na pozadí) přidaly k řečovému signálu a zkreslovaly ho.

Dále je signál rozdělen na krátké časové rámce, které jsou později zpracovávány odděleně. Pro účely rozpoznávání řeči se osvědčilo rozdělení na rámce délky 25 ms a šířkou překryvu 15 ms. To znamená, že první rámec pokrývá úsek 0 ms až 25 ms, druhý rámec pokrývá úsek 10 ms až 35 ms a tak dále. Při obvyklé vzorkovací frekvenci to odpovídá 200 vzorkům na jeden rámec.

Zpracování signálu se provádí pouze jednou pro každý časový rámec a navíc je z větší části zajištěno pomocí hardware. Nemá tedy významný vliv na rychlost rozpoznávání, ale

přitom může výrazně ovlivnit kvalitu výsledků. Proto se na výsledné rámce aplikují další transformace, které mají pomoci při další práci se vzorky:

- Ustřednění
- Vážení Hammingovým oknem

Ustřednění se používá pro odečtení stejnosměrné složky nasnímaného signálu. Protože stejnosměrná složka signálu je pouze chybou kvantizace nebo snímání, nemá vliv na rozpoznávání řeči. Může mít ale negativní vliv na použité algoritmy, které předpokládají signál s nulovou stejnosměrnou složkou. Proto je z každého rámce odečtena podle vzorce 2.1, kde N je délka rámce a s_n je n-tý vzorek v rámci.

$$s_n^{'} = s_n - \frac{1}{N}\sum_{i=0}^{N-1} s_i \tag{2.1}$$

Vážení Hammingovým oknem je důležité pro další zpracování signálu ve frekvenční oblasti. Pokud by rámec byl zpracováván tak, jak byl „vytržen“ ze záznamu, měl by na svých okrajích z pohledu dalšího zpracování ostré hrany (protože signál mimo rámec se považuje za nulový). Tyto hrany by se projevily ve zkreslení frekvenční charakteristiky, která je základem pro další zpracování. Proto se signál směrem k okrajům rámce utlumí podle vzorce 2.2

$$s_n^{'} = s_n(0,54 - 0,46cos\frac{2\pi n}{N-1}) \tag{2.2}$$

2.2 Extrakce příznaků

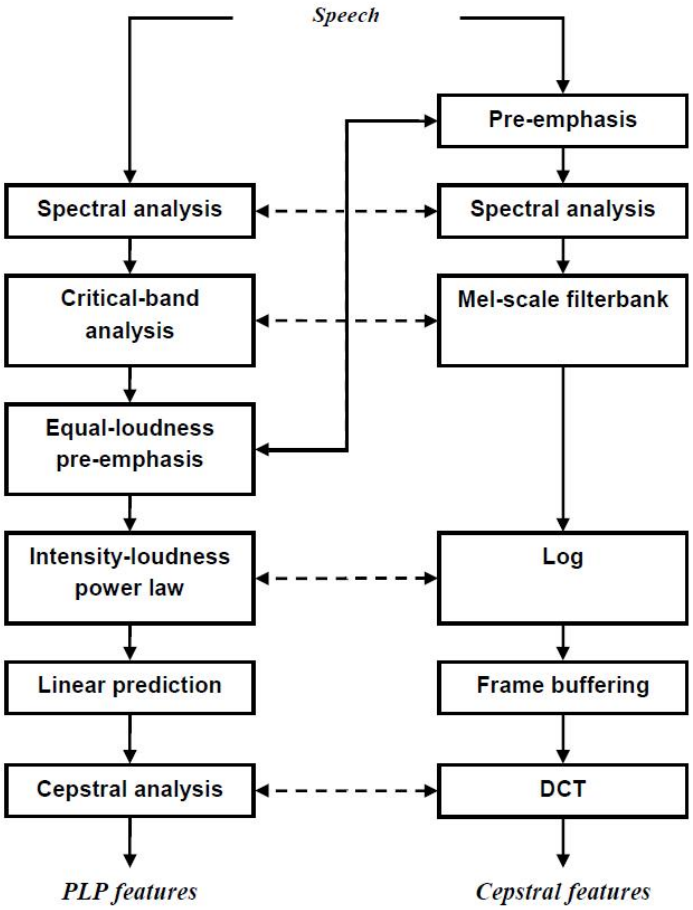
Signál má po základním zpracování stále velkou redundanci proti samotné řeči a pro další zpracování je vhodné tuto redundanci omezit. Pro každý rámec se tedy spočítá omezené množství příznaků, které tento rámec „nejlépe“ charakterizují pro účely rozpoznávání řeči. Požadavky na kvalitu příznaků jsou shrnuty například v [3]:

„*Příznaky by měly reprezentovat význačné rysy řečového signálu. Měly by pokud možno zachycovat takové charakteristiky, které jsou důležité pro vnímání a pochopení významu projeveného. Naopak by měly potlačovat šum a individuální charakteristiky mluvčího. Typicky např. požadujeme, aby příznaky nebyly závislé na frekvenci základního tónu. Na příznaky také klademe požadavek, aby byly dekorelované.*“

Pro výpočet příznaků se používají zejména dvě metody:

- Mel-frekvenční kepstrální koeficienty (MFCC)
- Perceptuální lineární predikce (PLP)

Postup vyhodnocení těchto metod je názorně vidět na obrázku 2.1 Podrobnější popis těchto metod lze nalézt například v [8] nebo případně v [3]. Jak je vidět na obrázku, v obou metodách lze nalézt odpovídající si části. Obě jsou založeny na výpočtu spektra signálu (*Spectral analysis*), na které je poté aplikována sada filtrů rozložená po frekvenční ose nelineárně podle vnímání různých frekvencí člověkem (*Critical-band analysis*). Člověk totiž v nižší části spektra (do 1 KHz) rozlišuje frekvenci zvuku daleko citlivěji než ve vyšší. Rovněž hlasitost je upravena v závislosti na frekvenci (*Pre-emphasis*) protože člověk vnímá nižší frekvence jako hlasitější. Dále následuje nelineární vyrovnání hlasitosti, čehož je u MFFC dosaženo logaritmováním.



Obrázek 2.1: Postup výpočtu PLP a MFCC. Převzato z [3] (upraveno)

Obě metody jsou si tedy podobné z hlediska výpočetní náročnosti. Podle testů [7] vycházejí MFCC jako mírně spolehlivější v přesnosti reprezentace řeči, ale v diplomové práci [3] byla nakonec vybrána perceptuální lineární predikce. Protože pro výstupy PLP byla natrénována neuronová síť (viz podkapitolu 2.3), bude tato metoda použita i zde.

Obě tyto metody mají nevýhodu v tom, že pro charakteristiku řeči používají pouze krátký úsek (jeden rámec). Pro přesnější reprezentaci lze příznaky MFCC počítat z delšího časového kontextu, nebo u PLP přidat derivace původních koeficientů podle času. Díky tomu by se sice zvýšila přesnost rozpoznávání, ale zároveň také počet vstupů neuronové sítě a tím by výrazně stoupla výpočetní náročnost celého rozpoznávače.

2.3 Klasifikace fonémů

Nadpis kapitoly je trochu zavádějící, protože po extrakci příznaků nemusí nutně následovat klasifikace fonémů. Například v případě jednodušších systémů zde může být rovnou klasifikátor klíčových slov. Klasifikace fonémů má ale výhodu v obecnosti řešení, protože lze klasifikátor natrénovat na obecné fonémy a klíčová slova zadávat až za běhu programu pomocí fonetického přepisu. Přepis může být automatický, takže je možné zadávat klíčová

slova v textové formě.

Algoritmů použitelných pro klasifikátor existuje několik. Mezi nejznámější patří skryté Markovovy modely (HMM), neuronové sítě, Algoritmy podpůrných vektorů (SVM) nebo rozhodovací stromy. V práci Tomáše Cipra [3] byly pro klasifikaci fonémů vybrány neuronové sítě. Protože cílem této práce není porovnávání jednotlivých algoritmů, budu předpokládat, že neuronové sítě byly vybrány jako optimální pro mobilní zařízení.

Klasifikace fonémů patří k výpočetně nejnáročnějším částem zpracování řeči. Každý rámec zpracovávaného signálu je totiž nutné porovnat s modely všech fonémů a pro každý určit míru podobnosti. V případě neuronové sítě sice neporovnáváme explicitně s jednotlivými modely, ale síť musí rovněž pro každý foném určit míru podobnosti a její vyhodnocení je časově náročné.

Jako výchozí algoritmus pro tuto část je tedy ponechána neuronová síť podle diplomové práce [3]. V ní byla navržena třívrstvá síť (tedy s jednou skrytou vrstvou) s dopředným průchodem. V uvedené práci jsou také shrnuty výsledné úspěšnosti klasifikace jednotlivých fonémů v závislosti na velikosti neuronové sítě, resp. na počtu neuronů ve skryté vrstvě. Z výsledků uvedených v tabulce 2.1 je zřejmé, že výsledná přesnost této konkrétní neuronové sítě nezávisí příliš na počtu neuronů ve skryté vrstvě. Proto byla vybrána menší velikost sítě, která obsahuje 100 neuronů ve skryté vrstvě. Dále je ve výše uvedené práci

Velikost sítě	100	500
Úspěšnost – frame accuracy	51,38 %	50,99 %
Přesnost – phoneme accuracy	31,57 %	32,42 %

Tabulka 2.1: Úspěšnost klasifikace v závislosti na velikosti neuronové sítě

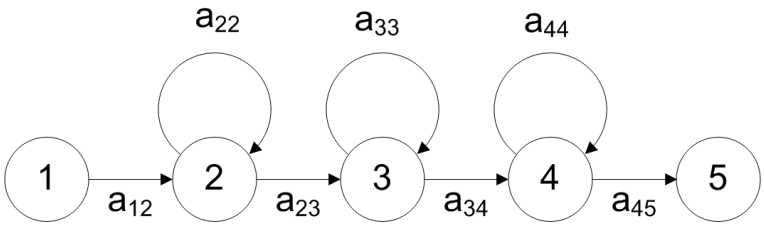
navrženo rozšíření vektoru příznaků o delta koeficienty, tj. derivaci původních koeficientů podle času. Tato úprava by sice negativně ovlivnila rychlost klasifikace, ale mohla by zvýšit její přesnost. Protože by však rozšíření znamenalo i přetrénování neuronové sítě, je v implementaci ponechán vektor PLP příznaků a v závislosti na výsledném výkonu celé knihovny se rozhodne o případném rozšíření.

2.4 Dekódování

Výstup z klasifikátoru fonémů obsahuje pro každý rámec vektor pravděpodobností pro jednotlivé fonémy. Protože úspěšnost klasifikátoru většinou nepřesahuje 60 %, je potřeba slovo z výstupu dekódovat. K tomu lze využít například skryté Markovovy modely. Pomocí těchto modelů jsou reprezentována jednotlivá slova jako množina stavů propojená orientovanými hranami (obrázek 2.2) Jednotlivé stavy v modelu odpovídají obecně určitému úseku promluvy. Mohou to být fonémy, dvojice fonémů nebo i větší celky. V případě použití větších úseků se sice zvyšuje přesnost rozpoznávání, ale výrazně se zvyšuje výpočetní náročnost[1]. Pravděpodobně proto byly pro implementaci do mobilního zařízení v [3] použity modely, u kterých každý stav odpovídá jednomu fonému.

S každým novým fonémem je pro všechny modely klíčových slov nutné vyhodnotit aktuální pravděpodobnost jejich výskytu. Pro procházení modelů je takřka standardem algoritmus Viterbiho dekodéru, který je názorně popsán v diplomové práci [3]. Tento algoritmus sice má (pro tyto dané modely) lineární časovou složitost, ale je nutné jej vyhodnocovat

[1] to je podrobněji rozebráno v [10]



Obrázek 2.2: Znázornění skrytého Markovova modelu

pro každý model zvlášť. Celková náročnost této fáze tedy závisí na velikosti slovníku rozpoznávaných slov a optimalizaci je nutné přizpůsobit předpokládanému množství slov ve slovníku.

Protože je obecný Viterbiho algoritmus dostatečně podrobně popsán v literatuře, nemělo by smysl jej zde znovu popisovat obecně. Proto je v následující části pouze stručně shrnut postup vyhodnocení pro dané modely. Tyto konkrétní modely mají proti obecným modelům řady omezení. Jednak v nich neexistují zpětné hrany, jelikož by vznikající cykly ve slově nedávaly smysl. Dále dopředné hrany vedou vždy pouze k následujícímu stavu a u každého stavu existuje právě jedna smyčka.

Díky výše uvedeným omezením lze ohodnocení modelu zjednodušit. Předpokládejme, že po každém průchodu modelem je u jednotlivých stavů uložená maximální pravděpodobnost, se kterou bylo možné se do daného stavu dostat. V následujícím průchodu existují pro každý stav právě dvě možnosti:

- Do daného stavu se model dostal v předcházejících průchodech a setrvává v něm.
- Do daného stavu se model dostal v aktuálním průchodu z předcházejícího stavu.

V obou případech je možné následující pravděpodobnost spočítat jako maximum dle vzorce 2.3, kde $p_n{}'$ značí následující pravděpodobnost, p_n značí původní pravděpodobnost a p_{n-1} značí původní pravděpodobnost předcházejícího stavu v modelu. Přechodová pravděpodobnost mezi stavy je označena t, pravděpodobnost setrvání ve stavu je označena s a pravděpodobnost výskytu odpovídajícího fonému (tj. výstup neuronové sítě) je označena f_n. V implementaci se místo hodnot pravděpodobností používají jejich logaritmy a násobení tak přejde na sčítání.

$$p_n{}' = max(p_{n-1}t, p_n s)f_n \tag{2.3}$$

Takto je možné spočítat následující maximální pravděpodobnost daného stavu pouze na základě předchozí pravděpodobnosti tohoto stavu a jemu předcházejícího. Pokud se tedy stavy prochází od posledního, lze průchod provést v lineárním čase.

Výše uvedený popis je samozřejmě značně zjednodušující a platí pouze pro tyto konkrétní typy modelů. Podrobnější a obecnější popis algoritmu lze nalézt například v [3].

Kapitola 3

Architektura ARM Cortex

Mobilním zařízením dnes dominují procesory založené na architektuře ARM. Byly totiž od začátku vyvíjeny pro nízkou spotřebu a standardizovaná architektura (popsaná například v [11]) umožňuje snadný vývoj aplikací. Kvůli standardizaci je sice architektura licencovaná, avšak na trhu je mnoho výrobců procesorů se zakoupenou licencí. Standardizace navíc umožňuje binární přenositelnost aplikací. Postupem času se samozřejmě standard vyvíjel, takže dnes existuje několik „rodin“ architektur. Používají se především poslední tři:

- ARMv5
- ARMv6
- ARMv7

Jednotlivé rodiny je možné dále rozdělit do verzí podle podporovaných rozšíření. Každá rodina má však definovaný základ, který je podporován všemi verzemi a zajišťuje binární kompatibilitu aplikací (tzn. aplikací, které byly překládány bez využití rozšíření dané verze).

Bylo by neúčelné zde rozebírat vývoj jednotlivých verzí, proto se zaměřím pouze na nejnovější rodinu ARMv7. Ta má tři hlavní větve:

- ARMv7-A
- ARMv7-R
- ARMv7-M

První verze je určená pro „běžné“ operační systémy a podporuje instrukční sady ARM, Thumb a Thumb-2 (viz podkapitolu 3.1). Druhá verze je určena pro real-time operační systémy a podporuje stejné instrukční sady jako předchozí. Poslední verze (ARMv7-M) je zjednodušená a podporuje pouze instrukční sadu Thumb-2. Díky tomu by měla být levnější než obě předchozí verze.

V této práci se zaměřím pouze na verzi ARMv7-A (konkrétně procesor ARM Cortex-A8), jelikož právě ta by měla být určena do výkonných mobilních zařízení. První dvě verze jsou si navíc velmi podobné, až na drobné úpravy v obsluze přerušení (pro real-time operační systémy). Zájemci o bližší popis mají k dispozici referenci [11]. Detailní popis a specifikace architektury jsou dostupné pouze partnerům s licencí, ale popis procesorů s touto architekturou je k dispozici zdarma.

Pro realizaci práce byl použit procesor OMAP 3530 od firmy Texas Instruments. Tento 32-bitový procesor je založený na architektuře ARMv7-A a taktovaný na frekvenci 600MHz.

Podporuje tedy instrukční sady ARM, Thumb a Thumb-2, které jsou podrobněji popsány v následující podkapitole. Dále také obsahuje bezpečnostní rozšíření *TrustZone*, matematický koprocesor *NEON* pro multimediální aplikace, jednotku *VFPv3* pro výpočty v plovoucí řádové čárce a částečně i rozšíření *Jazelle*. V dalších podkapitolách jsou tyto technologie popsány podrobněji.

3.1 Instrukční sady

V této podkapitole jsou podrobněji rozebrány výše zmiňované instrukční sady ARM, Thumb a Thumb-2. V závěru je popsáno rozšíření ThumbEE. Protože instrukční sada ovlivňuje významnou měrou výkon programů i velikost kódu, prošla u architektury ARM postupným vývojem.

Původní instrukční sada ARM obsahuje kompletní sadu 32-bitových instrukcí. Tuto sadu podporují téměř všechny[1] procesory založené na architektuře ARM.

Instrukční sada Thumb je pouze doplňkem základní sady. Obsahuje podmnožinu instrukcí ze základní sady a jednotlivé instrukce jsou komprimované na 16 bitů. Díky tomu dosahuje vyšší hustoty kódu, což je důležité u zařízení s malou pamětí pro zdrojové kódy programu. Není však kompletní, takže procesor se musí přepínat mezi dvěma módy (buď zpracovává instrukce z ARM, nebo Thumb). Hlavní nevýhodou je však nižší výpočetní výkon. V praxi se tedy doporučuje pro časově kritické části programu používat instrukční sadu ARM a pro rozsáhlé části kódu, které nejsou časově kritické, používat instrukční sadu Thumb.

Nejnovější instrukční sada Thumb-2 spojuje dohromady obě dvě předchozí. Komprimuje určité instrukce, čímž dosahuje snížení velikosti kódu. Zároveň si ale zachovává stejnou výkonnost jako základní instrukční sada. Také je kompletní, takže odpadá režie spojená s přepínáním módů. U některých procesorů se již upouští od předchozích instrukčních sad a podporuje se pouze Thumb-2. Lze tedy čekat, že se postupně přejde pouze na tuto instrukční sadu.

Rozšíření ThumbEE je podobné prvnímu rozšíření Thumb. Instrukce z tohoto rozšíření jsou podporovány pouze tehdy, pokud je procesor přepnutý do speciálního módu. Obsahuje pouze instrukce pro práci s pamětí a jeho výhodou je podpora hardware při kontrole přístupu do paměti[2].

3.2 Matematické koprocesory

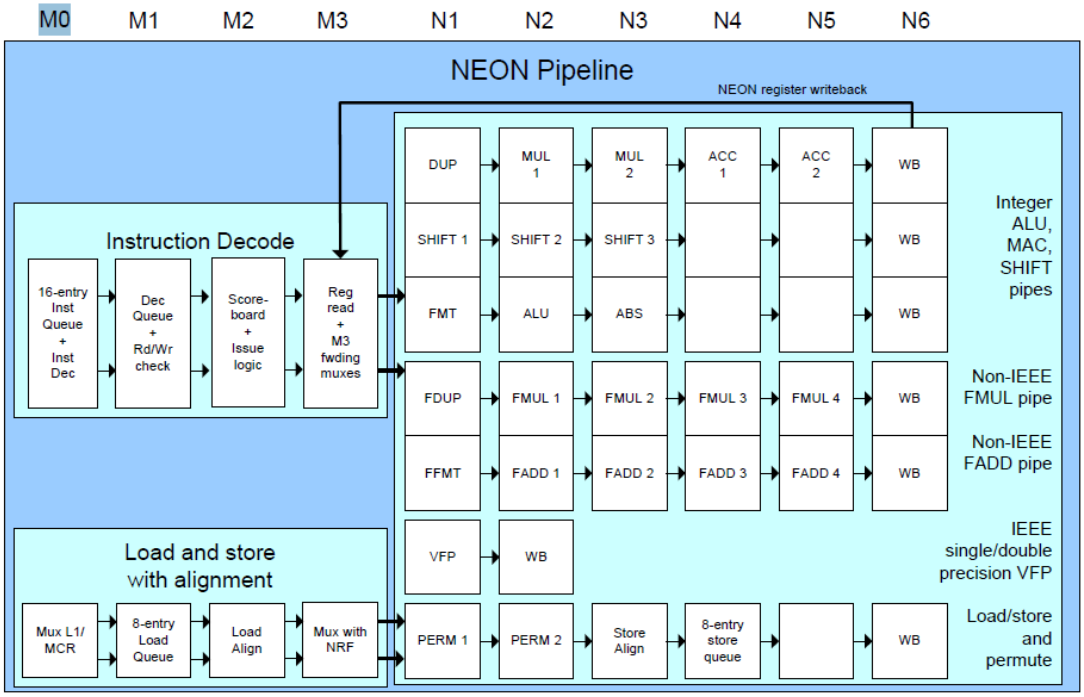
Procesor OMAP 3530 obsahuje dvě jednotky určené pro urychlení výpočtů v plovoucí řádové čárce. První z nich – jednotka NEON – je nepovinným rozšířením architektury a nemusí tedy nutně být přítomná ve všech procesorech založených na této architektuře. Vzhledem k jejímu výkonu a současným požadavkům na mobilní zařízení se však dá očekávat, že bude dostupná ve většině nových procesorů. Druhá jednotka – VFP – byla dostupná i v předchozích verzích architektury a je udržována zejména kvůli zpětné kompatibilitě. V následujících dvou částech jsou obě jednotky popsány podrobněji.

[1] kromě výjimek, které podporují pouze Thumb-2

[2] například kontrola mezí polí, nebo nulového ukazatele

3.2.1 Jednotka NEON

Tento koprocesor je založen na architektuře SIMD a je určen pro urychlení multimediálních výpočtů pomocí vektorového zřetězeného zpracování aritmetických instrukcí. Je oddělen od jádra procesoru a má vlastní přístup do paměti cache (tj. vlastní load/store jednotku). Struktura koprocesoru je podrobně znázorněna na obrázku 3.1. Na obrázku je vidět vstupní fronta instrukcí (Inst Queue), která je plněna z jádra procesoru. Za ní následuje zřetězené dekódování instrukcí a vydávání instrukcí do aritmeticko-logické jednotky. Všechny předchozí části jsou zdvojeny, takže je možné v jednom taktu teoreticky zpracovávat až dvě instrukce. Toto je samozřejmě omezeno typem instrukcí, protože koprocesor obsahuje pouze jednu jednotku pro celočíselné výpočty a jednu jednotku pro výpočty v plovoucí řádové čárce. V praxi je tak často zároveň s aritmetickou instrukcí vydávána druhá instrukce pouze pro vstupní/výstupní operaci, která je nezávislá na výpočetních jednotkách. Vlastní výpočetní jednotky mají šest zřetězených stupňů, takže v jednom taktu lze mít rozpracováno až dvanáct instrukcí.



Obrázek 3.1: Struktura jednotky NEON (převzato z [5])

Každou instrukcí lze zpracovávat až 128-bitové slovo, které může být rozděleno na:

- 32-bitová čísla v plovoucí řádové čárce
- 8-bitová až 64-bitová čísla v pevné řádové čárce
- 8-bitová až 64-bitová pole bitů (bitfields)
- 8-bitové nebo 16-bitové polynomy s 1-bitovým koeficientem

Jednotlivé položky jsou zpracovávány v rámci jedné instrukce, takže koprocesor může během jedné instrukce zpracovat až 16 (8-bitových) čísel.

K dalším výhodám tohoto koprocesoru patří podpora nezarovnaného přístupu do paměti a automatizovaného načítání struktur z paměti. Tím jsou částečně eliminovány (bez zásahu software) problémy s načítáním klasických struktur z vyšších programovacích jazyků. Koprocesor do jednotlivých registrů umožňuje načítat data s pevně daným rozestupem (1 až 4 Byte) takže je například možné načítat odděleně barevné složky z pole, ve kterém jsou jednotlivé barvy uloženy sekvenčně (za sebou). Poslední výhodou je možnost nahlížet na každý z registrů jako na dva 64- bitové registry. Díky tomu lze využít naplno všechny registry. Například při násobení jsou potřeba dva 64-bitové registry pro vstupní hodnoty a jeden 128-bitový registr pro výstupní hodnoty. Ve skutečnosti se však použijí pouze dva 128-bitové registry.

3.2.2 VFP koprocesor

V mobilních zařízeních měly procesory často podporu pouze pro výpočty v pevné řádové čárce. Výpočty v plovoucí řádové čárce se pak emulovaly pomocí software. Ale stejně jako u prvních osobních počítačů se i v mobilních zařízeních začaly prosazovat matematické koprocesory. U architektury ARM je k tomu určena jednotka VFP (Vector Floating Point). Ta podporuje výpočty s jednoduchou (32-bitů) i dvojitou (64-bitů) přesností dle normy IEEE-754[1]. Je možné ji přepnout do čtyř různých módů:

- Full-compliance
- Default NaN
- Flush-to-zero
- RunFast

V případě prvního módu jsou všechny nestandardní stavy ošetřeny výjimkou, pro kterou si uživatel může napsat vlastní obsluhu (nebo zakázat výjimku). V módu „Default NaN“ je v případě chybného vstupu (NaN) generován okamžitě chybný výstup (NaN) bez zásahu software. V módu „Flush-to-zero“ jsou příliš malé[3] hodnoty automaticky převedeny na nulu. Tím je na úkor snížení přesnosti zvýšena rychlost výpočtu pro čísla blízká nule. Poslední mód „RunFast“ implikuje oba dva předchozí módy. Navíc u všech dalších výjimečných stavů vrací defaultní hodnoty definované v IEEE-754 bez zásahu software. Tím je zajištěna deterministická doba výpočtu na úkor možnosti ošetření výjimek.

V jádře Cortex-A8 je použita třetí verze tohoto koprocesoru (VFPv3), která má 32 registrů pro dvojitou přesnost (místo původních 16), instrukci pro přímé načtení konstanty do registru a instrukci pro převod čísla do celočíselného formátu. Registrovou sadu sdílí s jednotkou NEON (tzn. nezávislou na jádře procesoru). Jednotka sice není zřetězená a všechny instrukce musí zpracovávat sekvenčně, ale je nezávislá na jádře procesoru, takže v průběhu výpočtu lze provádět jiné instrukce. Doba výpočtu jednotlivých instrukcí je pro představu shrnuta v tabulce 3.1. Protože je jednotka oddělená od jádra procesoru, je k těmto časům nutné připočítat i latenci způsobenou dekódováním instrukce v koprocesoru. Tuto latenci je však možné skrýt, pokud za poslední instrukcí pro tento koprocesor následují instrukce nezávislé na výsledku výpočtu.

[3] tj. „tiny“ dle normy IEEE-754 [1]

Instrukce	Jednoduchá přesnost	Dvojitá přesnost
FADD	9-10	9-10
FSUB	9-10	9-10
FMUL	10-12	11-17
FDIV	20-37	29-65
FCMP	4-7	4-7

Tabulka 3.1: Doba výpočtu VFP v taktech

3.3 Další rozšíření

V této podkapitole jsou shrnuty další důležité vlastnosti použité architektury, které však nemají vliv na výkon. Jsou zde zařazeny pro úplnost, ale protože nejsou významné pro tuto práci, je jejich popis velice stručný. Zájemci mohou podrobnější informace lze nalézt v [11].

Rozšíření Jazelle

Tato technologie kombinuje rozšíření hardware i software za účelem optimalizace běhu Java Virtual Machine[4]. V tomto procesoru je však implementována pouze minimální podpora, kvůli kompatibilitě software. To znamená, že na tomto procesoru je možné spustit optimalizované prostředí Java Virtual Machine, avšak jeho běh není nijak urychlován.

Bezpečnostní rozšíření TrustZone

Kromě známého privilegovaného módu, určeného pro operační systém, podporuje tato architektura i takzvaný *Secure state*. Veškerý kód, včetně jádra operačního systému, se pak rozděluje na „zabezpečený“ a „nezabezpečený“. Některé části paměti mohou být vyhrazeny pouze pro zabezpečený kód a přechod mezi zabezpečeným a nezabezpečeným módem může být programově kontrolován.

3.4 Podpora překladačů

Stejně jako u každého procesoru i u mobilních zařízení záleží výkon programu na použitém překladači. Protože je ARM standardizovaná architektura, vzniklo v průběhu jeho vývoje několik komerčních překladačů. Ke známějším patří například překladače od firem IAR a Keil. Tyto překladače jsou však určeny pro menší architektury a nepodporují všechna rozšíření Cortex-A8.

K plnohodnotným překladačům bych zařadil pouze komerční překladač firmy RealView (RVCT), překladač firmy Texas Instruments (TMS470) a open-source GNU Compiler Colection (GCC). Tyto překladače podporují všechna rozšíření architektury Cortex-A8. Liší se spíše v podpoře při vývoji než ve výkonnosti kódu. RVCT je stejně jako TMS470 součástí rozsáhlého vývojového prostředí, které obsahuje i nástroje pro automatickou vektorizaci kódu pro jednotku NEON. Překladač GCC ve verzi 4.4.1 již má také částečnou podporu automatické vektorizace. I když je možné jednotku NEON využít pomocí vestavěných funkcí, automatická vektorizace kódu je výhodnější z hlediska údržby kódu (jak je popsáno v kapitole 3.5).

[4]programové prostředí pro běh aplikací napsaných v jazyce Java

Výhodou architektury ARM jsou instrukční sady zaměřené na minimalizaci velikosti kódu (detailněji popsané v kapitole 3.1). Tyto sady však samozřejmě pro plné využití musí být podporovány v překladači. Instrukční sada Thumb je starší a je proto podporována prakticky v každém překladači. Zejména překladače od firem Keil a IAR se specializují na tuto sadu a na minimální velikost výsledného kódu. Jsou totiž určeny pro menší vestavěné systémy. Tam se úspory v paměti pro program projeví u sériové výroby menší potřebnou pamětí ROM a tím i nižšími náklady. Instrukční sada Thumb-2 je novější a určená spíše pro zvýšení výkonu při zachování velikosti kódu předchozí verze Thumb. Většina procesorů podporuje obě verze instrukční sady a možná i proto není novější verze plně podporována v překladačích firem Keil a IAR. Plně podporována je pouze u překladačů GCC, RVCT a TMS470.

Přizpůsobení překladače se týká také volacích konvencí pro funkce vyšších jazyků. Protože systém předávání parametrů funkcím musí být jednotný v celém programu a lze využívat knihovny překládané různými překladači, je nutné specifikovat jednotné rozhraní, aby byly programy kompatibilní na binární úrovni. Takzvané „ABI“ (application binary interface) specifikuje rovněž předávání parametrů funkcím. V souvislosti s jednotkou NEON je podstatné rozšířené ABI, které umožňuje předávat funkcím parametry přes registry této jednotky (podrobněji viz literaturu [4]). V základním ABI je nutné všechny parametry předávat přes základní registrovou sadu, což způsobuje zbytečné přenosy výsledků mezi těmito sadami registrů. Vzhledem k vysoké latenci těchto přenosů (přes 20 taktů procesoru) má podpora rozšířeného ABI významný vliv na výkon. V překladači GCC se tato podpora nastavuje přepínačem `-mhard-float`, ale není v současné verzi (4.4.1-4ubuntu8) implementována.

3.5 Možnosti využití jednotky NEON

Stejně jako rozšíření SSE u běžných architektur, i jednotka NEON je založena na architektuře SIMD. Zpracování více dat jednou instrukcí má sice pozitivní vliv na výkon procesoru, ale přináší řadu problémů do překladačů z vyšších programovacích jazyků. Pro využití těchto jednotek existují v zásadě dva odlišné přístupy:

- Použití vestavěných funkcí
- Automatická vektorizace

Jednotlivé přístupy jsou podrobněji popsány v následujících podkapitolách.

3.5.1 Použití vestavěných funkcí

Při použití vestavěných funkcí má programátor téměř plnou kontrolu nad využitím SIMD jednotky. Každá vestavěná funkce se obvykle přeloží do strojového kódu jako jedna instrukce. Tento přístup má proti přímému vepisování instrukcí velkou výhodu — programátor se nemusí starat o mapování registrů na proměnné.

Protože SIMD jednotky zpracovávají více dat najednou, jsou pro tyto funkce vytvořeny i speciální datové typy (pro přehlednost kódu). Použitím vestavěných funkcí se tak kód stává závislý na cílové architektuře (musí podporovat NEON). Závislost lze sice obejít podmíněným překladem, ale vznikají pak dvě verze kódu. To přinejmenším znesnadňuje údržbu kódu a může být zdrojem špatně odhalitelných chyb.

Jednotka NEON je navíc oddělená od jádra procesoru a pracuje tedy paralelně s ním. To je sice výhodné, protože lze zpracovávat dvě instrukce zároveň, ale je pak nutné řešit

synchronizaci přístupu do paměti. Synchronizace je zajištěna pomocí hardware, ale dokumentace této problematiky je poněkud strohá[5].

Rovněž dokumentace vestavěných funkcí překladače není příliš obsáhlá[6] a zejména struktura datových typů pro načítání dat pomocí prokládaného přístupu do paměti nebyla na první pohled zřejmá. Proto je zde uveden příklad kódu sečtení jednotlivých prvků struktur, kde každá obsahuje tři prvky typu float. V kódu jsou nejprve čtyři struktury načteny do tří registrů, přičemž v každém registru jsou uloženy čtyři odpovídající si prvky struktur. K prvnímu prvku je pak postupně přičten druhý a třetí. Výsledek je pak uložen do pole. V každém prvku výsledného pole je tedy součet jednotlivých prvků odpovídající struktury.

```
tStruktura pole[4];
float vysledky[4];
float32x4x3_t reg = vld3q_f32 (pole);
reg.val[0] = vaddq_f32 (reg.val[0], reg.val[1]);
reg.val[0] = vaddq_f32 (reg.val[0], reg.val[2]);
vst1q_f32 (vysledky, reg.val[0]);
```

Účelem tohoto fragmentu kódu bylo ukázat práci s datovými typy pro načítání dat pomocí prokládaného přístupu do paměti. Práce s ostatními datovými typy a vestavěnými funkcemi je již poměrně intuitivní a lze ji odvodit z dokumentace jednotlivých instrukcí. Ta je dostupná přímo ve zveřejněné dokumentaci k architektuře [11].

3.5.2 Automatická vektorizace

S rozvojem SIMD architektur se jim začaly přizpůsobovat i překladače z vyšších programovacích jazyků. Novější překladače již umí v kódu vyhledat úseky, které lze výhodně zpracovat v SIMD jednotce a tyto úseky přepsat na vektorové instrukce pro tuto jednotku. Hlavní výhodou tohoto přístupu je, že stačí udržovat pouze jednu verzi kódu. Pro architektury s podporou SIMD jednotky se pak pouze zapne automatická vektorizace kódu. I když může být výkon oproti ruční vektorizaci o něco nižší, vyváží ho urychlení vývoje a možnost vektorizace již vytvořených kódů.

Překladač GCC ve verzi 4.4.1 podporuje automatickou vektorizaci kódu i s využitím jednotky NEON. Vektorizace tedy není omezena jen na běžné architektury. Pro vynucení vektorizace lze použít přepínač `-ftree-vectorize`, nebo zapnout maximální optimalizaci pomocí `-O3`, nebo `-Ofast`. Často si však optimalizátor neporadí se strukturou smyčky ve zdrojovém kódu a smyčku ponechá bez vektorizace. V takovém případě je potřeba mít informaci o tom, které smyčky v programu dokázal optimalizátor vektorizovat a které nikoliv. Toho lze dosáhnout pomocí parametru překladače `-ftree-vectorizer-verbose=X`, kde za X lze dosadit číslo od 0 do 7. Tento parametr nastavuje úroveň vypisovaní ladících informací (kde 0 značí bez výpisů). Při zapnutí překladač vypisuje, které smyčky se mu nepodařilo vektorizovat a stručně shrne i z jakého důvodu.

K nejčastějším problémům s vektorizací patří následující:

- Nepodporovaná instrukce (no vectype for stmt)
- Nevyhodnocené datové závislosti (unhandled data-ref)

Bohužel vypisované shrnutí může být často zavádějící. U prvního problému je příčina většinou zřejmá. Jednotka NEON například neobsahuje instrukci pro dělení v plovoucí řádové

[5] vliv na výkon je stručně nastíněn na http://hardwarebug.org/2008/12/31/

[6] spíše jde pouze o seznam funkcí na http://gcc.gnu.org/onlinedocs/gcc/ARM-NEON-Intrinsics.html

čárce, takže nelze vektorizovat cykly obsahující dělení. Druhé shrnutí se například objevuje v případě, kdy je v těle cyklu volána funkce. Protože uvedená verze překladače nemůže funkcím předávat parametry přes registry vektorové jednotky (viz podkapitolu 3.4), je pro vektorizaci nemožné vyhodnotit návratovou hodnotu funkce. Stejné shrnutí se však objeví pro cykly, ve kterých je podmíněné vykonání kódu, nebo proměnná s globální platností (invariantní v těle cyklu). Na lepším hlášení problémů s vektorizací se průběžně pracuje, jak je vidět například z diskuzních skupin věnovaných chybám překladače [7].

3.5.3 Výkon jednotky NEON

Pro orientační porovnání výkonu byly vytvořeny tři verze triviální smyčky, která sčítala čísla ze dvou vstupních polí a výsledek ukládala do třetího. První verze v těle smyčky obsahovala jednoduché sečtení a byla přeložena bez použití vektorizace. Druhá verze byla totožná, ale byla přeložena s použitím automatické vektorizace GCC. Ve třetí verzi jsem pro uložení hodnot použil pole nativního typu vektorové jednotky (`int32x4_t`) a pro výpočet vestavěnou funkci (`vaddq_s32`), která sčítá vždy čtyři čísla zároveň. Následně jsem měřil čas běhu celého programu a průměrné výsledky shrnul v tabulce 3.2. Z tabulky je zřejmé, že jednotka NEON nejvíce urychluje výpočty v plovoucí řádové čárce. Rozdíl je propastný i přes použití VFP koprocesoru (emulovaný výpočet by byl mnohem pomalejší). Pro výpočty s pevnou řádovou čárkou se více uplatnila režie smyčky, jelikož samotná doba výpočtu je proti ní zanedbatelná. To ovlivnilo i automatickou vektorizaci, protože ta musí k režii smyčky přidat ještě kontrolu mezí pole a zarovnání.

Vektorizace	Žádná	Automatická	Ruční
Doba běhu – celočíselně	4,078 s	2,977 s	1,836 s
Doba běhu – jednoduchá přesnost	22,772 s	3,049 s	1,910 s

Tabulka 3.2: Porovnání doby běhu programu, podle typu vektorizace

Výše uvedené porovnání je samozřejmě pouze orientační a nemůže sloužit k přesnému porovnání výkonu u automatické vektorizace. Na to by bylo potřeba více testů a různé případy použití. Lze si však podle něj udělat představu o možnostech využití vektorizace. Pro velkou část programu stačí použít automatickou vektorizaci. Ta zajistí výrazné urychlení výpočtů s plovoucí řádovou čárkou a částečně urychlí i ostatní výpočty. Pro nejkritičtější části programu je výhodné použít ruční vektorizaci, protože lze získat až poloviční nárůst výkonu. Ale s ohledem na přenositelnost kódu je potřeba ručně vektorizovat pouze menší části kódu.

[7] např. na http://www.mail-archive.com/gcc-bugs@gcc.gnu.org/msg282874.html

Kapitola 4

Návrh optimalizací

Tato část práce obsahuje shrnutí vybraných algoritmů a návrh jejich optimalizace pro výše uvedenou architekturu ARM Cortex-A8. Pro přehlednost je rozdělena do podkapitol podle jednotlivých fází zpracování řečového signálu.

Výjimkou je zpracování vstupního signálu, které není uvedeno v samostatné kapitole. Jak bylo již uvedeno výše, tato fáze má minimální vliv na rychlost rozpoznávání [1] a nemělo by tedy smysl v ní ručně optimalizovat kód. Výsledný efekt optimalizací by byl zanedbatelný, takže je tato část ponechána bez optimalizací.

Samotná implementace těchto algoritmů byla převzata z diplomové práce [3] a upravena pro překladač GCC. Původní implementace byla totiž určena pro operační systém symbian a využívala některé jeho knihovny pro správu paměti a práci s řetězci. Tyto funkce byly nahrazeny pomocí C++ knihovny STL.

4.1 Extrakce příznaků

Tato část se může podílet na výsledných časových nárocích, jelikož veškeré výpočty v ní jsou prováděny v plovoucí řádové čárce. Přenesením těchto výpočtů do SIMD jednotky lze výpočet několikrát urychlit a tím snížit celkovou dobu výpočtu.

Protože se však extrakce příznaků provádí pouze jedenkrát pro každý rámec, není tato optimalizace kritická pro další rozšiřování systému, jako například u dekodéru. Proto je optimalizace této části ponechána na automatické vektorizaci kódu pomocí překladače a pozornost je věnována více následujícím částem algoritmu, které jsou pro výkon podstatnější.

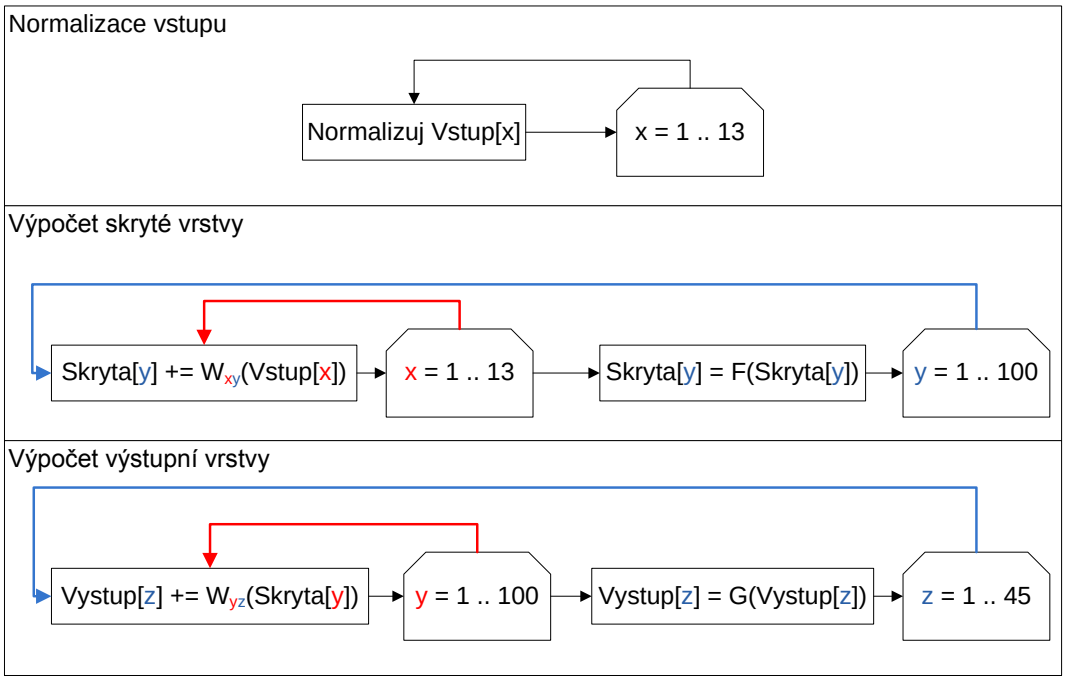
Výraznější optimalizace této části by měla význam pouze v případě dodatečných rozšíření. Pokud by se ukládal delší časový kontext příznaků (jak je navrženo například v [7]), nebo pokud by se použil delší vektor příznaků, mělo by smysl uvažovat o automatické vektorizaci tohoto kódu. Ve stávající implementaci však tato část nemá výrazný vliv na celkovou dobu výpočtu.

Výjimkou z tohoto je výpočet algoritmu rychlé Fourierovy transformace, který se na celkové době výpočtu podílí významnou měrou, jak bude uvedeno v kapitole 4.4. Tento algoritmus je tedy jedinou částí z extrakce příznaků, která bude optimalizována jinak než překladačem.

[1] Například v [6] tvoří cca 0,4% doby výpočtu

4.2 Klasifikace fonémů

Tato fáze zpracování je klíčová pro výkon celého systému. Jednak je časově náročná a jednak ovlivňuje výslednou přesnost rozpoznávání řeči. Navíc bylo i podle profilování nejnáročnější částí při zpracování právě vyhodnocení sítě, takže se optimalizace zaměří právě na tuto část. Řešením podle [3] může být „optimalizace algoritmu Forward pass tak, že se převede na násobení a sčítání matic, pro které existují optimalizované knihovny (např. BLAS)“. Většina těchto knihoven však zatím není optimalizována pro jednotku NEON a pokusy o kompilaci s vektorizací (u knihovny BLAS) byly neúspěšné. Alternativou by mohla být knihovna OpenMAX[2], která mimo jiné podporuje vektorové operace a zpracování signálů (např. výpočet Fourierovy transformace). Tuto knihovnu je sice možné stáhnout zdarma, ale lze ji přeložit pouze v komerčním vývojovém prostředí firmy RealView (RVDS).

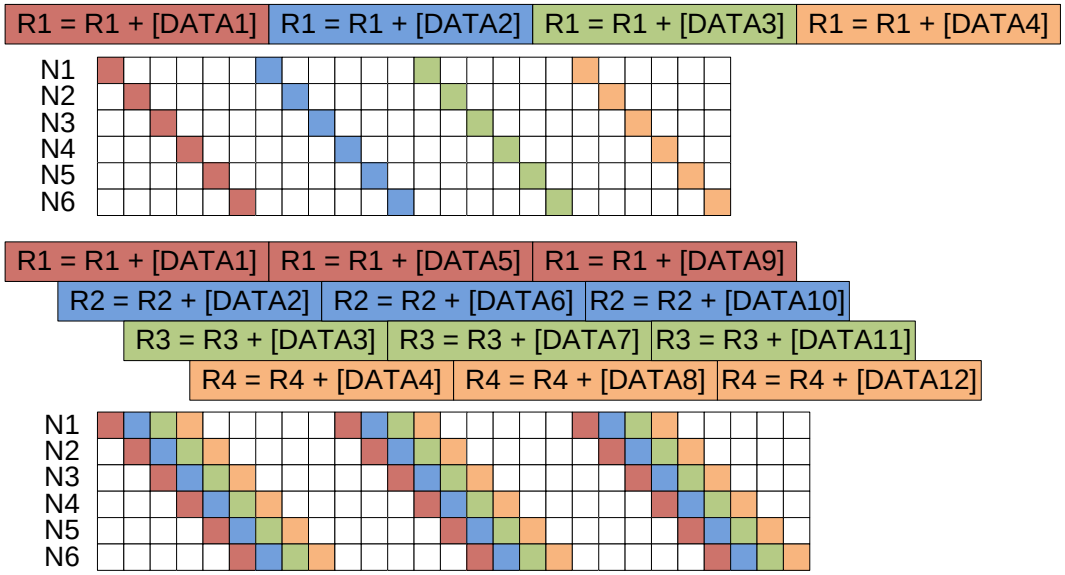


Obrázek 4.1: Schéma vyhodnocení neuronové sítě

Jako výsledné řešení je tedy navržena ruční vektorizace kódu pro vyhodnocení neuronové sítě. Postup vyhodnocení je zjednodušeně naznačen na obrázku 4.1, kde jsou pro názornost dosazeny parametry implementované sítě — ta má 13 vstupů, 100 perceptronů ve skryté vrstvě a 45 ve výstupní vrstvě. V obrázku je také abstrahováno váhování vstupů (jako funkce W) a aktivační funkce perceptronů (jako funkce F a G). Váhování se provádí jednoduchým násobením vstupní hodnoty a definované váhy vstupu, jejíž hodnota je uložena v konstantním poli. Problematice uložení a načtení těchto polí se blíže věnuje kapitola 5.2.2. Jako aktivační funkce jsou použity sigmoida (F) a softmax (G), které jsou podrobněji popsány například v [3]. Cílem této abstrakce schématu je znázornění vnitřních a vnějších smyček výpočtu. Vnitřní smyčka je vždy označena červeně, vnější modře, včetně použitých řídících proměnných cyklu. Z tohoto rozdělení se bude dále vycházet v popisu implementace optimalizací.

[2] bližší popis na http://www.arm.com/products/multimedia/openmax/index.html

Ruční vektorizace pomocí vestavěných funkcí sice na jednu stranu umožňuje vytvářet přehledný zdrojový kód, ale na druhou stranu neumožňuje kontrolovat mapování proměných na registry procesoru. To může být nevýhodné při vektorizaci, ve které na sobě sice jednotlivé průchody cyklem nezávisí, ale překladač pro každý průchod použije stejné mapování registrů, takže instrukce musí čekat na dokončení předchozího cyklu. To je znázorněno na obrázku 4.2. V první části uvedeného obrázku je znázorněno, jak by byla počítána suma hodnot při použití stále stejného registru pro průběžný součet. Aby mohl být obsah tohoto registru načten a inkrementován, musí být nejprve dokončena předchozí instrukce. V druhé části obrázku je znázorněno, jak by vypadal stejný výpočet při použití čtyř různých registrů pro průběžný součet. Tímto způsobem je možné mít rozpracovaných až 6 instrukcí zároveň. Protože mezi jednotlivými instrukcemi nejsou datové závislosti, lze dosáhnout až šestinásobného zrychlení. Samozřejmě je na konci cyklu nutné tyto registry sečíst dohromady, ale tato režie je zanedbatelná vzhledem k urychlení výpočtu těla cyklu. Pro názornost je v obrázku vyznačeno i využití jednotlivých stupňů zřetězené aritmeticko-logické jednotky. Tyto stupně jsou označeny N1 až N6 stejně jako ve schématu 3.1. Zde uvedený obrázek je samozřejmě zjednodušený, jelikož neobsahuje instrukce pro načítání dat z paměti a instrukce pro řízení cyklu. Slouží tedy pouze pro ilustraci principu vektorizace smyčky.



Obrázek 4.2: Závislost výpočtu sumy na mapování registrů

Jak je vidět z obrázku, mapování proměnných na registry má výrazný vliv na dobu výpočtu. Proto se ruční vektorizaci vyhodnocení neuronové sítě zaměří i na tuto oblast. V zásadě existují dvě různá řešení. První řešení spočívá v částečném rozepsání těla cyklu, tak aby několik stejných instrukcí bylo za sebou. Optimalizátor překladače by je poté mohl namapovat podle dostupných volných registrů. Druhým řešením je explicitní součet do čtyř různých proměnných, které jsou za cyklem sečteny. Tím by bylo vynuceno namapování na rozdílné registry.

4.3 Dekodér

I když je výpočet Viterbiho algoritmu poměrně rychlý, zásadním způsobem ovlivňuje možnosti rozšíření detektoru. Pro každé slovo, které má být detekováno, je totiž obvykle potřeba udržovat několik modelů s jeho fonetickým přepisem. A při samotné detekci je nutné pro každý klasifikovaný foném procházet všechny tyto modely. Tato část se tedy jako jediná rozšiřuje spolu s počtem slov ve slovníku. Proto by se optimalizace měla soustředit i na tuto část, i když ne tolik jako na klasifikaci fonémů.

Vektorizace vyhodnocení Viterbiho algoritmu pro jeden model bohužel není možná, kvůli postupným závislostem mezi jednotlivými průchody přes stavy modelu. Vyhodnocení každého stavu závisí na původní hodnotě předcházejícího stavu, takže nelze vyhodnocovat více stavů souběžně. Další možností pro vektorizaci by bylo souběžné vyhodnocení stavů z různých modelů. Jednotlivé modely by však musely mít stejný počet stavů, což nelze obecně zaručit. Vzhledem k poměrně nízkému počtu průchodů cyklem by vektorizace pravděpodobně ani neměla významný vliv. Proto bylo od vektorizace upuštěno a optimalizace bude provedena pouze přepsáním výpočtu do instrukcí jednotky NEON pomocí vestavěných funkcí překladače. I tento přepis může přinést výkonový zisk, jelikož jednotka NEON pracuje paralelně s jádrem procesoru. Na tomto případě je však vidět, že pro vektorovou jednotku jsou vhodnější jednoduché algoritmy založené na hromadných výpočtech (jako například neuronové sítě) namísto algoritmů s komplikovanou strukturou.

Další částí bude optimalizace funkce pro detekci prahu, která je v původní implementaci řešena neoptimálně. Při detekci slova je nutné určit minimální (nejlepší) skóre v daném časovém okně a toto skóre pak porovnávat s prahem pro detekci. Implementované řešení tedy testuje skóre uprostřed tohoto okna, zda není minimální. Teprve poté je porovnává s prahem pro detekci. Tímto způsobem se většinu času zbytečně testuje minimum přes celé časové okno, zatímco jeho hodnota je nad prahem a nemá tedy smysl ji testovat. Efektivnější je nejprve testovat skóre proti hodnotě prahu a až poté kontrolovat, zda je toto skóre minimální v celém časovém okně.

4.4 Výsledky profilování

Pro optimalizaci jakéhokoliv programu je profilování jednou z nejdůležitějších částí. Umožňuje určit části programu, které mají největší vliv na celkovou dobu výpočtu a které se tedy vyplatí optimalizovat. Existuje sice mnoho různých profilovacích nástrojů, ale jen některé z nich jsou nezávislé na architektuře procesoru (případně určené přímo pro architekturu ARM). Pro profilování je například velmi vhodný balíček nástrojů valgrind, jelikož umožňuje detekovat i výpadky v paměti cache. U něj je však podpora pro architekturu ARM zatím pouze plánovaná.

V současnosti nejznámějším nástrojem je patrně gprof, který je takřka v každé linuxové distribuci. Tento nástroj v pravidelných intervalech zjišťuje, ve které funkci se program aktuálně nachází, a podle toho vytváří statistický odhad o době výpočtu jednotlivých funkcí. Na použité platformě sice zobrazoval chybně informaci o celkové době běhu programu[3], ale procentuální odhad doby výpočtu funkcí umožňoval alespoň určit funkce vhodné k optimalizaci.

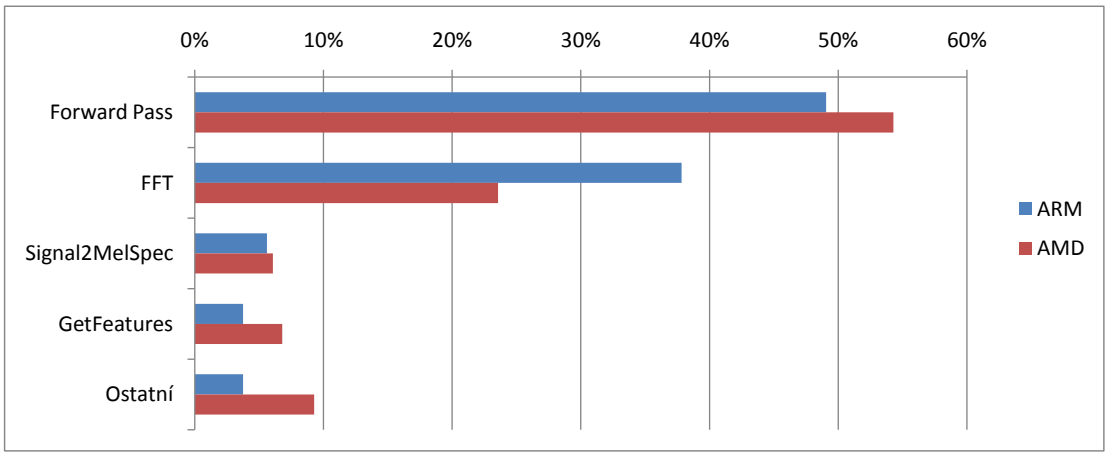
Dalším vhodným nástrojem je oprofile. Tento nástroj sice podporuje architekturu ARM, ale vyžaduje také podporu v jádře operačního systému. Na použitém vývojovém kitu se

[3]Bohužel se nepodařilo zjistit příčinu.

bohužel nepodařilo přeložit jádro s podporou oprofile [4], takže tento nástroj nebyl nakonec použit.

Pro profilování byl nakonec vybrán nástroj gprof. Pomocí něj byly vybrány funkce vhodné k optimalizaci. Přesnější doba výpočtu jednotlivých funkcí pak byla zjištěna jako rozdíl běhu dvou verzí programu, kdy v jedné verzi byla vždy odstraněna všechna volání měřené funkce (výsledky jsou uvedeny v kapitole 5).

Program bez optimalizací byl pro porovnání nejprve profilován na procesoru AMD Opteron s instrukční sadou x86-64 a poté přímo na vývojovém kitu s procesorem OMAP 3530. Protože byl k profilování použit nástroj gprof, bylo vlastní měření provedeno pětkrát a z výsledků vytvořen průměr. Výsledky jsou shrnuty v grafu 4.3. Přesné hodnoty nejsou, vzhledem k rozptylu výsledků nástroje gprof, relevantní — graf slouží pouze pro ilustraci rozdílů mezi architekturami.



Obrázek 4.3: Podíl funkcí na celkové době výpočtu.

Z grafu je vidět, že na obou architekturách zhruba polovinu celkové doby tvoří vyhodnocení neuronové sítě (funkce ForwardPass). Velký rozdíl je však u výpočtu Fourierovy transformace (FFT). Zatímco na „běžné“ architektuře tvořil necelou čtvrtinu celkové doby výpočtu, na architektuře ARM tvořil téměř třetinu této doby. Při optimalizaci se tedy nelze zcela spoléhat na výsledky profilování na běžných architekturách. Je potřeba program testovat na reálném zařízení, nebo alespoň emulátoru. Optimalizace samotného výpočtu FFT je podrobněji popsána v kapitole 5.1.

[4] Překlad pomocí konfigurace z http://people.canonical.com/~ogra/arm/OMAP35x_EVM/kernel/.

Kapitola 5

Implementace optimalizací

Jednotlivé navržené optimalizace byly postupně implementovány a byl zjišťován jejich vliv na dobu výpočtu. Zároveň bylo průběžně zjišťováno rozložení doby výpočtu mezi jednotlivé funkce. Pro testování byl použit vývojový kit s procesorem OMAP 3530 taktovaným na 600MHz.

Pro zjištění rozložení doby výpočtu byla použita stejná metodika jako v kapitole 4.4. Pro zjištění celkové doby výpočtu byl program pětkrát spuštěn pomocí nástroje time. Jako výsledná hodnota byl vzat průměr těchto hodnot. Započítávala se samozřejmě pouze samotná doba běhu programu, tedy položka „user time“. Pokud není u konkrétního testu uvedeno jinak, byl program vždy testován na datech odpovídajících šedesáti sekundám záznamu. Základní parametry pro překlad byly následující:

```
-mfpu=neon -O3 -Wall -ffast-math -fno-exceptions -pg -g
```

Zároveň bylo testováno pomocí průběžných ladících výpisů, zda optimalizace neovlivňují výsledky výpočtu (a tím tedy i přesnost rozpoznávání řeči). Tyto ladící výpisy však výrazně ovlivňovaly dobu výpočtu, proto byl program vždy překládán ve dvou verzích. V jedné verzi byly pomocí maker preprocesoru odstraněny veškeré výstupní operace, druhá verze sloužila pro ověření výsledků. Zhodnocení vlivu veškerých optimalizací na přesnost výsledků je shrnuto v části 5.2.4.

5.1 Extrakce příznaků

I když by podle předpokladů měla tato část rozpoznávání tvořit minimální část celkové doby výpočtu, ukázal se při profilování (blíže viz kapitolu 4.4) pravý opak. Vlivem výpočtu FFT byla tato část téměř nejkritičtější pro rychlost rozpoznávání (srovnatelná s vyhodnocením neoptimalizované neuronové sítě). Proto byl původní výpočet FFT nahrazen knihovnou FFTW. Ta sice zatím nepodporuje výpočty v jednotce NEON, ale i přesto dosahuje lepších výsledků díky optimalizaci algoritmu. Navíc je podpora této knihovny pro jednotku NEON ve vývoji[1].

Výsledné urychlení výpočtu nasazením knihovny FFTW je vidět z tabulky 5.1 V tabulce je uvedena jak doba samotného výpočtu FFT, tak i celková doba výpočtu programu. Doba výpočtu FFT byla získána jako rozdíl doby běhu dvou verzí programu, kde v jedné verzi bylo odstraněno volání funkce pro vyhodnocení FFT. Do této doby je tedy započítáno i volání

[1] podle několika internetových diskuzních skupin, např. http://groups.google.com/group/beagleboard

funkce, které by však mělo být zanedbatelné vzhledem k době výpočtu. Navíc je započítáno do obou výsledků, takže nemá vliv na porovnání metod.

Výsledné urychlení (zhruba 6 % doby výpočtu FFT) sice není výrazné, ale jak už bylo napsáno výše, knihovna FFTW zatím nepodporuje SIMD instrukce jednotky NEON. Lze tedy předpokládat výrazné urychlení, pokud bude program překládán s verzí knihovny, která bude mít tuto podporu doimplementovanou. Toto je také důvodem, proč nebyl původní výpočet FFT ručně vektorizován.

Algoritmus	Původni FFT	Knihovna FFTW
Celková doba běhu programu	9,88 s	9,70 s
Doba běhu funkce FFT	3,23 s	3,04 s
Podíl na celkové době	32,6 %	31,3 %

Tabulka 5.1: Zrychlení výpočtu nasazením knihovny FFTW

5.2 Neuronová síť

Z výsledků profilování vyplývá, že vyhodnocení neuronové sítě je nejdéle trvající částí programu (viz podkapitolu 4.4). Optimalizace této části je tedy nejpodstatnější pro celkovou výkonnost rozpoznávače. Dalším důvodem pro optimalizaci je možnost využití optimalizovaného algoritmu i v jiných oblastech než je rozpoznávání řeči. Neuronové sítě jsou totiž poměrně obecným algoritmem pro klasifikaci.

5.2.1 Automatická vektorizace

Nejsnazší možností optimalizace je automatická vektorizace. Ta je zároveň vhodná i z hlediska přenositelnosti kódu, proto byla testována jako první. Překladač GCC (ve verzi 4.4.1-4ubuntu8) nedokázal automaticky vektorizovat žádnou smyčku ve funkci pro vyhodnocení neuronové sítě. Vektorizoval pouze smyčku pro normalizaci vstupů. To může být způsobeno mnoha okolnostmi, které jsou podrobněji popsány v [12].

Základním předpokladem pro automatickou vektorizaci pomocí jednotky NEON je zapnutí přepínače `--ffast-math`. Jednotka NEON totiž nevyhovuje normě IEEE-754[1] pro výpočty v plovoucí řádové čárce. Výsledky výpočtů se tedy mohou při použití vektorových instrukcí lišit a vektorovou jednotku nelze použít pro urychlení výpočtů, kde je požadována přesnost dle IEEE-754. Přepínač `--ffast-math` značí překladači, že preferujeme rychlost výpočtů nad přesností a shodou s normou. Díky tomu může překladač použít automatickou vektorizaci pomocí instrukcí jednotky NEON.

První úpravou programu bylo přepsání kódu tak, aby struktura těla smyček co nejvíce odpovídala vzorovým smyčkám uvedeným v [12]. Jako krátký příklad zde uvedu přepis smyčky pro výpočet skryté vrstvy neuronové sítě:

```
for (y = 0; y < HiddenLength; ++y) {
 for (x = 0; x < InputLength; ++x)
    HiddenLayer[y] += Input[x] * HiddenWeights[y][x];
  HiddenLayer[y] += HiddenBias[i];
  HiddenLayer[y] = 1.0f / (1.0f + Exp(-HiddenLayer[y]));
}
```

Ta byla přepsána na následující kód:

```
for (y = 0; y < HiddenLength; ++y) {
  float sum = 0;
  const float * weightRow = HiddenWeights[y];
  for (x = 0; x < InputLength; ++x)
    sum += Input[x] * weightRow[x];
  HiddenLayer[y] = sum;
  HiddenLayer[y] += HiddenBias[y];
  HiddenLayer[y] = 1.0f / (1.0f + Exp(-HiddenLayer[y]));
}
```

Z úryvku kódu je vidět, že se funkčnost nijak nezměnila a překladač by pravděpodobně obě verze kódu přeložil úplně stejně (s využitím optimalizací). Pro automatickou vektorizaci se však „zjednodušená“ smyčka stala vektorizovatelnou a překladač ji tedy přeložil s využitím instrukcí pro jednotku NEON. Takto bylo možné vektorizovat pouze vnitřní cyklus výpočtu skryté vrstvy. U dalších smyček se vyskytovalo vždy alespoň jedno z následujících omezení (která jsou podrobněji popsána níže):

- Volání funkce v těle cyklu.
- Nízký počet cyklů.
- Nepodporovaná instrukce v těle cyklu.

Aby překladač GCC mohl vektorizovat tělo cyklu, nesmí se v něm vyskytovat žádné volání funkce s parametry, které jsou součástí výpočtu. Toto omezení vychází z toho, že v GCC zatím není implementována podpora takzvaného „hard-float abi“. Problém podpory ABI je blíže popsán v kapitole 3.5. Protože není podporováno „hard-float abi“, nelze funkcím předávat parametry přes registrovou sadu jednotky NEON, takže se parametry musí kopírovat mezi jednotlivými registrovými sadami. Tím se samozřejmě degraduje výkon (latence těchto přenosů je vysoká) a vektorizace ztrácí smysl.

Další podmínkou pro vektorizaci je vysoký počet cyklů. Protože překladač musí při automatické vektorizaci přidat před smyčku inicializaci (s kontrolou zarovnání), musí se tělo smyčky zopakovat tolikrát, aby se tato přidaná režie na začátku smyčky vyrovnala urychlením těla smyčky. Proto se překladač nepokouší vektorizovat krátké smyčky s nízkým počtem opakování. Dále je vhodné zarovnat počet opakování na násobek čtyř, aby nebylo nutné za výpočet smyčky přidávat dopočítání nezarovnaných položek.

Pro automatickou vektorizaci je také nutné, aby vektorová jednotka podporovala všechny instrukce nutné pro výpočet. Ve verzi architektury ARM Cortex-A8 nemá jednotka NEON instrukci pro dělení čísel — ta je přidána až ve verzi ARM Cortex-A9. Proto nelze automaticky vektorizovat smyčky obsahující ve výpočtu dělení. Rovněž výpočet exponenciály je v knihovně `math.h` implementován jako volání vestavěné funkce překladače, které samozřejmě nelze nahradit instrukcí jednotky NEON. I když by bylo teoreticky možné toto omezení obejít (například pomocí instrukcí VFP), je to patrně příliš komplexní problém pro automatickou vektorizaci.

Vzhledem k výše uvedeným omezením, byla vektorizována pouze vnitřní smyčka výpočtu skryté vrstvy a smyčka pro normalizaci vstupu. Ostatní smyčky, tj. vnější cyklus skryté vrstvy a vyhodnocení vstupní a výstupní vrstvy, nebyly automaticky vektorizovány. I přes to byl vliv úprav na dobu výpočtu značný, jak ukazuje tabulka 5.2.

Verze smyčky	Původní	Původní	Upravená	Upravená
Použita vektorizace	Ne	Ano	Ne	Ano
Celková doba běhu	9,51 ± 0,03 s	9,70 ± 0,03 s	6,85 ± 0,06 s	6,42 ± 0,05 s
Neuronová síť	4,90 ± 0,03 s	5,04 ± 0,03 s	2,24 ± 0,05 s	1,77 ± 0,08 s
Podíl na celkové době	51,5 %	52,0 %	32,7 %	27,5 %

Tabulka 5.2: Zrychlení dosažené automatickou vektorizací výpočtu neuronové sítě

V tabulce jsou uvedeny doby běhu pro 4 verze programu. V první verzi nebyla povolena automatická vektorizace při kompilaci funkce pro vyhodnocení neuronové sítě. Díky tomu nebyla vektorizována ani normalizace vstupů ani výpočet skryté vrstvy. Ve druhé verzi byla povolena vektorizace, ale byla použita původní verze kódu, takže došlo k vektorizaci pouze u smyčky pro normalizaci vstupů. U třetí a čtvrté verze byly použity optimalizované smyčky, takže ve čtvrté verzi překladač automaticky vektorizoval jak smyčku pro normalizaci vstupů, tak i vnitřní smyčku výpočtu skryté vrstvy neuronové sítě.

V tabulce je uvedena jak celková doba výpočtu programu, tak i doba samotného vyhodnocení neuronové sítě (včetně normalizace vstupů). Doba vyhodnocení neuronové sítě byla získána jako rozdíl doby běhu dvou verzí programu, kde v jedné verzi bylo odstraněno volání funkce pro výpočet neuronové sítě. Do této doby je tedy započítáno i volání funkce, které by však mělo být zanedbatelné vzhledem k době výpočtu. Protože již jde o poměrně nízké hodnoty, může být výsledek ovlivněn chybou měření. Pro orientační určení přesnosti byly do tabulky přidány i hodnoty směrodatné odchylky zaokrouhlené nahoru na jednu platnou číslici.

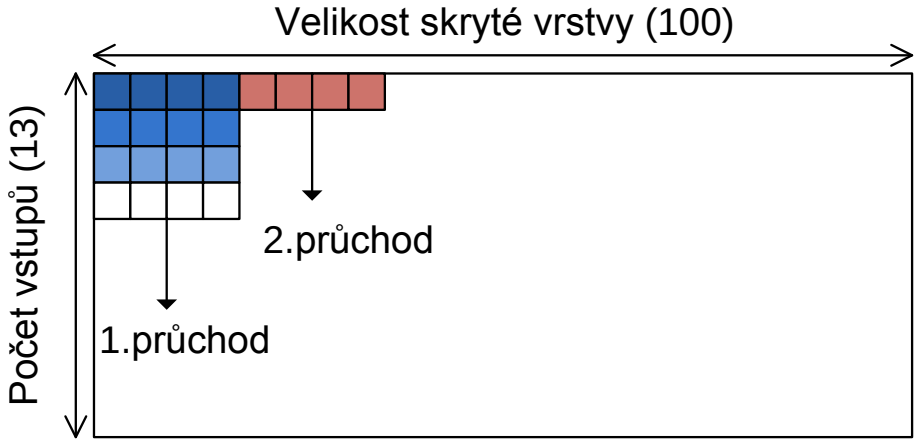
Další úpravou bylo označení všech ukazatelů použitých ve smyčce pomocí kvalifikátoru `__restrict__`. Ten značí překladači, že oblasti paměti, na které ukazatele odkazují, se nepřekrývají. Pro vektorizaci to sice není nutná podmínka, ale toto označení značně zjednodušuje vektorizaci. Překladač totiž není schopen z kódu zjistit, zda se oblasti nepřekrývají, a musí vytvořit dvě verze kódu. Za běhu se potom vybere vektorizovaná verze kódu jen tehdy, pokud se oblasti paměti nepřekrývají. Tato úprava však neměla žádný vliv na výkon a nebyl zaznamenán ani žádný významný vliv na velikost výsledného kódu. Pro kontrolu byl program znovu přeložen s parametrem překladače `-ftree-vectorizer-verbose=3`. Ten zajistí vypsání podrobnějších informací o průběhu vektorizace. Ve verzi programu bez kvalifikátoru `__restrict__` se opravdu vytvořily dvě verze kódu smyčky a kontrola překrytí oblastí v paměti (dle zprávy překladače „created 2 versioning for alias checks“), zatímco ve verzi s kvalifikátorem `__restrict__` byla vytvořena pouze jedna verze kódu.

5.2.2 Ruční vektorizace

Protože neuronová síť zásadním způsobem ovlivňuje přesnost rozpoznávání, je předpokládáno její rozšíření a pravděpodobně se tedy bude na celkové době výpočtu podílet významněji. Proto byl její výpočet kromě automatické vektorizace urychlen i ručním přepsáním smyček pomocí vestavěných funkcí překladače které přímo generují instrukce pro vektorovou jednotku NEON. Jak už bylo uvedeno v kapitole 3.5.3, lze tímto způsobem zkrátit dobu výpočtu až o třetinu oproti automatické vektorizaci.

Nejprve byla přepsána smyčka pro výpočet skryté vrstvy (viz obrázek 4.1). Ve vnitřní smyčce se zde provádí pro každý perceptron skryté vrstvy váhování vstupních hodnot. Koeficienty pro toto váhování lze uložit v paměti jako matici o rozměrech $N_{in} \times N_{hidden}$, nebo

$N_{hidden} \times N_{in}$, kde N_{in} značí délku vstupního vektoru a N_{hidden} počet perceptronů ve skryté vrstvě. Vektorová jednotka musí načítat hodnoty po řádcích, protože jsou v paměti uloženy bezprostředně za sebou a jednotka neumí načítat data z větším rozestupem než 4. Z tohoto důvodu je výhodnější uspořádání $N_{hidden} \times N_{in}$, kde je délka řádku (tj. 100 pro implementovanou síť) dělitelná čtyřmi a není tedy potřeba ošetřovat výpočet zbývajících prvků na konci řádku. Ve vnitřním cyklu je však nutné projít všechny vstupní hodnoty pro jednotlivý perceptron skryté vrstvy a teprve potom lze vyhodnotit aktivační funkci perceptronu. Tím vzniká neoptimální průchod maticí, jak je naznačeno na obrázku 5.1. Do vektorové jednotky se sice načítají data po čtveřicích, ale v každém kroku z jiného řádku matice. Tím může často docházet k výpadkům v paměti cache. Navrženým řešením tohoto problému je projít matici po řádcích a postupně tímto způsobem vypočítat sumu váhovaných vstupů, bez výpočtu aktivační funkce. Aktivační funkci pak lze vyhodnotit v následujícím cyklu přes všechny perceptrony skryté vrstvy. Tím by sice vznikla ve výpočtu další smyčka (spolu s další režií), ale omezily by se výpadky paměti cache. Pro tuto konkrétní síť však nelze čekat výkonový zisk v testech. Procesor má 16 kB paměti cache (L1) pro data a matice koeficientů má velikost 5,1 kB, takže velmi pravděpodobně by k žádným výpadkům nedocházelo (kromě počátečního načtení). Protože na vývojovém kitu nebyl k dispozici nástroj pro zjištění počtu výpadků cache a výkonový zisk by se projevil až u větší neuronové sítě, nebyla tato optimalizace implementována.



Obrázek 5.1: Schéma průchodu maticí váhových koeficientů.

Pro ruční přepsání kódu bylo nutné změnit i datové typy použité při výpočtu. Proto byla, zároveň s přepsáním smyčky pro vyhodnocení skryté vrstvy, přepsána i smyčka pro vyhodnocení výstupní vrstvy (aby se kód nestal příliš nepřehledným). U té je situace jednodušší, jelikož pole vah pro výpočet vstupů perceptronů má již z principu více sloupců než řádků a není potřeba jej transponovat pro optimální postup výpočtu jako v předchozím případě. Pro každý perceptron se sčítají výstupy skryté vrstvy násobené odpovídajícím váhovým koeficientem. Pro násobení a sečtení lze využít vektorovou instrukci `vmlaq`, která vynásobí jednotlivé složky dvou vektorů a výsledek přičte (po složkách) k třetímu vektoru. Tato konkrétní instrukce pracuje s vektory o čtyřech složkách (což je maximum pro tuto verzi architektury NEON). Protože však v tomto případě výsledkem musí být skalár (vstup jednoho perceptronu výstupní vrstvy), je nutné čtyři složky vektoru po ukončení smyčky sečíst a teprve tento výsledek uložit do paměti. Tato režie navíc je však zanedbatelná, vzhledem k urychlení dosaženému vektorovým zpracováním v těle smyčky.

Výsledky ruční optimalizace jsou shrnuty v tabulkách 5.3 a 5.4. Ve druhé tabulce jsou výsledky po zpracování dvojnásobné délky dat (tj. 120 sekund), aby byla omezena chyba měření. Chyba totiž u doby vyhodnocení samotné neuronové sítě přesáhla 5 %. Původní tabulka 5.3 je zde ponechána pro porovnání s předchozími optimalizacemi. V tabulkách je opět uvedena jak celková doba výpočtu programu, tak i doba samotného vyhodnocení neuronové sítě (včetně normalizace vstupů). Doba vyhodnocení neuronové sítě byla získána jako rozdíl doby běhu dvou verzí programu, kde v jedné verzi bylo odstraněno volání funkce pro výpočet neuronové sítě.

Použita vektorizace	Automatická	Ruční
Celková doba běhu programu	6,42 ± 0,08 s	5,82 ± 0,05 s
Doba vyhodnocení neuronové sítě	1,77 ± 0,1 s	1,18 ± 0,06 s
Podíl na celkové době	27,6 %	20,2 %

Tabulka 5.3: Zrychlení dosažené ruční vektorizací výpočtu neuronové sítě

Použita vektorizace	Automatická	Ruční
Celková doba běhu programu	12,70 ± 0,08 s	11,61 ± 0,06 s
Doba vyhodnocení neuronové sítě	3,48 ± 0,09 s	2,38 ± 0,07 s
Podíl na celkové době	27,4 %	20,5 %

Tabulka 5.4: Zrychlení dosažené ruční vektorizací výpočtu neuronové sítě při použití dvojnásobné délky vstupních dat

Poslední optimalizací bylo rozepsání vektorizovaného kódu tak, aby bylo v těle cyklu více stejných instrukcí za sebou (viz návrh v kapitole 4.2). Díky rozepsání by měl mít optimalizátor překladače možnost mapovat proměnné na různé registry a tím odstranit závislosti mezi výpočty. Po zpětném překladu vygenerovaného kódu bylo ale zjištěno, že překladač všechny instrukce vygeneroval prakticky totožným způsobem a vliv na výkon byl samozřejmě zanedbatelný. Proto byl kód znovu přepsán a pro průběžné součty byly použity různé proměnné, které byly na konci cyklu sečteny. Díky poslední úpravě již překladač vygeneroval kód s různým mapováním registrů, ale vliv na výkon nebyl znatelný (i přes opakované měření). To je pravděpodobně způsobeno režií cyklu, která může být vyšší než je doba výpočtu ve vektorové jednotce. Díky tomu se doba výpočtu ve vektorové jednotce neprojeví na celkové době průchodu cyklem.

5.2.3 Výpočet exponenciální funkce

Posledním urychlením neuronové sítě byla optimalizace výpočtu exponenciální funkce. Nejprve bylo testováno makro `FAST_EXP` popsané v diplomové práci [3]. Původní kód tohoto makra, který navrhl Nicol N. Schraudolph, se mi nepodařilo nalézt. Odkaz na článek citovaný v [3] totiž není platný. Pravděpodobně je to způsobeno tím, že přístup k článku je nyní zpoplatněn (viz literaturu [9]). K dispozici je ale novější verze tohoto makra implementovaná jako inline funkce v jazyce C++. Implementace i matematický základ jsou podrobně popsány v [2]. Implementace se v principu neliší od makra implementovaného v [3], jelikož také vychází z algoritmu, který navrhl Nicol N. Schraudolph. Toto makro je však pouze aproximací výpočtu exponenciální funkce a musí být tedy kontrolováno, zda (resp. jakou

měrou) ovlivňuje výsledky vyhodnocení neuronové sítě. Porovnání výstupů neuronové sítě je shrnuto v kapitole 5.2.4 spolu se zhodnocením vlivu dalších optimalizací na přesnost výpočtu. Zhodnocení vlivu této optimalizace na dobu výpočtu je uvedeno v závěru této kapitoly.

Další možností pro optimalizaci exponenciální funkce je přesunutí jejího výpočtu do jednotky NEON. Standardní knihovna matematických funkcí totiž nevyužívá instrukce pro tuto jednotku, což může mít vliv na dobu výpočtu. Přepisem knihovny matematických funkcí pro jednotku NEON se zabývá projekt math-neon[2]. Protože zatím nebyla publikována žádná stabilní verzi knihovny, byla z jeho zdrojových kódů převzata pouze implementace exponenciální funkce. Ta kvůli nepodporovanému „hard-float abi“ (viz kapitolu 3.4) musí parametr i návratovou hodnotu předávat přes základní registrovou sadu, což může prodloužit dobu vyhodnocení funkce. Navrhovaným řešením bylo použití funkce inline[3]. Díky tomu se kód volání funkce rozepíše do instrukcí, které má k dispozici optimalizátor. Zbytečný přenos se tedy eliminuje optimalizátorem překladače. Do inline funkce musel být nakonec vložen celý kód výpočtu, protože při tomto způsobu řešení nesmí být ve funkci volání pomocné funkce pro výpočet. Toto volání by totiž z pohledu překladače nemělo žádné parametry (ty se předávají přes registry jednotky NEON) a optimalizátor by poté zoptimalizoval zbytek výpočtu bez ohledu na volání pomocné funkce (většinou tedy chybně).

Stejně jako v případě makra `FAST_EXP`, i u této optimalizace byl zkoumán vliv na přesnost výsledku. Zhodnocení vlivu na přesnost je uvedeno v části 5.2.4.

Vliv optimalizace výpočtu exponenciální funkce na dobu výpočtu programu byl relativně nízký (vzhledem k celkové době). Program byl tedy testován nad dvojnásobnou délkou dat (tj. 120 sekund), aby byla omezena nepřesnost měření. Výsledky jsou shrnuty v tabulce 5.5. V prvním sloupci jsou uvedeny časy pro verzi programu s exponenciální funkcí ze standardní knihovny cmath. V druhém sloupci je verze programu s výpočtem pomocí makra `FAST_EXP` a ve třetím sloupci jsou odpovídající výsledky pro inline funkci[4] s využitím instrukcí jednotky NEON. Ve všech verzích byl použit ručně optimalizovaný algoritmus pro výpočet neuronové sítě. Doba vyhodnocení neuronové sítě byla opět získána jako rozdíl doby běhu dvou verzí programu, kde v jedné verzi bylo odstraněno volání funkce pro výpočet neuronové sítě. Podobným způsobem byla zjišťována doba vyhodnocení samotné exponenciální funkce.

Použitá metoda	cmath	`FAST_EXP`	math-neon
Celková doba běhu programu	11,59 ± 0,06 s	10,70 ± 0,06 s	10,78 ± 0,03 s
Doba vyhodnocení neuronové sítě	2,38 ± 0,08 s	1,49 ± 0,07 s	1,57 ± 0,06 s
Podíl na celkové době	20,5 %	14,0 %	14,5 %

Tabulka 5.5: Zrychlení dosažené optimalizací výpočtu exponenciální funkce (použití dvojnásobné délky vstupních dat)

Nebyly však odstraňovány volání funkce, ale funkce byla upravena tak, aby pouze vracela svůj parametr. Tím vznikne velmi hrubý odhad o době výpočtu. Je totiž možné, aby optimalizátor při této úpravě inline funkce zoptimalizoval i související výpočty a tím celkovou dobu zkrátil o větší část, než je pouhá doba výpočtu exponenciální funkce. Tato doba byla navíc příliš malá na to, aby měření bylo vypovídající. Proto je nutné brát výsledky v tabulce

[2] dostupný na http://code.google.com/p/math-neon/ pod licencí GNU Lesser General Public License

[3] původní verze kódu vyhovovala standardu ANSI C, proto nevyužívala inline funkce

[4] funkce byla vytvořena na základě implementace z projektu math-neon

5.6 pouze jako orientační. Lze na nich však ukázat, že doba výpočtu exponenciální funkce již není podstatná pro celkovou dobu běhu programu a nemělo by smysl ji dále optimalizovat.

Použitá metoda	cmath	`FAST_EXP`	math-neon
Celková doba běhu programu	11,59 ± 0,06 s	10,70 ± 0,06 s	10,78 ± 0,03 s
Výpočet exponenciální funkce	1,04 ± 0,08 s	0,15 ± 0,06 s	0,22 ± 0,09 s
Podíl na celkové době	9,0 %	1,4 %	2,1 %

Tabulka 5.6: Orientační měření doby výpočtu exponenciální funkce

5.2.4 Vliv na přesnost výpočtů

Protože optimalizace navržené v této kapitole nemusí vždy zachovávat přesnost výpočtů, byly výstupy neuronové sítě y_i porovnávány s výstupy verze bez optimalizací b_i. Z rozdílu každých jednotlivých výstupů pak byla spočítána odchylka Δ podle vzorce 5.1.

$$\Delta_i = \frac{y_i - b_i}{b_i} \cdot 100\% \tag{5.1}$$

Z absolutních hodnot těchto odchylek byla poté spočítána maximální a průměrná hodnota. Výsledky shrnuje tabulka 5.7. Z tabulky je zřejmé, že u většiny optimalizací je rozdíl dán

Metoda optimalizace	Průměrná odchylka (%)	Maximální odchylka (%)
Automatická vektorizace	$1,5 \cdot 10^{-5}$	$3,8 \cdot 10^{-4}$
Ruční vektorizace	$1,8 \cdot 10^{-5}$	$3,8 \cdot 10^{-4}$
Math-neon exp	$1,5 \cdot 10^{-5}$	$3,8 \cdot 10^{-4}$
Makro `FAST\_EXP`	0,44	4,0

Tabulka 5.7: Orientační měření doby výpočtu exponenciální funkce

pouze zaokrouhlovací chybou při zaokrouhlování výsledku a je zanedbatelný. Pouze u výpočtu exponenciální funkce pomocí makra `FAST_EXP` (viz podkapitolu 5.2.3) jsou rozdíly ve výstupech neuronové sítě významné. Maximální rozdíl tvořil 4% hodnoty výstupu, což by pravděpodobně již mohlo ovlivnit výsledky rozpoznávání. Pro výpočet exponenciální funkce tedy přicházejí v úvahu tyto alternativy:

- Pro výpočet i trénování neuronové sítě použít makro `FAST_EXP`.
- Pro výpočet neuronové sítě použít inline funkci optimalizovanou pro jednotku NEON.

Pokud je pro trénování neuronové sítě použita stejná funkce jako pro její vyhodnocení, nemá nepřesnost při výpočtu vliv na kvalitu rozpoznávání a lze s výhodou využít makro `FAST_EXP`. Pokud je však síť trénována s přesným výpočtem exponenciální funkce (například pomocí různých nástrojů pro trénování), lze využít optimalizovanou exponenciální funkci.

Ostatní optimalizace měly na přesnost výpočtu zanedbatelný vliv, takže lze předpokládat, že využití vektorových instrukcí jednotky NEON neovlivní výslednou přesnost detekce slov.

5.3 Viterbiho dekodér

I když by se z předchozích částí mohlo zdát, že dekodér je z hlediska doby výpočtu zanedbatelný, opak je pravdou. Samotný algoritmus detekce je sice poměrně rychlý, ale detekce se provádí pro každý model slova a závisí na počtu fonémů v modelu. Doba výpočtu této části je tedy přímo úměrná počtu a délce slov ve slovníku. V předchozích kapitolách na optimalizované části neměla velikost slovníku vliv, takže ve slovníku byla pouze tři detekovaná slova. V této části však již záleží na velikosti slovníku, takže byl naplněn 200 stejnými slovy o délce sedm znaků. Rozdílnost slov nemá na výsledky testů vliv, protože dekodér zpracovává všechny modely nezávisle na sobě. Kvůli tomuto rozšíření však nelze výsledky z této kapitoly srovnávat s předchozími.

Jak už bylo popsáno v části 4.3, původní implementace dekodéru nebyla zdaleka optimální. Proto musela být před vektorizací upravena na úrovni zdrojového kódu. Nejprve byla upravena smyčka určená pro detekci prahu, která v původní implementaci vypadala následovně:

```
for (TInt i = 1; i <= KMinWinLen; ++i)
{
    TState score = i < KMinWinLen ? aModel->scores[i] : newScore;
    if (score.score < minScore.score)
        isMinimal = false;
    aModel->scores[i-1] = score;
}
```

Ve smyčce se zjišťuje, zda je ve vybraném časovém okně (KMinWinLen) skóre uprostřed tohoto okna minimální (viz podkapitolu 4.3). Zároveň s tím je toto okno posouváno v čase. Hned první podmínka v cyklu se zdá být zbytečná a může být přesunuta až za cyklus. Přesunutí podmínky však mělo negativní dopad na rychlost výpočtu, jak je vidět z druhého sloupce tabulky 5.8. To je pravděpodobně způsobeno chybnou predikcí skoků, která je obecně náchylná na skoky uvnitř těla smyčky. Pokud přepíšeme i druhou podmínku, aby v těle cyklu nebyl žádný skok, dosáhneme poměrně výrazného urychlení, jak je vidět ve třetím sloupci tabulky 5.8.

Optimalizace	Žádná	1. podmínky	2. podmínky
Celková doba běhu programu	9,36 ± 0,04 s	9,99 ± 0,06 s	8,95 ± 0,04 s
Viterbiho dekodér	4,34 ± 0,06 s	4,97 ± 0,08 s	3,93 ± 0,06 s
Podíl na celkové době	46,4 %	49,8 %	43,9 %

Tabulka 5.8: Vliv vynechání podmínky v těle cyklu.

Další úpravou byla počáteční kontrola hodnoty uprostřed časového okna proti prahu detekce. Pokud je tato hodnota nad prahem, je použita alternativní smyčka bez kontroly minima (jak bylo navrženo v 4.3). Vliv této úpravy je vidět v tabulce 5.9. Tato úprava se sice neprojeví v případě, kdy je skóre modelu pod úrovní prahu, ale tento případ není tak častý. Většina modelů je po celou dobu detekce nad tímto prahem a pouze v případě, kdy je vysloveno odpovídající (nebo podobné) slovo, klesne tato hodnota pod nastavený práh. Proto se u většiny modelů tato optimalizace uplatní.

Poslední optimalizací bylo přepsání funkce pro vyhodnocení modelu. Ta byla přepsána pomocí vestavěných funkcí překladače tak, aby se na jejím výpočtu co nejvíce podílela

Kontrola prahu	Zpětná	Dopředná
Celková doba běhu programu	9,36 ± 0,04 s	8,08 ± 0,03 s
Viterbiho dekodér	4,34 ± 0,06 s	3,04 ± 0,05 s
Podíl na celkové době	46,4 %	37,7 %

Tabulka 5.9: Vliv optimalizace dopřednou kontrolou prahu.

jednotka NEON. I když se nejednalo o vektorizaci, vliv na výkon byl znatelný, jak je vidět v tabulce 5.10. Tento vliv na výkon je dán za prvé tím, že jednotka NEON pracuje paralelně s jádrem procesoru. Za druhé je zrychlení dáno také použitím speciální instrukce pro výběr maxima ze dvou hodnot. Tato instrukce je generována vestavěnou funkcí `vmax_f32`. Díky tomu odpadá další podmínka v těle cyklu, která mohla mít vliv na predikci skoků.

Kód algoritmu	Původní	Přepsaný
Celková doba běhu programu	8,08 ± 0,03 s	7,52 ± 0,04 s
Viterbiho dekodér	3,04 ± 0,05 s	2,50 ± 0,06 s
Podíl na celkové době	37,7 %	33,2 %

Tabulka 5.10: Optimalizace Viterbiho algoritmu přepsáním na instrukce jednotky NEON.

Kapitola 6

Zhodnocení

V předchozí kapitole byly popsány implementované optimalizace a u každé jednotlivé části byl zhodnocen její přínos na zrychlení celkové doby výpočtu. V této kapitole budou sepsány doby výpočtu jednotlivých částí a jejich zastoupení v celkové době běhu programu (tzn. profil). Zároveň bude také zhodnoceno dosažené zrychlení.

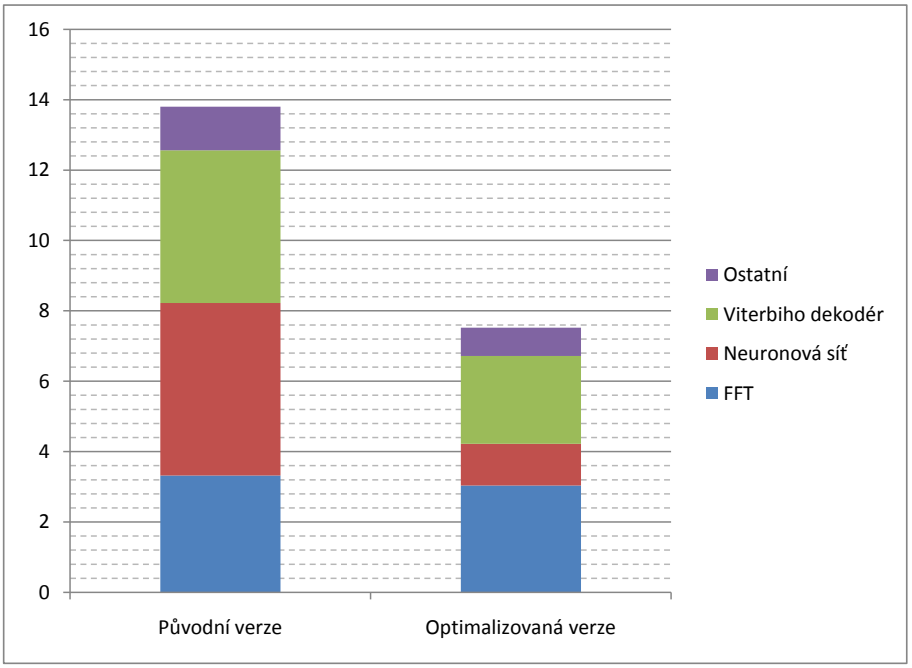
V tabulce 6.1 jsou shrnuty jednotlivé optimalizované algoritmy, jejich doba výpočtu před optimalizací a po optimalizaci. V posledním sloupci tabulky je uvedeno procentuální urychlení, kde je jako základ brána původní doba výpočtu. V tabulce jsou zahrnuty pouze algoritmy, které se významnou měrou podílely na celkové době výpočtu.

	Původní verze	Optimalizovaná verze	Urychlení
FFT	3,32 s	3,04 s	8,4 %
Neuronová síť	4,90 s	1,18 s	75,9 %
Viterbiho dekodér	4,34 s	2,50 s	42,4 %

Tabulka 6.1: Urychlení výpočtu jednotlivých algoritmů.

Z tabulky je vidět, že nejvýrazněji bylo urychleno vyhodnocení neuronové sítě. Základem tohoto urychlení bylo zjednodušení vnitřních smyček výpočtu a jejich vektorizace pomocí instrukcí koprocesoru NEON. Tím byla doba vyhodnocení sítě zkrácena o více jak 3 sekundy. Další zrychlení (přes 0,4 sekundy) bylo dosaženo optimalizací výpočtu exponenciální funkce. Ta byla optimalizována natolik, že se nakonec podílela zanedbatelným způsobem na době vyhodnocení neuronové sítě (přibližně deseti procenty). Viterbiho dekodér byl rovněž urychlen velice výrazně, zejména díky obecné optimalizaci detekce prahu. Výpočet Fourierovy transformace byl nahrazen optimalizovanou knihovnou FFTW. To sice nepřineslo výrazné urychlení, ale v budoucnu lze předpokládat dokončení optimalizace této knihovny pro vektorovou jednotku NEON, takže by výsledné urychlení bylo znatelnější.

Na obrázku 6.1 je zobrazen graf celkové doby běhu programu, ve kterém bylo do slovníku rozpoznávaných slov vloženo 200 slov o délce 7 znaků. V grafu je porovnána původní verze programu bez jakýchkoliv optimalizací a optimalizovaná verze programu. Pro výpočet exponenciální funkce bylo v optimalizované verzi použito makro `FAST_EXP`. Celkově byla doba výpočtu snížena z původních 13,8 sekund na výsledných 7,5 sekundy. To odpovídá celkovému zrychlení o 45,5 % původní doby výpočtu. Pokud by detektor byl použit pro online detekci klíčových slov, pracoval by pouze při dostupných datech a implementované optimalizace by se projevily pouze na snížení zátěže procesoru. Výše uvedeným časům odpovídá využití procesoru na 23 % s původní verzí detektoru a využití na 12,5 % s optimalizovanou verzí.



Obrázek 6.1: Graf rozložení celkové doby běhu programu.

Implementovaný detektor je tedy možné použít pro online detekci klíčových slov, tak jak byl původně navržen. Díky implementovaným optimalizacím lze výrazně snížit zátěž procesoru. Toho se dá využít buď ke snížení spotřeby zařízení, nebo k implementaci rozšíření na zvýšení přesnosti detekce. Protože tento detektor často detekuje falešné výskyty klíčového slova (viz literaturu [3]), lze také uvažovat o použití dvou různých stupňů detekce. Na pozadí by pracoval tento detektor s nastaveným vyšším prahem citlivosti a při detekci klíčového slova by toto slovo bylo verifikováno druhým spolehlivějším (tzn. i výpočetně náročnějším) algoritmem. Tím by byla zajištěna nízká spotřeba v „pohotovostním režimu“ a zároveň přesná detekce vyslovení klíčového slova.

Díky obecnému využití neuronových sítí lze tento optimalizovaný algoritmus použít i v jiných oblastech než je rozpoznávání řeči. Proto v následujícím seznamu shrnu její parametry, pro účely porovnání s ostatními sítěmi. Tím lze orientačně určit možnosti použití jiné neuronové sítě na mobilním zařízení. Podrobněji je síť popsána v kapitole 2.3.

- Počet vstupů: 13
- Počet neuronů ve skryté vrstvě: 100
- Počet výstupů: 45
- Aktivační funkce: Sigmoida a softmax
- Průměrná doba vyhodnocení: 496μs

Kapitola 7

Závěr

Rozpoznávání řeči je poměrně obsáhlá problematika a kompletní optimalizace detektoru klíčových slov, včetně porovnání jednotlivých algoritmů z hlediska poměru úspěšnost/složitost, by překročila rámec této práce. Proto se zaměřuje pouze na optimalizace pro danou architekturu ARM Cortex-A8, která by v budoucnu měla pokrývat většinu trhu s mobilními zařízeními. Asi nejvýznamnější částí této architektury je SIMD jednotka NEON. Ta je určena pro multimediální aplikace a několikanásobně zvyšuje výkon u výpočtů v plovoucí řádové čárce. Zároveň také umožňuje zhruba dvojnásobné urychlení vektorových výpočtů v pevné řádové čárce. Nevýhodou této jednotky je nutnost použití speciálních vektorových instrukcí, takže pro optimalizaci programu je nutné použít buď automatickou, nebo ruční vektorizaci kódu.

V této práci byl optimalizován detektor klíčových slov, převzatý z diplomové práce Ing. Tomáše Cipra. Ten byl již při návrhu určen pro online detekci na mobilním zařízení a optimalizován na úrovni algoritmů, ale nebyl nikdy předtím testován na reálném zařízení. Proto nebylo zpočátku zřejmé, zda bude detektor použitelný pro online detekci. Původní implementace detektoru byla tedy přepsána, aby ji bylo možné přeložit bez knihoven operačního systému Symbian, a následně byly jednotlivé algoritmy z tohoto detektoru optimalizovány na úrovni strojového kódu přímo pro danou architekturu. Využitím instrukcí vektorové jednotky se podařilo zkrátit dobu vyhodnocení neuronové sítě na 25 % původní doby a celkově se detektor podařilo zrychlit o 45 %.

Výsledný detektor na testovacím vývojovém kitu zpracoval 60 sekund záznamu řeči za 7,5 sekundy. Vzhledem k této výkonové rezervě by dalším pokračováním této práce mělo být rozšíření vektoru příznaků a použité neuronové sítě, aby se zvýšila spolehlivost detekce.

Mimo implementované optimalizace je hlavním přínosem práce otestování detektoru na reálném zařízení. Díky tomu je možné získat pro další práce alespoň hrubou představu o výpočetním výkonu současných mobilních zařízení. Přínos práce však nemusí zůstat pouze v oblasti rozpoznávání řeči. Protože neuronové sítě jsou poměrně obecným algoritmem, lze optimalizovaný algoritmus využít i v mnoha jiných oblastech.

Literatura

[1] IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2008*, 29 2008: s. 1–58, doi:10.1109/IEEESTD.2008.4610935.

[2] Cawley, G.: On a Fast Comact Approximation of the Exponential Function.

[3] Cipr, T.: *Implementace detektoru klíčových slov do mobilního telefonu (Symbian 60)*. FIT VUT v Brně, 2007, diplomová práce.

[4] Earnshaw, R.: Procedure call standard for the ARM architecture. 2005: str. 32.

[5] Kolektiv autorů: Architecture and Implementation of the ARM® Cortex™-A8 Microprocessor. `http://www.arm.com/pdfs/TigerWhitepaperFinal.pdf (leden 2010)`, 2005.

[6] Král, T.: *Fixed-point implementácia rozpoznávania reči*. FIT VUT v Brně, 2007, diplomová práce.

[7] Miiner, B.: A comparison of front-end configurations for robust speech recognition.

[8] Rosell, M.: An Introduction to Front-End Processing and Acoustic Features for Automatic Speech Recognition.
`http://www.nada.kth.se/ rosell/courses/rosell_acoustic_features.pdf (2010)`, January 17, 2006.

[9] Schraudolph, N.: A fast, compact approximation of the exponential function. *Neural Computation*, ročník 11, č. 4, 1999: s. 853–862.

[10] Szöke, I.; Schwarz, P.; Burget, L.; aj.: Phoneme based acoustics keyword spotting in informal continuous speech.
`https://www.fit.vutbr.cz/šzoke/papers/radioelektronika_2005.pdf (leden 2010)`, 2005.

[11] WWW stránky: ARM Documentation home page.
`http://infocenter.arm.com/help/index.jsp (leden 2010)`.

[12] WWW stránky: Auto-vectorization in GCC.
`http://gcc.gnu.org/projects/tree-ssa/vectorization.html (duben 2010)`.

Použité zkratky

- MFCC — kepstrální koeficienty (Mel-frequency cepstral coeficients)
- SVM — algoritmy podpůrných vektorů (Support vector Machines)
- PLP — perceptuální lineární predikce (Perceptual linear prediction)
- GCC — open-source překladač (GNU Compiler Collection)
- RVCT — překladač firmy RealView (RealView Compiler tool)
- TMS470 — překladač firmy Texas Instruments
- NEON — matematický vektorový koprocesor pro architekturu ARM
- SIMD — vektorová architektura pro hromadné zpracování dat (Single Instruction Multiple Data)
- VFP — koprocesor pro výpočty v plovoucí řádové čárce (Vector Floating Point)
- OMAP 3530 — procesor firmy Texas Instruments založený na architektuře ARM Cortex A8
- FFT — algoritmus pro rychlý výpočet Fourierovy transformace (Fast Fourier transform)
- FFTW — optimalizovaná knihovna pro výpočet FFT

Seznam příloh

Příloha 1: Fonetická abeceda SAMPA

SAMPA SYMBOL	ORTHOGRAPHIC TRANSCRIPTION	SAMPA TRANSCRIPTION	ENGLISH TRANSLATION
i	myš	miS	mouse
e	les	les	forest
a	pas	pas	passport
o	rok	rok	year
u	kus	kus	piece
i:	pít	pi:t	to drink
e:	lék	le:k	drug
a:	rád	ra:t	glad
o:	móda	mo:da	fashion
u:	půl	pu:l	half
o_u	mouka	mo_uka	flour
a_u	auto	a_uto	car
p	pes	pes	dog
b	bota	bota	shoe
t	tam	tam	there
d	dům	du:m	house
c	tito	cito	these
J\	děd	J\et	grandfather
k	krk	krk	neck
g	kde	gde	where

Tabulka 1: Abeceda SAMPA pro fonetický přepis slov (1/2)

SAMPA SYMBOL	ORTHOGRAPHIC TRANSCRIPTION	SAMPA TRANSCRIPTION	ENGLISH TRANSLATION
t_s	cíl	t_si:l	aim
d_z	leckdy	led_zgdi	at times
t_S	čas	t_Sas	time
d_Z	léčba	le:d_Zba	medical treatment
f	forma	forma	form
v	vak	vak	bag
s	sen	sen	dream
z	zub	zup	tooth
Q\	tři	tQ\i	three
P\	řád	P\a:t	order
S	šaty	Sati	clothes
Z	žal	Zal	regret
j	jas	jas	brightness
x	chata	xata	cottage
h\	had	h\at	snake
r	ret	ret	lip
l	led	let	ice
m	mák	ma:k	poppy
n	noc	not_s	night
N	banka	baNka	bank
J	nic	Jit_s	nothing

Tabulka 2: Abeceda SAMPA pro fonetický přepis slov (2/2)