Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra kybernetiky

DIPLOMOVÁ PRÁCE

Plzeň 2014

Josef Michálek

Prohlášení

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

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

Plzeň 29. srpna 2014 .....

podpis

Poděkování

Chtěl bych poděkovat Ing. Janu Vaňkovi, Ph.D., vedoucímu mé diplomové práce, za odborné vedení, cenné rady, věcné připomínky a čas, který mi věnoval při zpracování této práce.

Anotace

Tématem této diplomové práce je využití grafické karty pro urychlení zpracování řeči. Práce obsahuje popis použitých metod parametrizace (MFCC, PLP, TRAPS), adaptace (fMLLR) a také stručný popis programování GPU. Dále se práce zabývá popisem vytvořeného kódu s přihlédnutím k architektuře GPU. Na konci práce je vyhodnocení včetně porovnání s jinými programy.

Klíčová slova: rozpoznávání řeči, parametrizace řečového signálu, adaptace akustického modelu, MFCC, PLP, TRAPS, fMLLR, CUDA, OpenCL

Abstract

The topic of this thesis is the use of graphics card for speech processing acceleration. The thesis contains the description of used methods of parameterization (MFCC, PLP, TRAPS), adaptation (fMLLR) and also a brief description of GPU programming. The next part of the thesis discusses the implemented code and it's advantages for GPU usage. The last part presents the results and compares this implementation against other programs.

Key words: speech recognition, speech signal parameterization, acoustic model adaptation, MFCC, PLP, TRAPS, fMLLR, CUDA, OpenCL

Obsah

Seznam obrázků

Seznam výpisů kódu

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.

Postupem času se vyvíjí stále nové metody zpracování řeči. Jejich zdokonalování probíhá současně se zdokonalováním technologie a je vhodné pro zpracování řeči používat nejmodernější prostředky. Jedním z nejrychleji se vyvíjejících dílů domácích počítačů je dnes grafická karta. Hlavní její výhodou je rychlé zpracování paralelních operací. A protože mnoho metod zpracování řeči lze snadno paralelizovat, jeví se grafická karta jako ideální prostředek k jejich nasazení.

Cílem této diplomové práce je navrhnout a implementovat některé metody zpracování řeči na grafické kartě. Hlavní účel je několikanásobně rychlejší výpočet než na samotném procesoru, který používá většina dnešních systémů zpracování řeči. Tohle téma jsem si vybral, protože se zajímám o zpracování řeči a s programováním grafických karet mám zkušenosti.

Na začátku práce je teoretický popis implementovaných metod zpracování řeči. Dále je stručný popis, z čeho se skládají programy na grafické kartě a jak probíhá jejich vývoj. Pak následuje popis implementace zvolených metod zpracování řeči. Na konci práce je porovnání mé implementace s několika jinými používanými programy.

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).

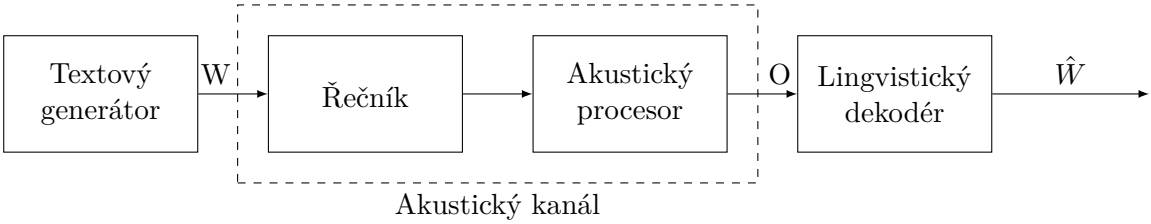
Z těchto důvodů se používá algoritmus na principu dynamického programování. Principem algoritmu je nelineární časová normalizace, kde je kolísání v časové ose modelováno časově nelineární DTW (dynamic time warping) funkcí. Při porovnávání dvou obrazů se jedna časová osa upraví tak, aby se eliminovaly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



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

vektorů příznaků 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[2]. 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]

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:

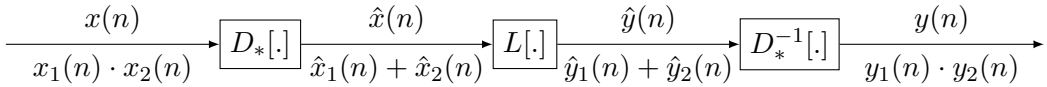
   (a) počet úrovní kvantování (obvykle se volí ve tvaru 2^B, kde B je počet bitů v binárním kódu)

   (b) kvantizační krok Δ.

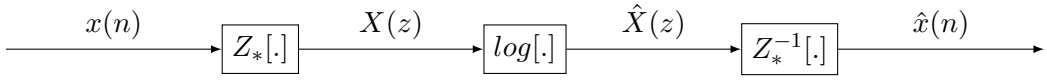
   Jestliže S_{max} je maximální úroveň vzorkovaného signálu, $|s(nT)| \leq S_{max}$, dostaneme

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

[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 $F_v \geq 2F_m$.



Obrázek 2.2: Obecné schéma homomorfního systému



Obrázek 2.3: Schéma charakteristického systému D_*

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.

Obecné schéma homomorfního systému je na obrázku 2.2. Modul D_* se nazývá charakteristický systém a jeho struktura je na obrázku 2.3[2]. 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.

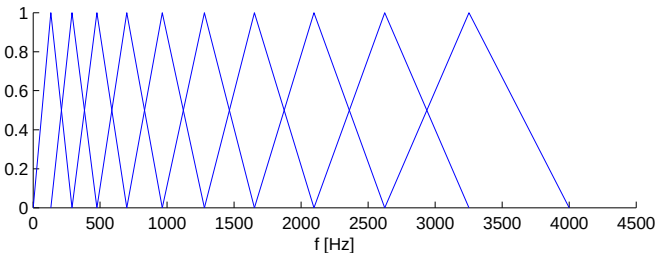
V současnosti jsou preferovány dvě modifikace homomorfního zpracování řeči, a to kepstrální koeficienty odvozené z koeficientů lineární predikce a melovské kepstrální koeficienty (Mel-frequency cepstral coefficients – MFCC).

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

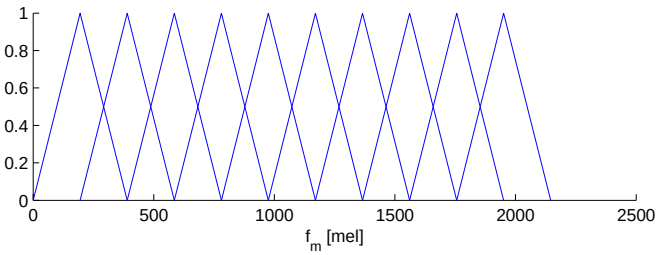
- **Subjektivní vnímání výšky tónu**
  Experimentálně bylo zjištěno, že člověk vnímá výšku tónu subjektivně. Byla zavedena stupnice subjektivní výšky zvuku s jednotkou **mel**. Frekvence v melech m se z frekvence f v [Hz] spočte vzorcem

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

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



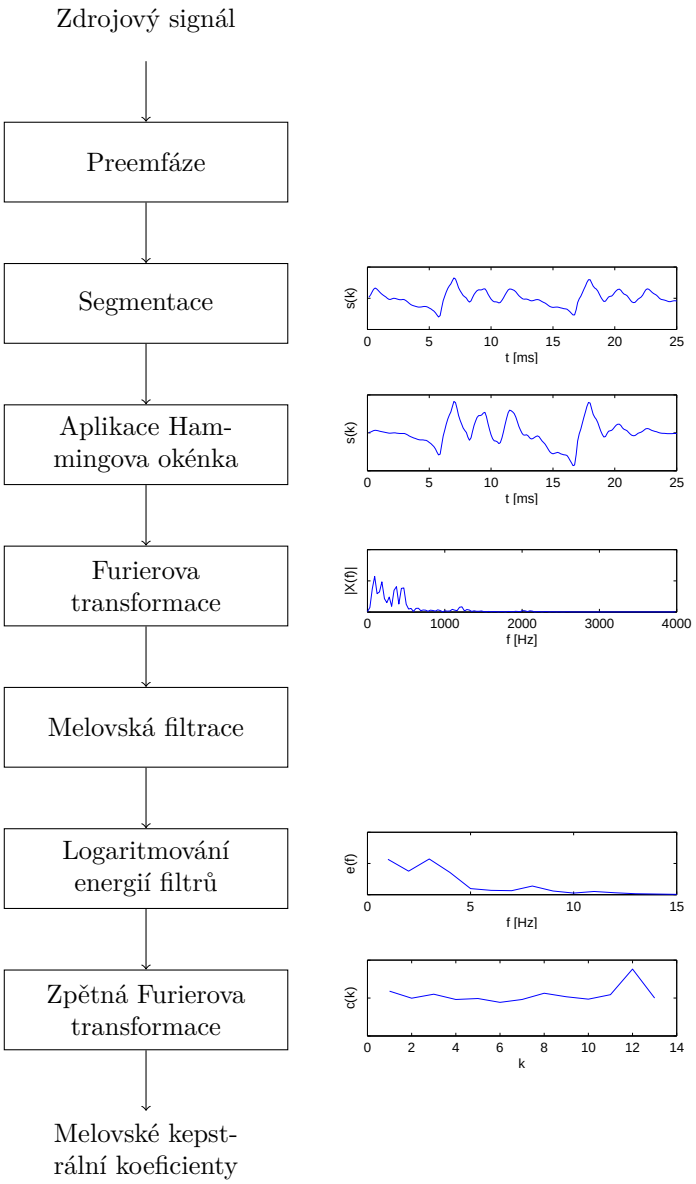
(a) v původní stupnici v [Hz]



(b) v melovské stupnici

Obrázek 2.4: Banka melovských filtrů

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.



Obrázek 2.5: Postup výpočtu melovských kepstrálních koeficientů

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.

Z vlastností diskrétní kosinové transformace vyplývá, že většina energie je soustředěna na nižších frekvencích. Proto kepstrální koeficienty s vyššími indexy nabývají nižších hodnot a v praxi se provádí tzv. liftering podle vztahu

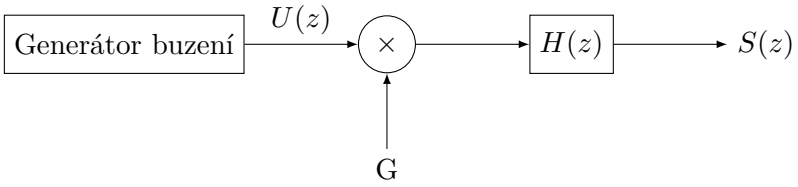
$$c_{lift}(n) = \left[1 + \frac{L}{2}\sin(\pi n/L)\right] c_m(n), \tag{2.3}$$

kde L se typicky volí $L = 22$ [2].

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 transformaci frekvenční osy provést posunem melovských filtrů.

Pro 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



Obrázek 2.6: Model vytváření řeči

frekvence a je obvykle rovna polovině vzorkovací frekvence. Často používanou warpovací funkcí je funkce využívající bilineární transformaci:

$$\eta_\alpha(\omega) = \omega + 2\arctan\left(\frac{(1-\alpha)\sin\omega}{1-(1-\alpha)\cos\omega}\right) \tag{2.4}$$

2.3.2 PLP

Perceptivní lineární prediktivní analýza je metoda, která se snaží odhadnout parametry modelu vytváření řeči z krátkodobé analýzy řečového signálu. Model vytváření řeči se skládá z časově proměnného přenosu a generátoru budící funkce, viz obr. 2.6. Při vytváření znělých zvuků je budící funkcí posloupnost impulzů a při vytváření neznělých zvuků je to náhodný šum. Základním principem metody PLP je předpoklad, že k-tý vzorek signálu $s(k)$ lze vyjádřit jako lineární kombinaci Q předchozích vzorků a buzení $u(k)$[2].

$$s(k) = -\sum_{i=1}^{Q} a_i s(k-i) + Gu(k),$$

kde Q je řád modelu a G je zesílení budící funkce. Přenos modelu lze pak zapsat ve tvaru

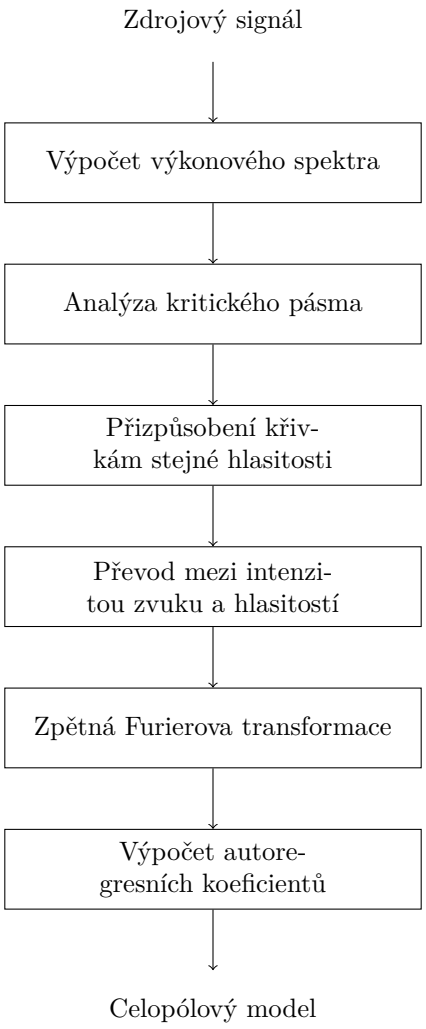
$$H(z) = \frac{S(z)}{U(z)} = \frac{G}{A(z)} = \frac{G}{1+\sum_{i=1}^{Q} a_i z^{-i}} \tag{2.5}$$

Dále jsou popsány jednotlivé kroky metody PLP. Postup výpočtu je také zobrazen na obr. 2.7.

Výpočet výkonového spektra

Pomocí krátkodobé Fourierovy transformace se určí výkonové spektrum řečového signálu. Řečový signál se rozdělí na mikrosegmenty, ty jsou váženy Hammingovo okénkem a pomocí algoritmu FFT se získají vzorky spektra $S(\omega)$. Výkonové spektrum řečového signálu je definováno jako

$$P(\omega) = |S(\omega)|^2 = [\mathrm{Re}S(\omega)]^2 + [\mathrm{Im}S(\omega)]^2 \tag{2.6}$$



Obrázek 2.7: Postup výpočtu PLP

Nelineární transformace frekvencí a kritická pásma

Podobně jako u metody MFCC se i zde využívá subjektivního vnímání výšky tónu člověkem a maskování zvuků. PLP tyto jevy realizuje pomocí nelineární transformace původní osy frekvencí a konstrukcí maskujících křivek, které simulují kritická pásma slyšení. Frekvence se transformuje z $\omega[\mathrm{rad}/s]$ na $\Omega(\omega)[\mathrm{bark}]$ podle vztahu

$$\Omega(\omega) = 6\mathrm{arcsinh}\left(\frac{\omega}{1200\pi}\right) = 6\ln\left(\frac{\omega}{1200\pi} + \sqrt{\left(\frac{\omega}{1200\pi}\right)^2 + 1}\right), \tag{2.7}$$

kde $\omega = 2\pi f$ a f je původní frekvence v Hz. Pásmové propusti jsou popsány vztahem

$$\Psi(z) = \begin{cases} 0 & \text{pro } z < -2,5 \\ 10^{z+0,5} & \text{pro } -2,5 \leq z \leq -0,5 \\ 1 & \text{pro } -0,5 < z < 0,5 \\ 10^{-2,5(z-0,5)} & \text{pro } 0,5 \leq z \leq 1,3 \\ 0 & \text{pro } z > 1,3 \end{cases},$$

kde z je v jednotkách [bark]. Filtry jsou rozmístěné lineárně s krokem přibližně 1 bark, viz obr. 2.8a. Na obr. 2.8b jsou zobrazeny filtry ve stupnici v Hz. První filtr má obvykle střed v počátku přenášeného pásma (0 bark) a poslední filtr na konci přenášeného pásma. Hodnoty těchto filtrů se obvykle nepočítají, ale protože jsou pro další kroky algoritmu potřebné, určují se tak, že se položí rovny hodnotě sousedního filtru.

Přizpůsobení filtrů křivkám stejné hlasitosti

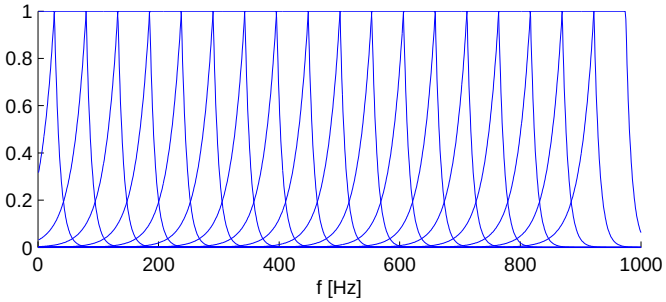
Hlasitost zvuku vnímaného člověkem závisí na intenzitě zvuku a na frekvenci. Pro přizpůsobení výkonového spektra $P(\omega)$ této vlastnosti lidského sluchu se provádí preemfáze diskrétních vzorků křivek pásmového filtru m-tého kritického pásma a odpovídajících hodnot aproximující křivky $E(\omega)$ vztahem

$$\Phi(\Omega(\omega)) = E(\omega)\Psi(\Omega(\omega) - \Omega_m),$$

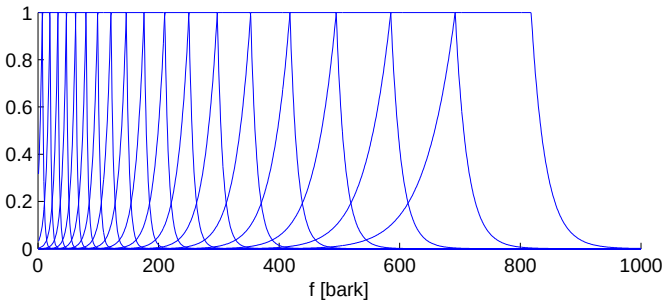
kde Ω_m [bark] je střední frekvence m-tého kritického pásmového filtru a $m = 0, \ldots, M-1$. Funkce $E(\omega)$ představuje aproximaci na stejnou citlivost lidského sluchu v odlišných frekvencích [2].

$$E(\omega) = K\frac{\omega^4(\omega^2 + 56,9 \cdot 10^6)}{(\omega^2 + 6,3 \cdot 10^6)^2(\omega^2 + 379,4 \cdot 10^6)(\omega^6 + 9,6 \cdot 10^26)},$$

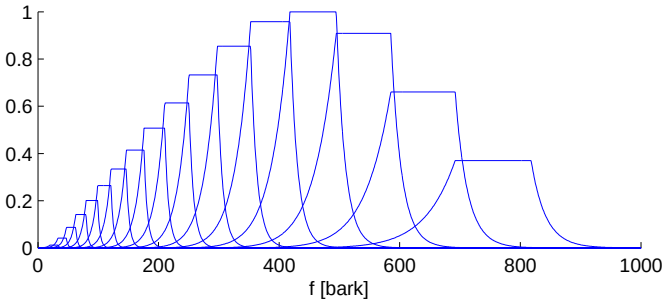
kde K je konstanta, která je volena tak, aby kritický pásmový filtr dosáhl úrovně 1 pro nejvyšší hodnoty intenzity. Filtry po přizpůsobení křivkám stejné hlasitosti jsou zobrazeny na obr. 2.8c.



(a) v původní stupnici v [bark]



(b) ve stupnici v Hz



(c) ve stupnici v Hz po přizpůsobení hlasitosti

Obrázek 2.8: Banka PLP filtrů

Vážená spektrální sumarizace vzorků výkonového spektra

Pokud známe hodnoty výkonového spektra $P(\omega)$ a tvary kritických pásmových filtrů, lze výstup filtrů spočítat vztahem

$$\Xi(\Omega_m) = \sum_{\omega=\omega_d}^{\omega_h} P(\omega)\Phi(\Omega(\omega))$$

Protože jsou hodnoty křivek filtrů nulové mimo rozsah od -2,5 do 1,3 bark, provádí se suma pouze v tomto intervalu. Sumační meze se spočtou pomocí inverzní funkce k (2.7).

Závislost mezi intenzitou zvuku a hlasitostí

Výstupy kritických pásmových filtrů se dále umocní na 0,3. Tento vztah vyjadřuje závislost mezi vnímanou hlasitostí člověkem a intenzitou zvuku.

$$\xi(\Omega_m) = (\Xi(\Omega_m))^{0,3}$$

Aproximace spektrem celopólového modelu

Dalším krokem algoritmu PLP je aproximace hodnot $\xi(\Omega_m)$ spektrem celopólového modelu. Pro chybu predikce platí

$$e(k) = \sum_k \left[s(k) + \sum_{i=1}^{Q} a_i s(k-i) \right]$$

Po aplikaci z-transformace a s využitím vztahu (2.5) dostaneme

$$E(z) = \left[1 + \sum_{i=1}^{Q} a_i z^{-i} \right] S(z) = A(z)S(z), \tag{2.8}$$

kde $E(z)$ a $S(z)$ jsou z-transformace $e(k)$ a $s(k)$ a $A(k)$ je inverzní filtr. Po aplikaci Parsevalova teorému se celková chyba predikce vyjádří vztahem

$$E = \sum_k e^2(k) = \frac{1}{2\pi} \int_{-\pi}^{\pi} |E(\mathrm{e}^{j\omega}|\mathrm{d}\omega,$$

kde $E(\mathrm{e}^{j\omega})$ se získá z $E(z)$ po dosazení $z = \mathrm{e}^{j\omega}$. S využitím rovnic (2.6) a (2.8) můžeme vztah upravit na

$$E = \frac{1}{2\pi} \int_{-\pi}^{\pi} P(\omega)A(\mathrm{e}^{j\omega})A(\mathrm{e}^{-j\omega})\mathrm{d}\omega$$

Celkovou chybu predikce je třeba minimalizovat. Za využití autokorelačního přístupu dostaneme inverzní Fourierovou transformací z výkonového spektra $P(\omega)$ autokorelační funkci

$$R(i) = \frac{1}{2\pi} \int_{-\pi}^{\pi} P(\omega)\cos(i\omega)\mathrm{d}\omega, \quad i = 0, \ldots, Q$$

Protože výkonové spektrum $P(\omega)$ není v praxi spojité, ale jsou známé hodnoty pouze pro konečný počet frekvencí, je třeba autokorelační funkci $R(i)$ definovat pomocí sumy

$$R(i) = \frac{1}{N} \sum_{n=0}^{N-1} P(\omega_n) \cos(i\omega_n), \quad i = 0, \ldots, Q$$

kde N je celkový počet spektrálních bodů na jednotkové kružnici. Platí, že $P(\omega_n)$ je sudá funkce a $\omega_0 = 0$ a $\omega_{N/2} = \pi$. Předpokládáme, že celkový počet spektrálních hodnot v jedné polovině kružnice, kromě bodů 0 a π, je $M - 2$. Celkem je tedy $N = 2(M - 1)$ spektrálních hodnot.

Pro případ aproximace $\xi(\Omega_m)$ spektrem celopólového modelu je třeba vztah pro autokorelační funkci upravit, protože první a poslední filtr ($\xi(\Omega_0)$ a $\xi(\Omega_{M-1})$) leží na hranicích přenášeného pásma. Upravený vztah pro autokorelační funkci je

$$R(i) = \frac{1}{2(M-1)} \left\{ \xi(\Omega_0) \cos(i\omega_0) + 2 \left[\sum_{m=1}^{M-2} \xi(\Omega_m) \cos(i\omega_m) \right] + \xi(\Omega_{M-1}) \cos(i\omega_{M-1}) \right\}, \tag{2.9}$$

kde $i = 0, \ldots, Q$ a $\omega_{M-1} = \pi$. Jak bylo uvedeno dříve, často se volí $\xi(\Omega_0) = \xi(\Omega_1)$ a $\xi(\Omega_{M-1}) = \xi(\Omega_{M-2})$.

Vztah pro výpočet koeficientů celopólového modelu lze zapsat maticově pomocí rovnice

$$\begin{bmatrix} R_n(0) & R_n(1) & R_n(2) & \cdots & R_n(Q-1) \\ R_n(1) & R_n(0) & R_n(1) & \cdots & R_n(Q-2) \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ R_n(Q-1) & R_n(Q-2) & R_n(Q-3) & \cdots & R_n(0) \end{bmatrix} \cdot \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_Q \end{bmatrix} = \begin{bmatrix} -R_n(1) \\ -R_n(2) \\ \vdots \\ -R_n(Q) \end{bmatrix} \tag{2.10}$$

Protože je matice symetrická a v Töplitzově tvaru, lze k výpočtu koeficientů a_i využít Levinsonův algoritmus modifikovaný Durbinem. Výpočet probíhá rekurzivně pro $i = 1, \ldots, Q$ pomocí rovnic

$$E_n^{(0)} = R_n(0) \tag{2.11}$$

$$k_i = -\left[R_n(i) + \sum_{j=1}^{i-1} a_j^{(i-1)} R_n(i-j) \right] / E_n^{(i-1)} \tag{2.12}$$

$$a_i^{(i)} = k_i \tag{2.13}$$

$$a_j^{(i)} = a_j^{(i-1)} + k_i a_{i-j}^{(i-1)}, \quad 1 \leq j \leq i-1 \tag{2.14}$$

$$E_n^{(i)} = (1 - k_i^2) E_n^{(i-1)}, \tag{2.15}$$

kde $a_j^{(i)}$ je j-tý parametr prediktoru řádu i. Za předpokladu, že budící funkce má tvar jednotkového impulzu nebo bílého šumu lze odvodit vztah pro zesílení G.

$$G^2 = R_n(0) + \sum_{i=1}^{Q} a_i R_n(i) = E_n$$

Kepstrální koeficienty LPC

Lineární systém modelující vytváření řeči lze také popsat pomocí kepstrálních koeficientů. Pro jejich určení nejdříve spočteme logaritmus přenosu, podobně jako v homomorfním zpracování řeči v kapitole 2.3.1.

$$\log H(z) = \log \frac{G}{A(z)}$$

Jestliže polynom $A(z)$ proměnné z^{-1} je Q-tého řádu, všechny jeho kořeny leží uvnitř jednotkové kružnice a $A(\infty) = 1$, pak lze provést Taylorův rozvoj

$$\log \frac{G}{A(z)} = c(0) + c(1)z^{-1} + \cdots = \sum_{k=0}^{\infty} c(k)z^{-k},$$

kde $c(k)$ jsou kepstrální koeficienty. Rovnici derivujeme, abychom se zbavili algoritmu.

$$-\sum_{i=1}^{Q} i a_i z^{-i} = \left[\sum_{k=1}^{\infty} k c(k) z^{-k}\right] \left[\sum_{i=0}^{Q} a_i z^{-i}\right]$$

Platí, že $a_0 = 1$. Z předchozího vztahu můžeme odvodit vztah pro výpočet kepstrálních koeficientů.

$$c(1) = -a_1 \tag{2.16}$$

$$c(k) = -a_k - \sum_{i=1}^{k-1} \left(\frac{i}{k}\right) c(i) a_{k-i}, \quad \text{pro } k = 2, 3, \ldots, Q \tag{2.17}$$

$$c(k) = -\sum_{i=1}^{Q} \left(\frac{k-i}{k}\right) c(k-i) a_i, \quad \text{pro } k = Q+1, Q+2, \ldots \tag{2.18}$$

Tyto kepstrální koeficienty se vztahují ke spektrální obálce LPC, proto jsou odlišné od kepstrálních koeficientů z kapitoly 2.3.1. Pro správnou reprezentaci spektrální obálky je třeba vyčíslit alespoň Q koeficientů.

Kepstrální koeficienty jsou obecně dekorelované. Díky tomu se často používají v systémech rozpoznávání řeči založených na skrytých Markovových modelech s diagonálními kovariančními maticemi.

2.3.3 Dynamické koeficienty

Příznaky popsané v předchozích kapitolách (MFCC i PLP) jsou statické, tzn. vektory příznaků popisují pouze jeden mikrosegment řečového signálu. 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. Ty představují časové změny vektorů příznaků a určují se z $2L + 1$ po sobě jdoucích mikrosegmentů řečového signálu pomocí vztahů

$$[\Delta c(i)]_n = \frac{\sum_{k=-L_1}^{L_1} k[c(i)]_{n+k}}{\sum_{k=-L_1}^{L_1} k^2} \tag{2.19}$$

$$[\Delta^2 c(i)]_n = \frac{\sum_{k=-L_2}^{L_2} k[\Delta c(i)]_{n+k}}{\sum_{k=-L_2}^{L_2} k^2}, \tag{2.20}$$

kde $c = [c(0), \ldots, c(M)]_n^T$ je vektor příznaků odpovídající mikrosegmentu n. Typická hodnota L je od 1 do 3[2]. Pro každý mikrosegment pak dostáváme vektor příznaků ve tvaru

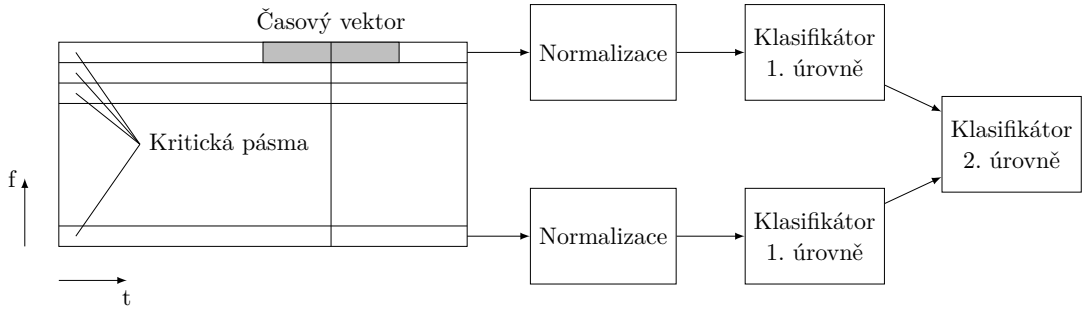
$$c^{\text{out}} = [c, \Delta c, \Delta^2 c]^T$$

2.3.4 TRAPS

Předchozí metody získávají statické příznaky, které se doplňují dynamickými koeficienty. Jiný přístup je použit v metodě TempoRAl PatternS (TRAPS)[3]. 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 2.9.

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 na-



Obrázek 2.9: Diagram TRAPS systému

hradit 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.[3] 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. Tenhle 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

2.4 Gradientní metoda fMLLR

Dnešní systémy rozpoznávání řeči jsou založeny převážně na skrytých Markovových modelech, jejichž výstup je popsán pomocí Gaussovských směsí. Pro rozpoznání řeči je možné natrénovat systém závislý na řečníkovi, ale pro to by bylo potřebné velké množství promluv od jednoho řečníka a systém by nebyl příliš vhodný na rozpoznání promluv jiných řečníků. Proto je vhodnější natrénovat systém nezávislý na řečníkovi z promluv mnoha různých lidí a poté model adaptovat pro rozpoznání řeči konkrétní osoby [4].

Při adaptaci dochází k maximalizaci pravděpodobnosti rozpoznávání. V této práci je pro to použita metoda Maximum Likelihood Linear Regression (MLLR). Je možné měnit buď řečový model nebo vektory příznaků. Změna příznaků je časově méně náročná, proto je použita. Tato metoda se nazývá feature Maximum Likelihood Linear Regression (fMLLR). Pro její řešení je použita Newtonova metoda.

V této metodě se provádí minimalizace podle vztahu

$$\lambda^* = \underset{\lambda}{\operatorname{argmin}}\, \mathcal{F}(O, \lambda)$$

$\mathcal{F}(O, \lambda)$ je kritérium definované jako

$$\mathcal{F}(O, \lambda) = -p(O|\lambda)p(\lambda),$$

kde $p(\lambda)$ značí apriorní informaci o rozložení vektoru λ obsahujícího parametry modelu a $O = \{o_1, o_1, \ldots, o_T\}$ je posloupnost T vektorů příznaků náležejícím jednomu řečníkovi.

Vektory příznaků $o(t)$ se transformují na $\overline{o}(t)$ podle vztahu

$$\overline{o}(t) = Ao(t) + b$$

V případě jednoho normálního rozdělení je parciální derivace pro jeden prvek a_{ij} matice A rovna

$$\frac{\partial \mathcal{F}}{\partial a_{ij}} = \frac{\mu_i - \overline{o}_i(t)}{\sigma_i^2} o_j(t)$$

a druhá parciální derivace

$$\frac{\partial^2 \mathcal{F}}{\partial a_{ij}^2} = -\frac{o_j^2(t)}{\sigma_i^2}$$

Pro vektor b jsou parciální derivace dány vztahy

$$\frac{\partial \mathcal{F}}{\partial b_i} = \frac{\mu_i - \overline{o}_i(t)}{\sigma_i^2}$$

$$\frac{\partial^2 \mathcal{F}}{\partial b_i^2} = -\frac{1}{\sigma_i^2}$$

K celkové derivaci se kromě součtu těchto parciálních derivací také musí přidat derivace $\log(\det(A))$.

Poté lze určit nový odhad matice A

$$A_{(n+1)} = A_{(n)} - \alpha \frac{1}{2} \frac{\frac{\partial \mathcal{F}}{\partial A_{(n)}}}{\frac{\partial^2 \mathcal{F}}{\partial A^2_{(n)}}},$$

kde α je stabilizační konstanta z intervalu $\langle 0, 1 \rangle$.

V této práci je implementována zobecněná verze algoritmu na plnokovarianční matice.

Kapitola 3

GPGPU

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. To se označuje jako GPGPU (General-purpose computing on graphics processing units).

3.1 CUDA

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*.

3.1.1 Architektura GPU

Většina plochy GPU se skládá z čipů zvaných streaming multiprocessory. 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, dojde k divergenci. To znamená, že všechna vlákna ve warpu provádí stejné instrukce, ale výsledek se uloží pouze u některých. Důsledkem divergence je zbytečné vykonání instrukcí a promarněný výkon. Při respektování tohoto SIMT omezení je možné psát kód tak, aby vlákna v jednotlivých warpech spouštěla stejný kód, a dosáhnout tak zrychlení výpočtu.

3.1.2 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 a u některých karet má vyrovnávací paměť.

- **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 N bank, ke kterým se může přistupovat ve stejný časový okamžik. Banky jsou uspořádány tak, aby N po sobě jdoucích 32 bitů velkých částí paměti patřilo do N různých bank. Na kartách s compute capability menší než 2.0 je počet banek 16, u ostatních karet 32.

- **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.

3.1.3 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 je 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 nějaký 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.

Programy pro architekturu CUDA lze psát ve více jazycích. Pro nativní kód v jazyku C++ jsou k dispozici 2 různé API, které v novějších verzích CUDA navzájem kombinovat.

- CUDA Driver API – jedná se o nízkoúrovňové API, které umožňuje detailní správu grafické karty, je nezávislé na jazyku, ale je složitější s ním pracovat a s kódem pro grafickou kartu se pracuje ve formě *cubin* souborů.

- CUDA Runtime API – jde o API postavené nad Driver API. Umožňuje automatickou správu kontextů a zařízení, podporuje jednoduché a přehledné volání GPU kódu a programy v něm napsané lze jednoduše přeložit a vložit do `*.exe` souborů.

3.2 OpenCL

Open Computing Language (OpenCL) je průmyslový standard pro paralelní programování heterogenních počítačových systémů. Tento standard je otevřený a je spravován neziskovým průmyslovým konsorciem Khronos Group. Na rozdíl od architektury CUDA nevyžaduje proprietární hardware jedné firmy. Programy psané v OpenCL lze spouštět na velkém množství procesorů, grafických karet, digitálních signálových procesorů atd.

Na rozdíl od architektury CUDA nemá nic podobného jako CUDA Runtime API. S OpenCL rozhraním se komunikuje ve formě funkcí jazyka C. Kód pro grafickou kartu se překládá při spuštění programu a je na uživateli, kam ho uloží a v jaké podobě.

Paměťový a programový model je shodný s CUDA, některé pojmy mají pouze odlišné názvy.

Kapitola 4

Implementace

V této práci byl pro parametrizaci vytvořen program pojmenovaný Afet (Accelerated feature extraction tool), který umožňuje počítat příznaky MFCC, PLP a TRAPS. Program podporuje výpočty na platformách CUDA, OpenCL a také na procesoru. Na GPU jsou akcelerovány všechny kroky výpočtů, včetně dynamických koeficientů a normalizace.

Při vývoji jsem nejdříve napsal kód pro procesor. Poté jsem vytvořil kód pro platformu CUDA. Hlavním důvodem pro zvolení platformy CUDA před OpenCL je lepší podpora ve vývojovém prostředí MS Visual Studio včetně ladění a profilování kódu. Po dokončení kódu pro CUDA byl přepsán do OpenCL.

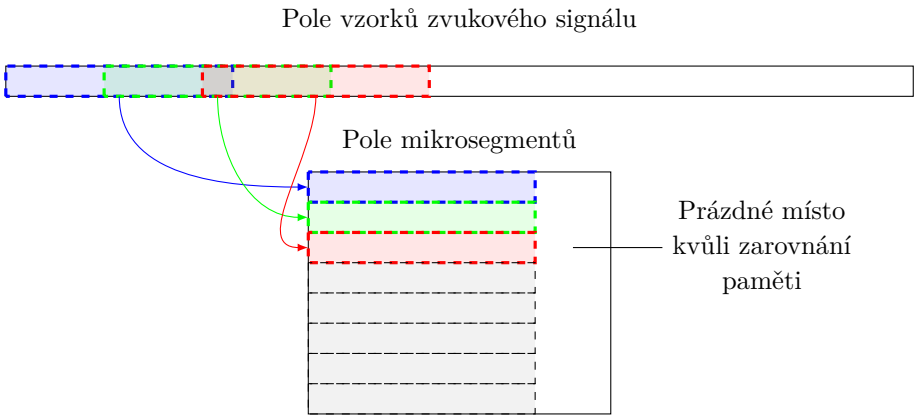
Při spuštění programu se alokuje všechna potřebná paměť a vytvoří se datové struktury nutné k výpočtu, např. matice DCT koeficientů. Při zpracování souborů se pouze kopírují data na kartu, spouští se kernely a data se kopírují zpět z karty. Díky tomu je program vhodný pro dávkové zpracování mnoha souborů.

Všechna data jsou v paměti uložena ve formě matic, kde jednotlivé řádky představují mikrosegmenty, pokud není řečeno jinak. Šířka všech takových matic je doplněna na nejbližší vyšší mocninu dvou z důvodu lepšího zarovnání v paměti a rychlejšího výpočtu FFT. Výška matic je doplněna na násobek velikosti warpu, to zajistí správné zarovnání paměti v místech výpočtu, kde se pracuje s transponovanými maticemi.

4.1 MFCC

4.1.1 Segmentace

Prvním krokem metody MFCC je rozdělení vstupního signálu na okénka a aplikace váhové funkce. Data do paměti grafické karty lze dostat více způsoby. V bakalářské práci [1] jsem ověřil, že nejrychlejší je přesunout všechna data ke zpracování do jednoho velkého bufferu a ty dále rozdělit do segmentů samostatným kernelem. Při přesunu každého segmentu zvlášť by většinu času zabralo samotné spuštění paměťové transakce.



Obrázek 4.1: Znázornění segmentace

Kernel pro segmentaci byl spojen s kernelem pro vážení váhovou funkcí okénka. Na začátku kódu kernelu se zkopírují koeficienty váhové funkce do sdílené paměti. Kernel se spouští v jednorozměrném bloku, kde jeho dimenze je rovna minimu z 256 a velikosti okénka v paměti. To zaručí správné zarovnání warpu a bank sdílené paměti. Grid je dvourozměrný. První rozměr je zvolen takový, aby vlákna zpracovala všechny prvky okénka, a druhý rozměr je roven 256. Bylo experimentálně ověřeno, že při této velikosti gridu je spuštěn dostatek vláken na zakrytí latence při čtení z globální paměti a také vláken není zbytečně mnoho. To se může negativně projevit zejména na kartách, které nemají cache globální paměti. Na obr. 4.1 je znázorněna segmentace pro první 3 mikrosegmenty.

```
1  static ___global___ void kernelSegmentWindow(float * tmp, float * data, float * window, int
       window_count, int window_size, int window_size2, int shift)
2  {
3      int win_index = blockDim.x * blockIdx.x + threadIdx.x,
4          frame_index = blockDim.y * blockIdx.y;
5      extern ___shared___ float win[];
6      win[threadIdx.x] = window[win_index];
7      ___syncthreads();
8      while (frame_index < window_count)
9      {
10         if (win_index < window_size)
11         {
12             int indexIn = frame_index * shift + win_index;
13             data[window_size2 * frame_index + win_index] = tmp[indexIn] * win[threadIdx.x];
14         }
15         else
16             data[window_size2 * frame_index + win_index] = 0;
17         frame_index += gridDim.y * blockDim.y;
18     }
19 }
```

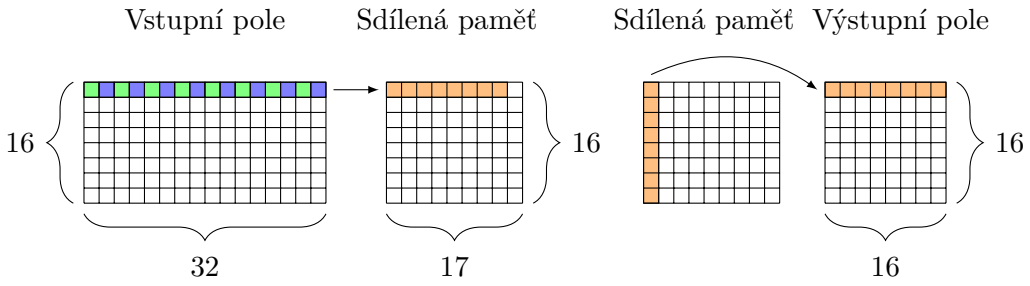
Výpis kódu 4.1: Kernel pro segmentaci dat

4.1.2 FFT

Pro výpočet Furierovy transformace byl použit algoritmus FFT implementovaný firmou NVIDIA v knihovně CUFFT. Žádná alternativní implementace FFT pro platformu CUDA nebyla nalezena.

Knihovna CUFFT umožňuje transformovat data z komplexních do komplexních, z reálných do komplexních a z komplexních do reálných čísel. Protože výstupem segmentace jsou reálná čísla, byla použita verze algoritmu dávající na výstupu komplexní čísla. Pro Furierovu transformaci reálných hodnot platí, že druhá polovina výstupu je komplexně sdružená s první polovinou. Proto je v knihovně CUFFT pro úsporu paměti výstupem transformace o délce N pouze $N/2+1$ hodnot. Zbytek hodnot do N se dá spočítat z prvních $N/2+1$. Dále předpokládám platnost Shannonova teorému, tzn. nejvyšší frekvence, o kterou se v signálu zajímám, nepřekročí polovinu vzorkovací frekvence. Za platnosti tohoto předpokladu je prvních $N/2+1$ hodnot výstupu Furierovy transformace dostatečných a ostatní hodnoty se nedopočítávají.

Z důvodů uvedených v kapitole 4.1.3 je výstup Furierovy transformace nutné transponovat. Následující kernel provádí výpočet absolutní hodnoty komplexních čísel na výstupu FFT a zároveň transpozici dat v paměti. Absolutní hodnota je vypočtena pomocí funkce `cuCabsf` z knihovny CUDA. Pro transpozici dat je použita sdílená paměť o velikosti 16×17 (použitá hodnota makra `TRANSPOSE_TILE_SIZE` je 16). Vlákna se spouští v bloku 16×16. Nejdříve zapíšou spočtené absolutní hodnoty FFT koeficientů do sdílené paměti, synchronizují se s ostatními vlákny v bloku a poté přečtou hodnoty koeficientů ze sdílené paměti transponovaně a uloží je do globální paměti, viz obr. 4.2. Zeleně jsou označeny reálné a modře imaginární části komplexních Furierových koeficientů. Oranžově pak jejich absolutní hodnoty. Šipky naznačují, která data jsou odkud kam kopírována prvním warpem v bloku.



Obrázek 4.2: Transpozice dat

Po sobě následující vlákna provádí čtení i zápis po sobě následujících hodnot v globální paměti během jedné paměťové transakce. Bez využití sdílené paměti by nedocházelo k takovému zarovnanému zápisu hodnot a to by způsobilo 16× více paměťových transakcí. Dále by rozměry sdílené paměti neměly být rovny rozměrům bloku vláken. Pokud

ano, pak by transponované čtení hodnot vedlo ke čtení ze stejné banky pro všechna vlákna ve warpu. Zvětšení jednoho rozměru sdílené paměti o 1 tento problém eliminuje.

```
1  static ___global___ void kernelTranspose(cufftComplex * data_in, float * data_out, int width, int
       height, int opitch, float norm_factor)
2  {
3      ___shared___ float tile[TRANSPOSE_TILE_SIZE][TRANSPOSE_TILE_SIZE + 1];
4      int xIndex = blockDim.x * blockIdx.x + threadIdx.x,
5          yIndex = blockDim.y * blockIdx.y + threadIdx.y,
6          xIndexO = blockDim.y * blockIdx.y + threadIdx.x,
7          yIndexO = blockDim.x * blockIdx.x + threadIdx.y;
8      int rep = (height + gridDim.y * blockDim.y - 1) / (gridDim.y * blockDim.y);
9      for (int i = 0; i < rep; i++)
10     {
11         int shift = gridDim.y * blockDim.y * i;
12         if (xIndex < width && yIndex + shift < height)
13             tile[threadIdx.y][threadIdx.x] = cuCabsf(data_in[width * (yIndex + shift) + xIndex]) *
                   norm_factor;
14         ___syncthreads();
15         if (xIndexO + shift < height && yIndexO < width)
16             data_out[opitch * yIndexO + xIndexO + shift] = tile[threadIdx.x][threadIdx.y];
17         ___syncthreads();
18     }
19 }
```

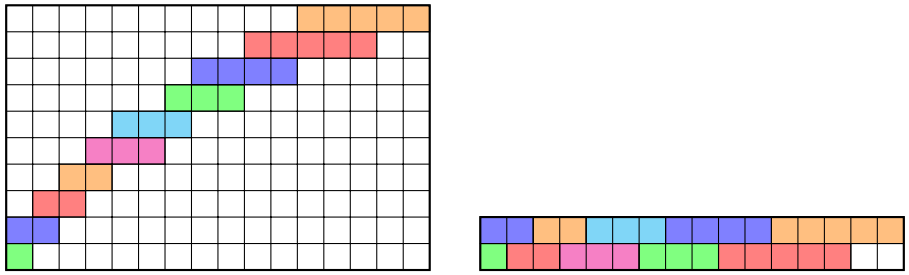
Výpis kódu 4.2: Kernel pro transpozici a výpočet magnitudy FFT koeficientů

4.1.3 Melovská filtrace

V bakalářské práci jsem filtraci bankou melovských filtrů vyřešil pomocí maticového násobení. Vytvořil jsem matici filtračních koeficientů a tou jsem vynásobil výstup Furierovy transformace. V této práci jsem zjistil, že se nejedná o nejrychlejší řešení, a navrhl jsem nový způsob výpočtu filtrace. Většinu matice filtračních koeficientů tvoří nuly, pouze hodnoty blízko diagonály jsou nenulové a proto většinu operací při maticovém výpočtu tvoří násobení nulou. Pro zrychlení filtrace bylo třeba navrhnout jiný algoritmus výpočtu. Matice filtračních koeficientů je zobrazena na obr. 4.3a. Barevná pole odpovídají jednotlivým filtrům a prázdná pole jsou nulová.

V každém místě se překrývají 2 filtry, proto jsem změnil uspořádání filtračních koeficientů do matice o 2 řádcích, viz obr. 4.3b. Samotnou filtraci jsem vyřešil vlastním kernelem. Kernel se spouští s jednorozměrným blokem o velikosti 128. Větší velikost bloku by vedla k pomalejšímu výpočtu, protože kernel vyžaduje velký počet registrů a nedošlo by k zaplnění multiprocesorů aktivními vlákny.

Každé vlákno spočte hodnoty všech filtrů v bance pro každé okénko. Vstupní hodnoty se čtou pouze jednou a kernel počítá najednou 2 filtry. V poli `sum` se akumuluje výsledná hodnota právě počítaných filtrů. Při dosažení konce filtru se logaritmus jeho hodnoty zapíše do výstupního pole, příslušný prvek pole `sum` se vynuluje a dále se v něm počítá hodnota dalšího filtru. Vstupní data je nutné mít v paměti transponované, tzn. jsou uložena po řádkách, které odpovídají Furierovým koeficientům pro jednotlivé frekvence



(a) Původní (b) Změněné

Obrázek 4.3: Uspořádání koeficientů melovských filtrů v paměti

a sloupce odpovídají okénkům. Tohle uspořádání zaručuje zarovnaný přístup do paměti a po sobě následující vlákna ve warpu čtou po sobě následující hodnoty v paměti. Důvod je stejný jako v předchozí kapitole.

```
1  template <bool transposeOutput>
2  static __global__ void kernelFFTAndFilter(float * data_in, float * data_out, int pitch, int
       window_count, int window_size2, int num_banks, int num_banks2, float log_threshold)
3  {
4      int frame_index = blockDim.x * blockIdx.x + threadIdx.x;
5      while (frame_index < window_count)
6      {
7          float sum[2] = {0,0};
8          int curf = 0;
9          int lastf = c_filter_beg[num_banks + 1];
10         for (int i = c_filter_beg[0]; i <= lastf; i++)
11         {
12             float v = data_in[pitch * i + frame_index];
13
14             while (i == c_filter_beg[curf + 1])
15             {
16                 curf++;
17                 if (curf >= 2)
18                 {
19                     int sumidx = curf % 2;
20                     if (transposeOutput)
21                         data_out[pitch * (curf - 2) + frame_index] = logf(max(sum[sumidx],
                               log_threshold));
22                     else
23                         data_out[num_banks2 * frame_index + curf - 2] = logf(max(sum[sumidx],
                               log_threshold));
24                     sum[sumidx] = 0;
25                 }
26             }
27             sum[0] += c_filters[i] * v;
28             sum[1] += c_filters[window_size2 + i] * v;
29         }
30         frame_index += gridDim.x * blockDim.x;
31     }
32 }
```

Výpis kódu 4.3: Kernel pro aplikaci banky melovských filtrů

4.1.4 DCT

V této práci je použita DCT typu II. Výpočet DCT je řešen pomocí maticového násobení. Po spuštění programu je vytvořena matice DCT koeficientů. Tyto koeficienty jsou poté upraveny rovnicí (2.3) pro liftering. Knihovna CUBLAS umožňuje specifikovat, zda se matice mají číst transponovaně nebo ne. Pro použité rozměry matic se ukázalo, že nejrychlejší je číst obě matice transponovaně, proto je transponovaně uložen výstup melovské filtrace. Pokud si uživatel nepřeje spočítat kepstrální koeficienty, ale pouze výstup banky melovských filtrů, výstup kernelu pro filtraci se transponovaně neuloží.

4.2 PLP

4.2.1 Filtrace bankou filtrů

První část výpočtu PLP, která se liší od MFCC, je filtrace výstupu Furierovy transformace. Stejně jako v případě MFCC je i zde třeba transponovat výstup FFT. Kernel pro transpozici se liší pouze v tom, že na výstupu není absolutní hodnota komplexních koeficientů, ale její druhá mocnina. PLP používá energetické spektrum a ne magnitudové, jako MFCC.

V jednom bodě se překrývá mnoho filtrů a proto nelze výpočet filtrace zjednodušit podobně jako u MFCC. Koeficienty všech filtrů jsou uloženy v matici, kde řádky odpovídají jednotlivým filtrům v bance. Kernel se spouští ve dvourozměrném bloku, kde první dimenze je rovna 16 a druhá je rovna minimu z 16 a počtu filtrů v bance. Každé vlákno spočte jednu výstupní hodnotu.

Na začátku kernelu se z indexu vlákna a bloku určí, který filtr a které okénko se má použít pro výpočet. Během výpočtu hodnoty filtru přistupují vlákna stejného warpu k po sobě následujícím vstupním hodnotám a ke stejným filtračním koeficientům. Takový přístup do paměti je rychlý ze stejných důvodů jako v metodě MFCC. Do výstupního pole se uloží hodnota filtru umocněná na hodnotu 0,3.

Tento kernel vynásobí 2 matice, výsledek umocní na 0,3 a také využívá informace, kde začínají a končí filtry, pro eliminaci nadbytečných čtení z paměti a násobení nulou.

```
1  template <bool transposeOutput>
2  static __global__ void kernelFilter(float * data_in, float * data_out, float * d_filters, int pitch, int
       window_count, int window_size2, int num_banks, int num_banks2, float nfft_over_sr, float
       minbark, float stepbark)
3  {
4      int frame_index = blockDim.x * blockIdx.x + threadIdx.x,
5          bank_index = blockDim.y * blockIdx.y + threadIdx.y;
6      float midbark = minbark + bank_index * stepbark;
7      int firstidx = max((int)round(bark2hz(midbark - 2.5) * nfft_over_sr), 0),
8          lastidx = min((int)round(bark2hz(midbark + 1.3) * nfft_over_sr), window_size2 / 2);
9
10     if (bank_index >= num_banks)
11         return;
```

```
12      while (frame_index < window_count)
13      {
14          float sum = 0;
15          for (int i = firstidx; i <= lastidx; i++)
16          {
17              float v = data_in[pitch * i + frame_index];
18              float f = d_filters[window_size2 * bank_index + i];
19              sum += f * v;
20          }
21          sum = pow(sum, 0.3f);
22          if (transposeOutput)
23              data_out[pitch * bank_index + frame_index] = sum;
24          else
25              data_out[num_banks2 * frame_index + bank_index] = sum;
26          frame_index += gridDim.x * blockDim.x;
27      }
28  }
```

Výpis kódu 4.4: Kernel pro filtraci v metodě PLP

4.2.2 Autokorelační funkce

Po filtraci je dalším krokem výpočet hodnot autokorelační funkce. Pro to je použit vztah (2.9). Tento vztah pro výpočet $R(i)$ z $\xi(\Omega)$ lze zapsat vektorově

$$R(i) = \frac{1}{2(M-1)} \begin{bmatrix} \cos(i\omega_0) & 2\cos(i\omega_1) & \dots & 2\cos(i\omega_{M-2}) & \cos(i\omega_{M-1}) \end{bmatrix} \begin{bmatrix} \xi(\Omega_0) \\ \xi(\Omega_1) \\ \vdots \\ \xi(\Omega_{M-2}) \\ \xi(\Omega_{M-1}) \end{bmatrix}$$

a po sloučení všech rovnic pro $i = 1, \dots, Q$ dostaneme vztah

$$R = \frac{1}{2(M-1)} \begin{bmatrix} \cos(0\omega_0) & 2\cos(0\omega_1) & \dots & 2\cos(0\omega_{M-2}) & \cos(i\omega_{M-1}) \\ \cos(1\omega_0) & 2\cos(1\omega_1) & \dots & 2\cos(1\omega_{M-2}) & \cos(i\omega_{M-1}) \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ \cos(Q\omega_0) & 2\cos(Q\omega_1) & \dots & 2\cos(Q\omega_{M-2}) & \cos(i\omega_{M-1}) \end{bmatrix} \begin{bmatrix} \xi(\Omega_0) \\ \xi(\Omega_1) \\ \vdots \\ \xi(\Omega_{M-2}) \\ \xi(\Omega_{M-1}) \end{bmatrix},$$

kde R je vektor hodnot autokorelační funkce, $\xi(\Omega)$ je výstup předchozího kernelu a hodnoty ω_n jsou rovnoměrně rozmístěny tak, aby $\omega_0 = 0$ a $\omega_{M-1} = \pi$.

Násobení matic na grafické kartě je velmi rychlá operace, proto se hodnoty autokorelační funkce počítají pomocí uvedeného maticového vztahu. Matici na levé straně násobení je třeba spočítat pouze jednou při startu programu, to dále urychluje samotný výpočet $R(i)$.

Z důvodů lepšího přístupu do paměti následujících kernelů je maticové násobení provedeno tak, že je výstup transponovaný. Násobení matic je rychlejší, když se jedna matice

čte z paměti transponovaně a druhá ne, ale zhoršení rychlosti výpočtu Levinsonova-Durbinova algoritmu kvůli špatnému přístupu do paměti je příliš velké, v nejhorším případě až 32-krát více paměťových transakcí, podobně jako u transpozice. Proto je zvolena pomalejší metoda násobení matic, ale doba celkového výpočtu je kratší.

4.2.3 Levinsonův-Durbinův algoritmus

Pro výpočet koeficientů a_i celopólového modelu z autokorelační funkce byl zvolen Levinsonův-Durbinův algoritmus, viz rovnice (2.11) až (2.15). Tento algoritmus je velmi rychlý, ale není snadno aplikovatelný na grafické karty, protože je rekurzivní a hlavní výhoda grafických karet je v paralelních výpočtech.

Bylo by možné vytvořit kernel, ve kterém jedno vlákno spočte jednu výslednou hodnotu, ale to by přineslo několik problémů. Jedním by byla složitá synchronizace mezi vlákny z důvodu nutnosti rekurzivního výpočtu. Dalším a ještě závažnějším problémem by byla divergence vláken, díky které by multiprocesory grafické karty strávily většinu času zpracováním divergentních vláken.

Proto byl zvolen přístup, kdy jedno vlákno spočte všechny hodnoty koeficientů a_i pro jeden mikrosegment. Tento přístup eliminuje veškeré problémy s rekurzí, ale jedno vlákno musí několikrát číst i zapsat hodnotu a_i během výpočtu. Čtení z i zapisování do globální paměti je pomalá operace, zejména na kartách bez vyrovnávací paměti. Před spuštěním kernelu se otestuje, zda karta obsahuje dostatek sdílené paměti, a pokud ano, tak se tato paměť použije pro všechny mezivýpočty. Poté se konečný výsledek zapíše do globální paměti pouze jednou.

V i-té iteraci algoritmu se podle vztahu (2.14) spočte $i-1$ mezivýsledků $a_j^{(i)}$. Při tom se využívá hodnot $a_j^{(i-1)}$ z minulé iterace. Bez úpravy by bylo nutné zachovávat hodnoty všech $a_j^{(i-1)}$ a to by vedlo k dvojnásobným paměťovým nárokům. Mezivýsledky se ukládají do sdílené paměti a její velikost je velmi omezená, proto je výhodné algoritmus upravit.

Původní vztah

$$a_j^{(i)} = a_j^{(i-1)} + k_i a_{i-j}^{(i-1)}$$

pro $j = 1, \ldots, i-1$ byl upraven a rozdělen na následující kroky

1. $t_1 = a_j^{(i-1)}$
2. $t_2 = a_{i-j}^{(i-1)}$
3. $a_j^i = t_1 + k_i t_2$
4. $a_{i-j}^i = t_2 + k_i t_1$

pro $j = 1, \ldots, j_{max}$, kde j_{max} je rovno hodnotě $\frac{i-1}{2}$ zaokrouhlené nahoru na nejbližší celé číslo. Tento postup umožňuje určit hodnoty dvou $a_j^{(i)}$ během jednoho kroku smyčky a zároveň nevyžaduje zachovávat hodnoty $a_j^{(i-1)}$ z minulé iterace, protože si ukládá mezivýsledky do proměnných t_1 a t_2.

```
1  template <bool useSMem>
2  static ___global___ void kernelLevinson(float * data_in, float * data_out, int window_count, int
       order, int pitch_in)
3  {
4      int frame_index = blockDim.x * blockIdx.x + threadIdx.x,
5          pitch_out,
6          row_out;
7      float * out;
8
9      extern ___shared___ float s_out[];
10     if (useSMem)
11     {
12         out = s_out;
13         row_out = threadIdx.x;
14         pitch_out = blockDim.x;
15     }
16     else
17     {
18         out = data_out;
19         pitch_out = pitch_in;
20     }
21
22     while (frame_index < window_count)
23     {
24         if (!useSMem)
25             row_out = frame_index;
26
27         float E = data_in[frame_index];
28         out[row_out] = 1;
29         for (int i = 1; i <= order; i++)
30         {
31             float k = data_in[pitch_in * i + frame_index];
32             for (int j = 1; j <= i - 1; j++)
33                 k += out[pitch_out * j + row_out] * data_in[pitch_in * (i - j) + frame_index];
34             k /= -E;
35             out[pitch_out * i + row_out] = k;
36             int jmax = ceil((i - 1) / 2.f);
37             for (int j = 1; j <= jmax; j++)
38             {
39                 float a1 = out[pitch_out * j + row_out];
40                 float a2 = out[pitch_out * (i - j) + row_out];
```

```
41                  out[pitch_out * j + row_out] = a1 + k * a2;
42                  out[pitch_out * (i - j) + row_out] = a2 + k * a1;
43              }
44              E *= (1 - k * k);
45          }
46          for (int i = 0; i <= order; i++)
47              data_out[pitch_in * i + frame_index] = out[pitch_out * i + row_out] / E;
48          frame_index += gridDim.x * blockDim.x;
49      }
50  }
```

Výpis kódu 4.5: Kernel pro aplikaci Levinsonova-Durbinova algoritmu

4.2.4 Kepstrální koeficienty LPC

Výpočet kepstrálních koeficientů podle vztahů (2.16) až (2.18) je také rekurzivní, podobně jako Levinsonův-Durbinův algoritmus. Proto byl využit stejný přístup, kdy se spustí blok vláken tak, že jedno vlákno spočte všechny kepstrální koeficienty jednoho mikrosegmentu. Opět se využívá sdílené paměti, pokud je k dispozici, jinak se mezivýsledky ukládají do globální paměti.

```
1   template <bool useSMem>
2   static ___global___ void kernelCepsCoef(float * a, float * c, int window_count, int order, int
        pitch_in)
3   {
4       int frame_index = blockDim.x * blockIdx.x + threadIdx.x,
5           pitch_out = order + 1,
6           row_out;
7       float * out;
8
9       extern ___shared___ float s_out[];
10      if (useSMem)
11      {
12          out = s_out;
13          row_out = threadIdx.x;
14      }
15      else
16          out = c;
17
18      while (frame_index < window_count)
19      {
20          if (!useSMem)
21              row_out = frame_index;
22
23          float a0 = a[frame_index],
24              a0inv = 1.f / a0;
25          out[pitch_out * row_out] = -log(a0);
26          out[pitch_out * row_out + 1] = -a[pitch_in + frame_index] * a0inv;
27          if (useSMem)
28          {
29              c[pitch_out * frame_index ] = s_out[pitch_out * threadIdx.x ];
30              c[pitch_out * frame_index + 1] = s_out[pitch_out * threadIdx.x + 1];
31          }
32          for (int k = 2; k <= order; k++)
```

```
33          {
34              float sum = 0;
35              for (int i = 1; i <= k - 1; i++)
36                  sum += (k - i) * out[pitch_out * row_out + k - i] * a[pitch_in * i + frame_index];
37              out[pitch_out * row_out + k] = (-a[pitch_in * k + frame_index] - sum / k) * a0inv;
38
39              if (useSMem)
40                  c[pitch_out * frame_index + k] = s_out[pitch_out * threadIdx.x + k];
41          }
42          frame_index += gridDim.x * blockDim.x;
43      }
44  }
```

Výpis kódu 4.6: Kernel pro výpočet kepstrálních koeficientů

4.3 TRAPS

V metodě TRAPS je třeba určit hodnoty kritických pásmových filtrů. Pro to se použije stejný postup jako v metodě MFCC. Po filtraci bankou melovských filtrů je zavolán kernel, který z hodnot těchto filtrů spočte konečný výstup metody TRAPS.

Při spuštění programu se vytvoří matice dvě DCT koeficientů. Jedna pro levý kontext a druhá pro pravý kontext. Tyto koeficienty jsou dále váženy Hammingovým okénkem. Výsledné matice se uloží do konstantní paměti grafické karty. Tato paměť umožňuje velmi rychlé čtení a pro tyto matice je dostatečně velká.

Kernel zpracovává vstupní data v blocích o velikosti 16×16 prvků. Nejdříve si do sdílené paměti zkopíruje úsek vstupních dat, potřebných k výpočtu. Poté v cyklu provede násobení matic, kde data ve sdílené paměti představují jednu matici a druhá matice je matice DCT koeficientů. Tato operace se provádí dvakrát, jednou pro levý a podruhé pro pravý kontext. Na začátku kódu kernelu se spočte hodnota `rep` představující počet úseků dat, které budou spočteny jedním blokem vláken. Každý blok vláken spočte `rep` po sobě následujících úseků dat. Tento přístup zajistí zarovnané čtení z paměti a také lepší využití vyrovnávací paměti, je-li k dispozici.

```
1   static ___global___ void kernelTraps(float * data_in, float * data_out, int num_banks, int
        num_banks2, int half_context, int dct_len, int trap_len, int window_count)
2   {
3       int rep = ceil(float(window_count) / gridDim.y),
4           frame_index = blockIdx.y * rep,
5           bank_index = blockDim.x * blockIdx.x + threadIdx.x,
6           r = 0,
7           window_count_in = window_count + 2 * (half_context - 1);
8
9       ___shared___ float tile[32][16];
10      tile[ threadIdx.y][threadIdx.x] = safe_read(data_in, frame_index + threadIdx.y, bank_index,
            num_banks2, window_count_in);
11      tile[16 + threadIdx.y][threadIdx.x] = safe_read(data_in, frame_index + threadIdx.y + 16,
            bank_index, num_banks2, window_count_in);
12      ___syncthreads();
13
```

```
14      while (r < rep && frame_index < window_count)
15      {
16          //left context
17          int maxshift = min(min(rep - r, 16), window_count - frame_index);
18          for (int shift = 0; shift < maxshift; shift++)
19          {
20              int cur_frame = frame_index + shift;
21              float sum = 0;
22              for (int i = 0; i < half_context; i++)
23                  sum += tile[shift + i][threadIdx.x] * c_dct_matrix[half_context * threadIdx.y + i];
24              if (threadIdx.y < dct_len && threadIdx.x < num_banks)
25                  data_out[trap_len * cur_frame + dct_len * threadIdx.x + threadIdx.y] = sum;
26          }
27
28          __syncthreads();
29          if (threadIdx.y < 15)
30              tile[threadIdx.y][threadIdx.x] = tile[15 + threadIdx.y][threadIdx.x];
31          else
32              tile[31][threadIdx.x] = safe_read(data_in, frame_index + threadIdx.y + 31, bank_index,
                        num_banks2, window_count_in);
33          __syncthreads();
34          tile[15 + threadIdx.y][threadIdx.x] = safe_read(data_in, frame_index + threadIdx.y + 30,
                    bank_index, num_banks2, window_count_in);
35          __syncthreads();
36
37          //right context
38          for (int shift = 0; shift < maxshift; shift++)
39          {
40              int cur_frame = frame_index + shift;
41              float sum = 0;
42              for (int i = 0; i < half_context; i++)
43                  sum += tile[shift + i][threadIdx.x] * c_dct_matrixr[half_context * threadIdx.y + i];
44              if (threadIdx.y < dct_len && threadIdx.x < num_banks)
45                  data_out[trap_len / 2 + trap_len * cur_frame + dct_len * threadIdx.x + threadIdx.
                        y] = sum;
46          }
47
48          frame_index += 16;
49          r += 16;
50
51          __syncthreads();
52          float tmp = tile[2 * threadIdx.y][threadIdx.x];
53          tile[2 * threadIdx.y][threadIdx.x] = tile[2 * threadIdx.y + 1][threadIdx.x];
54          __syncthreads();
55          if (threadIdx.y == 0)
56              tile[31][threadIdx.x] = safe_read(data_in, frame_index + 31, bank_index, num_banks2,
                        window_count_in);
57          else
58              tile[2 * threadIdx.y - 1][threadIdx.x] = tmp;
59          __syncthreads();
60      }
61  }
```

Výpis kódu 4.7: Kernel pro výpočet metody TRAPS

4.4 Delta koeficienty

Pro výpočet delta i akceleračních koeficientů je použit stejný kernel. Akcelerační koeficienty se spočtou voláním tohoto kernelu, kde delta koeficienty představují vstupní hodnoty.

Výpočet probíhá v blocích jako u transpozice dat. Nejdříve se do sdílené paměti zkopíruje blok dat o velikosti $16 \times (16 + 2L)$, kde L odpovídá parametru kernelu `t1`, viz rovnice (2.19) (delta koeficienty v čase t se spočtou ze statických koeficientů od času $t - L$ do času $t + L$). Velikost sdílené paměti pro tento kernel se určí při volání kernelu podle zvoleného parametru L. Po kopírování se otestuje, jestli byl dosažen konec vstupních dat a pokud ano, vlákno se ukončí. Tento test se provádí až po kopírování do sdílené paměti proto, že jsou vstupní data posunuté o L pozic, a i když nějaké vlákno nespočte žádnou výstupní hodnotu, tak všechna vlákna zkopírují vstupní hodnoty do sdílené paměti k dispozici ostatním vláknům. Poté se pomocí vzorce pro výpočet delta koeficientů spočtou jejich hodnoty a uloží se do výstupního pole.

```
1  static ___global___ void kernelDelta(float * data_in, float * data_out, int cols, int window_count,
       int l1)
2  {
3      int frame_index = blockDim.y * blockIdx.y + threadIdx.y,
4          col = blockDim.x * blockIdx.x + threadIdx.x;
5
6      extern ___shared___ float tile[][DELTA_TILE_SIZE];
7      while (true)
8      {
9          ___syncthreads();
10         tile[threadIdx.y][threadIdx.x] = safe_read(data_in, frame_index, col, cols, window_count +
               2 * l1);
11         if (threadIdx.y < 2 * l1)
12             tile[DELTA_TILE_SIZE + threadIdx.y][threadIdx.x] = safe_read(data_in, frame_index
                   + DELTA_TILE_SIZE, col, cols, window_count + 2 * l1);
13         ___syncthreads();
14
15         if (frame_index >= window_count)
16             return;
17
18         float num = 0,
19             den = 0;
20         for (int l = 1; l <= l1; l++)
21         {
22             num += l * (tile[threadIdx.y + l1 + l][threadIdx.x] - tile[threadIdx.y + l1 - l][threadIdx.x
                   ]);
23             den += l * l;
24         }
25         data_out[cols * frame_index + col] = num / (2 * den);
26
27         frame_index += gridDim.y * blockDim.y;
28     }
29 }
```

Výpis kódu 4.8: Kernel pro výpočet delta koeficientů

4.5 Normalizace

Některé aplikace vyžadují normalizaci příznaků. Program podporuje 3 druhy normalizace:

1. CMN (Cepstral mean normalization)
   Každý prvek je normalizován na nulovou střední hodnotu vzorcem:
   $$\hat{x}_i = x_i - \mu \tag{4.1}$$
2. CVN (Cepstral variance normalization)
   Každý prvek je normalizován na nulovou střední hodnotu a jednotkovou směrodatnou odchylku vzorcem:
   $$\hat{x}_i = \frac{x_i - \mu}{\sigma} \tag{4.2}$$
3. Normalizace podle minima/maxima
   Každý prvek je normalizován na nulovou střední hodnotu a absolutní hodnota normalizovaného prvku nepřesáhne 1 podle vzorce:
   $$\hat{x}_i = \frac{x_i - \mu}{\max\{|x_{min} - \mu|, |x_{max} - \mu|\}} \tag{4.3}$$

kde x_i je prvek na pozici i, $\hat{x}_i$ je normalizovaný prvek, μ je střední hodnota, σ je směrodatná odchylka a platí

$$x_{min} = \min_i x_i, \quad x_{max} = \max_i x_i$$

Pro správné výsledky je třeba počítat normalizaci na celém zpracovávaném zvukovém souboru najednou. Program zpracovává soubory po blocích na grafické kartě a paměť RAM používá pouze ke vstupu a výstupu dat. Dostatečně velký soubor, pro jehož zpracování není dostatek paměti na grafické kartě, není možné normalizovat celý najednou. Soubor by bylo třeba zpracovat ve dvou fázích. V první by se získaly požadované charakteristiky dat pro normalizaci (např. průměr, směrodatná odchylka) a poté by se soubor znovu načetl a normalizoval. Do paměti karty se nevejdou pouze velmi dlouhé soubory o délce v řádu milionů okének. Za předpokladu neměnných podmínek pořizování zvukové nahrávky lze říci, že se budou charakteristiky jako průměr a směrodatná odchylka jednotlivých bloků zpracovávaných dat lišit velmi málo a chyba bude zanedbatelná. Z důvodu vyšší rychlosti zpracování se proto normalizace provádí na každém zpracovávaném bloku zvlášť s výjimkou posledního. Ten pro normalizaci používá parametry předchozího bloku, protože obsahuje velmi málo dat. Např. u metod MFCC a PLP je velikost posledního bloku v řádu jednotek, bez delta koeficientů je rovna jednomu okénku.

Výpočet celé normalizace bylo třeba rozdělit do 3 kernelů. První kernel spočte částečné součty hodnot a maxima/minima pro 256 úseků vstupních dat. Požadované charakteristiky jsou u tohoto i ostatních kernelů určené pomocí šablon jazyka C++.

```
template <typename Float, bool wantVar, bool wantMinMax>:
static ___global___ void kernelSum(float * data, Float * mean, Float * var, float2 * minmax, int cols,
      int window_count)
{
    int rep = ceil(float(window_count) / gridDim.y),
        frame_index = blockIdx.y * rep;
    int xIndex = blockDim.x * blockIdx.x + threadIdx.x;

    Float sum = 0,
        sum2 = 0;
    float2 minmaxv = make_float2(FLT_MAX, -FLT_MAX);
    for (int r = 0; r < rep && frame_index < window_count; r++, frame_index++)
    {
        float v = data[cols * frame_index + xIndex];
        sum += v;
        if (wantVar)
            sum2 += v * v;
        if (wantMinMax)
        {
            minmaxv.x = min(minmaxv.x, v);
            minmaxv.y = max(minmaxv.y, v);
        }
    }
    mean[cols * blockIdx.y + xIndex] = sum;
    if (wantVar)
        var[cols * blockIdx.y + xIndex] = sum2;
    if (wantMinMax)
        minmax[cols * blockIdx.y + xIndex] = minmaxv;
}
```

Výpis kódu 4.9: Kernel pro první krok normalizace

Po spočtení částečných součtů se spustí druhý kernel. Ten z nich určí konečné charakteristiky dat, jako jsou průměr, směrodatná odchylka a hodnota minimálního a maximálního prvku. Výsledné charakteristiky se uloží na první pozice vstupních polí.

```
template <typename Float, bool wantVar, bool wantMinMax>
static ___global___ void kernelFinalizeSum(Float * mean, Float * var, float2 * minmax, int cols, int
     rows, int window_count)
{
    int xIndex = blockDim.x * blockIdx.x + threadIdx.x;

    Float sum = 0,
        sum2 = 0;
    float2 minmaxv = make_float2(FLT_MAX, -FLT_MAX);
    for (int i = 0; i < rows; i++)
    {
        sum += mean[cols * i + xIndex];
        if (wantVar)
            sum2 += var[cols * i + xIndex];
        if (wantMinMax)
        {
            minmaxv.x = min(minmaxv.x, minmax[cols * i + xIndex].x);
            minmaxv.y = max(minmaxv.y, minmax[cols * i + xIndex].y);
        }
    }
    mean[xIndex] = sum / window_count;
```

```
21      if (wantVar)
22      {
23          Float v = rsqrt((sum2 - sum * (sum / window_count)) / (window_count - 1));
24          var[xIndex] = v;
25      }
26      if (wantMinMax)
27          minmax[xIndex] = minmaxv;
28  }
```

Výpis kódu 4.10: Kernel pro druhý krok normalizace

Po určení požadovaných charakteristik se může provést normalizace dat podle vztahu (4.1), (4.2) nebo (4.3). Pro každý druh normalizace byl napsán zvláštní kernel. Každý z těchto kernelů si na začátku zkopíruje do sdílené paměti charakteristiky potřebné k normalizaci. V těle kernelu se k nim pak přistupuje pouze skrz sdílenou paměť.

```
1   template <typename Float>
2   static ___global___ void kernelNormalizeCMN(float * data, Float * mean, int cols, int window_count
        )
3   {
4       int xIndex = blockDim.x * blockIdx.x + threadIdx.x,
5           frame_index = blockDim.y * blockIdx.y + threadIdx.y;
6
7       extern ___shared___ float smean[];
8
9       if (threadIdx.y == 0)
10          smean[threadIdx.x] = (float)mean[xIndex];
11      ___syncthreads();
12
13      while (frame_index < window_count)
14      {
15          int idx = cols * frame_index + xIndex;
16          float v = data[idx];
17          data[idx] = v - smean[threadIdx.x];
18
19          frame_index += gridDim.y * blockDim.y;
20      }
21  }
```

Výpis kódu 4.11: Kernel pro normalizaci typu CMN

```
1   template <typename Float>
2   static ___global___ void kernelNormalizeCVN(float * data, Float * mean, Float * var, int cols, int
        window_count)
3   {
4       int xIndex = blockDim.x * blockIdx.x + threadIdx.x,
5           frame_index = blockDim.y * blockIdx.y + threadIdx.y;
6
7       extern ___shared___ float smean[];
8       float * svar = smean + cols;
9
10      if (threadIdx.y == 0)
11      {
12          smean[threadIdx.x] = (float)mean[xIndex];
13          svar[threadIdx.x] = (float)var[xIndex];
14      }
```

```
15      __syncthreads();
16
17      while (frame_index < window_count)
18      {
19          int idx = cols * frame_index + xIndex;
20          float v = data[idx];
21          data[idx] = (v - smean[threadIdx.x]) * svar[threadIdx.x];
22
23          frame_index += gridDim.y * blockDim.y;
24      }
25  }
```

Výpis kódu 4.12: Kernel pro normalizaci typu CVN

```
1   template <typename Float>
2   static __global__ void kernelNormalizeMinMax(float * data, Float * mean, float2 * minmax, int
        cols, int window_count)
3   {
4       int xIndex = blockDim.x * blockIdx.x + threadIdx.x,
5           frame_index = blockDim.y * blockIdx.y + threadIdx.y;
6
7       extern __shared__ float smean[];
8       float * sminmax = smean + cols;
9
10      if (threadIdx.y == 0)
11      {
12          float m = (float)mean[xIndex];
13          smean[threadIdx.x] = m;
14          sminmax[threadIdx.x] = 1.f / max(abs(minmax[xIndex].x - m), abs(minmax[xIndex].y - m));
15      }
16      __syncthreads();
17
18      while (frame_index < window_count)
19      {
20          int idx = cols * frame_index + xIndex;
21          float v = data[idx];
22          data[idx] = (v - smean[threadIdx.x]) * sminmax[threadIdx.x];
23
24          frame_index += gridDim.y * blockDim.y;
25      }
26  }
```

Výpis kódu 4.13: Kernel pro normalizaci podle minima/maxima

Kernely pro normalizaci umožňují jako parametr šablony specifikovat datový typ pro výpočet charakteristik příznaků. Pokud je k dispozici grafická karta podporující datový typ `double` (compute capability 1.3), tak je použit. Jinak se použije `float`. Aritmetické operace nad typem `double` jsou pomalejší, ale zaručují větší přesnost výsledků, protože při normalizaci velkého množství dat může u typu `float` nastat ztráta přesnosti, kdy se sčítá velké množství malých hodnot.

4.6 OpenCL

Kód pro platformu OpenCL vznikl přepisem kódu pro platformu CUDA. Zdrojové soubory s kódem pro grafickou kartu jsou na systému MS Windows vložené do spustitelného souboru ve formě zdrojů (resources). Pokud při překladu není detekován překladač MS Visual Studio, použije se standardní načítání kernelů z textových souborů, které musí být ve stejné složce jako program.

Pro násobení matic je použita knihovna clAmdBlas (součástí clMath, dříve APPML).

Výpočet FFT je realizován pomocí 2 různých knihoven, clAmdFft (opět součást projektu clMath) a implementace FFT od firmy Apple. Na kartách NVidia knihovna clAmdFft nevrací správné výsledky. Která knihovna se má použít pro výpočet se určí pomocí parametru programu.

V tabulce 4.1 je vidět porovnání délky výpočtu FFT. Jedná se o hromadný výpočet 65 536 různých FFT délky 128, 256 a 512 prvků. Některá pole jsou prázdná, protože CUFFT podporují pouze karty od NVidia a na těchto kartách clAmdFft nefunguje. Z tabulky je vidět, že pro některé karty je rychlejší clAmdFft a pro některé implementace od Apple. Proto program podporuje obě knihovny.

Tabulka 4.1: Porovnání doby výpočtu FFT v [ms] pro různé délky a GPU

	128 prvků			256 prvků			512 prvků		
GPU	CUFFT	clAmdFft	Apple	CUFFT	clAmdFft	Apple	CUFFT	clAmdFft	Apple
NVidia GTX 660	1,4	–	1,6	2,5	–	6,6	4,9	–	4,9
NVidia GT 640	5,2	–	5,5	10,3	–	10,3	20,5	–	20,6
AMD HD 5670	–	4,4	3,9	–	10,9	12,2	–	15,0	147,4
AMD HD 7700	–	10,3	20,1	–	20,2	38,7	–	38,6	242,1
Intel HD 4000	–	16,0	14,3	–	25,3	26,9	–	52,3	54,8

4.7 Adaptace řečového modelu

Pro adaptaci byl dodán vedoucím práce program, který ji počítá na procesoru pomocí instrukční sady SSE2. V rámci této diplomové práce byl program rozšířen o možnost výpočtu na platformě CUDA. Soubor `fMLLRCUDA.cu` obsahuje všechny změny a nové funkce, které jsem do programu přidal.

Výpočet gradientní metody probíhá v iteracích. V každé iteraci se spočtou derivace matice A a vektoru B, použijí se pro jejich změnu a v další iteraci proběhne stejný výpočet s novými hodnotami A a B.

4.7.1 Transformace vektorů příznaků

Nejdříve je třeba transformovat vstupní hodnoty vektorů příznaků x podle vztahu

$$x_t = A \cdot x + B, \tag{4.4}$$

kde x označuje jeden vektor příznaků.

Na grafické kartě je dostatek paměti pro všechna vstupní data a pomocné paměťové struktury, proto je možné transformovat všechny vektory příznaků během jediného volání kernelu. Následující kernel provede výpočet rovnice (4.4) pro všechny vektory x najednou. Kernel se spouští v blocích o velikosti 16×16 vláken a každé vlákno vypočte hodnotu jednoho prvku vektoru x_t. Vstupní data používané v této práci jsou vždy dimenze 36, tzn. matice A bude mít rozměry 36×36. Všechny běžně používané karty podporující CUDA mají dostatek sdílené paměti, proto je do ní uložena celá matice A. Pro násobení $A \cdot x$ je pak používána pouze kopie matice ve sdílené paměti.

Každé vlákno při celém svém běhu používá takový prvek vektoru B, který odpovídá jeho indexu. Proto není pro vektor B třeba použít sdílenou paměť, potřebný prvek se načte pouze jednou a uloží se do registru. Po vynásobení vektoru x maticí A se k výsledku pouze přičte hodnota odpovídajícího prvku B z tohoto registru.

Výstupem kernelu je pole vektorů x_t.

```
1  __global__ void KernelTransformT(int dim, int numFrames, float * xt, const float * x, const float
       * A, int ld, const float * B)
2  {
3      int ix = blockDim.x * blockIdx.x + threadIdx.x,
4          f = blockDim.y * blockIdx.y + threadIdx.y;
5
6      extern __shared__ float sA[];
7      for (int y = threadIdx.y; y < dim; y += blockDim.y)
8          for (int x = threadIdx.x; x < dim; x += blockDim.x)
9              sA[dim * y + x] = A[ld * y + x];
10     __syncthreads();
11
12     if (ix < dim)
13     {
14         float B_ = B[ix];
15         while (f < numFrames)
16         {
17             float sum = 0;
18             for (int i = 0; i < dim; i++)
19                 sum += x[ld * f + i] * sA[dim * i + ix];
20             xt[ld * f + ix] = sum + B_;
21             f += gridDim.y * blockDim.y;
22         }
23     }
24 }
```

Výpis kódu 4.14: Kernel pro transformaci příznaků

4.7.2 Výpočet pomocných hodnot x_c

Během celého výpočtu gradientní metody se několikrát používá hodnota

$$x_c = C_{sg}^{-1} \cdot (\mu_{sg} - x_t),$$

kde C_{sg}^{-1} je inverzní kovarianční matice odpovídající Gaussovké složce g stavu s, μ_{sg} je odpovídající vektor středních hodnot a x_t je transformovaný vektor příznaků x odpovídající stavu s.

Hodnotu x_c je třeba spočítat pro všechny Gaussovské složky g odpovídajících vektorů x a stavů s. Protože počet složek pro každý stav není konzistentní (pohybuje se v řádu jednotek až do několika desítek) a také počet vektorů x odpovídajících jednotlivým stavům je velmi proměnlivý (od několika jednotek do několika tisíc), nebylo možné data uložit do pravidelné paměťové struktury s jednoduchým výpočtem indexu prvku z hodnot s a g a zároveň zbytečně nealokovat velké množství paměti.

Je možné data rozdělit a v jednu chvíli zpracovat pouze určitý počet stavů, ale pro další výpočty je žádoucí určit hodnoty všech x_c najednou za cenu složitější indexace prvků v paměti.

Byla zvolena taková struktura paměti, kde jsou za sebou uloženy po sobě následující hodnoty x_c odpovídající jednotlivým stavům s a složkám stavů g. Tyto hodnoty nejsou v paměti pravidelně, např. pokud má první stav 4 složky a odpovídají mu 2 vektory x a druhý stav má 10 složek s 12 vektory x, pak hodnoty x_c pro druhý stav začínají na indexu $4 \cdot 2$ a pro třetí stav na indexu $4 \cdot 2 + 10 \cdot 12$. Pro správnou indexaci prvků bylo třeba vytvořit několik polí.

Následující pole mají velikost rovnou počtu vektorů x_c a značí, kterým hodnotám s, g a x vektory x_c náleží:

- `stateArr` – obsahuje index stavu s
- `frameArr` – obsahuje index vektoru x, hodnota 0 odpovídá prvnímu vektoru x pro odpovídající stav s
- `gaussArr` – obsahuje index Gaussovské složky g, hodnota 0 odpovídá první složce stavu s

Další pole mají velikost rovnou počtu stavů a obsahují informace o pozici následujících hodnot v paměti:

- `gaussOffset` – index první Gaussovské složky stavu s v poli všech Gaussovských složek všech stavů
- `frameOffset` – index prvního vektoru x náležejícího stavu s v poli všech vektorů x

Za pomoci těchto polí se v kernelu určí indexy všech potřebných hodnot pro výpočet konkrétního vektoru x_c. Parametr kernelu `xc_size` obsahuje celkový počet vektorů x_c. Kernel se spouští stejně jako předchozí v bloku vláken o velikosti 16×16 a každé vlákno spočte jeden prvek jednoho vektoru x_c. Celý blok spočte 16 vektorů x_c.

Opět je využito sdílené paměti. Ukládají se do ní hodnoty vektorů $\mu_{sg} - x_t$, protože při násobení inverzní kovarianční maticí každé vlákno čte všechny prvky těchto vektorů a sdílená paměť zamezí mnohonásobnému přístupu do globální paměti. Na inverzní kovarianční matici není ve sdílené paměti místo, protože v nejhorším případě může každý z 16 následujících vektorů x_c zpracovávaných blokem vláken odpovídat jinému stavu, to by si vyžádalo 16 různých inverzních kovariančních matic o celkové velikosti 83 kB a nejnovější karty (s compute capability 5.0) mají pouze 64 kB sdílené paměti na multiprocesor.

```
1  __global__ void KernelCompXC(int dim, int xc_size, const int * stateArr, const int * frameArr,
       const int * gaussArr, const int * gaussOffset, const int * frameOffset, float * xc, const float * xt
       , const float * mu, int ld, const float * icov)
2  {
3      int ix = blockDim.x * blockIdx.x + threadIdx.x,
4          iy = blockDim.y * blockIdx.y + threadIdx.y;
5
6      extern __shared__ float sdx[];
7
8      int g;
9      if (iy < xc_size)
10     {
11         int s = stateArr[iy],
12             f = frameOffset[s] + frameArr[iy];
13         g = gaussOffset[s] + gaussArr[iy];
14         for (int i = threadIdx.x; i < dim; i += blockDim.x)
15             sdx[dim * threadIdx.y + i] = mu[ld * g + i] - xt[ld * f + i];
16     }
17     __syncthreads();
18     if (ix < dim && iy < xc_size)
19     {
20         float sum = 0;
21         for (int i = 0; i < dim; i++)
22             sum += sdx[dim * threadIdx.y + i] * icov[ld * (ld * g + i) + ix];
23         xc[ld * iy + ix] = sum;
24     }
25 }
```

Výpis kódu 4.15: Kernel pro výpočet hodnot x_c

4.7.3 Výpočet logaritmu hustoty pravděpodobnosti

Po výpočtu pomocných hodnot x_c je lze využít k výpočtu logaritmu hustoty pravděpodobnosti.

Nejdříve je třeba určit hustoty pravděpodobnosti $\gamma_{sg}(t)$ pro všechny složky g stavů s a vstupní vektory příznaků t podle vztahu

$$\gamma_{sg} = \mathrm{gconst}_g + 0,5 \cdot x_c \cdot (x_t - \mu_{sg})$$

`gconst` je předpočítaná hodnota, která odpovídá konstantní části hustoty pravděpodobnosti. Je rovna

$$\text{gconst}_g = -\frac{1}{2}\left[M \ln(2\pi) - \ln|\det C_g|\right]$$

kde M je dimenze vektorů příznaků a C je kovarianční matice normální hustoty pravděpodobnosti pro složku stavu g[9].

Pro každý vektor x_c je výstupem jedna hodnota, proto je výpočet realizován pomocí jednoduché smyčky, ve které se provede skalární násobení vektorů x_c a $x_t - \mu_{sg}$. Kernel se spouští s velikostí bloku 256 a každé vlákno spočte jednu výstupní hodnotu γ_{sg}. Pro správnou indexaci jsou opět použita pole z předchozí kapitoly.

```
1  __global__ void KernelCompGammas(int dim, int xc_size, const int * stateArr, const int *
       frameArr, const int * gaussArr, const int * gaussOffset, const int * frameOffset, float * gammas,
        const float * xc, const float * xt, const float * mu, int ld, const float * gconst)
2  {
3      int iy = blockDim.x * blockIdx.x + threadIdx.x;
4
5      if (iy < xc_size)
6      {
7          int s = stateArr[iy],
8              f = frameArr[iy],
9              g = gaussArr[iy],
10             goffset = gaussOffset[s],
11             foffset = frameOffset[s];
12         float se = 0;
13         for (int i = 0; i < dim; i++)
14             se += xc[dim * iy + i] * (xt[dim * (foffset + f) + i] - mu[ld * (goffset + g) + i]);
15         gammas[iy] = gconst[goffset + g] + 0.5f * se;
16     }
17 }
```

Výpis kódu 4.16: Kernel pro výpočet hodnot γ

Pro sčítání logaritmů platí vztah

$$\log_b(a + c) = \log_b a + \log_b(1 + b^{\log_b c - \log_b a})$$

Po dosazení

$$\begin{aligned} b &= \mathrm{e} \\ p_1 &= \log_b a = \ln a \\ p_2 &= \log_b c = \ln c \end{aligned}$$

lze vztah upravit na

$$\ln(a + c) = p_1 + \ln(1 + \mathrm{e}^{p_2 - p_1})$$

Pokud je $c > a$, tzn. $p_2 > p_1$, pak by se hodnoty p_1 a p_2 na pravé straně rovnice měly zaměnit.

V této metodě adaptace se počítá s logaritmy pravděpodobností a funkce `DevAddLog` slouží k jejich sčítání pomocí předchozího vztahu.

Poté co známe hustoty pravděpodobnosti pro jednotlivé složky stavů můžeme určit hustoty pravděpodobnosti vektorů příznaků, v kódu označených jako pole `lp`. Jedná se o logaritmy pravděpodobnosti, proto pro součet použijeme předchozí funkci. Spouští se opět 256 vláken v bloku, kde každé vlákno spočte logaritmus hustoty pravděpodobnosti pro jeden vektor příznaků. Tento kernel používá další pole pro správnou indexaci dat:

- `frameState` – umožňuje zjistit index stavu z indexu vektoru příznaků
- `xcOffset` – z indexu stavu lze určit, kde začínají jemu odpovídající prvky v poli x_c

```
1  __device__ float DevAddLog(float p1, float p2)
2  {
3      if (p1 > p2)
4          return p1 + logf(1.0f + expf(p2-p1));
5      else
6          return p2 + logf(1.0f + expf(p1-p2));
7  }
8
9  __global__ void KernelCompLP(int numFrames, const int * frameState, const int * numGauss,
       const int * frameOffset, const int * xcOffset, float * lp, const float * gammas)
10 {
11     int fidx = blockDim.x * blockIdx.x + threadIdx.x;
12
13     if (fidx < numFrames)
14     {
15         int s = frameState[fidx],
16             nGauss = numGauss[s],
17             f = fidx - frameOffset[s],
18             gammaidx = xcOffset[s] + nGauss * f;
19         float lp_ = gammas[gammaidx];
20         for (int g = 1; g < nGauss; g++)
21             lp_ = DevAddLog(lp_, gammas[gammaidx + g]);
22         lp[fidx] = lp_;
23     }
24 }
```

Výpis kódu 4.17: Kernel pro výpočet logaritmů pravděpodobnosti

Posledním krokem je normalizace γ_{sg} za pomoci dříve spočtených hodnot hustoty pravděpodobnosti.

```
1  __global__ void KernelNormalizeGammas(int xc_size, const int * stateArr, const int * frameArr,
       const int * frameOffset, float * gammas, const float * lp, float threshold)
2  {
3      int iy = blockDim.x * blockIdx.x + threadIdx.x;
4
5      if (iy < xc_size)
6      {
7          int s = stateArr[iy],
8              f = frameArr[iy],
9              foffset = frameOffset[s];
10         float tmp = gammas[iy] - lp[foffset + f];
11         gammas[iy] = tmp > threshold ? exp(tmp) : 0.0f;
12     }
13 }
```

Výpis kódu 4.18: Kernel pro normalizaci γ

4.7.4 Derivace vektoru b

Pro výpočet derivace vektoru b je použit jeden kernel. Spouští se v bloku o rozměru 256×1. Velikost gridu je 1×36 bloků. Kernel postupně spočte dílčí derivace vektoru b pro všechny vstupní vektory příznaků. Protože je výsledná celková derivace b rovna součtu dílčích derivací pro všechny vektory příznaků a Gaussovské složky odpovídající jednotlivým stavům, lze výpočet realizovat smyčkou přes všechny hodnoty dříve spočteného pole x_c. Kvůli paralelizaci výpočtu je vstupní pole x_c rozděleno na 256 různých částí. Jednotlivá vlákna každého bloku spočtou částečný součet jedné z těchto 256 částí a výsledek uloží do sdílené paměti. Poté první vlákno každého bloku spočte celkovou derivaci b z těchto mezisoučtů a uloží ji do výstupního pole. Jeden blok vláken počítá vždy jeden prvek vektoru b. Dimenze vektoru b je 36, proto je taková i velikost gridu.

```
1  template<typename T>
2  __global__ void KernelUpdateDB(int dim, int xc_size, const int * stateArr, const int * frameArr,
       const int * gaussArr, const int * gaussOffset, const int * numFrames, T * dB, T * ddB, const
       float * gammas, const float * xc, int ld, const float * ivar, float tau)
3  {
4      int idx = blockDim.x * blockIdx.x + threadIdx.x;
5      int idy = blockDim.y * blockIdx.y + threadIdx.y;
6
7      __shared__ float sdB1[2*256];
8      __shared__ float * sddB1;
9      sddB1 = sdB1 + 256;
10     T dB1, ddB1;
11
12     if (idy < dim)
13     {
14         dB1 = 0;
15         ddB1 = 0;
16         for (int i = threadIdx.x; i < xc_size; i += blockDim.x)
17         {
18             int s = stateArr[i],
19                 g = gaussOffset[s] + gaussArr[i];
20             T alpha = 1.0 / (tau + numFrames[s]);
```

```
21              dB1 += alpha * gammas[i] * xc[ld * i + idy];
22              ddB1 += alpha * gammas[i] * ivar[ld * g + idy];
23          }
24          sdB1[threadIdx.x] = dB1;
25          sddB1[threadIdx.x] = ddB1;
26      }
27      __syncthreads();
28      if (threadIdx.x == 0)
29      {
30          dB1 = 0,
31          ddB1 = 0;
32          for (int i = 0; i < blockDim.x; i++)
33          {
34              dB1 += sdB1[i];
35              ddB1 += sddB1[i];
36          }
37          dB[idy] = dB1;
38          ddB[idy]= ddB1;
39      }
40  }
```

Výpis kódu 4.19: Kernel pro výpočet derivace vektoru b

4.7.5 Derivace matice A

Pro výpočet derivace matice A byl nejdříve použit podobný přístup, jako při výpočtu derivace vektoru B. Při tom ale docházelo k mnoha přístupům do globální paměti a vlákna strávila většinu času čekáním na data. Proto bylo nutné navrhnout jiný algoritmus pro výpočet derivace.

Vnější smyčka kernelu neprochází přes jednotlivé prvky x_c jako dříve, ale přes všechny stavy řečového modelu. Na začátku každé iterace si kernel přečte z globální paměti informace o stavu jako počet jemu náležejících mikrosegmentů, počet Gaussovských složek, offset mikrosegmentů v poli x atd. V původním přístupu docházelo ke čtení mnoha redundantních hodnot, k tomu už nedochází, každá hodnota se přečte, když je třeba, ale celý kernel se hůře paralelizuje. Po načtení informací o stavu kernel postupně iteruje přes všechny mikrosegmenty a Gaussovské složky a spočte dílčí derivace matice A.

Kernel se spouští s blokem o velikosti 16×16 vláken. Rozměry gridu v ose x a y jsou dostatečně velké na pokrytí celé matice A (tzn. 3×3) a rozměr v ose z je zvolen 64. Kernel spočte 64 různých částečných derivací a uloží je postupně za sebou do výstupního pole. Výstupem každého bloku je jeden částečný součet spočtený se stavů $z, z+64, z+128, \ldots$, kde z je index bloku v ose z. Hodnota 64 byla experimentálně ověřena jako optimální.

Předchozí kernel (pro výpočet derivace B) nepotřebuje tolik vstupních dat a výstupem je pouze 36 hodnot, proto se v něm neprojeví negativně přístup do globální paměti jako v tomto kernelu.

```
1  template<typename T>
2  __global__ void KernelUpdateDA(int dim, int numStates, const int * numFrames, const int *
       numGauss, const int * frameOffset, const int * gaussOffset, const int * xcOffset, float * dA,
       float * ddA, const float * gammas, const float * x, const float * xc, int ld, const float * ivar,
       float tau)
3  {
4      int ix = blockDim.x * blockIdx.x + threadIdx.x,
5          iy = blockDim.y * blockIdx.y + threadIdx.y;
6
7      if (ix < dim && iy < dim)
8      {
9          T dA1 = 0,
10             ddA1 = 0;
11         for (int s = blockIdx.z; s < numStates; s += gridDim.z)
12         {
13             int nFrames = numFrames[s];
14             int nGauss = numGauss[s];
15             int goffset = gaussOffset[s];
16             int idx = xcOffset[s];
17             int fidx = frameOffset[s];
18             float alpha = 1.0 / (tau + nFrames);
19             for (int f = 0; f < nFrames; f++)
20             {
21                 float x_ = x[ld * fidx + iy];
22                 for (int g = 0; g < nGauss; g++)
23                 {
24                     float gamma = alpha * gammas[idx];
25                     dA1 += gamma * x_ * xc[ld * idx + ix];
26                     ddA1 += gamma * x_ * x_ * ivar[ld * (goffset + g) + ix];
27                     idx++;
28                 }
29                 fidx++;
30             }
31         }
32         dA[ld * (ld * blockIdx.z + iy) + ix] = dA1;
33         ddA[ld * (ld * blockIdx.z + iy) + ix] = ddA1;
34     }
35 }
```

Výpis kódu 4.20: Kernel pro výpočet částečných derivací matice A

Po spočtení 64 dílčích součtů se spustí další kernel, který z nich vypočte výslednou derivaci matice A. Kernel se spouští s blokem 16×16 a gridem 3×3 (dostatečně velký na pokrytí matice A). Každé vlákno spočte hodnotu jednoho prvku derivace matice A odpovídající jeho indexu. Aktualizace matice A a vektoru B podle spočtených derivací probíhá na CPU.

```
1  template<typename T>
2  __global__ void KernelSumA(int dim, int num, T * dA, T * ddA, const float * dAacc, const float *
       ddAacc, int ld)
3  {
4      int ix = blockDim.x * blockIdx.x + threadIdx.x,
5          iy = blockDim.y * blockIdx.y + threadIdx.y;
6
7      if (ix < dim && iy < dim)
8      {
```

```
9         float dA1 = 0,
10            ddA1 = 0;
11        for (int i = 0; i < num; i++)
12        {
13            dA1 += dAacc[ld * (ld * i + iy) + ix];
14            ddA1 += ddAacc[ld * (ld * i + iy) + ix];
15        }
16        dA[ld * iy + ix] = dA1;
17        ddA[ld * iy + ix] = ddA1;
18    }
19 }
```

Výpis kódu 4.21: Kernel pro výpočet konečné derivace matice A

Kapitola 5

Vyhodnocení

5.1 Podmínky experimentu

Jako referenční implementace MFCC a PLP bylo zvoleno HTK. Rychlost zpracování byla také porovnána s programem openSMILE. Metoda TRAPS byla porovnávána s referenčním programem dodaným vedoucím k bakalářské práci. Tento program využívá knihovnu Intel IPP pro velmi rychlé matematické operace na procesoru.

Bylo vytvořeno několik množin testovacích zvukových souborů. Celková délka všech souborů v každé množině je 10 hodin. Vzorkovací frekvence souborů byla zvolena 8 kHz a 44 kHz, odpovídající běžným vzorkovacím frekvencím telefonu a CD. Délka zvukových souborů byla volena náhodně ze 2 rozsahů. Krátké soubory o délce 10-20 sekund a dlouhé soubory o délce 5-10 minut.

Všechny testy byly provedeny na počítači s následující konfigurací:

- CPU: Intel Core 2 Quad Q6600
- Základní deska: Asus P5Q-E
- GPU: NVidia GTX 660
- SSD: Samsung SSD 830 256 GB
- HDD: Seagate Barracuda ST2000DM001

Pro testy zpracování souborů z HDD do HDD a z SSD do SSD byl použit stejný fyzický disk, pouze testy z HDD do SSD a z SSD do HDD používají odlišné fyzické disky pro vstupní a výstupní data. Rychlost SSD byla dále omezena použitím sběrnice SATA II.

Konfigurace MFCC:

- Délka okénka: 20 ms

- Posun okénka: 10 ms
- Počet filtrů: 15 pro 8 kHz, 25 pro 44 kHz
- Počet výstupních kepstrálních koeficientů: 13 včetně nultého
- Velikost delta a akceleračního okna: 3

Konfigurace PLP:

- Délka okénka: 20 ms
- Posun okénka: 10 ms
- Počet filtrů: 17 pro 8 kHz, 27 pro 44 kHz
- Řád celopólového modelu: 12
- Počet výstupních kepstrálních koeficientů: 13 včetně nultého
- Velikost delta a akceleračního okna: 3

Konfigurace TRAPS:

- Délka okénka: 25 ms
- Posun okénka: 10 ms
- Počet filtrů: 15
- Délka časového vektoru: 31
- Počet výstupních DCT koeficientů: 11 včetně nultého

Pro metody MFCC a PLP výstupní vektory příznaků byly normalizovány na nulovou střední hodnotu. Tohle nastavení odpovídá příznaku `_Z` v HTK parametru `TARGETKIND`. Příznaky metody TRAPS byly normalizovány na nulovou střední hodnotu a jednotkovou směrodatnou odchylku.

Program pro parametrizaci metod MFCC, PLP i TRAPS, který je výsledkem této diplomové práce, je open source a je k dispozici zdarma na internetové adrese
https://sourceforge.net/projects/accelfeatextr/

Jako referenční program pro adaptaci řečového modelu byl použit program dodaný vedoucím diplomové práce. Celý algoritmus adaptace je v něm zpracován na procesoru pomocí SSE2 instrukcí. Pro testování byl dodán řečový model s 4922 stavy. Model celkem obsahuje 45 416 normálních hustot pravděpodobnosti. Dále byl dodán soubor s vektory příznaků přiřazenými stavům řečového modelu. Celkem obsahuje 32 291 vektorů, které mají dimenzi 36.

5.2 Výsledky parametrizace

Celková doba běhu programů pro metody MFCC a PLP je zanesena do tabulek 5.1 a 5.2. Sloupce „Afet“ odpovídají výsledkům této diplomové práce. Z tabulek je vidět, že bylo dosaženo významného zrychlení doby zpracování. V porovnání s HTK je zpracování 3-krát až 18-krát rychlejší a openSMILE dopadl ve všech testech nejhůře. Největší zrychlení nastalo pro dlouhé soubory o vzorkovací frekvenci 8 kHz. Důvodem je to, že v případě dlouhých souborů se na grafické kartě zpracovávají velké bloky dat najednou a u krátkých souboru se zpracovává mnoho krátkých bloků dat. Z tabulek je také zřejmé, že pro malé soubory se projevila krátká doba odezvy SSD. U dlouhých souborů není téměř žádný rozdíl, ale v případě krátkých souborů se vzorkovací frekvencí 44 kHz přináší SSD 3-násobné zrychlení. Proto je SSD doporučeno pro zpracování velkého množství kratších souborů.

Celkový výkon se pohybuje mezi 0,00017 a 0,005 RTF (Real time factor), kdy v nejlepším případě program zpracoval 10 hodin dat během 6,1 sekundy.

Výsledky pro metodu TRAPS jsou v tabulce 5.3. Sloupec „IPP“ označuje dobu výpočtu referenčního programu využívajícího knihovnu Intel IPP. Test byl proveden pouze pro soubory o vzorkovací frekvenci 8 kHz, protože referenční program jiné soubory bez úpravy zdrojového kódu nepodporuje. Z tabulky je vidět, že bylo dosaženo zhruba 18-krát kratší doby parametrizace.

Tabulka 5.1: Doba výpočtu metody MFCC v sekundách

	HDD → HDD			SSD → SSD			SSD → HDD			HDD → SSD		
Vstup	HTK	oS	Afet	HTK	oS	Afet	HTK	oS	Afet	HTK	oS	Afet
Krátký 8 kHz	143.2	347.3	37.5	109.2	346.4	28.6	61.1	346.5	33.9	137.1	344.6	29.7
Dlouhý 8 kHz	109.3	282.9	6.1	88.6	281.9	6.0	88.6	283.4	6.5	97.5	283.2	6.1
Krátký 44 kHz	364.6	998.2	172.3	273.8	1002.5	57.1	275.3	997.1	63.1	336.7	1001.0	109.1
Dlouhý 44 kHz	272.2	875.7	46.8	252.6	875.1	28.9	254.2	874.2	28.4	266.6	873.9	37.9

Tabulka 5.2: Doba výpočtu metody PLP v sekundách

	HDD → HDD			SSD → SSD			SSD → HDD			HDD → SSD		
Vstup	HTK	oS	Afet	HTK	oS	Afet	HTK	oS	Afet	HTK	oS	Afet
Krátký 8 kHz	108.2	361.5	36.6	76.1	363.3	29.9	43.5	360.4	34.6	101.6	358.6	30.4
Dlouhý 8 kHz	82.8	293.8	7.1	59.8	299.0	7.3	59.1	289.4	7.3	68.1	289.5	7.0
Krátký 44 kHz	298.8	1008.9	175.7	200.0	1012.7	62.0	202.7	1007.1	70.3	255.5	1008.2	120.3
Dlouhý 44 kHz	201.5	889.6	49.5	183.0	893.1	31.7	183.6	886.9	31.0	198.6	888.0	40.5

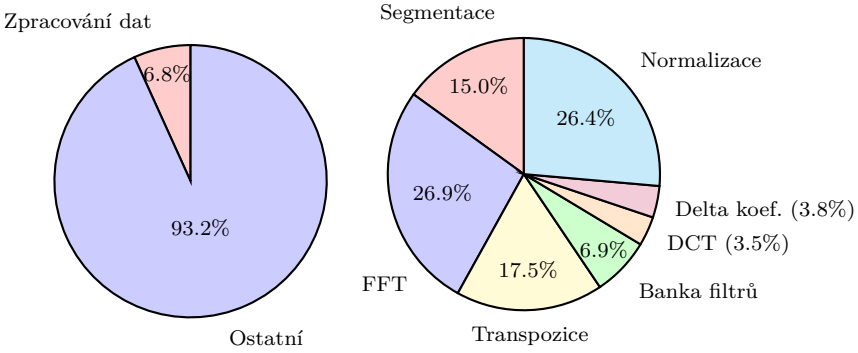
Dále bylo testováno, kolik času zaberou jednotlivé kroky výpočtu. Tento test byl spuštěn na dlouhých souborech o vzorkovací frekvenci 8 kHz a vstupní i výstupní soubory

Tabulka 5.3: Doba výpočtu metody TRAPS v sekundách

	HDD → HDD		SSD → SSD		SSD → HDD		HDD → SSD	
Vstup	IPP	Afet	IPP	Afet	IPP	Afet	IPP	Afet
Krátký 8 kHz	107,2	4,8	107,2	5,1	108,1	6,2	108,0	5,1
Dlouhý 8 kHz	118,9	6,1	118,6	5,3	118,4	6,7	118,4	6,0

se nacházely na stejném HDD. V grafech 5.1a a 5.2a je zobrazeno, kolik času programu zabere zpracování dat a kolik ostatní činnosti, které zahrnují převážně vstupní a výstupní operace a také inicializaci programu včetně předpočítání několika matic atd. V případě MFCC zabere zpracování dat pouze 6,8 % a v případě PLP 9,6 %. Z toho lze usoudit, že většinu času zabere čtení i zápis vstupních a výstupních souborů. Doba strávená v jednotlivých částech metod zpracování řeči je zobrazena v grafech 5.1b a 5.2b.

I přes dlouhou dobu potřebnou pro vstupněvýstupní operace bylo dosaženo významného zrychlení parametrizace řečového signálu.

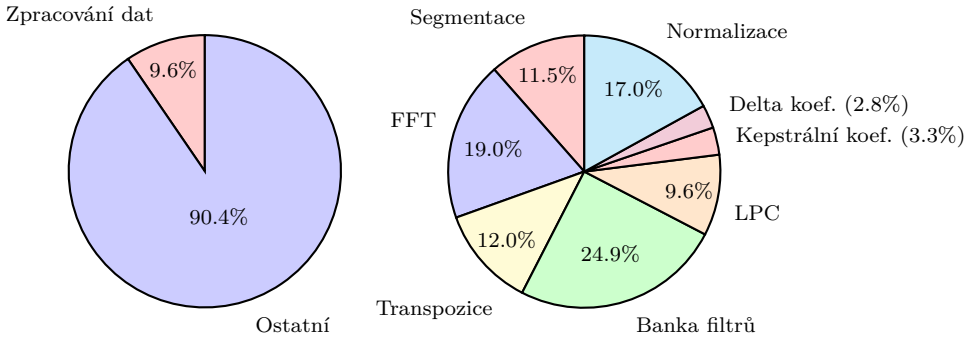


(a) Přehled aplikace (b) Zpracování dat

Obrázek 5.1: Čas strávený v jednotlivých částech programu pro MFCC (Dlouhé soubory, 8kHz, z HDD do HDD)

5.3 Výsledky adaptace

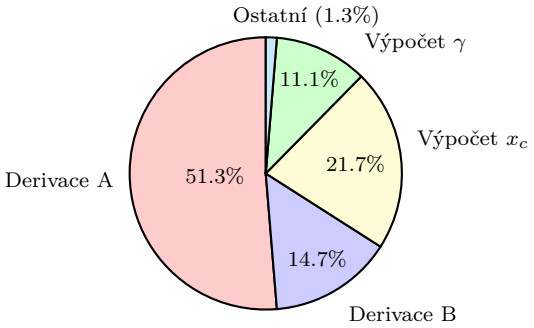
Adaptaci počítá referenční program na procesoru pomocí instrukční sady SSE2 celkem 55,4 sekund, zatímco na grafické kartě výpočet trvá 10,9 sekund. Tzn. na testovacím stroji se podařilo dosáhnout zhruba 5-krát kratší doby výpočtu. Běh všech kernelů během jedné iterace zabere průměrně 181,5 ms. Program adaptaci počítá pro 40 iterací, to odpovídá celkem 7,2 sekundám pro běh všech kernelů. Do celkového času výpočtu není zahrnuto načítání vstupních dat a výstup do souboru, takže zbytek času je nejspíš využit na přesuny dat na kartu a zpět a kroky algoritmu, které probíhají na CPU, jako



(a) Přehled aplikace (b) Zpracování dat

Obrázek 5.2: Čas strávený v jednotlivých částech programu pro PLP (Dlouhé soubory, 8kHz, z HDD do HDD)

např. samotná aktualizace matice A a vektoru b na konci každé iterace. Na obr. 5.3 je znázorněno, jakou část výpočtu tvoří jednotlivé kernely. Výpočet derivace matice A je nejnáročnější část, zabírá více než polovinu času celého výpočtu. Jedním z hlavních důvodů pro rychlý výpočet je to, že program zpracuje všechny data najednou, tzn. během jedné iterace se každý kernel volá pouze jednou. Z toho důvodu je nutné mít na kartě dostatek paměti na všechna data. Použitý řečový model zabere celkem 154 MB v paměti grafické karty a všechny ostatní paměťové struktury zaberou 95,6 MB. Dnešní karty mají alespoň 1 GB paměti, to je více než dostatek.



Obrázek 5.3: Čas strávený v jednotlivých částech programu pro adaptaci

Kapitola 6

Závěr

Tématem práce bylo implementovat výpočet několika metod zpracování řeči na grafické kartě. Implementoval jsem metody parametrizace MFCC, PLP, TRAPS a adaptaci řečového modelu s plnou kovarianční maticí.

Pro parametrizaci jsem vytvořil program, který umožňuje provádět výpočty na platformách CUDA i OpenCL a také na procesoru, pokud např. není k dispozici grafická karta. Vytvořený program provádí parametrizaci několikanásobně rychleji než referenční programy HTK a openSMILE. Oproti HTK byla parametrizace v nejhorším případě 3-krát kratší, v nejlepším případě dokonce 18-krát kratší, openSMILE podával vždy nejhorší výsledky.

Z výsledků plyne, že nejvyššího zrychlení se podařilo dosáhnout u dlouhých souborů. Také bylo zjištěno, že samotná parametrizace zabírá pouhých 6,8 % času celého zpracování pro MFCC a 9,8 % pro PLP. Zbytek času připadá na práci se soubory, přesuny dat na kartu a zpět a ostatní činnosti. Pokud by se podařilo samotné výpočty ještě zkrátit, na celkové době zpracování souboru by se to už moc neprojevilo. K dalšímu zrychlení parametrizace je možné např. využít SSD nebo RAM disků, pro snížení doby odezvy.

Metodu fMLLR pro adaptaci řečového modelu se podařilo na testovacím stroji urychlit 5-krát. Několik částí algoritmu nejde jednoduše paralelizovat a další urychlení by bylo problematické. Díky rychlejší adaptaci je možné také tento ušetřený čas využít k výpočtu dalších iterací a tím zpřesnit výsledek.

Přílohy

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

- Afet/src – zdrojový kód programu Afet
- Afet/bin – spustitelné soubory programu Afet přeložené pro systém Windows
- Adaptace/source – zdrojový kód adaptace řečového modelu
- Adaptace/AdaptGrad – projektové soubory pro MS Visual Studio 2010
- Adaptace/test – přeložený program pro systém Windows s testovacími soubory

Literatura

[1] *Josef Michálek*, **Zrychlení parametrizace akustického signálu pomocí GPU zařízení**
Plzeň, 2012. Bakalářská práce. Západočeská univerzita, Fakulta aplikovaných věd, Katedra kybernetiky

[2] *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

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

[4] *Jan Vaněk and Zbyněk Zajíc*, **A Direct Criterion Minimization based fMLLR via Gradient Descend**
Text, Speech, and Dialogue, Lecture Notes in Computer Science, vol. 8082, p. 52-59, Springer, 2013

[5] **CUDA C Programming Guide** [online], Nvidia, 2012 [cit. 29. srpna 2014]
Dostupné na: <http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf>

[6] **CUDA C Best Practices Guide** [online], Nvidia, 2012 [cit. 29. srpna 2014]
Dostupné na: <http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Best_Practices_Guide.pdf>

[7] **CUFFT Library User Guide** [online], Nvidia, 2012 [cit. 29. srpna 2014]
Dostupné na: <http://developer.download.nvidia.com/compute/DevZone/docs/html/CUDALibraries/doc/CUFFT_Library.pdf>

[8] **CUBLAS Library User Guide** [online], Nvidia, 2012 [cit. 29. srpna 2014]
Dostupné na: <http://developer.download.nvidia.com/compute/DevZone/docs/html/CUDALibraries/doc/CUBLAS_Library.pdf>

[9] *Steve J. Young, D. Kershaw, J. Odell, D. Ollason, V. Valtchev, P. Woodland*, **The HTK Book Version 3.4**
Cambridge University Press, 2006

Příloha A

Uživatelská dokumentace

A.1 Afet

Afet je konzolový program, pro výpočet příznaků MFCC, PLP a TRAPS. Podporuje výpočet na platformách CUDA i OpenCL a také na procesoru. Chování programu se nastavuje pomocí parametrů příkazové řádky a nebo pomocí konfiguračního souboru. Pokud je stejný parametr nastaven v konfiguračním souboru i na příkazové řádce, má přednost příkazová řádka. Pokud se program spustí bez parametrů nebo s parametrem `-h`, vypíše nápovědu a také seznam nalezených výpočetních platforem spolu s jejich ID.

Použití programu:

```
afet.exe [options] ([--wav-file] | --scp) file
```

Parametry programu:

-h [–help]	vypíše nápovědu a ukončí se
–dev arg	specifikuje zařízení, na kterém se má spustit výpočet. Pro platformu CUDA je to jedno číslo, ID zařízení. OpenCL zařízení lze specifikovat více způsoby. Jedno číslo zvolí zařízení s takovým indexem z první platformy. Dvě čísla oddělená dvojtečkou specifikují ID platformy i zařízení. Před dvojtečkou také může být jméno typu platformy, např. "gpu:2"spustí kód na zařízení s indexem 2 typu "gpu".
–ext arg (=htk)	přípona výstupního souboru
–sample-limit arg (=100000)	určuje velikost bloků, v jakých se soubor zpracovává. Velikost je v samplech
–input-dir arg	cesta k adresáři se vstupními soubory

–output-dir arg	cesta k adresáři, kam se uloží výstupní soubory
–wav-file arg	vstupní `*.wav` soubor
–scp arg	textový soubor obsahující seznam souborů ke zpracování
-c [–config] arg	konfigurační soubor
-p [–platform] arg	platforma pro výpočet, možné hodnoty jsou • Auto (defaultní hodnota) • CUDA • OpenCL • CPU

Parametry parametrizace:

-m [–method] arg	metoda parametrizace, možné hodnoty jsou • MFCC (defaultní hodnota) • PLP • TRAPS
–window-size arg (=200)	velikost mikrosegmentů ve vzorcích (je nutné přepočítat z [ms])
–shift arg (=80)	vzájemný posun mikrosegmentů ve vzorcích
–banks arg (=15)	počet filtrů v bance, melovských nebo PLP
–sample-rate arg (=8000)	vzorkovací frekvence vstupních souborů
–low-freq arg (=64)	frekvence prvního filtru v bance
–high-freq arg (=4000)	frekvence posledního filtru v bance
–c0 arg (=true)	true, pokud má být na výstupu nultý koeficient
–norm arg (=2)	použitý druh normalizace: • 0 - žádný • 1 - CMN • 2 - CVN • 3 - podle minima/maxima

`–dyn arg (=0)`	druh použitých dynamických koeficientů: • 0 - žádné • 1 - delta • 2 - delta i akcelerační
`–l1 arg (=3)`	šířka bloku pro výpočet delta koeficientů
`–l2 arg (=3)`	šířka bloku pro výpočet akceleračních koeficientů
`–norm-after-dyn (=1)`	true, pokud se normalizace má provést až po výpočtu dynamických koeficientů

Parametry pouze pro MFCC:

`–ceps arg (=11)`	počet výstupních kepstrálních koeficientů, 0 ruší jejich výpočet, výstupem je pak výstup melovské filtrace
`–lift-coef arg (=22)`	hodnota koeficientu pro liftering

Parametry pouze pro PLP:

`–model-order arg (=8)`	řád požadovaného celopólového modelu

Parametry pouze pro TRAPS:

`–traps-length arg (=31)`	velikost bloku, ze kterého se počítají časové vektory v metodě TRAPS
`–traps-dct arg (=10)`	počet výstupních DCT koeficientů

A.2 Adaptace

Pro adaptaci řečového modelu slouží program pojmenovaný `AdaptGrad.exe`. Při spuštění očekává 4 parametry příkazové řádky.

Příklad spuštění:

```
AdaptGrad.exe image.bin ___adapt_data_full.bin A.txt B.txt
```

Význam jednotlivých parametrů:

1. soubor s řečovým modelem
2. soubor s vektory příznaků
3. výstupní soubor pro matici A
4. výstupní soubor pro vektor b

Program provede adaptaci a do specifikovaných souborů uloží hodnoty nalezené matice A a vektoru b v textové podobě.