Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra kybernetiky

BAKALÁŘSKÁ PRÁCE

Plzeň 2012

Josef Michálek

Prohlášení

Předkládám tímto k posouzení a obhajobě bakalářskou práci zpracovanou na závěr studia na Fakultě aplikovaných věd Západočeské univerzity v Plzni.

Prohlašuji, že jsem bakalářskou práci vypracoval samostatně a výhradně s použitím odborné literatury a pramenů, jejichž úplný seznam je její součástí.

Plzeň 20. srpna 2012 .....

podpis

Anotace

Tématem této bakalářské práce je využití GPU pro zrychlení parametrizace řečového signálu. Práce obsahuje popis použitých metod analýzy řečového signálu (MFCC, LC-RC systém). Další část práce se zabývá návrhem kódu pro výpočet zvolených metod tak, aby se využilo architektury GPU. V poslední části práce je uvedeno vyhodnocení a porovnání kódu na CPU a GPU. Výsledkem práce je program, který je na přiloženém CD.

Klíčová slova: rozpoznávání řeči, parametrizace řečového signálu, MFCC, LC-RC systém, CUDA

Abstract

The subject of this bachelor thesis is the use of GPU for speech signal parameterization acceleration. The thesis contains the description of used methods of speech signal analysis (MFCC, LC-RC system). The next part of the thesis describes the process of writing the code to compute selected methods in order to take advantage of GPU architecture. The result of this thesis is a program on included CD.

Key words: speech recognition, speech signal parameterization, MFCC, LC-RC system, CUDA

Obsah

Seznam obrázků

Seznam výpisů

Kapitola 1

Úvod

Komunikace mluvenou řečí je nejzákladnější a nejpřirozenější způsob komunikace mezi lidmi. S rozvojem techniky vědci usilují o to, aby se partnerem člověka v mluveném rozhovoru mohl stát i stroj. Tento způsob komunikace by byl přirozenější a dovedl by výrazně usnadnit život i práci s počítačem. Pro komunikaci člověka s počítačem je však třeba vyřešit mnoho problémů z oblastí zpracování řečového signálu, počítačové syntézy řeči, rozpoznávání řeči a porozumění významu promluvy.

I přes řadu nedořešených problémů dnes dochází k nasazování systémů hlasové komunikace člověka s počítačem. Většinou se jedná o problémově orientované řešení. Oblast komunikace je omezená, systém pracuje v prostředí se známým rušivým pozadím apod. Dialogové systémy jsou nasazovány v úlohách hlasové komunikace s databázovými systémy. Lze je využít např. pro rezervační systémy, objednání zboží, automatickou telefonní ústřednu atd.

Využívány jsou i dílčí komponenty hlasových dialogových systémů:

- **Syntéza řeči**
  Jde o převod textu do mluvené podoby. Využívá se např. pro automatické čtení emailů, SMS zpráv a ostatních textů pro nevidomé.

- **Rozpoznávání řeči**
  Jedná se o převod mluvené řeči do textu. Uplatňuje se při automatickém generování titulků ke zvukové stopě a také generuje vstup pro systémy porozumění řeči.

- **Porozumění řeči**
  Jedná se o porozumění významu rozpoznané promluvy.

Se zlepšováním technologie je možné využívat pro komunikaci s počítačem mluvenou řečí složitější a výpočetně náročnější metody. Většina metod rozpoznávání řeči je založena na rozdělení řečového signálu na krátké segmenty o délce několika desítek milisekund. S těmito segmenty se pak provádí stejné operace. Této skutečnosti je možno využít a přesunout některé výpočty z procesoru na grafickou kartu. Grafické karty jsou vytvořeny za účelem rychlého provedení velkého množství stejných operací. Např. při vykreslování objektu v grafické aplikaci karta vypočte barvu mnoha pixelů stejným postupem nezávisle na sobě, pouze pro každý pixel použije jiné vstupní parametry. Tématem této práce je využití grafické karty pro rychlejší parametrizaci zvukového signálu.

Kapitola 2

Metody rozpoznávání řeči

Jak bylo uvedeno v úvodu, rozpoznáváním řeči se rozumí převod mluvené řeči na text. Využívá se pro mnoho různých aplikací jako např. automatické generování titulků, rozpoznávání příkazů mobilním telefonem (volání kontaktu podle jména) nebo bojovým letounem, kde usnadňuje práci pilotovi, je součástí dialogových systémů atd.

Metody rozpoznávání řeči se dělí na metody využívající porovnávání se vzory a statistické metody.

2.1 Porovnávání se vzory

Metoda pracuje se vzory jako s celky a klasifikuje je do té třídy, k jejímuž vzorovému obrazu má nejblíže. Každé slovo je zde reprezentováno posloupností vektorů příznaků. Důležité je určení vzdálenosti mezi obrazy.

Dva různé obrazy stejného slova nemají vždy stejnou délku, proto nelze porovnat přímo posloupnosti vektorů příznaků. Je možné upravit délku obou obrazů za pomoci lineární normalizace tak, aby byla stejná, ale ani tento postup nedá požadovaný výsledek. Pokud řečník vysloví stejné slovo v různých situacích nebo slovo vysloví dva různí řečníci, není různá pouze délka celého slova, ale i jeho částí (fonémů).

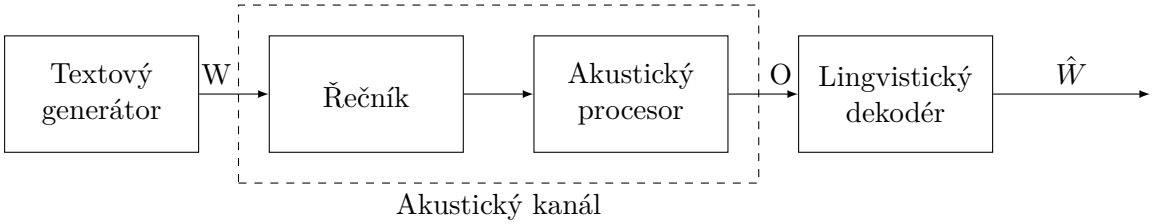
Z těchto důvodů se používá algoritmus na principu dynamického programování. Principem algoritmu je nelineární časová normalizace, kde je kolísání v časové ose modelováno časově nelineární DTW (dynamic time warping) funkcí. Při porovnávání dvou obrazů se jedna časová osa upraví tak, aby se eliminovaly časové rozdíly a tím se minimalizuje vzdálenost obou obrazů.

Metoda porovnávání se vzory se používá pro klasifikaci izolovaně vyslovených slov. Pro každé slovo je nutné mít záznam ve slovníku. Při klasifikaci se používá DTW funkce pro nalezení vzorového slova s nejmenší vzdáleností.

2.2 Statistické metody

Tento přístup ke klasifikaci je založen na modelování promluvy pomocí skrytých Markovových modelů. Jedním Markovovým modelem mohou být modelována celá slova nebo subslovní jednotky (slabiky, fonémy, trifony apod.). Promluva je modelována zřetězením těchto dílčích modelů. Procesem trénování se stanoví parametry odpovídajících Markovových modelů a neznámá promluva se klasifikuje podle toho, jaká posloupnost modelů subslovních jednotek generuje promluvu s největší aposteriorní pravděpodobností.

Schéma systému rozpoznávání řeči s využitím statistického přístupu je na obrázku 2.1. Akustický procesor převádí řečový signál produkovaný řečníkem na posloupnost vektorů příz-



Obrázek 2.1: Blokové schéma systému rozpoznávání řeči s využitím statistického přístupu

naků a lingvistický dekodér tyto posloupnosti překládá na posloupnosti slov.

Nechť $W = \{w_1, w_2, \ldots, w_N\}$ je posloupnost slov a $O = \{o_1, o_2, \ldots, o_T\}$ je posloupnost vektorů příznaků. Cílem je nalézt posloupnost slov $\hat{W}$ maximalizující pravděpodobnost $P(W|O)$, tj. nejpravděpodobnější posloupnost slov pro danou posloupnost vektorů příznaků. Použitím Bayesova pravidla lze odvodit

$$\hat{W} = \underset{W}{\operatorname{argmax}}\, P(W|O) = \underset{W}{\operatorname{argmax}}\, \frac{P(W)P(O|W)}{P(O)} \tag{2.1}$$

kde $P(W)$ je apriorní pravděpodobnost posloupnosti slov W na vstupu, $P(O|W)$ je pravděpodobnost, že při vyslovení slov W bude generována posloupnost vektorů příznaků O a $P(O)$ je apriorní pravděpodobnost vektorů příznaků na výstupu. Protože $P(O)$ není závislá na W, lze (2.1) upravit na

$$\hat{W} = \underset{W}{\operatorname{argmax}}\, P(W)P(O|W) = \underset{W}{\operatorname{argmax}}\, P(W, O) \tag{2.2}$$

Z rovnice (2.2) vyplývá, že problém nalezení posloupnosti slov $\hat{W}$ lze řešit pomocí dvou oddělených pravděpodobností $P(W)$ a $P(O|W)$. Pravděpodobnost $P(W)$ představuje jazykový model a pravděpodobnost $P(O|W)$ akustický model (model řečníka). Oba modely lze trénovat samostatně a je třeba je určit před samotným rozpoznáváním řeči.

Úloha rozpoznávání řeči s využitím statistických metod se skládá z těchto kroků:

1. Pomocí analýzy řečového signálu se určí posloupnost vektorů příznaků O.
2. Vytvoří se akustický model pro ocenění pravděpodobnosti $P(O|W)$.
3. Vytvoří se jazykový model pro ocenění pravděpodobnosti $P(W)$.
4. Nalezne se nejpravděpodobnější posloupnost slov $\hat{W}$.

2.3 Analýza řečového signálu

Akustická analýza je hlavní téma této práce. Lidské hlasivky si lze představit jako systém pomalu se měnící v čase. Dostatečně krátký řečový signál lze proto považovat za stacionární proces. Z tohoto předpokladu vychází většina metod analýzy řečového signálu a vede na aplikaci metod krátkodobé analýzy. Základem těchto metod je rozdělení vstupního signálu na množství segmentů o délce několika desítek milisekund, jejichž vlastnosti jsou považované za konstantní. Tyto segmenty jsou zpracovávány samostatně a výsledkem analýzy pro každý segment je vektor příznaků, který daný segment popisuje. Výsledkem analýzy celého řečového signálu je posloupnost vektorů příznaků popisujících celý řečový signál. Metody krátkodobé analýzy předpokládají, že vstupní hodnoty jsou získány digitalizací analogového signálu.

V práci pracuji s daty získanými metodou pulzní kódové modulace (PCM). Metoda se skládá ze dvou kroků:

1. **Vzorkování**

   Vzorkování je transformace signálu $s(t)$ spojitého v čase na posloupnost vzorků $s_n = s(nT)$ diskrétních v čase. Vzorkování probíhá v časových okamžicích $t_n = nT$, kde T je perioda vzorkování. Dále je na vzorkování kladeno omezení Shannonova teorému.[1]

[1] Jestliže je analogový signál frekvenčně omezen na pásmo 0 až F_m [Hz], lze $s(t)$ rekonstruovat z hodnot vzorků $s(nT)$, jestliže pro vzorkovací frekvenci $F_v = 1/T$ platí $F_v \geq 2F_m$.

2. **Kvantizace a kódování**

   Jedná se o aproximaci analogové hodnoty vzorku signálu jednou z konečného souboru číselných hodnot. Je prováděna A/D převodníkem. Pro návrh kvantizéru s rovnoměrně rozloženými úrovněmi stačí zadat:

   (a) počet úrovní kvantování (obvykle se volí ve tvaru 2^B, kde B je počet bitů v binárním kódu)

   (b) kvantizační krok Δ.

   Jestliže S_{max} je maximální úroveň vzorkovaného signálu, $|s(nT)| \leq S_{max}$, dostaneme

$$2S_{max} = \Delta 2^B$$

Před vlastním zpracováním řečového signálu se využívá preemfáze. Tj. zdůrazňování amplitud spektrálních složek řečového signálu s jejich vzrůstající frekvencí. Důvod pro tento proces vyplývá z chování řečového ústrojí (pokles amplitud spektrálních složek řečového signálu na vyšších frekvencích) a z citlivosti lidského sluchu (klesá se vzrůstající frekvencí).

Preemfáze může být zajištěna dvěma způsoby:

1. analogovým filtrem, který je předřazen vzorkovači a kvantizéru a jehož frekvenční charakteristika má strmost 20 dB/dek od frekvence 100 Hz.

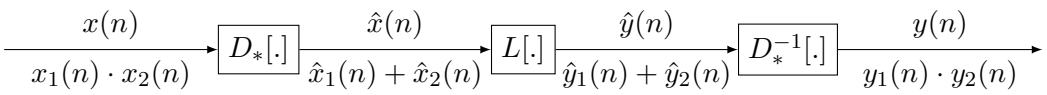
2. číslicovým filtrem, který je za vzorkovačem a kvantizérem a zpracovává signál podle vztahu:

$$y(n) = x(n) - ax(n-1)$$

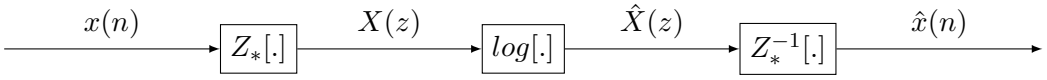
kde $x(n)$ je vstupní vzorek v čase n a $y(n)$ je výstup filtru. Parametr a se volí v rozsahu 0,9-1.

2.3.1 MFCC

Melovská frekvenční kepstrální filtrace je první metoda, o kterou se v práci zajímám. Jedná se o metodu homomorfního zpracování řeči. To se hodí pro analýzu signálů, které vznikly konvolucí nebo násobením dvou a více složek. Použití tohoto postupu je vhodné, protože proces vzniku řeči se dá popsat konvolucí budicího signálu (periodický sled pulzů pro znělé hlásky nebo šum pro neznělé hlásky) a impulzní funkce hlasového ústrojí. Cílem je určit parametry systému.



Obrázek 2.2: Obecné schéma homomorfního systému



Obrázek 2.3: Schéma charakteristického systému D_*

Obecné schéma homomorfního systému je na obrázku 2.2. Modul D_* se nazývá charakteristický systém a jeho struktura je na obrázku 2.3. Jestliže posloupnost $x(n)$ vznikla konvolucí posloupností $x_1(n)$ a $x_2(n)$

$$x(n) = x_1(n) * x_2(n)$$

Pak po aplikaci bloku D_* dostaneme

$$X(z) = Z\{x(n)\} = Z\{x_1(n) * x_2(n)\} = X_1(z)X_2(z)$$

$$\hat{X}(z) = log(X(z)) = log(X_1(z)) + log(X_2(z)) = \hat{X}_1(z) + \hat{X}_2(z)$$

$$\hat{x}(n) = Z^{-1}\{\hat{X}(z)\} = Z^{-1}\{\hat{X}_1(z) + \hat{X}_2(z)\} = \hat{x}_1(n) + \hat{x}_2(n)$$

Charakteristický systém D_* převádí konvoluci na součet modifikovaných signálů. L je lineární systém provádějící lineární filtraci sumy vstupních signálů a D_*^{-1} je systém inverzní k systému D_*.

Obvykle se místo z-transformace využívá Furierova transformace ($z = e^{j\omega}$). Pak můžeme napsat

$$\hat{X}\left(e^{j\omega}\right) = \log\left|X\left(e^{j\omega}\right)\right| + j\,\arg\left(X\left(e^{j\omega}\right)\right)$$

Poté můžeme určit komplexní kepstrum

$$\hat{x}(n) = \frac{1}{2\pi}\int_{-\pi}^{\pi} \hat{X}\left(e^{j\omega}\right) e^{j\omega n}\mathrm{d}\omega$$

a kepstrum

$$c(n) = \frac{1}{2\pi}\int_{-\pi}^{\pi} \log\left|X\left(e^{j\omega}\right)\right| e^{j\omega n}\mathrm{d}\omega$$

Kepstrum je tedy zpětná Furierova transformace logaritmu absolutní hodnoty Furierova obrazu vstupního signálu $x(n)$.

Krátkodobá kepstrální analýza řeči je metoda, která umožňuje ze signálu oddělit parametry buzení a parametry hlasového ústrojí. Proto se kepstrální koeficienty hodí pro systémy rozpoznání mluvené řeči.

V současnosti jsou preferovány dvě modifikace homomorfního zpracování řeči, a to kepstrální koeficienty odvozené z koeficientů lineární predikce a melovské kepstrální koeficienty (Mel-frequency cepstral coefficients – MFCC).

Metoda MFCC je metoda parametrizace řeči, která využívá procesu zpracování řečového signálu sluchovým ústrojím člověka. Především se jedná o:

- **Subjektivní vnímání výšky tónu**

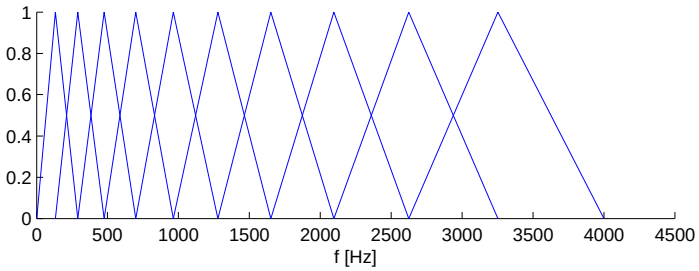
  Experimentálně bylo zjištěno, že člověk vnímá výšku tónu subjektivně. Byla zavedena stupnice subjektivní výšky zvuku s jednotkou **mel**. Frekvence v melech m se z frekvence f v [Hz] spočte vzorcem

  $$m = 2595 \cdot \log_{10}\left(1 + \frac{f}{700}\right)$$

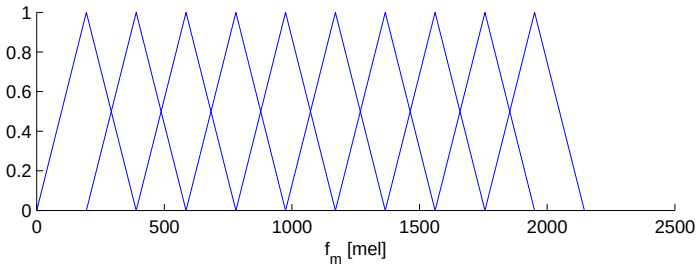
- **Kritická pásma**

  Pokud znějí dva tóny s různou frekvencí současně, jeden tón ovlivňuje vnímání tónu druhého. Tomuto jevu se říká maskování. Bylo zjištěno, že na maskování se podílí určité malé okolí kolem frekvence sledovaného tónu. Toto okolí se nazývá Kritické pásmo.

Metoda MFCC využívá banky trojúhelníkových filtrů. Umístění filtrů je dáno subjektivním vnímáním výšky tónu a jejich tvar (šířka) je dán kritickými pásmy. Filtry jsou obvykle rozloženy po celé frekvenční ose od nuly do Nyquistovy frekvence. Každý filtr v bance má trojúhelníkovou frekvenční odezvu. Filtry jsou na melovské frekvenční ose lineárně rozloženy. Každý filtr začíná ve střední frekvenci filtru předchozího a končí ve střední frekvenci filtru následujícího. Ukázka banky deseti filtrů pro frekvence 0 - 4000 Hz je na obrázcích 2.4a a 2.4b.



(a) v původní stupnici v [Hz]



(b) v melovské stupnici

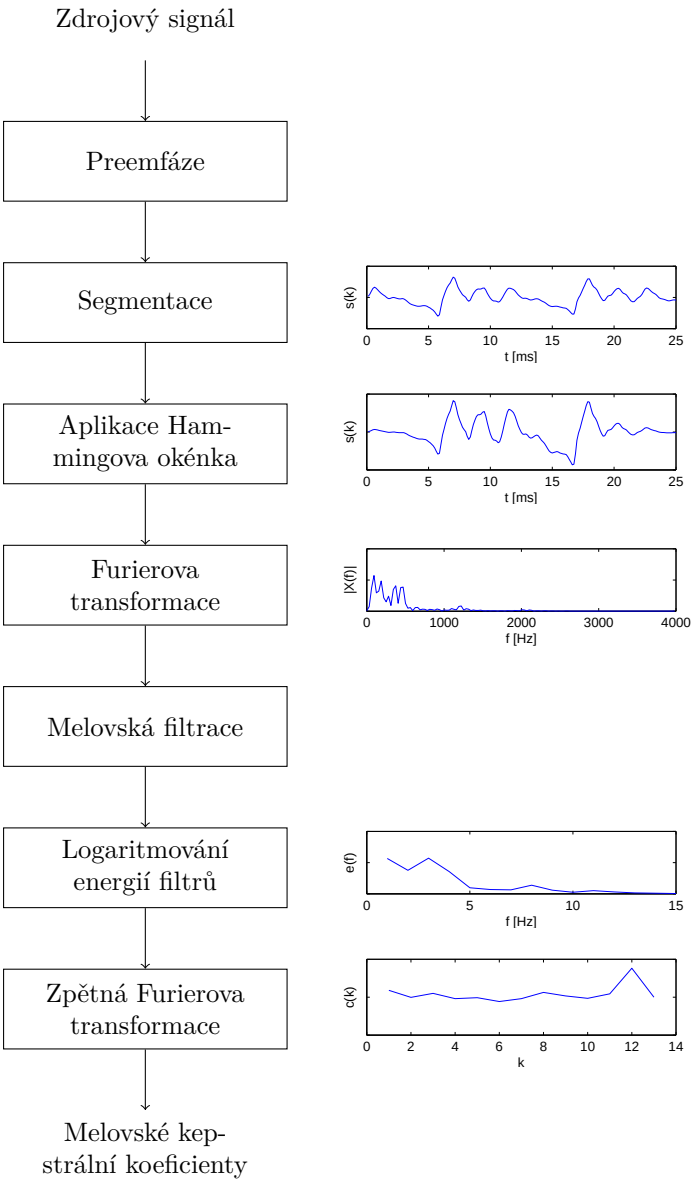
Obrázek 2.4: Banka melovských filtrů

Postup výpočtu MFCC (viz obr. 2.5):

1. Rozdělení vstupního signálu na segmenty o délce přibližně 25 ms
2. Aplikace Hammingova okénka
3. Výpočet krátkodobého výkonového spektra $P(f)$ (absolutní hodnota koeficientů diskrétní furierovy transformace)
4. Filtrace bankou melovských filtrů
5. Logaritmizace energií jednotlivých filtrů
6. Zpětná furierova transformace. Protože je výkonové spektrum $P(f)$ reálné a symetrické, výpočet zpětné furierovy transformace přejde na výpočet zpětné kosinové transformace podle vzorce:

$$c_m(j) = \sum_{i=1}^{M} y'_m(i) \cdot \cos\left(\frac{\pi}{M}(i-0,5)j\right) \quad \text{pro } j = 0, 1, \ldots, N$$

kde $y'_m(i) = \log_{10} y_m(i)$ a $y_m(i)$ je energie filtru m, M je počet filtrů a N je požadovaný počet melovských kepstrálních koeficientů. N se obvykle volí menší než M.



Obrázek 2.5: Postup výpočtu melovských kepstrálních koeficientů

Warpovací funkce

Délka hlasového traktu různých řečníků se liší. Může se pohybovat od 13 cm u dospělých žen až po 18 cm u dospělých mužů a to ovlivňuje polohy formantových frekvencí. Snahou metody normalizace hlasového traktu je kompenzovat tyto odlišnosti. Nejjednodušší řešení je transformovat frekvenční osu tak, aby se pozice formantů nového řečníka blížily k pozicím formantů referenčního řečníka. Protože metoda MFCC provádí melovskou filtraci ve frekvenčním spektru, je možné tuto transformaci frekvenční osy provést posunem melovských filtrů.

Pro tuto transformaci se využívá tzv. warpovací funkce $\tilde{\omega} = \eta_\alpha(\omega)$, která zobrazuje definiční obor proměnné $\omega \in \langle 0, \omega_{mez} \rangle$ zpět na množinu $\tilde{\omega} \in \langle 0, \omega_{mez} \rangle$, kde ω_{mez} je mezní frekvence a je obvykle rovna polovině vzorkovací frekvence. Často používanou warpovací funkcí je funkce využívající bilineární transformaci:

$$\eta_\alpha(\omega) = \omega + 2 \arctan \left(\frac{(1-\alpha)\sin\omega}{1-(1-\alpha)\cos\omega} \right) \tag{2.3}$$

2.3.2 LC-RC systém

Většina metod parametrizace řeči vychází z krátkodobých spekter. Z nich jsou spočteny vektory příznaků a průběžně předávány klasifikátoru. Z toho vyplývá, že popis daného mikrosegmentu závisí pouze na jediném krátkodobém spektru a okolní spektra na něj nemají vliv. Lidské hlasivky při mluvení postupně přechází mezi stavy a ukazuje se, že změny frekvenčního spektra v průběhu promluvy nesou informace vhodné pro klasifikaci fonémů.

Jeden z možných způsobů zahrnutí těchto informací do popisu řečového signálu je využití dynamických koeficientů označovaných delta a delta-delta. Tyto koeficienty vyjadřují dynamiku časové změny vektorů příznaků (derivaci) a počítají se z několika po sobě jdoucích mikrosegmentů. Delta koeficienty se určují přímo z původních vektorů příznaků získaných např. metodou MFCC a delta-delta koeficienty se určují z delta koeficientů. Výsledný vektor příznaků je pak rozšířen o delta i delta-delta koeficienty. Tyto vektory příznaků pak popisují delší časový úsek než původní vektory, obvykle 50 - 100 ms.

Jiný přístup je použit v metodě TempoRAl PatternS (TRAPS). Metoda využívá dvě úrovně klasifikátorů. Vstupem klasifikátoru první úrovně je dlouhý úsek výstupu jednoho kritického pásmového filtru. Takový úsek může být až 1 s dlouhý a obsahuje i informace o okolních fonémech. Ke každému kritickému pásmovému filtru je přidělen jeden klasifikátor.

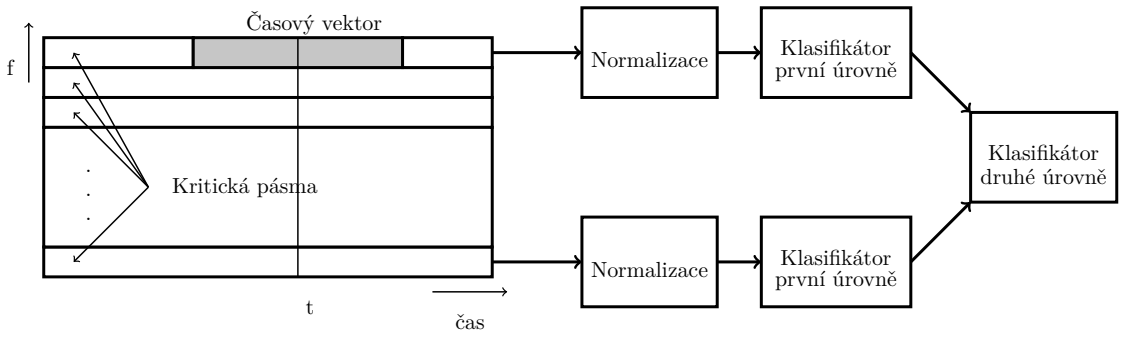
Jejich výstupy jsou použity v klasifikátoru druhé úrovně, který vrací výsledek rozpoznávání. Schéma systému je na obrázku 2.6.

Řetězec dat vycházející z kritického pásmového filtru se střední frekvencí f označíme $\tilde{o}_f(t) = \{o_{t-T,f}, \ldots, o_{t,f}, \ldots, o_{t+T,f}\}$. Pak jsou výstupem klasifikátoru první úrovně pravděpodobnosti $P(\omega_r|\tilde{o}_f(t))$ vyjadřující pravděpodobnost třídy ω_r za předpokladu řetězce dat $\tilde{o}_f(t)$. Všechny tyto pravděpodobnosti jsou vstupem klasifikátoru druhé úrovně a jeho výstupem jsou pravděpodobnosti jednotlivých tříd ω_r v čase t. Třídami se rozumí subslovní jednotky jako jsou monofony, trifony apod. Jako klasifikátory se často volí neurální sítě.

Požadavky na klasifikátor řeči zahrnují rychlost a jednoduchost výpočtů, ale výše uvedený systém je velmi složitý a pomalý. Proto je snahou systém zjednodušit. Neurální sítě představují nelineární transformaci. Pro zjednodušení systému je možné ji zaměnit za lineární transformaci. Dále předpokládáme, že výstup pásmových klasifikátorů nemusí odpovídat subslovním jednotkám. Poté můžeme každý pásmový klasifikátor nahradit PCA[2] transformací. Bylo zjištěno, že získané komponenty jsou velmi podobné těm získaných pomocí DCT a vážených Hammingovo okénkem. Experimenty potvrdily, že použití DCT má nepatrný vliv na výsledky.[2] Výstup systému se pak klasifikuje pomocí neurálních sítí. Takový systém se nazývá Simplified system.

Problémem při klasifikaci mohou být dlouhé trajektorie v obrazovém prostoru. Je jich mnoho a je možné, že velká část z nich se v trénovací množině nenachází. Za cenu ztráty části informace lze trajektorie rozdělit a modelovat je samostatně. Systém, který tato práce používá, rozděluje časové vektory na dvě části. Každá část je vážena odpovídající polovinou Hammingova okénka a je na ni aplikována DCT transformace. Tento systém se označuje Left context – Right context (LC-RC) system.

[2]Analýza hlavních komponent (Principal Component Analysis, PCA) je transformace, která zachová pouze dimenze s nejvyšší variancí



Obrázek 2.6: Diagram TRAPS systému

Kapitola 3

CUDA

Zatímco jsou procesory určeny pro rychlé spouštění jednoho vlákna, architektura grafických karet umožňuje spouštět mnoho pomalejších vláken najednou. S vývojem grafických karet přibyla možnost spouštění vlastních programů přímo na GPU. Tyto programy jsou určeny pro zpracování grafických dat, např. pixel shader počítá barvu výsledných pixelů obrazu, vertex shader transformuje body ze kterých jsou složeny grafické objekty atd. Tyto programy je možné využít pro jiné výpočty, ale jejich použití je omezené. Proto byla vyvinuta rozhraní umožňující využít výpočetní výkon grafických karet pro obecné výpočty.

Compute Unified Device Architecture (CUDA) je architektura určená pro paralelní výpočty vyvíjená firmou Nvidia. Umožňuje na grafické kartě spouštět kód napsaný v jazycích C/C++, FORTRAN nebo založený na architekturách OpenCL a DirectCompute. K dispozici jsou dále rozšíření pro další jazyky, např. Python, Perl, Java apod. CUDA je možné používat na všech grafických kartách od firmy Nvidia ze série G8x a novějších. Schopnosti jednotlivých karet jsou popsány hodnotou zvanou *compute capability*.

3.1 Architektura GPU

Většina plochy GPU se skládá z čipů zvaných streaming multiprocesory. Multiprocesor je čip, který je složen ze skalárních procesorů (až 32), pole registrů a sdílené paměti. Multiprocesory jsou založeny na architektuře SIMT (Single Instruction, Multiple Threads). Vlákna na multiprocesoru jsou spouštěny ve skupinách zvaných warpy. Při spuštění vláken na multiprocesoru jsou vlákna automaticky rozdělena do warpů a jednotlivé warpy běží nezávisle

na sobě. Každý skalární procesor má svůj vlastní čítač instrukcí, ale všechna vlákna ve stejném warpu provádí instrukce společně. Pokud dojde na větvení kódu a část vláken se chystá spustit instrukce v jiné větvi kódu než ostatní vlákna, začne se provádět jedna větev kódu a ostatní vlákna jsou pozastavena, dokud se nesloučí. Při respektování tohoto SIMT omezení je možné psát kód tak, aby vlákna v jednotlivých warpech spouštěla stejný kód, a dosáhnout tak zrychlení výpočtu.

3.2 Paměťový model

Zatímco program na procesoru pracuje zpravidla s jedním typem paměti (operační paměť), grafická karta obsahuje 6 typů paměti. Jednotlivé typy se liší velikostí, umístěním, rychlostí, použitím vyrovnávací paměti a možností zápisu/čtení.

- **Globální paměť**
  je paměť přístupná všem vláknům na všech multiprocesorech. Je možné z ní číst i do ní zapisovat, ale kvůli absenci vyrovnávací paměti je pomalá.
- **Konstantní paměť**
  je paměť určená pouze pro čtení. Zápis je do ní možný pouze z procesoru přes CUDA API. Mohou k ní přistupovat všechna vlákna, v podstatě se jedná o úsek globální paměti, ale využívá L1 cache. Díky tomu je přístup k ní rychlý.
- **Registry**
  jsou uloženy v poli na jednotlivých multiprocesorech. Každé vlákno může přistupovat pouze ke svým registrům, které se přiřadí při spuštění kódu. Jedná se o rychlou paměť s omezenou kapacitou.
- **Lokální paměť**
  je paměť pro proměnné, které se nevejdou do registrů. Každé vlákno může přistupovat pouze ke své lokální paměti. Rychlost lokální paměti je srovnatelná s globální pamětí, protože je na ní umístěná.
- **Sdílená paměť**
  je rychlá paměť sdílená bloku vláken. Spolu s registry je umístěná přímo na multiprocesoru. Přistupovat k ní mohou všechna vlákna ve stejném bloku. Sdílená paměť je

rozdělená do 16 bank, ke kterým se může přistupovat ve stejný časový okamžik. Banky jsou uspořádány tak, aby 16 po sobě jdoucích 32 bitů velkých částí paměti patřilo do 16 různých bank.

- **Paměť textur**

  je rychlá paměť určená pouze pro čtení. Využívá vyrovnávací paměti a je optimalizována na 2D prostorovou lokalitu. Vlákna ve stejném warpu čtou rychleji hodnoty na blízkých adresách. Při čtení z ní je možné využít filtračních jednotek grafické karty a automaticky provádět lineární filtraci dat nebo normalizaci na jednotkový rozsah.

3.3 Programový model

Program využívající rozhraní CUDA je složen ze 2 částí. Kód, který běží na CPU a kód, který běží na GPU. CPU a GPU bývají v CUDA aplikacích také nazývány host a zařízení. Kód, který se spouští na zařízení, je organizován ve funkcích zvaných kernely. Úkolem hosta je přesun data z paměti hosta do paměti zařízení, spuštění kernelu a přesun dat zpět do paměti hosta. Kernel se spouští ve formě vláken, jejichž konfigurace je zvolená hostem. Je popsána pojmy:

- **Blok vláken (thread block)**

  Vlákna jsou spouštěna v blocích. Vlákna jednoho bloku mohou využívat stejnou sdílenou paměť a navzájem synchronizovat svůj běh. Pro identifikaci vlákna v bloku určena zabudovaná proměnná `threadIdx`. Jedná se o 3-rozměrný vektor. Při spuštění kernelu se specifikuje velikost bloku jako počet vláken v každé dimenzi. Maximální počet vláken v bloku je omezen grafickou kartou (až 1024) a kernelem.

- **Mřížka (grid)**

  Podobně jako jsou vlákna organizována v blocích, jsou i bloky organizovány do mřížky. Pro identifikaci bloku v mřížce se používá zabudovaná proměnná `blockIdx`.

- **Warp**

  Warp je skupina vláken, které se zpracovávají najednou. Dnešní karty mají warp o velikosti 32 vláken. Při spuštění kernelu se vytvoří vlákna a rozdělí se do bloků a warpů. Na multiprocesoru se vytvoří více warpů než se může najednou zpracovávat. Pokud ně-

jaký např. čte z globální paměti, začne se vykonávat warp jiný. Tento přístup zmírňuje vliv latence paměti na průběh kernelu. Poměr počtu vytvořených warpů ku maximálnímu počtu warpů na multiprocesoru se nazývá *occupancy* a snahou je zvolit takovou konfiguraci kernelu, aby byla hodnota *occupancy* velká.

Kapitola 4

Implementace

4.1 MFCC

4.1.1 Segmentace

Vstupní data v RAM jsou jednotlivé hodnoty zvukového signálu. Pro práci s nimi je nutné je zkopírovat do globální paměti grafické karty a rozdělit je do segmentů. Úlohu je možné řešit dvěma způsoby:

- V cyklu kopírovat data segment po segmentu
- Zkopírovat všechna data najednou do dočasné paměti a poté spustit kernel, který je rozdělí do segmentů

Použitím prvního přístupu se data kopírují pouze jednou, ale jedná se o pomalejší řešení. Při n segmentech dojde k n přenosům malých bloků dat. V tomto případě není čas potřebný pro samotné zahájení a ukončení přenosu zanedbatelný. Proto jsem se rozhodl zvolit druhý způsob přenosu dat a jejich rozdělení do segmentů. Nejdříve se všechna vstupní data zkopírují do globální paměti grafické karty použitím jedné transakce. Poté se spustí kernel, který data segmentuje a uloží do jiné části globální paměti. Z důvodů rychlejších výpočtů (viz kapitola 4.1.2) je velikost segmentů prodloužena na nejbližší vyšší mocninu dvou a data jsou doplněna nulami.

```
__global__ void kernelPrepareData(float * tmp, float * data, int block_index,
    int window_count, int window_size, int shift)
{
    int index = blockDim.x * (block_index * MAX_GRID_SIZE + blockIdx.x) +
        threadIdx.x,
        window_size2 = ceil2(window_size);

    if (index % window_size2 < window_size)
    {
        int indexIn = (index / window_size2) * shift + (index % window_size2);
        data[index] = tmp[indexIn];
    }
    else
        data[index] = 0;
}
```

Výpis kódu 4.1: Kernel pro segmentaci dat

Pro tento způsob přípravy dat je nutno alokovat více paměti, je nutno spustit kernel, který data rozdělí do segmentů, ale funkci pro přenos dat z RAM do grafické karty stačí volat pouze jednou.

Dalším krokem analýzy akustického signálu je vážení okénkem. Váhovou funkci okénka určí uživatel. Její hodnoty se uloží do paměti karty a vážení segmentů probíhá v kernelu, který hodnotami váhové funkce vynásobí příslušné hodnoty v segmentu. Pro každý segment se spouští jeden blok vláken.

```
__global__ void kernelWindow(float * data, int window_count, float * window,
    int window_size2)
{
    int index = blockDim.x * blockIdx.x + threadIdx.x,
        k= index % window_size2;
    if (index < window_count * window_size2)
        data[index] *= window[k];
}
```

Výpis kódu 4.2: Kernel pro násobení váhovou funkcí okénka

Po provedení segmentace jsou data v paměti uložena ve formě matice. Počet řádků odpovídá počtu segmentů a počet sloupců je délka segmentu doplněná na mocninu dvou.

4.1.2 FFT

Pro výpočet rychlé Furierovy transformace je použita knihovna CUFFT od firmy NVidia. Knihovna byla vytvořena profesionály za účelem rychlého výpočtu a je nepravděpodobné, že bych svou implementací dosáhl lepších výsledků.

Obecná diskrétní Furierova transformace (dále DFT) se dá spočítat pomocí násobení matic a vektorů se složitostí $O(N^2)$. Knihovna CUFFT využívá Cooley-Tukey algoritmu. Jeho základ je v rozdělení původní DFT velikosti $N = N_1 \cdot N_2$ na menší DFT o velikosti N_1 a N_2. Toto dělení probíhá rekurzivně a jeho účelem je snížit složitost algoritmu na $O(N\log N)$ pro hladká čísla N. Knihovna implementuje dílčí bloky DFT pro faktorizaci čísly 2, 3, 5 a 7. Proto je knihovnou CUFFT optimalizována jakákoliv transformace o velikosti, která jde vyjádřit výrazem $2^a \cdot 3^b \cdot 5^c \cdot 7^d$, kde a, b, c a d jsou nezáporná celá čísla.

Pro další zvýšení výkonu jsou na velikost jednorozměrné transformace pro kladeny následující požadavky:

- Velikost transformace má být pouze násobek čísel 2,3,5 nebo 7. Např. transformace velikosti 3^n může být rychlejší než transformace velikosti $2^i \cdot 3^j$ i když je první transformace delší.
- Mocnina 2 v rozkladu délky transformace by měla být násobkem alespoň 16. Jde o omezení kvůli zarovnání paměti karet Tesla a Fermi.
- Mocnina 2 v rozkladu délky transformace by měla být násobkem 256. To dále zlepší zarovnání paměti.
- Délka transformace by měla jít vyjádřit pouze číslem 2 umocněném na hodnotu mezi 2 a 8192 pro karty Fermi nebo 2 a 2048 pro dřívější architektury. Transformace těchto velikostí jsou implementovány jako specializované zvláštně naprogramované kernely, které si drží všechny mezivýsledky ve sdílené paměti.

Knihovna obsahuje další omezení, ale ty se týkají dvojité přesnosti a in-place transformací, které se v programu nepoužívají.

Jak je uvedeno v kapitole 4.1.1, velikosti segmentů se doplňují na nejbližší vyšší mocninu čísla 2. Např. segmenty délky 200 se rozšiřují na segmenty o délce 256. Číslo 256 je 2-hladké číslo (žádný z jeho prvočíselných dělitelů není větší než 2). To umožňuje rychlejší průběh

algoritmu Cooley-Tukey. Dále tato velikost splňuje výše uvedené požadavky, které umožní další zvýšení rychlosti.

V jednom kroku se počítá FFT o velikosti n pro w různých segmentů. Data v paměti lze interpretovat jako matice o velikosti $n \times w$. Vstupem Furierovy transformace jsou reálná čísla, výstupem čísla komplexní. Knihovna CUFFT vrací pouze prvních $n/2+1$ koeficientů, protože ostatní koeficienty jsou na nich závislé (jsou komplexně sdružené) a dají se z nich spočítat.

Použitím knihovny CUFFT se získají FFT koeficienty, jejichž absolutní hodnotu je třeba spočítat. Pro její výpočet jsem se rozhodl napsat vlastní kernel. Kernel se spouští tak, že jeden blok vláken odpovídá jednomu segmentu. Pokud je dat příliš mnoho a nevejdou se do maximální velikosti gridu, v cyklu se spustí grid postupně na zbytek dat. Každé vlákno ze svého indexu a indexu bloku zjistí souřadnice koeficientu, kterého absolutní hodnotu má spočítat. Poté vypočte souřadnice vstupní hodnoty v závislosti na tom, jestli je k dispozici přímo spočtený koeficient (prvních $n/2 + 1$ hodnot) a nebo se musí spočítat jako číslo komplexně sdružené k jednomu z prvních $n/2 + 1$ koeficientů. Protože jsou výstupem kódu absolutní hodnoty koeficientů a absolutní hodnota komplexního čísla je rovna absolutní hodnotě čísla komplexně sdruženého, je možné znaménko imaginární složky ignorovat a počítat se všemi koeficienty stejně a vyhnout se tím větvení kódu.

```
__global__ void kernelFFTMag(cufftComplex * data_in, int window_count, int
    window_size2, float * data_out)
{
    int index = blockDim.x * blockIdx.x + threadIdx.x;
    if (index < window_count * window_size2)
    {
        int k= index % window_size2,
            row = index / window_size2,
            fft_size = window_size2 / 2 + 1,
            idx;

        if (k <= window_size2 / 2)
            idx = row * fft_size + k;
        else
            idx = row * fft_size + window_size2 + 1 - k;

        float x = data_in[idx].x,
```

```
            y = data_in[idx].y;
        data_out[index] = sqrt(x * x + y * y);
    }
}
```

Výpis kódu 4.3: Kernel pro výpočet absolutní hodnoty FFT koeficientů

4.1.3 Banka melovských filtrů

Pro každý segment se aplikuje banka melovských filtrů. Vstupem výpočtu pro každý segment je vektor x o velikosti n a výstupem vektor e o velikosti m, kde n je délka okénka a m počet melovských filtrů. Prvky vektoru x jsou magnitudy FFT koeficientů spočtené v kapitole 4.1.2 a prvky výstupního vektoru e jsou hodnoty energií jednotlivých filtrů. Energie filtru i se spočte následovně:

$$e_i = \sum_{j=1}^{n} f_{ij} \cdot x_j = f_i \cdot x \tag{4.1}$$

kde f_{ij} značí hodnotu funkce trojúhelníkového filtru i v bodě j. Rovnice 4.1 představuje skalární součin vektorů f_i a x. Protože vstupní data jsou uspořádaná v matici X, kde každý řádek reprezentuje segment, lze filtraci celého vstupního signálu o N segmentech provést vynásobením dvou matic:

$$E = \begin{bmatrix} x_{11} & x_{12} & \dots & x_{1n} \\ x_{21} & x_{22} & \dots & x_{2n} \\ \vdots & \vdots & & \vdots \\ x_{N1} & x_{N2} & \dots & x_{Nn} \end{bmatrix} \cdot \begin{bmatrix} f_{11} & f_{12} & \dots & f_{1m} \\ f_{21} & f_{22} & \dots & f_{2m} \\ \vdots & \vdots & & \vdots \\ f_{n1} & f_{n2} & \dots & f_{nm} \end{bmatrix} = X \cdot F$$

Pro určení matice filtrů F je třeba znát středy trojúhelníkových melovských filtrů, které jsou rovnoměrně rozloženy na melovské frekvenční stupnici mezi minimální a maximální frekvencí určenou podle vzorkovací frekvence řečového signálu. Pro banku m filtrů potřebuji $m + 2$ středů. Středy melovských filtrů jsou dále posunuty po frekvenční ose pomocí bilineární warpovací funkce, viz (2.3).

Grafické karty provádějí násobení matic mnohokrát rychleji než procesor a protože vstupní data X už jsou uložena v paměti karty, pro melovskou filtraci zbývá do paměti karty uložit matici filtrů F. Pro to je třeba určit středy filtrů. Nabízí se 2 postupy:

- Spočítat středy filtrů a matici na procesoru a poté ji zkopírovat do paměti grafické karty – jednodušší kód, nutnost kopírovat data.
- Spočítat středy filtrů a matici na grafické kartě – rychlejší výpočet

Kód pro procesor je jednodušší, ale pomalejší. Grafická karta může spustit pro každý koeficient jedno vlákno a tím je spočítat paralelně nezávisle na sobě. Navíc se matice nemusí kopírovat z RAM do paměti karty, i když při takové velikosti dat nejspíše více času zabere zahájení přenosu než přenos samotný. Při velikosti matice a za předpokladu, že se matice spočte pouze jednou při startu programu, není rychlost výpočtu relevantní, ale nejdříve jsem se rozhodl napsat kód pro výpočet filtrační matice na grafické kartě a dále nebyl důvod kód přepisovat na procesor.

```
__global__ void kernelGetMelCenters(float * centers, int num_banks, float
    sample_rate, float low_freq, float high_freq, float alpha)
{
    int index = blockIdx.x * blockDim.x + threadIdx.x;
    if (index <= num_banks + 1)
    {
        float minmel = hz2mel(low_freq),
            maxmel = hz2mel(high_freq),
            f = mel2hz(index / float(num_banks + 1) * (maxmel - minmel) +
                minmel);
        float o = 2 * (float)M_PI * f / sample_rate;
        o = o + 2 * atan(((1 - alpha) * sin(o)) / (1 - (1 - alpha) * cos(o)));
        centers[index] = sample_rate * o/ (2 * (float)M_PI);
    }
}
```

Výpis kódu 4.4: Kernel pro výpočet středů melovských filtrů

```
__global__ void kernelPrepareMelScaleFilterMatrix(float * matrix, int
    window_size2, float sample_rate, float low_freq, float high_freq, int
    num_banks, float * centers)
{
    int sample_index = blockDim.x * blockIdx.x + threadIdx.x,
        bank_index = blockDim.y * blockIdx.y + threadIdx.y,
        num_banks2 = ceil2(num_banks);

    if (sample_index < window_size2 && bank_index < num_banks2)
    {
        int mat_index = sample_index * num_banks2 + bank_index;
        if (bank_index < num_banks)
        {
            float cl = centers[bank_index],
                cc = centers[bank_index + 1],
                cr = centers[bank_index + 2],
                sample_freq = sample_index / float(window_size2) * sample_rate;

            if (sample_freq > cl && sample_freq <= cc)
                matrix[mat_index] = sample_freq / (cc - cl) - cl / (cc - cl);
            else if (sample_freq > cc && sample_freq < cr)
                matrix[mat_index] = sample_freq / (cc - cr) - cr / (cc - cr);
            else
                matrix[mat_index] = 0;
        }
        else
            matrix[mat_index] = 0;
    }
}
```

Výpis kódu 4.5: Kernel pro výpočet melovské filtrační matice

Pro násobení matic je použita knihovna CUBLAS optimalizovaná pro rychlé maticové a vektorové počty. Matice jsou v paměti uložené jako jednorozměrné pole. Jazyk C používá row-major[1] matice, ale CUBLAS kvůli kompatibilitě s jazykem Fortran používá column-major[2] matice. To znamená, že se funkce z knihovny CUBLAS chovají tak, jako kdyby pracovali

[1] matice jsou v paměti uloženy po řádcích

[2] matice jsou v paměti uloženy po sloupcích

s maticemi transponovanými. Je třeba spočítat součin matic $X \cdot F$. Pokud se zavolá funkce pro násobení matic $F \cdot X$, kvůli rozdílné indexaci C a CUBLAS funkce spočte $F^T \cdot X^T$. A podobně, kvůli rozdílné indexaci, kód v jazyce C přečte výsledek jako $\left(F^T \cdot X^T\right)^T$ a to je rovno požadovanému výsledku $X \cdot F$. Proto se v kódu volá funkce pro součin matic $F \cdot X$, i když operace $F \cdot X$ není kvůli rozměrům matic definovaná.

Výsledkem součinu je matice, kde řádky odpovídají segmentům a sloupce jednotlivým filtrům v bance.

Posledním krokem je zlogaritmování výsledných hodnot. Pro to je použit kernel, který zlogaritmuje každou hodnotu v jednorozměrném poli a při jeho volání se mu postupně předávají části matice.

```
__global__ void kernelLN(float * data, int data_size)
{
    int index = blockIdx.x * blockDim.x + threadIdx.x;
    if (index < data_size)
    {
        float v = data[index];
        if (v < 1e-30f)
            data[index] = 0;
        else
            data[index] = log(v);
    }
}
```

Výpis kódu 4.6: Kernel pro logaritmování energií filtrů

4.1.4 DCT

Pro výpočet diskrétní kosinové transformace je použit vzorec:

$$y_k = \sum_{n=0}^{N-1} x_n \cdot \cos\left(\frac{\pi}{N}(n+0,5)k\right) \quad k = 0, 1, \ldots, N-1 \tag{4.2}$$

Nejdříve jsem se rozhodl napsat kernel pro výpočet DCT přímo ze vzorce (4.2).

```
__global__ void kernelDCT(float * data, int window_count, int window_size2,
    float * data_out)
{
    int block_count = (window_size2 + blockDim.x - 1) / blockDim.x,
        index = blockDim.x * blockIdx.x + threadIdx.x,
        k~= index % window_size2,
        window_index = index / window_size2,
        window_beg = window_index * window_size2;

    float c = M_PI * (k~+ 0.5f) / window_size2,
        sum = 0;
    for (int block = 0; block < block_count; block++)
    {
        for (int i = 0; i < blockDim.x; i++)
        {
            int n = block * blockDim.x + i;
            sum += data[window_beg + n] * cos(c * (n + 0.5f));
        }
    }

    data_out[index] = sum;
}
```

Výpis kódu 4.7: Kernel pro výpočet DCT bez sdílené paměti

Kód se na kartě spouští paralelně a každá výstupní hodnota se počítá nezávisle na ostatních. Každý koeficient výstupního vektoru je spočten jedním vláknem. A protože se každý výstupní koeficient spočte z celého vstupního vektoru a počítá se N koeficientů, dochází během výpočtu ke čtení všech hodnot vstupního vektoru N-krát. Přístup do globální paměti karty je pomalý v porovnání s ostatními typy paměti. Výpočet se dá zrychlit použitím sdílené paměti. Přístup k ní je mnohokrát rychlejší a paměť je sdílená pro všechna vlákna běžící na stejném multiprocesoru. Délka vstupního a výstupního vektoru je stejná, proto je možné kód upravit tak, aby každé vlákno po svém spuštění zapsalo do sdílené paměti jednu hodnotu ze vstupního vektoru. Každý vstupní vektor je v paměti zarovnán, takže může dojít ke všem transakcím do sdílené paměti najednou. Poté dojde k synchronizaci vláken a samotný výpočet čte vstupní data ze sdílené paměti místo z globální.

```
__global__ void kernelDCT(float * data, int window_count, int window_size2,
    float * data_out)
{
    int block_count = (window_size2 + blockDim.x - 1) / blockDim.x,
        index = blockDim.x * blockIdx.x + threadIdx.x,
        k~= index % window_size2,
        window_index = index / window_size2,
        window_beg = window_index * window_size2;

    float c = M_PI * (k~+ 0.5f) / window_size2,
        sum = 0;
    int j = 0;
    for (int block = 0; block < block_count; block++)
    {
        int k2 = index % blockDim.x;
        extern __shared__ float dataIn[];
        dataIn[k2] = data[window_beg + block * blockDim.x + k2];
        __syncthreads();

        for (int i = 0; i < blockDim.x; i++)
            sum += dataIn[i] * cosf(c * (j++ + 0.5f));
        __syncthreads();
    }

    data_out[index] = sum;
}
```

Výpis kódu 4.8: Kernel pro výpočet DCT se sdílenou pamětí

Další způsob výpočtu DCT je pomocí matice koeficientů C. Prvky matice jsou $C_{nk} = \cos\left(\frac{\pi}{N}(n+0,5)k\right)$. Výpočet kosinové transformace vektoru o délce N se pak provádí násobením vektoru maticí C. A protože jsou všechny vstupní vektory energií filtrů uspořádané v matici po řádkách, lze celý výpočet vyjádřit násobením matic:

$$Y = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_{N-1} \end{bmatrix} = \begin{bmatrix} e_0 \\ e_1 \\ \vdots \\ e_{N-1} \end{bmatrix} \cdot C = E \cdot C$$

Prvky matice se spočtou na procesoru a poté se zkopírují do paměti grafické karty. Pro násobení matic je použita knihovna CUBLAS, a proto se matice v kódu násobí v opačném pořadí, viz kapitola 4.1.3.

Z navrhnutých způsobů řešení jsem experimentálně ověřil, že nejrychlejší výpočet DCT je pomocí násobení matic.

4.2 LC-RC systém

Vstupy této metody jsou:

- Zlogaritmované energie melovských filtrů (MFCC koeficienty bez DCT)
- Celková délka DCT, kterou metoda spočte
- Počet DCT koeficientů, které metoda vrátí, ostatní se zahodí

Pro získání logaritmů energií melovských filtrů je použit kód z předchozích kapitol.

Při výpočtu MFCC se provádí DCT pro každý časový okamžik přes všechny frekvence. Základem LC-RC systému je aplikace DCT pro každé kritické pásmo přes několik časových okamžiků do minulosti i budoucnosti od času t_0. Celková délka DCT na vstupu metody (označím `dctlen`) značí celkovou délku obou DCT v minulosti i budoucnosti. Protože se obě DCT překrývají v bodě t_0, je délka jednotlivých DCT rovna $\frac{\text{dctlen}-1}{2} + 1$. Celkem se tedy pro každé t_0 aplikuje DCT dvakrát, a to pro časové okamžiky $\{t_{-\frac{\text{dctlen}-1}{2}-1}, \ldots, t_0\}$ a pro $\{t_0, \ldots, t_{\frac{\text{dctlen}-1}{2}+1}\}$.

Pro t_0 blízké krajním hodnotám vstupního souboru dat není možné DCT aplikovat, protože není dostatečný počet koeficientů. V tomto případě se krajní segmenty duplikují tolikrát, aby bylo možné DCT aplikovat. A protože výstup metody v čase t_0 závisí i na hodnotách vstupu v čase $t > t_0$, metoda nevrací data pro stejný počet segmentů jako na vstupu, ale pouze ta, která lze spočítat. Ostatní data spočte při dalším volání, kdy jsou k dispozici potřebné vstupní hodnoty. Např. aplikace používá délku jednotlivých DCT rovnou 16 (`dctlen` rovno 31). První volání metody pro výpočet LC-RC koeficientů vrací data o délce `počet_segmentů` $-\,15$, protože transformace 15 posledních segmentů závisí na hodnotách, které metoda ještě nezná. Všechna další volání už vrací data o délce `počet_segmentů`. Zbývající segmenty se získají

metodou `flush`, která poslední segment několikrát zduplikuje a provede výpočet tak, aby byl celkový počet výstupních segmentů roven počtu segmentů vstupních.

Po analýze DCT v kapitole 4.1.4 jsem se rozhodl implementovat výpočet DCT pomocí maticového násobení bez porovnání s ostatními přístupy ve zmíněné kapitole.

Pro lepší ilustraci zvoleného postupu výpočtu DCT je vhodné si zvolit konkrétní hodnoty vstupu a výstupu metody. Např. je požadováno aplikovat metodu s délkou jednotlivých DCT rovnou 16, počtem výstupních koeficientů rovným 11 a vstupem metody jsou energie 15 melovských filtrů. Jinými slovy je vstupem matice o rozměrech `počet_segmentů` $\times$ 15 a výstupem matice o rozměrech `počet_segmentů` $\times$ 330. Nejdříve budu uvažovat DCT z dat směrem do minulosti. Je třeba spočítat několik DCT nezávislých na sobě. Vstupní data jsou uspořádaná v matici, kde každý sloupec obsahuje vstupní hodnoty pro jednu DCT a aplikuje se tolik DCT, kolik má matice sloupců. Jedná se o stejnou situaci jako v kapitole 4.1.4 s tím rozdílem, že výstupní data by neměla být uspořádaná v matici, ale ve vektoru. Prvních 11 hodnot budou koeficienty prvního DCT (z 1. sloupce matice), dalších 11 hodnot budou koeficienty druhého DCT (z 2. sloupce matice) atd. Celkem $11 \cdot 15 = 165$ hodnot, což je polovina výstupního vektoru pro jeden segment. Dalších 165 hodnot budou koeficienty všech DCT směrem do budoucnosti.

Matice jsou v paměti uloženy lineárně, řádek za řádkem, a zarovnání řádku v paměti jsem zvolil rovno délce řádku matice. Důsledkem této volby je rozdílná možnost interpretace matic. Pokud se DCT spočte stejným způsobem jako v kapitole 4.1.4, mohu s výslednou maticí o rozměrech 11×15 pracovat jako s maticí o rozměrech $(11 \cdot 15) \times 1$. Reprezentace obou matic v paměti je totožná. Z toho vyplývá, že výsledný vektor DCT koeficientů je možné spočítat jako v kapitole 4.1.4, pouze při násobení matice logaritmů energií melovských filtrů s maticí DCT koeficientů jsou zadány rozměry výstupní matice 11×15 a dále se s onou maticí pracuje jako s požadovaným výstupním vektorem o délce 165. Stejný postup je použit i pro DCT do budoucnosti.

Tento postup se provádí pro všechny segmenty ve smyčce. V jednom volání funkce pro násobení matic z knihovny CUBLAS se vypočtou všechny DCT pro jeden segment buď do minulosti nebo do budoucnosti. Pro výpočet všech DCT pro jeden segment (jeden časový okamžik) stačí provést 2 operace násobení matic.

Kapitola 5

Vyhodnocení

K vyhodnocení metod analýzy akustického signálu jsem použil dodanou aplikaci, která používá pro výpočty knihovnu Intel® Integrated Performance Primitives (IPP). Tato knihovna je balík vysoce optimalizovaných funkcí pro výpočty, které se týkají multimédií, zpracování signálů, šifrování apod.

Mým úkolem bylo rozšířit aplikaci o možnost výpočtu na grafické kartě a porovnat s kódem na procesoru.

Program jsem vyvíjel a testoval na svém počítači s konfigurací:

- Intel Core 2 Quad Q6600 (2.4 GHz)
- 4GB DDR2-800
- NVIDIA GeForce 8800 GT (512 MB)

Pro měření doby provádění kódu na procesoru byl použit časovač s vysokým rozlišením (funkce `QueryPerformanceCounter`). Na použitém systému tento časovač odpovídá ACPI PMT (Power Management Timer). Jedná se o jednoduchý časovač s frekvencí zhruba 3,58 MHz, který v každém cyklu zvýší svou hodnotu.

Pro měření kódu na grafické kartě bylo zvoleno rozhraní *CUDA Event API*, které používá časovač přímo na čipu karty. Časovač měří dobu trvání kernelů a protože je volání kernelů asynchronní operace, použití jiného časovače by vyžadovalo synchronizaci vláken před koncem měření. Dále asynchronní volání funkcí časovače zmírňuje Heisenbergův efekt[1] a *CUDA Event*

[1]Samotná přítomnost pozorovatele může ovlivnit výsledky experimentu. Při měření mnoha krátkých úseků kódu se na výsledném trvání kódu může projevit čas strávený voláním funkce časovače.

API dovoluje měřit překrývající se kernely a přenosy dat. Přesnost časovače na kartě je zhruba 0.5 µs.

Pro porovnání rychlosti kódu pro procesor a grafickou kartu je třeba programem zpracovat dostatečně dlouhý zvukový soubor. Pro tento účel jsem zvolil audioknihu s parametry:

- Vzorkovací frekvence: 8 kHz
- Bitů na vzorek: 16
- Počet vzorků: 24 379 351

To odpovídá téměř 51 minutám zvuku.

Měřil jsem jednotlivé části výpočtu a také jsem použil časovač s vysokým rozlišením pro zjištění doby trvání celého zpracování řečového signálu na hostu i zařízení. Do celkového času se zahrnují operace, které se na grafické kartě nepočítají, jako výpočet delta koeficientů nebo normalizace příznaků na nulovou střední hodnotu a jednotkový rozptyl.

Naměřené hodnoty jednoho spuštění programu pro obě metody jsou v tabulkách 5.1a a 5.2a. Pro porovnání jsem program spustil na mnohokrát menší soubor (23 807 vzorků) a výsledky zaznamenal do tabulek 5.1b a 5.2b.

Metoda	Délka výpočtu [ms]	
	CPU	GPU
Okénko	437,64	22,79
FFT	627,38	62,59
Magnitudy FFT	460,46	58,61
Melovská filtrace	526,37	189,46
Logaritmování	449,92	2,26
DCT	449,90	3,76
Celkem	6217,01	1309,85

(a) 24 379 351 vzorků

Metoda	Délka výpočtu [ms]	
	CPU	GPU
Okénko	0,38	0,31
FFT	0,59	0,88
Magnitudy FFT	0,44	0,78
Melovská filtrace	0,50	2,55
Logaritmování	0,43	0,02
DCT	0,42	0,04
Celkem	6,06	10,40

(b) 23 807 vzorků

Tabulka 5.1: Délky výpočtu dílčích metod MFCC

Dále jsem program spustil na různě velké soubory. Každý test proběhl 10× a výsledky jsou zprůměrovány, aby se redukovaly faktory ovlivňující průběh testu jako např. vyrovnávací paměť disku, krátkodobé vytížení systému jiným programem apod. Celkový čas analýzy

Metoda	Délka výpočtu [ms]	
	CPU	GPU
Okénko	436,92	22,87
FFT	616,10	62,66
Magnitudy FFT	451,82	58,80
Melovská filtrace	507,36	188,31
Logaritmování	448,15	2,23
DCT	86238,55	5371,38
Celkem	95642,31	11805,14

(a) 24 379 351 vzorků

Metoda	Délka výpočtu [ms]	
	CPU	GPU
Okénko	0,41	0,31
FFT	0,59	0,86
Magnitudy FFT	0,44	0,79
Melovská filtrace	0,57	2,49
Logaritmování	0,42	0,03
DCT	83,90	5,15
Celkem	95,47	28,44

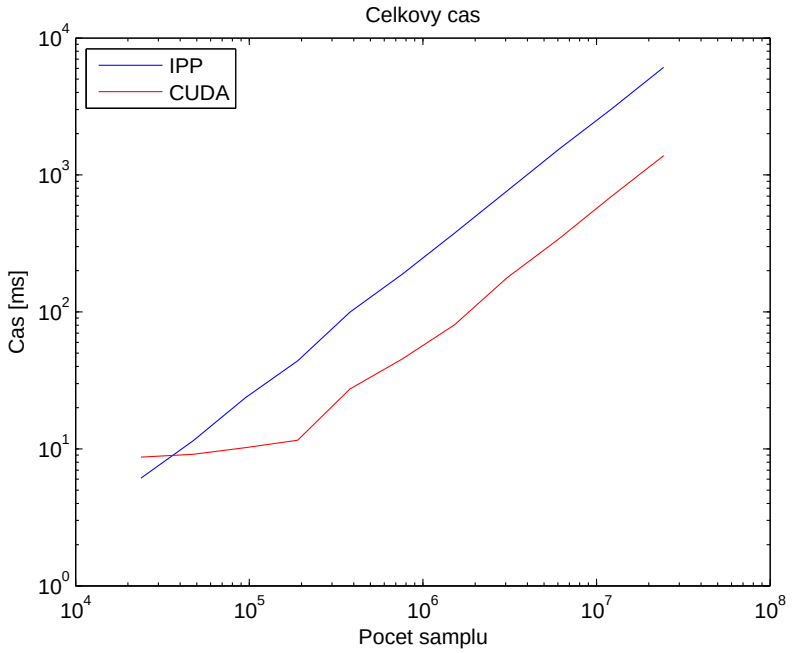
(b) 23 807 vzorků

Tabulka 5.2: Délky výpočtu dílčích metod LC-RC systému

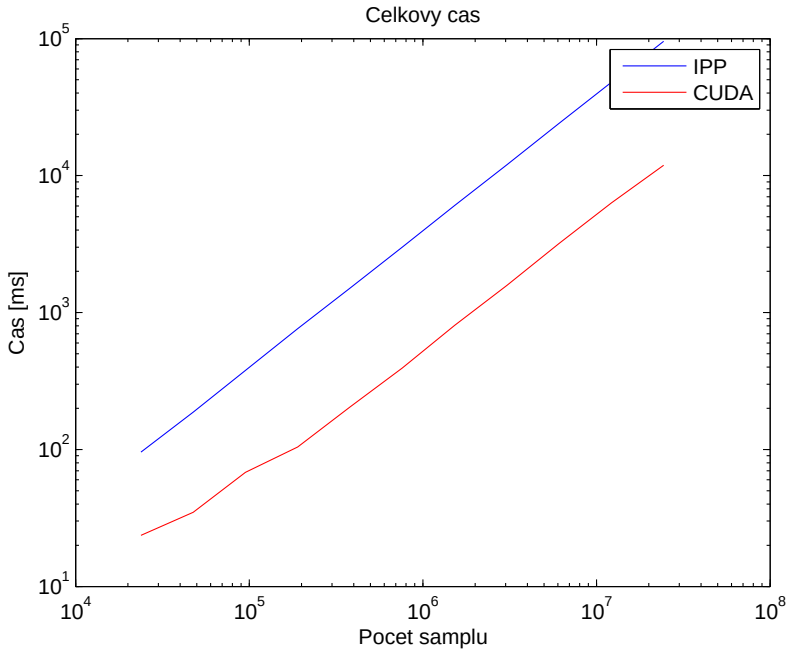
akustického signálu je zaznamenán v grafech 5.1 a 5.2. Osy grafů mají logaritmické měřítko z důvodu lepší přehlednosti. Z grafů je vidět, že pro malé soubory je výpočet na GPU pomalejší než na CPU. To lze vysvětlit tím, že dochází k mnoha přesunům malých bloků dat, které jsou časově náročnější než jeden přesun velkého bloku dat.

Dále je z grafu vidět, že závislost délky výpočtu na počtu vzorků v případě procesoru roste lineárně, zatímco v případě grafické karty je nejdříve téměř konstantní a od určité hodnoty začne lineárně růst. Důvodem je způsob přípravy dat a paměti pro výpočty na GPU. Alokace paměti grafické karty je časově náročná operace, proto se všechna pamět alokuje při startu programu. Program podporuje dávkové zpracování mnoha souborů, proto není možné při inicializaci zjistit, kolik vzorků bude třeba analyzovat. Program také využívá knihovny třetích stran (CUFFT), u kterých není možné odhadnout, kolik paměti alokují. Proto má program parametr, kterým se nastaví limit počtu segmentů, které se budou zpracovávat najednou. Všechna paměť na grafické kartě se alokuje vzhledem k tomuto počtu. Pokud je zpracovávaných vzorků více, vstupní signál se rozdělí a výpočet probíhá v cyklu postupně pro všechny bloky. Ale pokud je vzorků méně, doplní se nulami a počítá se s celým blokem dat. Proto je začátek grafu téměř konstantní.

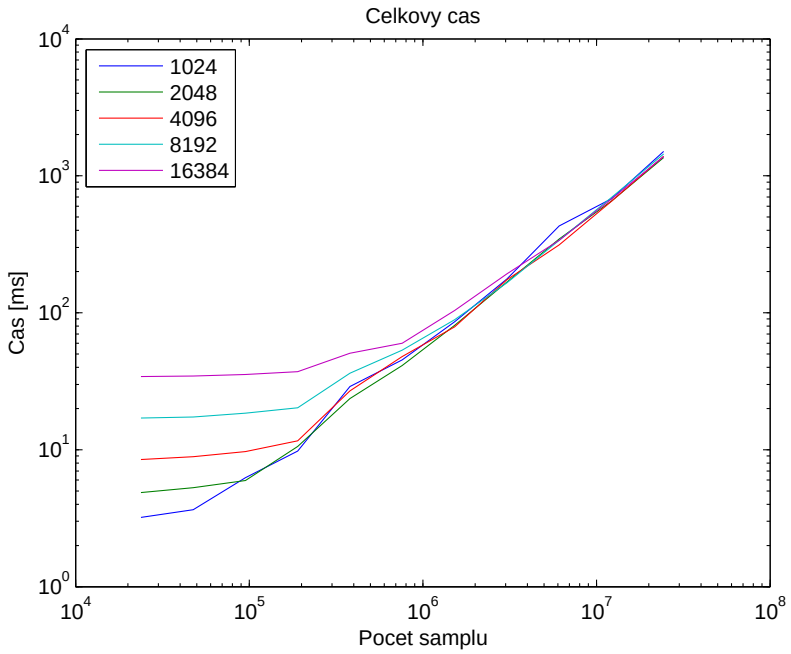
Vliv velikosti alokovaného bloku paměti ilustruje graf 5.3. Program byl spuštěn pro několik velikostí vstupního souboru a alokovaného bloku paměti. Z grafu lze vyčíst, že pro krátké signály běží program rychleji s menším limitem počtu segmentů, poté se rychlosti vyrovnají



Obrázek 5.1: Závislost doby výpočtu MFCC na počtu vstupních hodnot



Obrázek 5.2: Závislost doby výpočtu LC-RC systému na počtu vstupních hodnot



Obrázek 5.3: Závislost doby výpočtu MFCC na GPU na počtu vstupních hodnot pro různé počty zpracovávaných segmentů v jednom bloku

a pro dlouhé signály beží program nepatrně rychleji s větším alokovaným blokem. Čím větší blok je alokovaný, tím méně kernelů je třeba spustit pro zpracování všech dat, ale pro velké bloky není rozdíl velmi patrný.

Konečná data vypočtená na procesoru a grafické kartě se liší v průměru o 1 %. Největší vliv na rozdíl výsledných dat má výpočet FFT, ostatní části výpočtu mají na výslednou chybu v porovnání s FFT velmi malý vliv. Chyba je zřejmě způsobena menší přesností GPU. Program je psán pro karty s *compute capability* 1.1, které umí pracovat pouze s 32-bitovými reálnými čísly zatímco použitý procesor je schopen provádět operace s 80-bitovou přesností. Hodnoty v paměti hosta i zařízení jsou uloženy v 32-bitových číslech, ale všechny mezivýsledky během výpočtů na procesoru mají přesnost vyšší. Kód na zařízení provádí všechny výpočty v 32 bitech.

Kapitola 6

Závěr

Tématem práce bylo urychlení parametrizace řečového signálu s využitím grafické karty. Dodaný program provádí parametrizaci signálu na procesoru a mým úkolem bylo navrhnout vhodný způsob výpočtu zvolených metod parametrizace na GPU a dodaný program rozšířit.

Jak je uvedeno v kapitole 5, u obou zkoumaných metod parametrizace se podařilo dosáhnout cíle a zrychlit parametrizaci s využitím GPU. V případě LC-RC systému je zrychlení dokonce několikanásobné, na testovaném počítači v průměru 15×. Takto významného zrychlení bylo dosáhnuto pravděpodobně proto, že referenční program pro výpočet nepoužívá IPP ani jinou knihovnu, ale počítá DCT pomocí jednoduché smyčky v jazyce C.

Rychlejší parametrizace na GPU se podařilo dosáhnout pro dostatečně velké množství vstupních dat. Pokud je dat nedostatek, dochází k výraznému zpomalení z důvodu alokace paměti na grafické kartě a přenosu dat mezi hostem a zařízením. Z toho plyne omezení navrhovaného řešení. Grafickou kartu není vhodné využívat na zrychlení parametrizace krátkých řečových signálů, ale při parametrizaci dlouhých signálů je možné dosáhnout několikanásobnějšího výkonu. Z tohoto důvodu se nehodí pro online rozpoznávání řeči, kdy se v daný okamžik parametrizuje pouze jeden mikrosegment, ale je vhodná pro rozpoznávání jakýchkoliv delších promluv, např. při automatickém generování titulků k předem nahraným filmům.

Přílohy

Na přiloženém CD je elektronická podoba této bakalářské práce a soubory týkající se vytvořeného programu v těchto složkách:

- `source` – složka se zdrojovým kódem programu. Knihovna pro parametrizaci řečového signálu, která je výsledkem této práce, je v podsložce `gpulib`.
- `bin` – složka obsahující spustitelné soubory programů pro výpočet příznaků metodami MFCC a LC-RC systému. Podsložka `lib` obsahuje dynamicky linkované knihovny potřebné pro spuštění programů, pokud nejsou v systému nainstalovány.

Literatura

[1] *Prof. Ing. Psutka J., CSc., Doc. Ing. Matoušek J., Ph.D., Doc. Ing. Müller L., Ph.D., Doc. Dr. Ing. Radová V.*, **Mluvíme s počítačem česky**
Praha, Academia, 2006

[2] *Petr Schwarz*, **Phoneme recognition based on long temporal context** (Disertační práce)
Brno, Vysoké Učení Technické v Brně, 2008

[3] **CUDA C Programming Guide** [online], Nvidia, 2012 [cit. 20. srpna 2012]
Dostupné na: <http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf>

[4] **CUDA C Best Practices Guide** [online], Nvidia, 2012 [cit. 20. srpna 2012]
Dostupné na: <http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Best_Practices_Guide.pdf>

[5] **CUFFT Library User Guide** [online], Nvidia, 2012 [cit. 20. srpna 2012]
Dostupné na: <http://developer.download.nvidia.com/compute/DevZone/docs/html/CUDALibraries/doc/CUFFT_Library.pdf>

[6] **CUBLAS Library User Guide** [online], Nvidia, 2012 [cit. 20. srpna 2012]
Dostupné na: <http://developer.download.nvidia.com/compute/DevZone/docs/html/CUDALibraries/doc/CUBLAS_Library.pdf>

Příloha A

Programátorská dokumentace

Protože se práce nezabývá celým rozpoznáváním řeči, ale pouze analýzou řečového signálu, je výsledkem práce knihovna funkcí. Ta je určena k použití v programech, které potřebují analyzovat řečový signál na grafické kartě. Knihovna je psaná v jazyce C++ a pro vytvoření souborů nutných pro kompilaci je použit nástroj CMake.

A.1 Struktura knihovny

Knihovna je rozdělena do 3 tříd:

- Třída `GPUfft` slouží pro segmentaci dat a výpočet magnitud FFT koeficientů.
- Třída `GPUMelBanks` slouží pro výpočet energií melovských filtrů s následnou dekorelací použitím kosinové transformace.
- Třída `GPUTraps` slouží pro aplikaci LC-RC systému.

Každá třída je uložena ve 3 souborech o příponách:

- `.h` – hlavičkový soubor s deklarací třídy
- `.cuh` – hlavičkový soubor s deklarací proměnných a funkcí využívajících knihovny CUDA
- `.cu` – soubor s vlastním kódem funkcí a kernelů

Toto rozdělení umožňuje jinému programu použít pouze hlavičkový soubor s příponou `.h`, který se neodkazuje na knihovnu CUDA.

Při chybě metody vyvolají výjimku třídy `GPUException` oddědenou z třídy `std::exception`. Proto většina metod nemá návratovou hodnotu.

A.2 Třída GPUfft

Metody třídy:

void init(int input_data_length, int window_size, int shift)

Parametry:

`input_data_length:` délka vstupních dat v samplech

`window_size:` délka mikrosegmentu, do kterých se data rozdělí, v samplech

`shift:` vzdálenost počátku následujícího mikrosegmentu od počátku segmentu předchozího

Metoda nastaví všechny vnitřní proměnné třídy a alokuje paměť potřebnou pro všechny výpočty.

void cleanup()

Metoda deinicializuje všechny proměnné a vnitřní struktury a dealokuje paměť.

void set_window(const float * window)

Parametry:

`window:` koeficienty váhové funkce okénka. Jedná se o ukazatel do systémové paměti.

Metoda zkopíruje koeficienty do paměti grafické karty. To umožňuje zvolit vlastní váhovou funkci okénka.

void populate_buffer(const float * data)

Parametry:

`data:` PCM koeficienty vstupního signálu. Jeho délka je daná metodou `init`.

Metoda zkopíruje data do paměti grafické karty a rozdělí je na mikrosegmenty.

void apply_window()

Metoda vynásobí hodnoty každého mikrosegmentu v paměti karty hodnotami váhové funkce.

void fft()

Metoda aplikuje na data Furierovu transformaci a do pracovní paměti uloží magnitudy Furierova obrazu.

void apply(const float * data)

Parametry:

`data`: PCM koeficienty vstupního signálu. Parametr má stejný význam jako u metody `populate_buffer`.

Metoda postupně provede všechny operace implementované v této třídě. Postupně zavolá metody:

```
populate_buffer(data);
apply_window();
fft();
```

int get_input_data_length() const

Metoda vrací počet samplů, pro které je ve třídě alokovaná paměť.

int get_window_size() const

Metoda vrací délku mikrosegmentů, do kterých se rozdělí vstupní signál.

int get_window_size2() const

Metoda vrací délku mikrosegmentů zaokrouhlenou na nejbližší vyšší mocninu dvou.

int get_shift() const

Metoda vrací hodnotu, o kterou jsou mikrosegmenty vůči sobě posunuty.

int get_window_count() const

Metoda vrací počet mikrosegmentů, do kterých se rozdělil vstupní signál.

float * get_output_d()

Metoda vrací ukazatel do pracovní paměti třídy. V této paměti je vždy uložen výsledek poslední operace. Jde o paměť na grafické kartě.

void get_output(float * data_out, bool round)

Parametry:

`data_out:` ukazatel do systémové paměti, kam se zkopírují data z pracovní paměti třídy

`round:` určuje, jestli má být šířka výstupních dat zaokrouhlena na nejbližší vyšší mocninu dvou

Data jsou v paměti reprezentována jako matice o šířce rovné mocnině dvou. To umožňuje rychlejší výpočet. Pokud je parametr `round` roven `false`, pak metoda zkopíruje blok dat o šířce `window_size`. Jinak zkopíruje obsah celé pracovní paměti o šířce `window_size2`.

int estimated_window_count(int input_data_len)

Parametry:

`input_data_len:` délka vstupních dat

Metoda vrací, do kolika segmentů se rozdělí vstupní data o délce předané parametrem. Délka segmentů je nastavená metodou `init`.

A.3 Třída GPUMelBanks

Metody třídy:

void init(int window_size, int window_count, int num_banks, float sample_rate, float low_freq, float high_freq, int ceps_len, bool want_c0, float lift_coef)

Parametry:

`window_size:` délka mikrosegmentu, do kterých se data rozdělí, v samplech

`window_count:` počet mikrosegmentů v bloku, který se zpracovává najednou

`num_banks:` počet melovských filtrů v bance

`sample_rate:` vzorkovací frekvence vstupních dat

`low_freq:` frekvence prvního filtru v bance

`high_freq:` frekvence posledního filtru v bance

`ceps_len:` délka DCT transformace

`want_c0:` značí, jestli na výstupu požadujem první koeficient DCT transformace

`lift_coef:` koeficient pro lifter upravující DCT koeficienty

Metoda nastaví všechny vnitřní proměnné třídy a alokuje paměť potřebnou pro všechny výpočty.

void cleanup()

Metoda deinicializuje všechny proměnné a vnitřní struktury a dealokuje paměť.

void set_alpha(float alpha)

Parametry:

`alpha`: alfa koeficient warpovací funkce

Metoda nastaví alfa koeficient bilineární warpovací funkce.

float get_alpha() const

Metoda vrací aktuálně nastavený alfa koeficient warpovací funkce.

void apply_filter_d(float * d_data)

Parametry:

`d_data`: absolutní hodnoty koeficientů furierovy transformace. Jde o ukazatel do paměti grafické karty. Jako parametr lze předat výstup funkce `GPUfft::get_output_d`.

Metoda provede melovskou filtraci na vstupních datech a poté je zlogaritmuje.

void apply_filter(float * data)

Parametry:

`data`: absolutní hodnoty koeficientů furierovy transformace. Jde o ukazatel do paměti hosta.

Metoda zkopíruje data předaná parametrem do paměti grafické karty a poté na ně zavolá metodu `GPUMelBanks::apply_filter_d`.

void dct()

Metoda provede DCT transformaci energií melovských filtrů. Je nutno volat po volání `GPUMelBanks::apply_filter_d`.

int get_window_size() const

Metoda vrací délku mikrosegmentů, do kterých je rozdělen vstupní signál.

int get_window_size2() const

Metoda vrací délku mikrosegmentů zaokrouhlenou na nejbližší vyšší mocninu dvou.

int get_window_count() const

Metoda vrací počet mikrosegmentů, do kterých je rozdělen vstupní signál.

int get_num_banks() const

Metoda vrací počet melovských filtrů v bance.

int get_num_banks2() const

Metoda vrací počet melovských filtrů v bance zaokrouhlený na nejbližší vyšší mocninu dvou.

float get_sample_rate() const

Metoda vrací vzorkovací frekvenci vstupního signálu.

float get_low_freq() const

Metoda vrací frekvenci prvního filtru v bance.

float get_high_freq() const

Metoda vrací frekvenci posledního filtru v bance.

float * get_mfcc_d() const

Metoda vrací ukazatel do paměti grafické karty. Ukazuje na pole, ve kterém jsou uložena data před aplikací DCT.

float * get_dct_d() const

Metoda vrací ukazatel do paměti grafické karty. Ukazuje na pole, ve kterém jsou uložena data po aplikaci DCT.

void get_mfcc(float * data_out, bool round)

Parametry:

`data_out`: ukazatel do systémové paměti, kam se zkopírují data z pracovní paměti třídy

`round`: určuje, jestli má být šířka výstupních dat zaokrouhlena na nejbližší vyšší mocninu dvou

Metoda do paměti hosta zkopíruje data před aplikací DCT. Data jsou v paměti reprezentována jako matice o šířce rovné mocnině dvou. Pokud je parametr `round` roven `false`, pak metoda zkopíruje blok dat o šířce `num_banks`. Jinak zkopíruje obsah celé pracovní paměti o šířce `num_banks2`.

void get_dct(float * data_out, bool round)

Parametry:

`data_out:` ukazatel do systémové paměti, kam se zkopírují data z pracovní paměti třídy

`round:` určuje, jestli má být šířka výstupních dat zaokrouhlena na nejbližší vyšší mocninu dvou

Metoda do paměti hosta zkopíruje data po aplikaci DCT. Data jsou v paměti reprezentována jako matice o šířce rovné mocnině dvou. Pokud je parametr `round` roven `false`, pak metoda zkopíruje blok dat o šířce `dct_len`. Jinak zkopíruje obsah celé pracovní paměti o šířce `dct_len2`.

A.4 Třída GPUTraps

Metody třídy:

void init(int window_count, int num_banks, int trap_len, int dct_len, bool want_c0)

Parametry:

`window_count:` počet mikrosegmentů v bloku, který se zpracovává najednou

`num_banks:` počet melovských filtrů v bance

`trap_len:` délka časového vektoru (před rozdělením na levou a pravou část)

`dct_len:` počet výstupních DCT koeficientů pro jednu část časového vektoru

`want_c0:` značí, jestli na výstupu požadujem první koeficient DCT transformace

Metoda nastaví všechny vnitřní proměnné třídy a alokuje paměť potřebnou pro všechny výpočty.

void cleanup()

Metoda deinicializuje všechny proměnné a vnitřní struktury a dealokuje paměť.

int apply_d(float * d_data, int window_count)

Parametry:

`d_data:` ukazatel do paměti grafické karty. Ukazuje na pole obsahující energie melovských filtrů.

`window_count:` počet segmentů, pro který jsou uložena data v prvním parametru

Metoda aplikuje DCT pro levou i pravou část všech časových vektorů a uloží výstupní koeficienty do vnitřní paměti. Návratová hodnota značí počet segmentů, pro který jsou vypočtena data.

int apply(float * data, int window_count)

Parametry:

`data:` ukazatel do paměti hosta. Ukazuje na pole obsahující energie melovských filtrů.

`window_count:` počet segmentů, pro který jsou uložena data v prvním parametru

Metoda zkopíruje data do paměti grafické karty a poté na ně zavolá metodu `GPUTraps::apply_d` a vrátí její návratovou hodnotu.

int flush()

Metoda zduplikuje poslední segment v pracovní paměti a spočte zbývající výstupní koeficienty tak, aby byl počet výstupních segmentů stejný jako počet segmentů vstupních.

int get_trap_len() const

Metoda vrátí délku časového vektoru před rozdělením na levou a pravou část.

int get_trap_out_len() const

Metoda vrátí délku výstupního vektoru příznaků pro každý segment.

void get_traps(float * data_out, int window_count)

Parametry:

`data_out:` ukazatel do systémové paměti, kam se zkopírují data z pracovní paměti třídy

`window_count:` počet segmentů, pro který se mají zkopírovat výstupní data

Metoda zkopíruje z paměti zařízení do paměti hosta výstupní koeficienty. Počet segmentů, který se metodě předává jako parametr, by měl odpovídat návratové hodnotě posledního volání funkce `apply`, `apply_d` nebo `flush`.