

České vysoké učení technické v Praze
Fakulta elektrotechnická



Diplomová práce

Rozpoznávač hlasu na procesoru Cell

Pavel Bazika

Vedoucí práce: Ing. Miroslav Skrbek, Ph.D.

Studijní program: Elektrotechnika a informatika, dobíhající, Magisterský

Obor: Výpočetní technika

leden 2008

Poděkování

Na tomto místě bych rád poděkoval ing. Miroslavu Skrbkovi za jeho úspěšnou snahu zpřístupnit architekturu procesoru Cell mě i ostatním studentům.

Rovněž děkuji ing. Bořivoji Tydlitátovi, zaměstnanci firmy IBM ČR a.s., který mi poskytl mnohé dobré rady, jak psát optimalizovaný kód pro procesor Cell.

Prohlášení

Prohlašuji, že jsem svou diplomovou práci vypracoval samostatně a použil jsem pouze podklady uvedené v příloženém seznamu.

Nemám závažný důvod proti užití tohoto školního díla ve smyslu §60 Zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon).

V Praze dne 18.1. 2007

.....

Abstract

The work contains analysis and implementation of speech recognition frontend on Cell BE architecture. Developed frontend, optimized for one Cell SPU, is 2,5x faster than frontend running on common Pentium 4 processor. 66,5 million samples of input speech signal per second can be processed on one SPU. These results imply, that Cell is suitable for processing large amount of speech data, more than one speaker can produce.

Abstrakt

Práce se zabývá rozborem a implementací frontendu rozpoznávače řeči na procesoru Cell broadband engine. Výsledný optimalizovaný frontend běžící na jednom jádru SPU je až 2,5x rychlejší, než frontend běžící na běžně rozšířeném procesoru Pentium 4. Jedno jádro SPU je schopné zpracovat až 66,5 milionu vzorků vstupního řečového signálu za sekundu. Takový výkon předurčuje Cell spíše ke zpracování velkého množství různých řečových záznamů, než k rozpoznávání řeči jednotlivce.

Obsah

Seznam obrázků	xii
Seznam tabulek	xiii
1 Úvod	1
1.1 Význam strojového rozpoznávání řeči	1
1.2 Cíl práce	1
2 Algoritmy rozpoznávače řeči	2
2.1 Lidská řeč a její strojové rozpoznávání	2
2.2 Struktura lidské řeči	2
2.3 Princip klasického rozpoznávače řeči	2
2.4 Frontend rozpoznávače řeči	3
2.4.1 Zpracování v časové oblasti	3
2.4.2 Zpracování ve frekvenční oblasti	6
2.4.3 Vnitřní struktura frontendu rozpoznávače řeči	9
3 Návrh řešení na procesoru Cell	12
3.1 Programování Cell Broadband Engine	12
3.1.1 Architektura procesoru	12
3.1.2 Specifika programování SPU	13
3.2 Návrh vnitřní reprezentace dat	14
3.3 Návrh algoritmů	15
3.3.1 Výpočet DFT	16
3.3.2 Výpočet reálné FFT prostřednictvím komplexní FFT	17
3.3.3 Výpočet DCT pomocí FFT	21
4 Implementace na procesoru Cell	22
4.1 Filtrování střední hodnoty	22
4.1.1 Statické zpracování	22
4.1.2 Dynamické zpracování	23
4.1.3 Dynamické zpracování spolu s transpozicí	24
4.1.4 Odečtení střední hodnoty	24
4.2 Preemfáze	25
4.3 Váhování rámců okénkem	25
4.3.1 Funkce pro generování okének	25
4.3.2 Funkce pro okénkový výběr	26
4.4 Střední krátkodobá energie	27
4.5 Počet průchodů signálu nulou	28
4.6 Rychlá Fourierova transformace	29
4.6.1 Generování twiddle faktorů	29
4.6.2 Jednoduchá komplexní Fourierova transformace	29
4.6.3 Výpočet reálné FFT pomocí jednoduché komplexní FFT	32
4.6.4 Dvojitá komplexní Fourierova transformace	32
4.6.5 Sudo-lichá dekompozice	33
4.6.6 Výkonové spektrum	34
4.7 Logaritmus	35
4.8 Filtrace trojúhelníkovými filtry	35

4.9	Diskrétní kosinová transformace	36
4.9.1	Výpočet DCT z definice	36
4.9.2	Výpočet DCT prokládaných posloupností	37
4.9.3	Výpočet DCT pomocí FFT	37
4.9.4	Další funkce pro DCT	38
5	Testování	39
5.1	Principy testování	39
5.1.1	Testy správnosti algoritmů	39
5.1.2	Výkonnostní testy	39
5.2	Výsledky testů	39
5.2.1	Vliv rozvíjení cyklů na výkon	39
5.2.2	Porovnání výkonu algoritmů na různých platformách	41
5.2.3	Celkový výkon	46
6	Závěr	52
7	Literatura	53
A	Příložené CD	55
A.1	Obsah příloženého CD	55
A.2	Instalační příručka	55

Seznam obrázků

2.1	Struktura klasického rozpoznávače řeči	3
2.2	Průběh zvuku sykavky, konkrétně písmene S	5
2.3	Průběh zvuku samohlásky, konkrétně A	5
2.4	Průběh zvuku znělé hlásky B	6
2.5	Průběh zvuku neznělé hlásky P	6
2.6	Banka logaritmicky umístěných trojúhelníkových filtrů	8
2.7	Část frontendu provádějící předzpracování	10
2.8	Zpracování a extrakce příznaků z rámce	10
2.9	Detail výpočetního řetězce kepstra	11
3.1	Architektura procesoru Cell, převzato z [1]	12
3.2	Stavy pipeline při načítání skalárních dat pomocí SPU	13
3.3	Zpracovávání dat po jednotlivých rámcích s černě označeným vektorem	15
3.4	Zpracovávání čtyř rámců najednou s černě označeným vektorem	15
3.5	Motýlek - základní operace FFT	17
3.6	Průběh twiddle faktorů, reálná složka modře, imaginární červeně	17
3.7	Odezva FFT na jednotkový skok v reálné oblasti (zdroj [8])	19
3.8	Odezva FFT na jednotkový skok v imaginární oblasti (zdroj [8])	20
3.9	Operace mezi prvky při standardní FFT	20
3.10	Dvojitá FFT, nezávislá data jsou od sebe barevně odlišena	21
4.1	Paralelní redukce sčítání složek vektoru	22
4.2	Hammingovo okénko	26
4.3	Vykonávání funkce <code>civ_win_sel_f4</code>	27
4.4	Dvě komplexní čísla uložená prokládaně v jednom vektoru	29
4.5	Výběr vektorů ze vstupních polí při prvním kroku FFT	30
4.6	Bit-reversing aplikovaný na jednotlivá komplexní čísla	30
4.7	Výběr vektorů ze vstupního pole při druhé iteraci první fáze FFT	31
4.8	Výběr twiddle faktorů pro motýlky šestnáctic	32
4.9	Formát vstupního pole pro dvojitou komplexní FFT	33
4.10	Vektory vybírané pro sudo-lichou dekompozici	34
4.11	Banka logaritmicky umístěných trojúhelníkových filtrů	35
5.1	Vliv rozvinutí cyklů na algoritmus výpočtu střední hodnoty	40
5.2	Vliv rozvinutí cyklů na algoritmus preemfáze	40
5.3	Algoritmus výpočtu střední hodnoty pole	42
5.4	Algoritmus výpočtu krátkodobé energie	43
5.5	Algoritmus přičítání konstanty k prvkům pole	44
5.6	Algoritmus okénkového výběru	45
5.7	Výpočet střední hodnoty dynamicky	46
5.8	Algoritmus pro preemfázi	47
5.9	Diskrétní kosinová transformace s koeficienty počítanými za běhu	48
5.10	Diskrétní kosinová transformace počítaná maticovým násobením	48
5.11	Algoritmus <code>dct_scramble</code> užívaný pro výpočet DCT pomocí FFT	49
5.12	Algoritmus výpočtu logaritmu	49
5.13	Algoritmus výpočtu počtu průchodů signálu nulou	50
5.14	Filtrování trojúhelníkovými filtry, závislost na počtu filtrů	50
5.15	Porovnání výkonu platforem při FFT	51
5.16	Srovnání rychlosti výpočtu kepstra	51

Seznam tabulek

3.1	Tabulka trvání rámců při daných vzorkovacích frekvencích	16
3.2	Tabulka nutných vzorkovacích frekvencí pro rámeček trvající 25 ms	16

1 Úvod

1.1 Význam strojového rozpoznávání řeči

S postupujícím nasazováním informačních systémů do stále většího množství různorodých oblastí se klasické ovládání pomocí tlačítek, případně také pomocí myši, může stát nepohodlným či zcela nepoužitelným. Jednou z takových oblastí může být například použití řidičem ovládaných informačních systémů v autě. Řidič nemá volné ruce a především ani zrak, aby mohl se systémem komunikovat tradičními způsoby. Systémy implementující rozpoznávání řeči proto velice často obsahují i hlasový syntezátor, který slouží pro odpovědi na uživatelské dotazy.

Se vzrůstajícím výkonem výpočetní techniky již rozpoznávání řeči nepředstavuje tolik obtížný problém, jako tomu bylo dříve. Při vývoji kvalitního rozpoznávače řeči nejsme už tolik limitováni vlastní výpočetní technikou, ale spíše našimi omezenými znalostmi. Současným úkolem je tedy spíše navrhnout algoritmus zahrnující dostatečné množství informací pro co nejkvalitnější rozpoznávání, než pokusit se přimět nepříliš výkonnou techniku k realizaci složitých matematických postupů.

Dalším možným využitím současných procesorů je paralelní zpracování, což je pro procesor Cell, který obsahuje devět jader, obzvláště důležité téma. Můžeme se snažit zpracovávat více řečníků v reálném čase a nabídnout jim tak zajímavou možnost interakce. Nebo můžeme zavést jakési hlasové OCR a zpracovávat nahrávky uložené na médiích zabírajících velký prostor. Ať už jde o staré magnetofonové pásky, nebo dnešní terabytová datová úložiště.

1.2 Cíl práce

Cílem této diplomové práce je zvážit možnosti implementace rozpoznávače řeči na procesoru Cell a následně alespoň část z těchto algoritmů implementovat. Tyto algoritmy je poté třeba optimalizovat pro Cell a srovnat s klasickým řešením na jiných platformách. Po podrobnějším průzkumu principu rozpoznávače jsme stanovili, že cílem implementační části bude tzv. frontend rozpoznávače.

Podrobněji o rozpoznávačích a algoritmech v nich používaných pojednává kapitola 2. Kapitola 3 se zabývá specifikami platformy procesoru Cell. Nejprve obecně a pak konkrétně vzhledem k některým složitějším algoritmům v rozpoznávači používaných. Popis výsledné implementace optimalizovaných algoritmů čtenář najde v kapitole 4. V závěrečné kapitole 5 jsou uvedeny výsledky výkonnostních a srovnávacích testů s jinými platformami a rozborů naměřeného chování.

2 Algoritmy rozpoznávače řeči

2.1 Lidská řeč a její strojové rozpoznávání

Komunikace mezi lidmi nespočívá ve výměně slov s jednoznačným významem. Člověk nekomunikuje pouze řečí, ale také pomocí mimiky obličeje, gest rukou, postoje ap. Naopak strojové zpracování řeči vyžaduje pokud možno jednoznačný význam sdělení. Omezíme-li tedy komunikaci se strojem pouze na řeč, je třeba určitých ústupků ze strany člověka. Například v současné době není dost dobře možné zpracovávat delší promluvy, protože stroj nedokáže určit a udržet kontext. Pokud neznáme kontext sdělení, nelze často určit ani jeho význam.

Řeč však obsahuje i nadbytečné informace. Většinou jde o subjektivní vlastnosti určitého řečníka, jako je například výška jeho hlasu, intonace, rytmus řeči a vady řeči. Některé z těchto informací mohou dokonce zhoršit rozpoznávací schopnost stroje. Rozpoznávač tedy musí umět tyto nepodstatné informace nějakým způsobem odfiltrout, aby získal pouze relevantní část sdělení.

2.2 Struktura lidské řeči

Lidská řeč je z pohledu člověka tvořena slabikami, ze kterých jsou složená slova. Výslovnost slabik je do jisté míry individuálně podmíněná, avšak lidský mozek je schopen tyto nuance potlačit a porozumění mateřskému jazyku mu tak nečiní problémy. Dokonce je schopen porozumět i cizinci se značným přízvukem, pokud rozezná jednotlivé složky slova. Rozpoznávač řeči musí být schopen podobného úkolu.

Pro úspěšné rozpoznávání je třeba pracovat s jednotkami menšími než jsou celá slova. Na rozdíl od člověka je stroj schopen mnohem detailnější analýzy krátkých časových úseků. Výhodné je pracovat s tak krátkým úsekem řeči, aby během něj nedocházelo ke změně stavu artikulačního traktu. Optimální délka takového úseku je cca 25 ms, což odpovídá rezonanční frekvenci artikulačního traktu.

Z fonologických výzkumů plyne, že člověk je schopen svými artikulačními orgány modifikovat 12 různých tzv. příznaků, parametrů stavu artikulačního traktu. Tyto příznaky se uplatňují současně a jejich kombinací tak dostáváme tzv. fonémy, které jsou základní stavební jednotkou lidské řeči. Různé světové jazyky používají různý počet a druh fonémů. Například čeština používá 36 fonémů a angličtina 42. Počet fonémů se napříč světovými jazyky pohybuje od 12 do 60 (viz [7]).

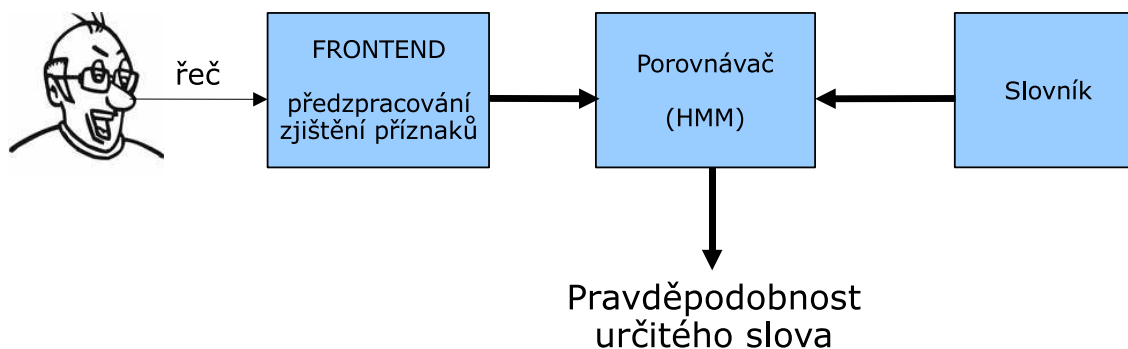
Jednotlivé fonémy se dále spojují do slabik, kterých je v češtině 2500-3500. Jedná se o slabiky z fonetického hlediska, kdy se stejně psaná slabika může vyslovovat různě v závislosti na okolních slabikách. Vyslovováním kombinací slabik dostáváme jednotlivá slova, kterých je v češtině asi 50000.

Kromě víceméně statického stavu artikulačního traktu v rámci fonémů přináší analýza na takto nízké úrovni další výhody – především to, že fonému je na rozdíl od slov málo a jsou platné celosvětově pro všechny jazyky, protože vyplývají z fyziologie člověka.

2.3 Princip klasického rozpoznávače řeči

Rozpoznávače řeči nejsou prozatím uzavřenou kapitolou výzkumu, stále je zde relativně velký prostor pro zlepšování. My se zde budeme věnovat rozpoznávači dnes již klasické struktury. Na obrázku 2.1 můžeme sledovat základní bloky rozpoznávače řeči.

Blokem na levé straně je takzvaný frontend. Vstupem frontendu je řečový signál reprezentovaný vzorky PCM původního signálu akustického. Hlavním úkolem frontendu je odfiltrout ze signálu neužitečné informace a ruchy a naopak získat informace reflektující řeč na vstupu. Získané informace by měly být pokud možno maximálně nezávislé na konkrétním řečníkovi. Jedná se



Obrázek 2.1: Struktura klasického rozpoznávače řeči

tedy opravdu o jakousi extrakci významové informace z řeči. Protože se tímto blokem zabývá zbytek práce, je podrobněji vysvětlen v kapitole 2.4.

Blok na pravé straně reprezentuje slovník slov, případně jejich částí a skupin. Ve slovníku jsou uloženy informace, které poskytuje frontend, případně přizpůsobené pro porovnávač. Na rozdíl od výstupu frontendu jsou však tyto informace trvalé, předem připravené nějakým jiným řečníkem.

Porovnávač je mostem mezi frontendem a slovníkem. Jeho úkolem je najít ve slovníku co nejpodobnější informaci s informací získanou z frontendu. Protože se frontendu nepodaří zcela odfiltrovat veškeré subjektivní vlivy řečníka, je třeba, aby porovnávač neporovnával absolutně, ale umožňoval určitou míru nepřesnosti. Výstupem porovnávače je pak pravděpodobnost, že řečník řekl to které slovo.

Porovnávače byly v minulosti řešeny algoritmem DTW, případně DDTW, česky zvaného borcení časové osy. Jde o porovnávání dvou nestejně dlouhých posloupností, které mělo reflektovat různou rychlost promluvy řečníka a promluvy uložené ve slovníku. Ukázalo se, že takový algoritmus není zcela dostačující, a proto se dnes více používají skryté Markovské modely (HMM). Jedná se o konečné automaty, které mají skryté a pozorovatelné stavy. Skutečný stav automatu je skrytý, ale určitým způsobem ovlivňuje stav pozorovatelný. Přechod mezi dvěma skrytými stavy je dán určitou pravděpodobností při daných vstupech. Při porovnávání dvou řečových vzorků přivedeme na vstupy modelu výstup frontendu a pozorované změny pozorovatelných stavů pak porovnááme se slovníkem.

2.4 Frontend rozpoznávače řeči

Následující odstavce se zabývají algoritmy frontendu rozpoznávače řeči. Po probrání jednotlivých algoritmů je vysvětlena jejich vzájemná návaznost v rámci frontendu.

2.4.1 Zpracování v časové oblasti

2.4.1.1 Stejnosečná složka

Stejnosečná složka je pro rozpoznávání řeči nepotřebná, a proto je vhodné se jí pro další výpočty odfiltrovat. Stejnosečnou složku můžeme buď odfiltrovat zvlášť pro každý rámeček (staticky), nebo použít dynamickou metodu postupného průměrování.

Střední hodnotu z pevně dané oblasti délky N vypočteme dle vzorce:

$$s = \frac{1}{N} \sum_{i=0}^N s[i]$$

Střední hodnotu můžeme též určovat průběžně užitím vztahu pro digitální derivační článek:

$$\bar{s}[n] = \gamma \bar{s}[n-1] + (1-\gamma)s[n], \text{ kde } \gamma \rightarrow 1$$

Filtrování pro každý rámeček zvlášť je nevhodné, pokud se rámečky překrývají - pak počítáme s každým vzorkem několikrát. Nevýhoda dynamické metody je, že je na počátku nepřesná, protože ještě není ustálena.

Dalším krokem po zjištění střední hodnoty je její odečtení. Pokud střední hodnotu počítáme dynamicky, je třeba nějakým způsobem ošetřit počáteční nepřesnost. Možností je nanečisto načíst několik vzorků před vlastním začátkem zpracování, z těchto vzorků spočítat střední hodnotu staticky a následně použít spočtenou hodnotu jako inicializační pro dynamickou metodu.

2.4.1.2 Preemfáze

Vlastností artikulačního traktu člověka je, že vyšší frekvence jsou produkovány s nižší energií. To poněkud ztěžuje zpracování, kdy by vyšší frekvence nemohly dosahovat rovnocenných amplitud s frekvencemi nižšími. Pro zvýšení úspěšnosti rozpoznávače se vyplatí tyto frekvence posílit. To se dá provést FIR filtrem 1. řádu:

$$H(s[n]) = s[n] - \kappa s[n-1] \text{ kde } \kappa \in \langle 0, 9; 1 \rangle$$

2.4.1.3 Řečové rámečky

Nejkratší časový úsek ve vstupním signálu nesoucí nějakou informaci se odvíjí od rezonanční frekvence artikulačního ústrojí člověka. Dle [14] jde o časový úsek dlouhý přibližně 25ms. Takový úsek budeme dále nazývat rámečkem. Vzhledem k zachování kontextu řeči je vhodné, aby se rámečky překrývaly, tedy aby byl jeden úsek řeči součástí několika rámečků. Doporučený překryv je cca 15ms.

Rámečky ze vstupního signálu vybíráme pomocí tzv. okna, které je definováno jako funkce $f(x)$, která splňuje $\forall x \in (-\infty; 0) \cup (l_{ram} - 1; \infty) : f(x) = 0$. Hodnotami funkce v intervalu $\langle 0; l_{ram} - 1 \rangle$ je dán typ okna. Vstupní signál se při výběru oknem násobí hodnotami odpovídajícími této funkci. Existuje hodně různých typů oken (mnoho typů lze nalézt na [13]), dva v rozpoznávacích řeči používané typy jsou okno pravoúhlé, které je definováno jako

$$w_{rect}(x) = \{1, x \in \langle 0; l_{ram} - 1 \rangle \tag{2.1}$$

$$0, jinak \tag{2.2}$$

a okno Hammingovo

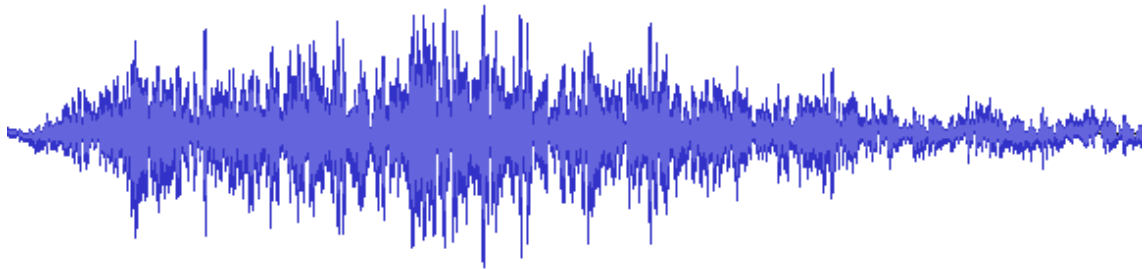
$$w_{rect}(x) = \{0.54 - 0.46 \cos \frac{2\pi n}{l_{ram} - 1}, x \in \langle 0; l_{ram} - 1 \rangle \tag{2.3}$$

$$0, jinak \tag{2.4}$$

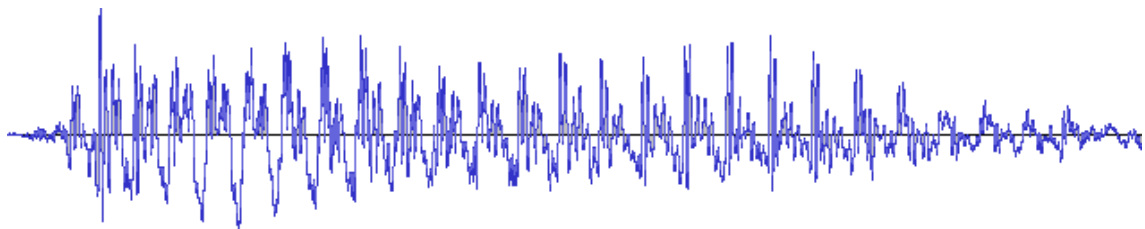
Výběr pravoúhlým oknem tedy spočívá ve zkopírování daného úseku vstupního signálu do rámečky, zatímco při výběru Hammingovým oknem je signál na počátku a konci rámečky utlumen. Hammingovo okno přináší výhodu, pokud rámeček následně převádíme z časové do frekvenční oblasti prostřednictvím Fourierovy transformace - v jeho spektru se neobjeví šum vzniklý vyříznutím rámečky z původního souvislého signálu. Výběr Hammingovým oknem představuje však krok navíc, protože každý vzorek je ještě třeba vynásobit funkcí Hammingova okna.

2.4.1.4 Počet průchodů nulou

Na základě počtu průchodů signálu v rámci nulou lze určit přibližnou frekvenci tónu v rámci. Tím je možné odlišit jednak sykavky (obr. 2.2) od ostatních hlásek (např. obr. 2.3), protože mají vysokou frekvenci. Dále tento parametr indikuje, zda jde o znělou či neznělou hlásku. Neznělé hlásky se více podobají šumu a proto vykazují vyšší počet průchodů signálu nulou. Pro určení tohoto parametru je třeba vyhodnocovat změny znaménka dvou po sobě následujících vzorků v rámci. Nevýhodou této charakteristiky je, že počet průchodů signálu nulou je značně náchylný na šum, je tedy problém odlišit pomlku v řeči od neznělých hlásek a sykavek. Dále je tento algoritmus závislý na odfiltrování stejnosměrné složky. Jako pomocný parametr je však užitečný.



Obrázek 2.2: Průběh zvuku sykavky, konkrétně písmene S



Obrázek 2.3: Průběh zvuku samohlásky, konkrétně A

Je výhodné počet průchodů nulou vyhodnotit po určitých úsecích ještě před rozdělením do rámců. Délku úseků volíme soudělnou s délkou rámců, každý rámeček pak bude obsahovat určitý počet celých úseků. Celkový počet průchodů signálu nulou v rámci můžeme spočítat jako součet průchodů signálu nulou v obsažených úsecích.

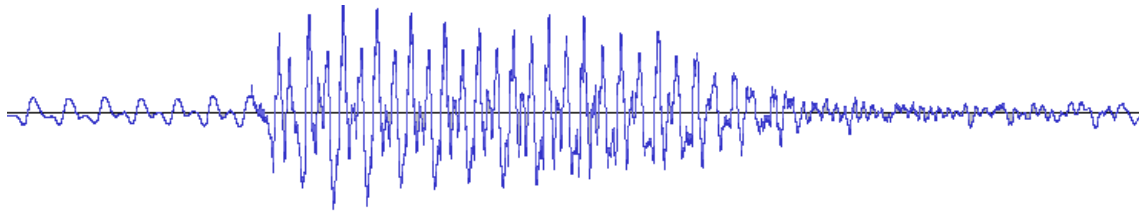
2.4.1.5 Střední krátkodobá energie

Střední krátkodobá energie slouží k podobným účelům jako počet průchodů signálu nulou - tedy rozlišení znělé (obr. 2.4) a neznělé (obr. 2.5) hlásky. Principem je, že znělé hlásky vykazují vyšší energii, protože se jejich vyslovování účastní hlasivky, působící v artikulačním traktu coby buzení. Neznělé hlásky jsou vyslovovány bez účasti hlasivek, pouze pomocí artikulace a proto jsou více podobné náhodnému šumu. Dále je pomocí tohoto parametru možno realizovat jednoduchou detekci řečové aktivity.

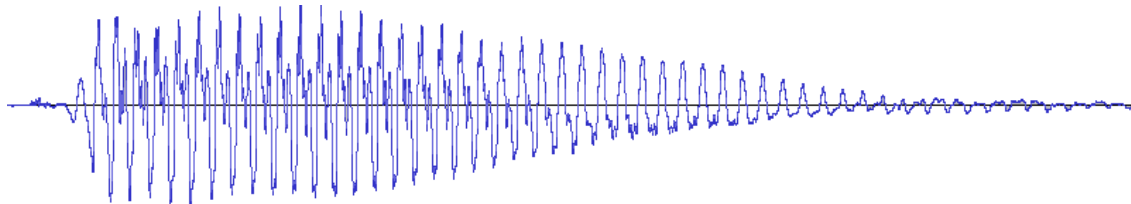
Střední krátkodobá energie je definovaná jako

$$E = \frac{1}{l_{ram}} \sum_{n=0}^{l_{ram}-1} (s[n])^2$$

tedy součet druhých mocnin vzorků v jednom rámci.



Obrázek 2.4: Průběh zvuku znělé hlásky B



Obrázek 2.5: Průběh zvuku neznělé hlásky P

2.4.1.6 Derivace vstupního signálu

Užitečnou informací pro rozpoznávač je také první, druhá a třetí derivace vstupního signálu. První derivace je někdy nazývána delta koeficienty a druhá akcelerační koeficienty. Derivace můžeme počítat podle jednoduchého vztahu

$$d_t = \frac{c_{t+W} - c_{t-W}}{2W}$$

kde W je šířka tzv. delta okna, která závisí na uživateli. Druhou (respektive třetí) derivaci obdržíme jednoduše dosazením delta (respektive akceleračních) koeficientů (tedy derivace první) do předešlého vztahu.

Delta koeficienty lze také počítat složitějším vztahem

$$d_t = \frac{\sum_{w=1}^W w(c_{t+w} - c_{t-w})}{2 \sum_{w=1}^W w^2}$$

postup pro výpočet dalších derivací je obdobný jako u jednoduchého případu. Výběr použitého vztahu závisí na vstupním řečovém signálu.

Algoritmy derivací řečového signálu jsme neimplementovali, proto se o nich nadále nebudeme zmiňovat.

2.4.2 Zpracování ve frekvenční oblasti

Operace v časové oblasti slouží spíše pro předzpracování signálu. Příznaky této oblasti získané poskytují pouze základní obrázek o vstupním signálu. Abychom ze signálu odfiltrovali informace závislé na řečníkovi (především výšku jeho hlasu) a získali pokud možno co nejčistší obsaženou informaci, musíme signál převést do frekvenční oblasti a zde ho dále zpracovávat.

2.4.2.1 Převod do frekvenční oblasti

Převod do frekvenční oblasti provádíme vzhledem k povaze vstupu pomocí Diskrétní Fourierovy transformace (DFT), která je definována vztahem

$$X(k) = \sum_{n=0}^{N-1} x(n)e^{-j2\pi\frac{nk}{N}}, \text{ pro } K \in \langle 0, N-1 \rangle$$

DFT je definovaná jako operace s komplexními čísly. To pro nás není nejvhodnější, protože transformujeme vzorky signálu, tedy čísla reálná. Můžeme samozřejmě za imaginární složky doplnit nuly, to však není efektivní. Existují však postupy, zmíněné v kapitole 4.6.3, které umožňují využít komplexní DFT pro transformaci reálných posloupností efektivně.

Algoritmus přímo vyplývající z definice by měl časovou složitost $O(n^2)$. Pomocí techniky rozděl a panuj lze dosáhnout časové složitosti $O(n \log n)$. Algoritmy využívající tuto techniku souhrnně nazýváme FFT algoritmy.

2.4.2.2 Spektrální výkonová hustota

Užívá se při zpracování náhodných signálů. Vypočte se jako druhá mocnina amplitudového spektra, tedy

$$G(\omega) = |F(\omega)|^2$$

2.4.2.3 Oddělení buzení od modifikace

Řečový signál se skládá ze dvou základních složek - z buzení a modifikace. Buzení je dáno základním kmitočtem hlasivek, zatímco modifikaci ovlivňuje artikulační trakt. Pro úspěšné rozpoznávání řeči je vhodné odfiltrovat složku buzení, protože samotná výška hlasu nemá na informační hodnotu slova vliv. V časové oblasti je řečový signál dán konvolucí těchto složek

$$s(t) = g(t) * h(t) = \int_{-\infty}^{\infty} g(\tau)h(t - \tau)d\tau$$

kde $g(t)$ představuje složku buzení a $h(t)$ složku modifikace. Převodem do frekvenční oblasti získáváme z konvoluce součin (díky vlastnostem DFT)

$$S(f) = G(f)H(f)$$

Složky buzení jsou bohužel rozesety po celém spektru – lidský hlas není jednoduchý sinus, ale skládám se z mnoha harmonických udávajících právě barvu hlasu. Nemůžeme je tedy jednoduše odfiltrovat. Pro oddělení využíváme kepstrální (kepstrum = spektrum spektra) koeficienty $c(n)$, které převádějí tento součin na součet. Kepstrum je definováno jako

$$\ln G(f) = \sum_{n=-\infty}^{\infty} c(n)e^{-j2\pi fn}$$

Suma v rovnici je vlastně definice DFT, můžeme tedy kepstrální koeficienty $c(n)$ vyjádřit jako

$$c(n) = F^{-1}(\ln G(f))$$

$G(f)$ odpovídá spektrální hustotě výkonu, kterou lze získat z DFT umocněním na druhou. Po dosazení máme

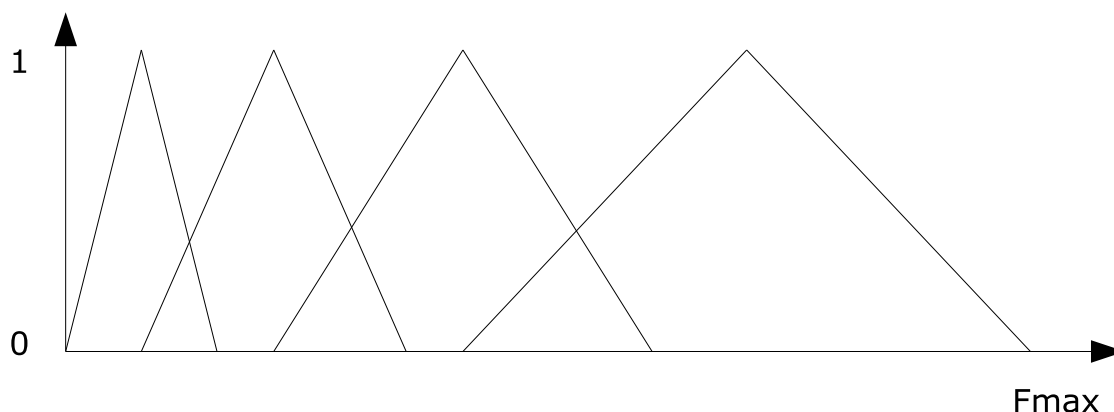
$$c(n) = F^{-1}(\ln |F(s[n])|^2)$$

Na kvěfrenční ose (ose kepstrálních koeficientů) je buzení umístěno na nižších hodnotách, zatímco modifikace na vyšších. Můžeme tedy jednoduchým hornopropustným filtrem (případně nepoužitím některých koeficientů s nižšími indexy) odstranit složku buzení.

2.4.2.4 Zohlednění frekvenční charakteristiky lidského ucha

DFT nezohledňuje fakt, že lidské ucho má na nižších frekvencích vyšší rozlišení než na frekvencích vyšších. Protože přizpůsobení rozpoznávače vlastnostem lidských orgánů zvyšuje úspěšnost rozpoznávání, je vhodné nějak zvýšit rozlišení na nižších frekvencích. To lze provést aplikací logaritmicky rozmístěných integračních filtrů. Každý filtr zabírá určitou část spektra a nasčítává hodnoty spektrálních čar, které pod něj spadají.

Filtry mohou mít různý tvar, nejčastěji jsou používány filtry trojúhelníkové, které se se zvyšující se frekvencí rozšiřují. Tvar filtrů udává součinitel, kterým jsou násobeny sčítané spektrální čáry. Spektrální čáry na krajích trojúhelníku jsou násobeny malým reálným číslem, takže se neuplatní tolik, jako spektrální čáry pod vrcholem trojúhelníku. Příklad trojúhelníkových filtrů uvádí obrázek 2.6



Obrázek 2.6: Banka logaritmicky umístěných trojúhelníkových filtrů

Filtry jiných tvarů se používají v rozpoznávacích pro speciální aplikace, viz např. kapitola 2.4.2.8.

Pro nelineární rozmístění filtrů v souladu s logaritmickým charakterem lidského ucha můžeme využít konverzi frekvenční osy z Hertzů na Mely (viz [14]) podle vztahu

$$F_{Mel} = 2959 \log_{10} \left(1 + \frac{F_{Hz}}{700} \right)$$

Tím získáme logaritmickou osu, nad kterou následně zkonstruujeme banku trojúhelníkových filtrů, ze kterých získáváme energii jednotlivých frekvenčních pásem namísto DFT. Protože je osa logaritmická, budou filtry na nižších frekvencích početnější a užší, což odpovídá vyšší citlivosti lidského ucha v této oblasti.

2.4.2.5 Mel-frekvenční keprální koeficienty

Úvahy z kapitol 2.4.2.3 a 2.4.2.4 můžeme spojit v jeden algoritmus, tedy postup, kdy určíme keprální koeficienty ze signálu přizpůsobenému charakteristikám lidského ucha s vyšším rozlišením v hloubkách. Nejprve potřebujeme získat amplitudy jednotlivých frekvencí po uplatnění trojúhelníkových filtrů. Na výsledné amplitudy uplatníme diskretní cosinovou transformaci (DCT), která zde zastupuje inverzní DFT [15] dle vztahu:

$$c_i = \sum_{j=0}^{N-1} m_j \cos\left(\frac{\pi i}{N} \left(j + \frac{1}{2}\right)\right)$$

Jedná se tedy o standardní DCT-II, často zvanou pouze DCT.

2.4.2.6 Relativizace vstupního signálu

V případě dlouhé promluvy mohou absolutní hodnoty signálu, odstup signálu od šumu apod. značně kolísat, což lidskému uchu nevádí, protože pracuje s relativními hodnotami, nikoliv s absolutními. Na hodnoty vypočtených kepstrálních koeficientů však absolutní hodnota signálu vliv má. Proto je vhodné před DCT zařadit relativní spektrální filtr (RASTA). RASTA spočívá ve třech krocích:

1. Spektrální složky jsou zkomprimovány nelineárním kompresním filtrem (např. logaritmem)
2. Na každou zkomprimovanou složku je uplatněn vyhlazovací filtr.
3. Na upravené složky je aplikován expanzní filtr, inverzní ke kompresnímu.

Definice kompresních a expanzních filtrů je možné najít v literatuře [4]. Aby byla relativizace účinná, je zapotřebí relativně složitých filtrů. Protože jsme tyto filtry neimplementovali, nebudeme se zde dále relativizací zabývat.

2.4.2.7 Normování délky artikulačního traktu

Z důvodů různých délek artikulačních traktů u různých lidí může být výhodné přizpůsobit měřítko Melovské frekvenční osy před aplikací trojúhelníkových filtrů. Jak plyne z [5], postačí lineární změna s koeficientem $\alpha \in < 0, 8; 1, 2 >$. Hodnotu parametru α je třeba stanovit experimentálně pro každého řečníka. Změny tohoto parametru provádíme automaticky zpětnovazebně podle úspěšnosti rozpoznávání.

2.4.2.8 Lombardův efekt

Lombardův efekt nastává, pokud se mluvčí nachází v hlučném prostředí, kde je šum tak silný, že ho lze stěží překonat. Mluvčí se snaží, aby byla řeč výkonově efektivnější, čímž značně změní charakteristiky své řeči. U mužů je tato změna charakteristik odlišná od žen. Úspěšnost rozpoznávání v případě Lombardova efektu může poklesnout až o 70%.

Tomuto poklesu je možné zabránit a přitom v podstatě zachovat vnitřní strukturu frontendu vhodnou modifikací uvedených algoritmů. První takovou změnou je dle [6] použití banky obdélníkových filtrů namísto banky filtrů trojúhelníkových (viz 2.4.2.4). Další účinnou změnou je použití LPC (linear predictive coding, viz [14]) místo DCT v konečném stupni výpočtu Mel-frekvenčních kepstrálních koeficientů.

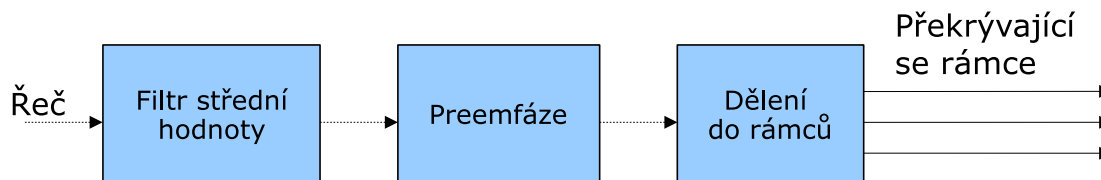
Úspěšnost upraveného rozpoznávače při Lombardově efektu je pak asi o 20% horší, než úspěšnost běžného rozpoznávače v tichém prostředí. Jde o velké zlepšení, které má však i své stinné stránky. Zhoršuje se totiž úspěšnost rozpoznávání řeči bez Lombardova efektu (v rádech jednotek procent). Proto je vhodné umět zařazovat tyto změny do předzpracovávacího řetězce dynamicky.

Algoritmus LPC není součástí implementované části. Lze jej však případně snadno zařadit na místo bloku DCT.

2.4.3 Vnitřní struktura frontendu rozpoznávače řeči

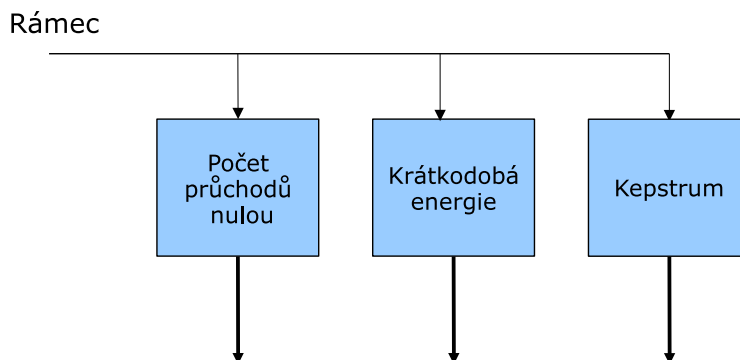
Algoritmy popsané v předcházejících odstavcích je třeba uspořádat do zpracovávajícího řetězce frontendu rozpoznávače řeči tak, aby efektivně plnily svůj účel. Kvůli překryvu dat v rámcích, do kterých je signál dělen, je vhodné provést nejprve operace, které dělení do rámců nevyžadují.

Jde o střední hodnotu a preemfázi. Pokud bychom zpracovávali jednotlivé rámce, aplikovali bychom tyto algoritmy několikrát na stejná data se stejným výsledkem, což je zbytečné plýtvání výpočetním výkonem. Frontend tedy začíná svoji práci filtrováním střední hodnoty (viz obr. 2.7). Toto filtrování je možné provádět staticky či dynamicky, to záleží na volbě implementátora. Následuje preemfáze, která posílí vyšší frekvence v audio signálu. Po preemfázi provádíme dělení do rámců dlouhých cca 25 ms s překryvem cca 15 ms.



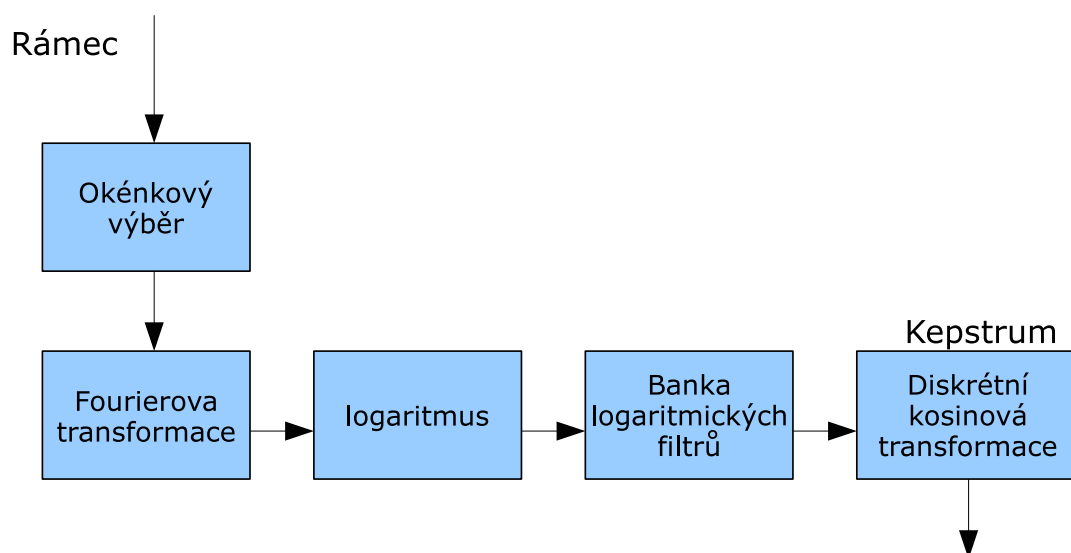
Obrázek 2.7: Část frontendu provádějící předzpracování

Po rozdělení signálu do jednotlivých rámců zpracováváme již samostatně tyto rámce (viz obr. 2.8). Cílem je extrahovat z rámců koeficienty reprezentující foném v daném rámci. K této extrakci slouží celkem tři algoritmy - vyhodnocení počtu průchodů signálu nulou, krátkodobá energie a kepstrum.



Obrázek 2.8: Zpracování a extrakce příznaků z rámce

Princip výpočtu kepstra je naznačen na obrázku 2.9. Nejprve provedeme okénkový výběr, čímž zpřesníme výsledky Fourierovy transformace, která následuje. Použité okénko je typicky Hammingovo. Protože se rámce překrývají, nepřicházíme o podstatné informace, které utlumíme na okrajích rámce – jsou totiž obsaženy v sousedních rámcích. Po Fourierově transformaci následuje logaritmus v souladu s kapitolou 2.4.2.3 o oddělování složky buzení a modifikace signálu. Po zlogaritmování provedeme filtraci bankou trojúhelníkových logaritmicky rozmístěných integračních filtrů, čímž reflektujeme vyšší citlivost lidského ucha na nižší frekvence. Závěrečnou operací je diskretní kosinová transformace, v případě eliminování Lombardova efektu nahrazená transformací LPC (viz kapitola 2.4.2.8).



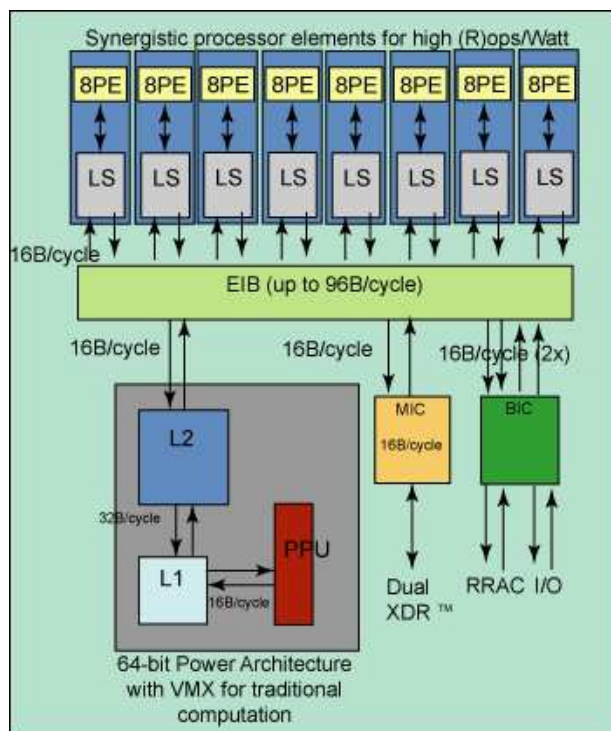
Obrázek 2.9: Detail výpočetního řetězce kepra

3 Návrh řešení na procesoru Cell

3.1 Programování Cell Broadband Engine

3.1.1 Architektura procesoru

Processor Cell Broadband Engine (dále jen Cell) vyvinuly firmy IBM a Sony v letech 2000 - 2005. Procesor sestává z devíti procesorových jader, z čehož jedno jádro je tzv. jednotka PPU (PowerPC processor unit) a ostatní jádra SPU (Synergic processor unit). Cell je také možno dodávat s omezeným počtem jader SPU. Celý procesor je taktován na 3,2 GHz. Schema architektury procesoru Cell je na obrázku 3.1



Obrázek 3.1: Architektura procesoru Cell, převzato z [1]

Narozdíl od jednotky PPU nemají jednotky SPU přímý přístup k hlavní (externí) paměti systému. Programový kód, který vykonávají, i data, nad kterými pracují, je uložen v tzv. Local store (LS), 256 kB velké paměti, která je vlastní každému SPU. Přenosy mezi hlavní pamětí a LS jsou realizovány prostřednictvím DMA. Přenos dat tedy řídí řadič DMA, zatímco jednotky mohou provádět výpočet nad daty, která se už přenesla. Díky tomu je možné využívat princip double buffering, kdy jednotka vždy provádí výpočet nad jedním ze dvou bufferů, zatímco druhý buffer je přes DMA plněn novými daty pro další zpracování. Po dokončení výpočtu a přenosu se úloha bufferů vymění.

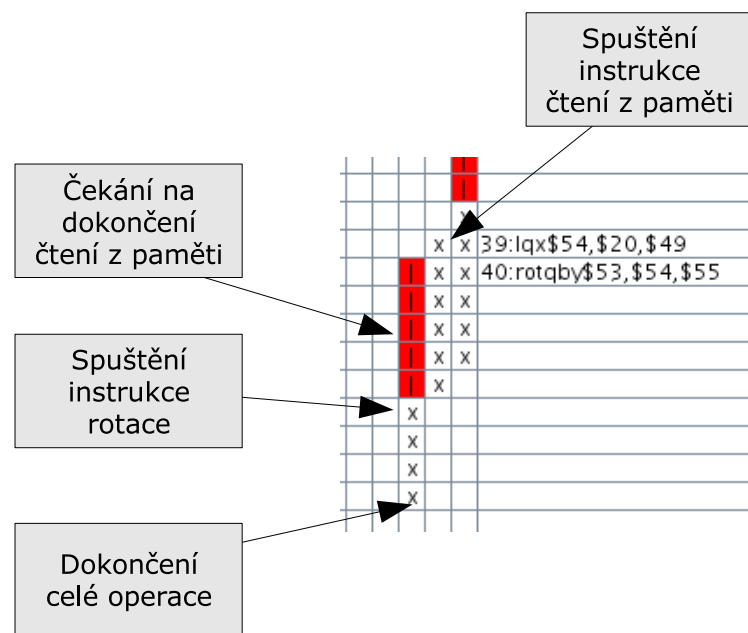
Jádro PPU je plnohodnotným superskalárním PowerPC procesorem včetně sady vektorových instrukcí AltiVec. Je vhodné jej využít pro řízení výpočtu (rozdělování úloh mezi SPU), běh operačního systému (jde o standardní PowerPC) či pro výpočty, které lze jen těžko vektorizovat (paralel na úrovni instrukcí) a na SPU by, díky jeho vlastnostem, nebyly efektivní.

Naopak jednotky SPU mají čistě vektorovou architekturu. Délka vektoru je 128 bitů, tedy 16 bytů. Jednotka SPU má k dispozici 128 šestnáctibytových registrů, které používá jak pro celočíselné operace, tak pro operace s plovoucí desetinnou čárkou. Registr tedy může obsahovat vektory různých datových typů, a to 16 nezávislých celých čísel o šířce jednoho bytu až po 4

čtyřbytové celočíselné hodnoty. Pro operace s desetinnými čísly jsou k dispozici datové typy o čtyřech číslech typu float či dvou číslech typu double. Kódování těchto čísel je v souladu s normou IEEE 754, viz [11]. Veškeré matematické operace probíhají nad celým vektorem. Výpočty se samostatnými skalárními čísly se provádějí nepřímo pomocí vektoru s jedinou platnou položkou. Přístup do paměti také podléhá vektorové architektuře jádra. Nejmenší objem dat, který lze mezi registrem a pamětí přenášet je 16 bytů (tedy délka registru). Data navíc musí být zarovnána na adresu dělitelnou šestnácti.

3.1.2 Specifika programování SPU

Při programování jednotek SPU je třeba zohlednit některé speciální vlastnosti této architektury, které je odlišují od ostatních procesorů. Především jde o vektorovou povahu procesoru. Nemá význam provádět na SPU skalární výpočty, k tomu slouží PPU. Prováděním skalárních algoritmů na SPU ztrácíme výkon především při operacích s pamětí – SPU umí načíst z LS pouze celý vektor, který následně musí transformovat na skalární číslo pomocí instrukce rotace. Skalární čísla jsou reprezentována jako vektory s hodnotou na pozici nula. Činnost SPU při takovéto operaci je vyobrazena na obrázku 3.2. Celé čtení trvalo 10 taktů. Přitom při správné optimalizaci a vytíženosti pipeline může zabrat prakticky jen jeden takt.



Obrázek 3.2: Stavy pipeline při načítání skalárních dat pomocí SPU

Operace SPU jsou vykonávány pomocí dvou pipeline. Každá pipeline může mít rozpracováno několik instrukcí. Pokud však mezi instrukcemi existují datové závislosti (některá instrukce potřebuje výsledky instrukce předchozí) musí pipeline nejprve dokončit instrukci, na které ostatní závisí a až pak může pokračovat vykonáváním instrukcí ostatních. Tento jev je samozřejmě nežádoucí, není při něm plně využita výpočetní kapacita SPU. Pipeline SPU rozlišujeme jako sudou a lichou pipeline, což souvisí s adresami instrukcí v paměti, které ta která pipeline vykonává. Každá pipeline je však dedikována jen pro určité operace. Lichá pipeline provádí instrukce pro práci s pamětí, skoky, rotace, posuvy a instrukce pro přeuspořádávání bytů v rámci registrů (instrukce `shuffle`), zatímco sudá pipeline provádí logické a aritmetické operace – jak celočíselné, tak s čísly v plovoucí řádové čáře.

Protože vykonání instrukce trvá určitý počet taktů a SPU může díky pipeline zpracovávat

několik instrukcí současně, je třeba udržovat dostatečnou míru datové nezávislosti mezi po sobě následujícími instrukcemi. Tedy pokud mezi dvěma operacemi existují datové závislosti, pak je vhodné mezi tyto dvě závislé operace vložit operaci další, nezávislou. V případě jednodušších algoritmů může být problém takovou nezávislou operaci najít.

Experimenty ukazují, že výkon SPU značně trpí krátkými cykly. Je to způsobeno především tím, že krátké cykly neumožní dostatečně využít hlubokou pipeline SPU. Cykly v algoritmech je pro zvýšení efektivity třeba nějakým způsobem prodloužit, aby docházelo k menšímu počtu vyhodnocování podmínky a skoků. Jednoduchým principem je rozvíjení cyklů (loop unrolling), kdy tělo cyklu vykonáme několikrát za sebou, než testujeme podmínku. V programu tedy umístíme tělo cyklu několikrát pod sebe.

Rozvíjení cyklů je možné dále využít pro zvýšení datové nezávislosti po sobě následujících instrukcí. V typickém případě, kdy tělo cyklu tvoří operace načtení hodnoty z paměti, výpočet s touto hodnotou a uložení výsledku, má rozvíjení cyklů vliv pouze na méně časté vyhodnocování podmínky a skok. Díky rozvinutí cyklu však lze pracovat s více sadami registrů, typicky se dvěma. Zatímco nad jednou sadou provádíme výpočet, druhou sadu plníme daty z paměti. Díky dvěma pipeline SPU je možné tyto operace provádět zároveň. Po dokončení začneme provádět výpočet nad druhou sadou, zatímco první sada přechází na práci s pamětí. Sady registrů střídáme v kruhové frontě, pokud využíváme pouze dvě, jedná se o jakýsi registrový double-buffering.

Konkrétní realizace může vypadat například takto: Při startu algoritmu načteme z paměti parametry potřebné pro první dvě iterace cyklu. V rámci cyklu spustíme požadovanou aritmetickou operaci nad první skupinou parametrů a ihned poté začneme načítat třetí sadu parametrů. Poté naplánujeme uložení výsledku, které optimalizující překladač umístí na vhodné místo po dokončení výpočtu. Pokračujeme zpracováním druhé sady parametrů a načítáním čtvrté. Třetí sada je již v tuto chvíli připravena pro další iteraci.

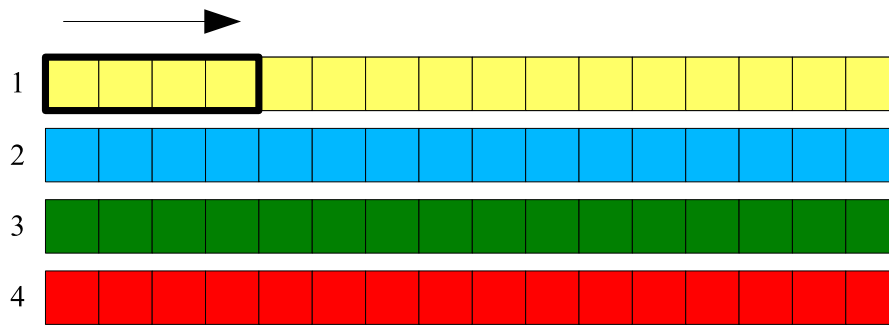
Další informace o programování SPU lze nalézt v literatuře [1].

3.2 Návrh vnitřní reprezentace dat

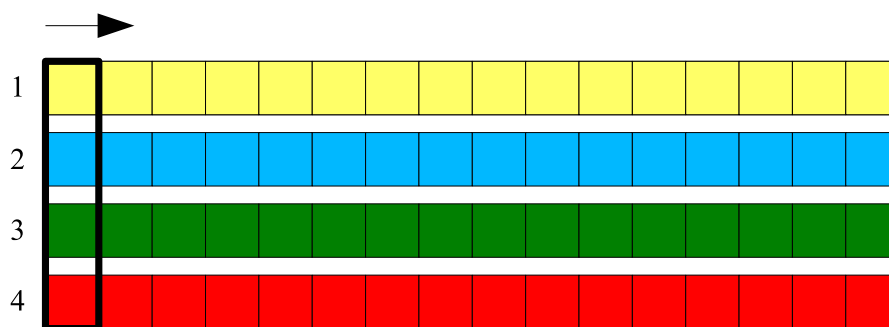
Až na malé výjimky disponuje SPU pouze takzvanými SIMD instrukcemi. SIMD instrukce vykonává určitou operaci nad větším počtem datových prvků najednou. Například existuje SIMD instrukce, která provede součet dvou čtveřic čísel typu `float`. Taková čtyři čísla nazýváme vektorem. Instrukce, která by provedla součet dvou skalárních čísel na SPU není k dispozici. Mají-li algoritmy plně využít možností SPU, je třeba navrhnout vnitřní reprezentaci zpracovávaných dat vhodnou pro tyto instrukce. K datům můžeme přistupovat v zásadě dvojím způsobem. První možností je zpracovávat čtyři položky běžného pole s daty (viz obr. 3.3), druhým způsobem pak zpracovávat vždy jednu položku čtyř nezávislých polí (viz obr. 3.4).

Výhodou prvního způsobu je především návaznost datových struktur na referenční verze algoritmů. Celý systém můžeme nejprve implementovat skalárně a jednotlivé funkce pak SIMDizovat. Další výhodou je, že jde o přirozený formát vstupních dat – digitální vzorky signálu tvoří souvislé posloupnosti. Nevýhody, které tento způsob zpracování přináší se projeví, pokud je třeba pracovat s jednotkou menší, než je jeden vektor. Pak je třeba vektory transformovat pomocí `shuffle` instrukcí apod. Mezi algoritmy, kterých se tento problém týká patří dynamický výpočet střední hodnoty (viz kapitola 2.4.1.1), preemfáze (kapitola 2.4.1.2) či sudo-lichá dekompozice.

Druhý způsob naopak pracuje se čtyřmi zcela nezávislými čísly typu `float`. Pokud existují nějaké blízké závislosti mezi skaláry v jednom poli, vyskytují se zde v podobě závislosti mezi jednotlivými vektory. Není proto třeba aplikovat na vektory instrukci `shuffle` a získávat z nich jednotlivá čísla typu `float`. Oproti předchozímu přístupu zde však musíme z paměti načíst více vektorů, zatímco v předešlém případě jsme obdrželi čtyři čísla najednou. Tato nevýhoda je však



Obrázek 3.3: Zpracovávání dat po jednotlivých rámcích s černě označeným vektorem



Obrázek 3.4: Zpracovávání čtyř rámců najednou s černě označeným vektorem

snáze řešitelná prostřednictvím použití většího počtu registrů, zatímco problém v předešlém algoritmu není prakticky řešitelný vůbec – je dán povahou algoritmu.

Další nevýhodou druhého způsobu je nutnost transformovat data na vstupu. V rámci vektoru jsou totiž uloženy vzorky ze čtyř různých posloupností. Tato transformace spočívá pouze v operacích čtení z paměti, ukládání do paměti a operace `shuffle`. Všechny tyto operace jsou vykonávány lichou pipeline, zatímco sudá pipeline je zcela nevytížená. Je tedy vhodné kromě samostatné funkce implementovat také funkci provádějící dvě operace – transformaci a ještě jednu další, aritmetickou. Ideálním kandidátem je dynamický výpočet střední hodnoty, která je umístěná na počátku zpracovacího řetězce a kde v sudé pipeline existují datové závislosti. Přidání kódu pro transformaci samostatných vstupních posloupností na prokládané tedy tuto funkci příliš nezpomalí.

Na základě úvah a experimentů jsme se rozhodli implementovat především druhý způsob reprezentace dat – tedy data zpracováváme jako čtyři nezávislé proudy. V rámci experimentování však vznikly i méně výkonné funkce podléhající prvnímu způsobu zpracování dat. Může je upotřebit ten uživatel, který nebude zpracovávat větší množství promluv najednou. Pro jednotlivé promluvy mohou být tyto funkce rychlejší, než kdybychom zpracovávali jednu promluvu opravdovou a k tomu navíc tři prázdné na doplnění nezbytného počtu.

3.3 Návrh algoritmů

V následujících odstavcích rozebereme možnosti implementace některých obtížnějších algoritmů potřebných pro frontend rozpoznávače řeči. Pro návrh implementace jednodušších algoritmů jsme využili postupy výše zmíněné – rozvíjení cyklů, zpracování čtyř rámců prokládaně, snahu o datovou nezávislost po sobě následujících instrukcích a využití pokud možno obou pipeline SPU

Vzorků v rámci	Vzorkovací frekvence	Trvání rámce
128	4 kHz	32 ms
256	8 kHz	32 ms
512	16 kHz	32 ms
1024	32 kHz	32 ms
2048	64 kHz	32 ms

Tabulka 3.1: Tabulka trvání rámců při daných vzorkovacích frekvencích

Vzorků v rámci	Vzorkovací frekvence
64	2,56 kHz
128	5,12 kHz
256	10,24 kHz
512	20,48 kHz
1024	40,96 kHz
2048	81,92 kHz

Tabulka 3.2: Tabulka nutných vzorkovacích frekvencí pro rámec trvající 25 ms

naráz.

3.3.1 Výpočet DFT

Pro výpočet diskretní Fourierovy transformace využijeme algoritmus FFT. Jde o algoritmus pracující na principu rozděl a panuj. Jeho složitost je logaritmická, na rozdíl od složitosti DFT plynoucí z definice, která je kvadratická. Algoritmů počítajících DFT v logaritmickém čase existuje několik. Výběr správného algoritmu je třeba učinit na základě dat, která zpracováváme. Především záleží na počtu prvků transformované posloupnosti. Existují algoritmy, které vyžadují, aby byl počet prvků mocninou dvou, jiné algoritmy pracují například pouze s prvočíselným počtem prvků.

V našem případě jsme se rozhodli pro nejstarší známý algoritmus FFT, takzvaný Cooley-Tukey algoritmus. Důvody byly pragmatické – tento algoritmus je implementován v Cell SDK. Algoritmus vyžaduje, aby délka transformované posloupnosti byla mocninou dvou. Toto omezení lze splnit vhodnou délkou rámce pro danou vzorkovací frekvenci, viz tabulky 3.1 a 3.2.

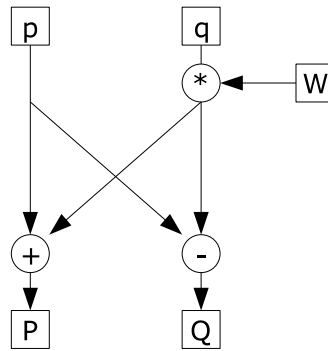
Algoritmus implementovaný v rámci Cell SDK transformuje komplexní posloupnost. V našem případě však potřebujeme transformovat reálnou posloupnost, proto jsme původní algoritmus upravili. Dále jsme algoritmus přizpůsobili zpracování čtyř prokládaných rámců najednou.

V definici Fourierovy transformace se vyskytují goniometrické koeficienty. Ty se v FFT objevují jakožto twiddle faktory a značí se zpravidla W . Jde o komplexní čísla. Základní operací FFT je tzv. motýlek, kdy spolu interagují dva prvky, přičemž jeden z nich (označovaný q) je před interakcí násoben určitým twiddle faktorem (viz obr. 3.5).

Twiddle faktory lze získat vyhodnocením výrazů:

$$\begin{aligned} \operatorname{Re}[W_i] &= \cos\left(\frac{2\pi i}{N}\right) \\ \operatorname{Im}[W_i] &= -\sin\left(\frac{2\pi i}{N}\right) \end{aligned}$$

pro $i \in \langle 0; \frac{N}{2} \rangle$. Počet potřebných koeficientů je tedy roven polovině velikosti transformované posloupnosti.



Obrázek 3.5: Motýlek - základní operace FFT

Twiddle faktory je vhodné během inicializační fáze předpočítat a uložit do tabulky pro pozdější využití. Protože se jedná o goniometrické funkce, je výpočet twiddle faktorů poměrně náročný. Twiddle faktory však vykazují určitou symetrii (viz obr. 3.6). Toho lze využít pro ušetření výpočetního času i paměti. Platí následující vztahy:

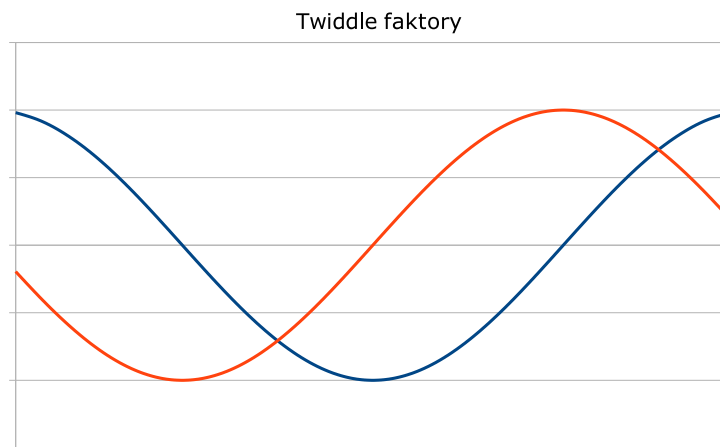
$$Re[W_{i+\frac{N}{4}}] = Im[W_i]$$

$$Im[W_{i+\frac{N}{4}}] = -Re[W_i]$$

$$Im[W_{\frac{N}{4}-i}] = -Re[W_i]$$

$$Re[W_0] = 1$$

$$Im[W_0] = 0$$



Obrázek 3.6: Průběh twiddle faktorů, reálná složka modře, imaginární červeně

Vztahy je možné odvodit z průběhu twiddle faktorů. Vyplývá z nich, že je možné všechny twiddle faktory předpočítat s použitím $\frac{N}{4}$ volání goniometrické funkce kosinus.

3.3.2 Výpočet reálné FFT prostřednictvím komplexní FFT

Rámce transformované prostřednictvím FFT v rozpoznávací řeči obsahují vzorky vstupního signálu, tedy reálná čísla. DCT je však definována pro čísla komplexní a i transformace reálné

posloupnosti má komplexní výsledek. Nejjednodušší metodou výpočtu reálné FFT je naplnit imaginární složky vstupu nulami a spustit komplexní transformaci.

Existuje však efektivnější způsob [10], který umožňuje provést reálnou FFT transformaci délky N pomocí komplexní transformace délky $\frac{N}{2}$ následované pomocným algoritmem – sudo-lichou dekompozicí (viz kapitolu 3.3.2.1).

Protože složitost FFT je $O(N \log N)$ a složitost sudo-liché dekompozice $O(N)$, je výsledná složitost

$$O(N \log N) + O(N) = O(N \log N)$$

Princip algoritmu vychází z možnosti výpočtu dvou reálných FFT pomocí jedné komplexní FFT stejné délky. Pokud bychom transformovali jednotkový skok v reálné oblasti, zatímco imaginární oblast bychom ponechali nulovou, získáme průběhy viditelné v pravé části obrázku 3.7. Je vidět, že výsledkem je sudá funkce v reálné části spektra a lichá funkce v imaginární části. Když budeme transformovat jednotkový skok v imaginární části vstupní posloupnosti (obr. 3.8), dostáváme lichou funkci v reálné části výsledku a sudou funkci v imaginární. Liché funkce jsou navzájem opačné. Protože pro FFT platí princip superpozice, můžeme oba jednotkové skoky transformovat zároveň. Výsledkem budou součty dosavadních výsledků. Nyní je zapotřebí tyto výsledky od sebe nějak oddělit. Získaný výsledek je v reálné i imaginární části součtem sudé a liché funkce. Rozklad libovolné funkce na funkci sudou a lichou lze provést pomocí sudo-liché dekompozice (viz 3.3.2.1). O výsledku platí:

- Sudá složka reálné části výsledku je reálná část spektra první posloupnosti.
- Lichá složka reálné části výsledku je imaginární část spektra druhé posloupnosti s opačným znaménkem.
- Sudá složka imaginární části výsledku je reálná část spektra druhé posloupnosti.
- Lichá složka imaginární části výsledku je imaginární část spektra první posloupnosti.

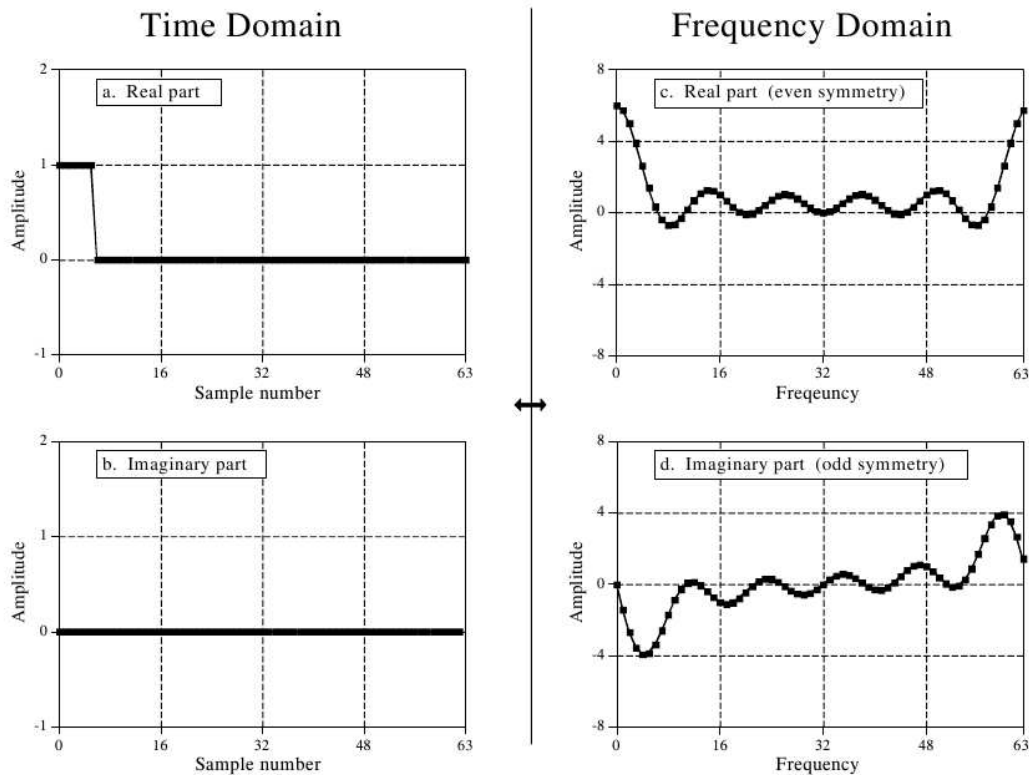
Tato tvrzení plynou přímo z obrázků 3.7 a 3.8.

Pro výpočet dvou reálných posloupností pomocí jedné komplexní tedy vytvoříme komplexní posloupnost, jejíž reálné složky prvků budou odpovídat prvkům první posloupnosti a imaginární složky prvkům druhé posloupnosti. Tuto komplexní posloupnost následně transformujeme pomocí FFT. Výsledek oddělíme sudo-lichou dekompozicí.

Chceme-li vypočítat pouze jednu reálnou FFT, opět vytvoříme komplexní posloupnost, jejíž reálné složky naplníme složkami se sudými indexy (počítáno od nuly) reálné posloupnosti a imaginární složkami s lichými indexy. Vzniklou komplexní posloupnost podrobíme FFT. Výsledek dekomponujeme sudo-lichou dekompozicí. Po dekompozici zbývá ještě jeden krok FFT, tentokrát již velikosti reálné posloupnosti.

Tímto postupem nesnižujeme počet kroků FFT, ten je nezbytně roven dvojkovému logaritmu velikosti FFT, avšak počítáme s FFT poloviční velikosti, čímž šetříme polovinu operací v každém kroku.

Požadavek na uložení prvků reálné posloupnosti střídavě do reálných a imaginárních složek komplexní posloupnosti, kterou poté FFT transformuje, pro nás nemusí být omezující. Budeme-li komplexní čísla reprezentovat jako posloupnost reálných čísel, kdy čísla na sudých indexech jsou reálné složky a čísla na lichých indexech složky imaginární (tedy složky jsou uloženy prokládaně), postačí transformovanou reálnou posloupnost pouze přetypovat a ihned máme správný vstupní formát.



Obrázek 3.7: Odezva FFT na jednotkový skok v reálné oblasti (zdroj [8])

3.3.2.1 Sudo-lichá dekompozice

Každou posloupnost lze rozložit na sudou a lichou složku, jejichž sečtením získáváme posloupnost původní. Rozklad je možno realizovat podle vzorců:

$$x_E[i] = \frac{x[i] + x[N - i]}{2}$$

$$x_O[i] = \frac{x[i] - x[N - i]}{2}$$

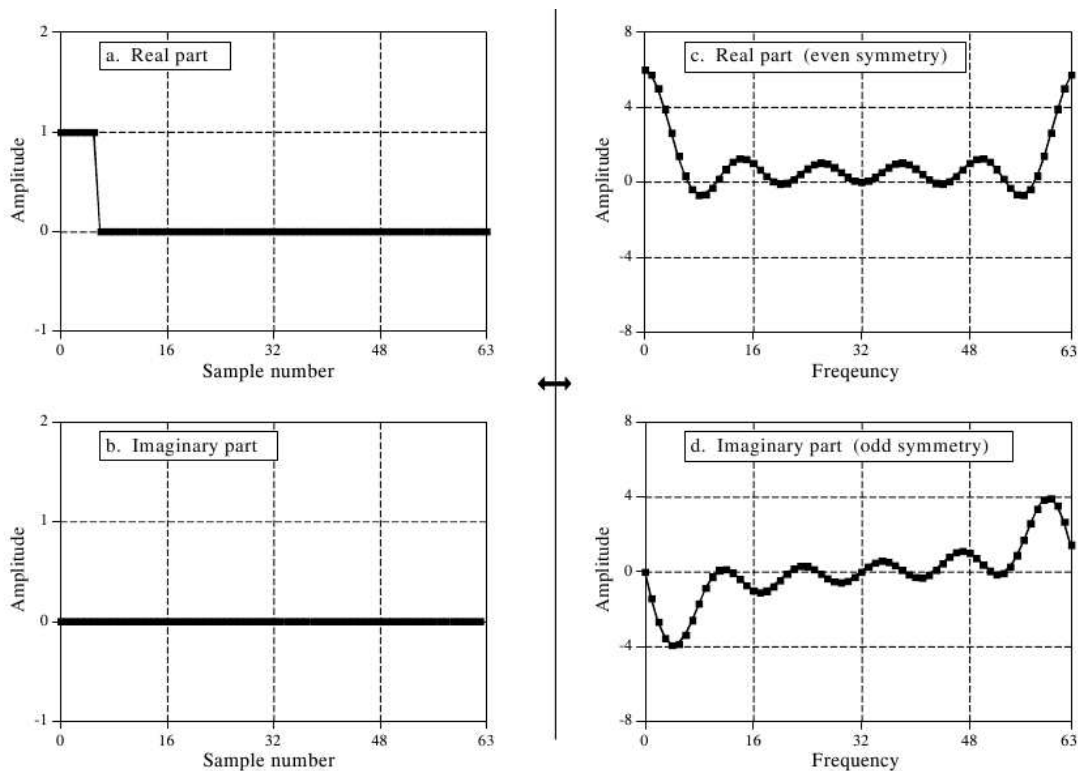
kde N je počet prvků posloupnosti a $i \in \{0; N/2 - 1\}$. Výsledná posloupnost x_E obsahuje sudou složku původní posloupnosti a x_O pak lichou.

Protože se jedná o přeuspořádání prvků pole během jednoho průchodu, je složitost sudo-liché dekompozice $O(N)$, tedy lineární.

3.3.2.2 Výpočet čtyř reálných FFT

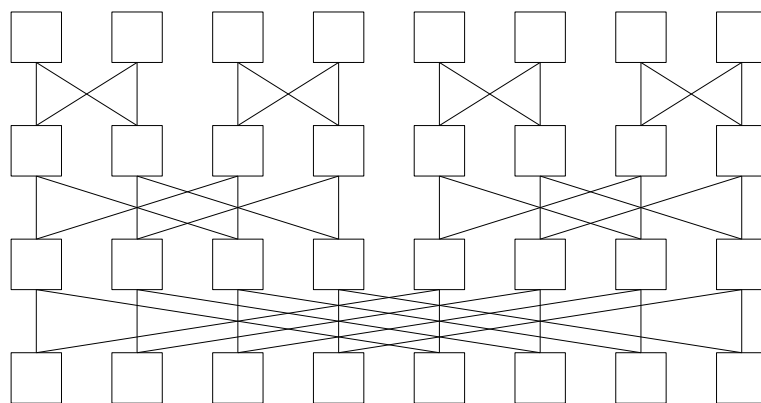
Vzhledem k navržené vnitřní reprezentaci dat, kdy pracujeme se čtyřmi nezávislými posloupnostmi najednou je třeba navrhnout FFT, která bude transformovat tyto čtyři posloupnosti. V kapitole 3.3.2 je popsán způsob, jak provést transformaci dvou reálných FFT pomocí jedné komplexní FFT stejné délky. Chceme-li spočítat čtyři FFT, můžeme buď použít tento algoritmus dvakrát nebo implementovat algoritmus vyhodnocující dvě komplexní FFT najednou následovaný sudo-lichou dekompozicí pro dvě komplexní posloupnosti.

Druhá možnost je pro nás výhodná, protože nám napomáhá snížit datové závislosti mezi posobě jdoucími operacemi. Implementace sudo-liché dekompozice pro dvě komplexní posloupnosti je triviální. Protože jsou vstupní posloupnosti i výsledné funkce na sobě zcela nezávislé,



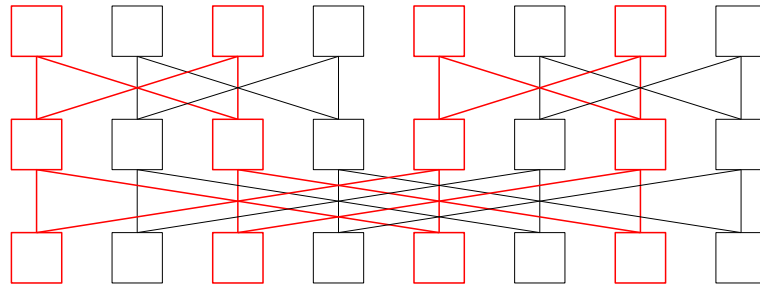
Obrázek 3.8: Odezva FFT na jednotkový skok v imaginární oblasti (zdroj [8])

jedná se o jednoduché zdvojení implementace pro jednu posloupnost. Pro výpočet FFT dvou komplexních posloupností je možné přímo využít stávající implementaci FFT jedné komplexní posloupnosti s drobnou úpravou - vypuštěním prvního kroku.



Obrázek 3.9: Operace mezi prvky při standardní FFT

Na obrázku 3.10 můžeme sledovat co se stane, pokud neprovedeme první krok FFT. Oproti standardní FFT (obrázek 3.9) nedojde k operaci mezi sousedícími prvky. Přivedeme-li tedy na vstup FFT bez prvního kroku dvě komplexní FFT uložené prokládaně, nedojde k jejich interakci. Abychom obdrželi správný výsledek, je ještě třeba upravit násobící koeficienty (twiddle faktory). Každý koeficient je oproti standardní FFT použit pro dva sousedící prvky. Lze tedy využít koeficientů předpočítaných pro jednoduchou FFT s tím, že zavedeme jinou formu inde-



Obrázek 3.10: Dvojitá FFT, nezávislá data jsou od sebe barevně odlišena

xování, nebo koeficienty uložíme do předpočítaného pole duplicitně. Pro naše účely jsme zvolili první možnost, protože nedojde k výraznému snížení výkonu a ušetříme tím paměť.

Vstup požadující prokládání komplexních posloupností navíc ve formátu střídajících se reálných a imaginárních složek přímo odpovídá formátu čtyř prokládaných reálných posloupností, které budeme touto FFT transformovat.

3.3.3 Výpočet DCT pomocí FFT

Výpočet DCT vede z definice na algoritmus složitosti $O(N^2)$. Tato transformace je však velice podobná DFT, proto se nabízí myšlenka, zda je možno DCT počítat podobným algoritmem jako FFT, jehož složitost je $O(N \log N)$. Dle [2] se ukazuje, že DCT je možno počítat přímo pomocí algoritmu FFT, přičemž je zapotřebí několika pomocných výpočtů, jejichž složitost je však lineární. Výsledná časová složitost je tedy $O(N \log N) + O(N) = O(N \log N)$.

Výpočet sestává ze tří kroků

1. Ze vstupní posloupnosti x vygenerujeme posloupnost y podle předpisu

$$y[i] = x[2i]$$

$$y[N - 1 - i] = x[2i + 1]$$

přičemž N je délka posloupnosti, která se má transformovat a $i \in \langle 0; \frac{N}{2} - 1 \rangle$.

Výsledkem je, že položky na sudých indexech jsou uspořádány na začátku pole a položky na lichých indexech v opačném pořadí od konce pole. Obě posloupnosti jsou reálné, protože DCT nepracuje s komplexními čísly.

2. Provedeme reálnou FFT nad získanou posloupností y . Získáváme posloupnost obrazů Y .
3. Z posloupnosti Y získáme posloupnost X , která odpovídá výsledné DCT, operací

$$X[i] = \text{Re}(Y[i] * (\cos(\frac{i\pi}{2N}) - j \sin(\frac{i\pi}{2N})))$$

kde N je délka transformované posloupnosti a $i \in \langle 0; N - 1 \rangle$.

Z komplexního výsledku FFT dostáváme reálnou posloupnost.

4 Implementace na procesoru Cell

Následující odstavce se vztahují přímo k implementovaným verzím algoritmů dostupných na příloženém CD. Doporučujeme čtenáři, aby měl při čtení následujícího textu vždy patřičný algoritmus připraven k nahlédnutí, usnadní se tím orientace v textu.

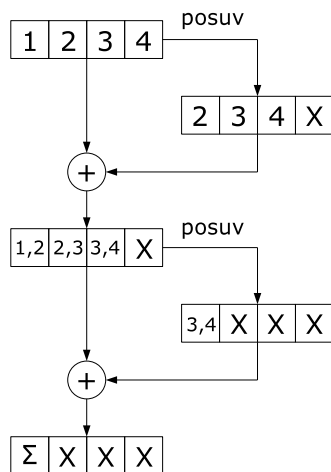
4.1 Filtrování střední hodnoty

K odfiltrování střední hodnoty potřebujeme znát její hodnotu. Tu lze určovat buď staticky na určitém bloku dat nebo též dynamicky pomocí digitálního derivačního článku.

4.1.1 Statické zpracování

Statické zpracování implementuje funkce `civ_mean_static`, která je k dispozici ve třech variantách:

- `civ_mean_static` je skalární implementace, která očekává na vstupu ukazatel na pole vzorků typu `float` a velikost tohoto pole a vrací jednu hodnotu typu `float`, která odpovídá vypočtené střední hodnotě. Funkce nejprve sečte hodnoty v poli a pak je vydělí počtem vzorků. Spočte tedy aritmetický průměr. Povolíme-li určitou míru rozvinutí cyklů, musí být velikost pole s touto mírou celočíselně dělitelná. Funkce pak v rámci cyklu přičítá k celkové sumě vždy několik položek pole. Jedná se o referenční verzi funkce, která není nijak optimalizovaná.
- `civ_mean_static_f4` na rozdíl od `civ_mean_static` požaduje na vstupu pole typu `vector float`. Funkce sčítá vektory vzorků, čímž nakonec obdrží čtyři součty, první je výsledek prvních položek v rámci vektorů, druhý druhých atd. Díky asociativitě součtu nyní postačí sečíst tato čísla. Provedeme tedy paralelní redukci (viz obr. 4.1), kdy pomocí posunu sečteme liché položky vektorů se sudými a následně tyto dva součty.



Obrázek 4.1: Paralelní redukce sčítání složek vektoru

Povolíme-li rozvinutí cyklů, jsou průběžné součty ukládány do více registrů. Počet těchto registrů je roven stupni rozbalení cyklu. Po průchodu vstupním polem tedy musíme provést paralelní redukci registrů a také paralelní redukci položek těchto registrů. Pořadí těchto operací je libovolné – můžeme nejprve sečíst položky v jednotlivých registrech a pak sečíst tato čísla, nebo nejprve sečíst registry a pak sečíst položky v jednom registru.

Součet samostatných čísel a vektorů se od sebe co do náročnosti provedení nijak neliší. Rozhodnutí je třeba učinit na základě algoritmu s rozvinutými cykly, v případě používání jednoho registru ztrácí tato úvaha smysl. První způsob nám poskytuje větší nezávislost mezi daty po sobě následujících operací součtu, druhý způsob vyžaduje méně operací součtu, avšak více závislých. Výběr tedy záleží na době zpracování instrukce součtu. Ukazuje se, že druhý způsob, kdy je součtů méně je výhodnější – při maximální úrovni rozbalení cyklů zabere cca 25 taktů, zatímco první způsob 35 (jedná se o výsledky získané statickou analýzou).

Na závěr je výsledný součet vydělen počtem vzorků v poli, což se provede vynásobením vektoru převrácenou hodnotou tohoto počtu. Výpočet převrácené hodnoty není nijak závislý na zbytku algoritmu, můžeme jej tedy provést kdekoliv tak, aby byla hodnota pro násobení včas připravena. Protože funkce vrací pouze jednu hodnotu, musíme ji z výsledného vektoru vyextrahovat. Zde využijeme znalosti reprezentace skalárních čísel na SPU – skalární číslo typu `float` je uloženo v první položce vektoru. Paralelní redukci tedy uspořádáme tak, aby finální součet a následně i podíl byl v této položce. Extrakci pak není třeba provádět.

Rozvineme-li cykly, musí být počet vzorků ve vstupním poli soudělný se čtyřnásobkem úrovně rozvinutí. Úzkým hrdlem algoritmu je závěrečná paralelní redukce, která při plném rozvinutí cyklů zabere $25 + 20 = 45$ taktů. Pro srovnání uvádíme, že jeden cyklus osmi vektorových součtů trvá 15 taktů hodin.

- `civ_mean_static_f4s` slouží pro výpočet čtyř středních hodnot čtyř polí najednou. Na vstupu je očekáváno pole typu `vector float`. Každý vektor v poli reprezentuje čtyři položky čtyř nezávislých polí. Výpočet probíhá obdobně funkci `civ_mean_static_f4`. Rozdíl je v tom, že se nevykonává závěrečná paralelní redukce po položkách, čímž je výpočet o 20 taktů kratší.

4.1.2 Dynamické zpracování

Dynamické určení střední hodnoty je založeno na digitálním derivačním článku. Funkce `civ_mean_dynamic`, která tento článek implementuje požaduje na vstupu pole vzorků, které má zpracovat, předchozí střední hodnotu a parametr `gamma`. Funkce provádí výpočet výrazu:

$$f(x) = \gamma * s_{n-1} + (1 - \gamma) * s_n \quad (4.1)$$

Funkce umí zpracovat i více vzorků. Návrátovou hodnotou je pak střední hodnota po dynamickém zpracování posledního z nich. Protože se předpokládá, že tato funkce bude volána často, je pro urychlení výpočtu kromě parametru `gamma` vstupem také parametr `mgamma`, který odpovídá hodnotě $1 - \gamma$. Výpočet dynamické střední hodnoty je implementován ve dvou verzích – funkcí `civ_mean_dynamic` a `civ_mean_dynamic_f4s`.

- `civ_mean_dynamic` je skalární implementací vztahu 4.1. Vstupními parametry je pole vzorků typu `float`. Dalšími parametry jsou počet položek v tomto poli, `gamma` a `mgamma` a předchozí vypočtená střední hodnota. Funkce navrácí dynamicky odhadnutou střední hodnotu po zpracování předaných vzorků. Zvýšením stupně rozvinutí cyklu je v rámci jednoho cyklu zpracován počet vzorků shodný se stupněm rozvinutí. Pole vzorků pak musí být soudělné s úrovní rozvinutí cyklů. Protože jde pouze o referenční verzi výpočtu, má rozvinutí vliv jen na počet skoků v rámci výpočtu, další optimalizace zde nejsou provedeny.
- `civ_mean_dynamic_f4s` dynamicky vyhodnocuje střední hodnotu několika posloupností vzorků najednou. Vstupem je jednak pole vektorů typu `vector float`, kde každý vektor

reprezentuje čtyři položky čtyř nezávislých polí. Dalším parametrem je počet vektorů v poli vektorů, který je shodný s počtem položek v každé posloupnosti. Vektory `vec_gamma` a `vec_mgamma` obsahují číslo γ resp. $1 - \gamma$ v každé své položce, čímž se šetří čas, který by byl spotřebován na výpočet $1 - \gamma$ a na rozkopírování jedné složky vektoru do všech ostatních. Posledním parametrem je vektor udávající poslední doposud vypočtenou střední hodnotu pro každou vstupní posloupnost.

Pokud nevyužijeme možnost rozvinutí cyklů, probíhá výpočet obdobně, jako u funkce `civ_mean_dynamic`. To není příliš výhodné, protože je zde velká závislost mezi daty. Při míře rozvinutí cyklů 2 a více je možno využít registrový double buffering (viz kapitolu 3.1.2), který tuto závislost mezi daty snižuje. Výpočetní vzorec nelze vyhodnotit pomocí jediné instrukce a obsahuje část na předchozím výpočtu závislou ($\gamma * s_{n-1}$) a nezávislou ($(1 - \gamma) * s_n$). Můžeme tedy po spuštění funkce spočítat dvě nezávislé části

$$((1 - \gamma) * s_1)$$

$$((1 - \gamma) * s_2)$$

Poté pokračujeme výpočtem $\bar{s}_1 = (\gamma * s_0) + ((1 - \gamma) * s_1)$, spustíme výpočet $((1 - \gamma) * s_3)$, spočítáme $\bar{s}_2 = (\gamma * \bar{s}_1) + ((1 - \gamma) * s_2)$ atd. Tento postup vede na určité zrychlení, avšak pro SPU jsou datové závislosti stále příliš těsné. Problém nastává především ve výpočetní, sudé pipeline, zatímco lichá pipeline je nevytížená.

4.1.3 Dynamické zpracování spolu s transpozicí

Pokud zpracováváme čtyři rámce najednou, je třeba je na vstupu převést do prokládaného formátu, tedy takového, kdy vektor čtyř čísel obsahuje vždy jeden vzorek z každého rámce. Běžným formátem vstupu budou totiž spíše oddělené rámce. Protože transformace dat z jednoho formátu do jiného je prací především pro lichou pipeline, je možné ji spojit s jinou operací do jedné funkce.

Vhodnou funkcí se jeví dynamické vyhodnocování střední hodnoty. V této funkci panují značné datové závislosti mezi instrukcemi vyhodnocovanými sudou pipeline. Přiměřená zátěž liché pipeline tak nezpůsobí příliš velké zpomalení. Navíc je tato funkce používána hned na začátku zpracovávajícího řetězce frontendu, což je pro konverzi formátu nezbytné.

Transpozice vstupních dat spolu s výpočtem střední hodnoty je implementována funkcí `civ_mean_dynamic_and_transp_f4s`. Vstupem funkce jsou čtyři rámce. Výstupem je spočtená střední hodnota a proud dat prokládaných rámců. Funkce ještě nebyla důkladně testována, proto je zmiňována pouze zde, jako doplňková možnost.

4.1.4 Odečtení střední hodnoty

Poté, co střední hodnotu získáme, je třeba ji odečíst od původního signálu. K tomu slouží funkce `civ_addconst` a `civ_addconst_f4s`. Úkolem obou funkcí je přičíst k zadanému poli zadanou hodnotu. Chceme-li odečíst střední hodnotu, zadáme ji jako parametr pro sčítání s opačným znaménkem.

- `civ_addconst` je referenční verzi funkce psanou ve standardním C. K poli čísel typu `float` přičítá jiné číslo typu `float`. Počet součtů, který je vykonán v rámci jedné iterace algoritmu je rovný úrovni rozvinutí cyklů. Proto musí být délka pole s touto úrovní soudělná.

- `civ_addconst_f4s` je optimalizovaná verze pracující se čtyřmi prokládanými proudy dat. Přičítanou hodnotou je vektor, jehož položky nemusí být stejné – každá položka je přičítána k jednomu proudy dat. Počet funkcí používaných registrů je roven úrovni rozvinutí cyklů. Funkce nejprve načte hodnoty do všech registrů, provede sečtení všech registrů s danou hodnotou a následuje uložení. Načtení a uložení vytěžuje lichou pipeline, zatímco součet sudou. Proto při rozvinutí cyklů začíná sčítání prvního registru ještě během načítání registrů dalších. Stejně tak ukládání výsledků probíhá paralelně se sčítáním. Přesné seřazení instrukcí ponecháváme na optimalizujícím překladači.

4.2 Preemfáze

Preemfáze slouží k zesílení amplitud vyšších frekvencí v řečovém signálu. Je implementována ve dvou verzích funkcemi `civ_preemp` a `civ_preemp_f4s`.

- `civ_preemp` je referenční, skalární, nepříliš optimalizovaná funkce pro posílení vyšších amplitud. Jsou-li rozvinuty cykly, pak algoritmus vykoná tolik kroků během jedné iterace, kolik je úroveň rozvinutí.
- `civ_preemp_f4s` slouží pro kalkulaci preemfáze čtyř prokládaně uložených datových proudů. Na vstupu očekává kromě pole pro zpracování a jeho velikosti také vektorový parametr `kappa`, který udává míru preemfáze. Tento parametr se běžně pohybuje v rozpětí 0,95 – 0,99. V rámci optimalizace jsme tento parametr učinili vektorovým, přičemž je očekáván vektor, jehož položky jsou všechny rovny κ . Vzhledem k tomu, že funkce potřebuje k výpočtu dvě hodnoty z pole (současnou a předcházející), využili jsme optimalizace typu registrový double buffering – pro výpočet jsou používány dvě sady registrů, přičemž s jednou je počítáno, zatímco s druhou se provádí operace s pamětí. Úlohy registrových sad se poté vymění (viz kapitolu 3.1.2).

4.3 Váhování rámců okénkem

Funkce pro práci s okénky lze rozdělit do dvou základních kategorií – funkce, které okénka generují a funkce, které vygenerovaná okénka aplikují na signál.

4.3.1 Funkce pro generování okének

Implementovány jsou funkce pro vygenerování dvou různých okének – pro obdélníkové a pro Hammingovo.

- `civ_win_get_rect` generuje obdélníkové okénko, tedy násobení jedničkou v zadaném rozsahu. Parametry je alokované pole v paměti, kam se mají násobící koeficienty uložit a velikost požadovaného pole. Jedná se o neoptimalizovanou referenční verzi algoritmu. Pokud nastavíme určitou úroveň rozvíjení cyklů, musí být požadovaná délka okénka s touto úrovní soudělná. V rámci jednoho cyklu algoritmu se bude generovat tolik násobících koeficientů, kolik je úroveň rozvinutí cyklů.
- `civ_win_get_rect_s` slouží k vygenerování okna pro okénkování prokládaných posloupností vzorků. Pracuje na stejném principu, jako funkce `civ_win_get_rect`. Výsledek je ukládán do pole typu `vector float`.
- `civ_win_get_hamming` generuje Hammingovo okénko (viz obr. 4.2) do pole typu `float`. Jedná se o základní skalární verzi. Rozvinutím cyklů získáme výpočet několika koeficientů v jednom cyklu algoritmu, což obnáší podmínku soudělnosti požadované délky okna s úrovní rozvinutí cyklů.



Obrázek 4.2: Hammingovo okénko

- `civ_win_get_hamming_s` generuje Hammingovo okno pro váhování prokládaných posloupností. Jeho funkce je totožná s `civ_win_get_hamming`, pouze výsledek výpočtu není ukládán do jedné proměnné typu `float`, ale do proměnné typu `vector float`, přičemž všechny položky mají stejnou hodnotu.

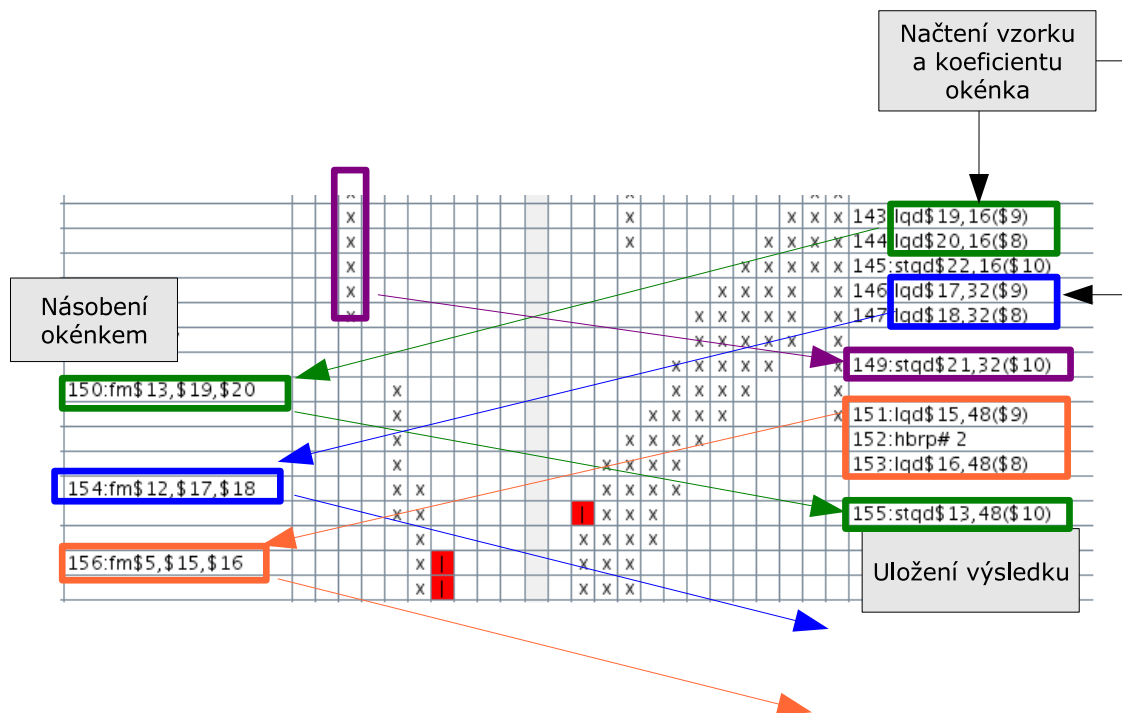
Tato a předešlá funkce nejsou nijak optimalizovány. Předpokládá se, že budou využity jen jednou, při inicializaci aplikace. Statická analýza provedená nad pokusnou optimalizovanou verzí ukázala, že zdaleka nejefektivnější, jak pro čas programátora, tak pro čas výpočetní, bude výpočet Hammingova okna na nevektorovém PPU a jeho následný přenos do lokální paměti SPU.

4.3.2 Funkce pro okénkový výběr

Okénko na signál aplikujeme pomocí operace násobení. Násobíme signál (o délce jednoho rámece) s vygenerovaným okénkem. Formáty vygenerovaného okénka jsou dva – pro jeden rámeček a pro čtyři prokládané rámečky. První z nich generují funkce bez přípony, druhé z nich funkce s příponou `_s`. Okénkový výběr zajišťují funkce rodiny `civ_win_sel`:

- `civ_win_sel` je referenční verze pracující s jednou posloupností (jedním rámečkem). Není optimalizovaná a je napsána prostřednictvím standardního jazyka C. Vstupní pole, výstupní pole i předgenerované okénko této funkce očekává ve formě ukazatele na typ `float`. Parametr `size` udává počet čísel ve zpracovávaném rámečku. Provedeme-li rozvinutí cyklu, je v rámci jednoho cyklu vynásobeno okénkovou funkcí více vzorků v rámci. Délka okénka, vstupu i výstupu musí být soudělná s úrovní rozvinutí cyklu. Datové závislosti mezi operacemi zde nejsou nijak řešeny.
- `civ_win_sel_f4` provádí obdobnou činnost jako `civ_win_sel`, avšak jeho vstupními parametry jsou ukazatele typu `vector float`. Funkce zpracovává vstupní data po vektorech o velikosti čtyř vzorků. Pokud je stupeň rozvinutí cyklů alespoň dva (tedy cykly jsou alespoň minimálně rozvinuty), využívá se registrový double buffering (viz kapitolu 3.1.2). Po několika úvodních krocích implementace využívá dvou pipeline SPU tak, že zatímco je sudou pipeline násoben z paměti načtený koeficient okénka s načteným vzorkem, pracuje lichá pipeline na uložení předchozího výsledku a načtení dalšího vzorku a koeficientu okénka. Logicky je tak lichá pipeline více vytížená. Instrukce uložení a načtení však trvá stejný počet taktů, jako násobení reálným číslem (konkrétně 6 taktů). Protože jsou instrukce ukládání a načítání nezávislé, jsou všechny tři dokončeny během osmi taktů.

Optimalizující překladač seřadí instrukce tak, že se ihned načítají další hodnoty z paměti, prodlevy tedy pak vznikají naopak čekáním na operaci násobení.



Obrázek 4.3: Vykonávání funkce `civ_win_sel_f4`

Na obrázku 4.3 je část statické analýzy vykonávání okénkového výběru. Práce s jednotlivými vektory je barevně odlišena. Je vidět, že zatímco je sudou pipeline násoben fialový vektor, je lichou pipeline načítán vektor zelený. Po dokončení načítání je sudou pipeline násoben zelený vektor, zatímco lichá pipeline ukládá výsledek fialového výpočtu a následně načítá další vzorek a koeficient okénka ve schématu označen modře. Dobrým rozvržením operací tak dochází k využití obou pipeline.

Použitím ještě většího počtu registrů při rozvíjení cyklu by bylo možno výpočet ještě urychlit. Vzhledem k vytíženosti liché pipeline by však zrychlení bylo zanedbatelné proti složitosti kódu.

- `civ_win_sel_f4s` implementuje okénkový výběr nad čtyřmi prokládanými rámci. Operace je zcela stejná s `civ_win_sel_f4`, pouze okénko je očekáváno v jiném formátu. Parametr `size` má význam počtu vzorků v jednom ze stejně dlouhých prokládaných rámců, což je rovno počtu vektorů ve čtyřnásobně dlouhém jednoduchém rámci. Lze tedy přímo využít funkci `civ_win_sel_f4`.

4.4 Střední krátkodobá energie

Implementace výpočtu střední krátkodobé energie rámce je do značné míry podobná implementaci výpočtu střední hodnoty statickou metodou (viz 4.1.1). Návrh struktury výpočtu je totiž stejný, liší se pouze vyhodnocovaný výraz, jde v podstatě o střední hodnotu druhých mocnin vzorků. K dispozici jsou, stejně jako u střední hodnoty, tři funkce:

- `civ_energy` je referenční skalární verze. Rozvinutí cyklů zajistí součet několika druhých mocnin vzorků v rámci jednoho cyklu algoritmu najednou. Délka vstupních dat musí být

opět soudělná s úrovní rozvinutí cyklů. Funkce vrací jedno číslo datového typu `float`, které je součtem druhých mocnin vstupních hodnot děleným počtem těchto hodnot.

- `civ_energy_f4` vrací také jedno číslo typu `float`, které má hodnotu střední krátkodobé energie vstupního pole. To je však tentokrát zadáno jako pole vektorů. Funkce používá několik registrů pro dílčí součty (v případě rozvinutí cyklů), přičemž v závěru provede paralelní redukci těchto registrů a následně i paralelní redukci složek součtu, čímž získá střední hodnotu energie, kterou vrátí. Součet složek je, stejně jako u střední hodnoty, koncipován tak, aby výsledná hodnota byla ve složce vektoru, kterou SPU používá pro reprezentaci skalárních čísel.
- `civ_energy_f4s` zpracovává čtyři prokládané rámce. Princip výpočtu je shodný s předchozí funkcí až na závěrečnou redukci složek výsledného součtu, která se zde neprovádí.

4.5 Počet průchodů signálu nulou

Počítání počtu průchodu signálu nulou je založeno na porovnávání znamének čísel, které po sobě ve vstupních datech následují. Jde o jeden z těch algoritmů, kdy pro zpracování pole vzorků o délce `N` musíme provést `N - 1` cyklů algoritmu. To přináší komplikaci, pokud cykly rozvíjíme. Řešením je provést `N - UNROLL_SIZE` a pak ještě `UNROLL_SIZE - 1` cyklů, kde `UNROLL_SIZE` je úroveň rozvinutí cyklů. Dostáváme tak

$$N - UNROLL_SIZE + UNROLL_SIZE - 1 = N - 1$$

cyklů, což je počet, který požadujeme. Počet průchodů nulou vyhodnocují dvě funkce:

- `civ_zero_crossings` je skalární neoptimalizovanou implementací, která znaménka po sobě následujících vzorků porovnává pomocí podmíněného příkazu, přičemž předešlé znaménko je uloženo v proměnné typu `bool`. Změny znaménka jsou čítány, po projití celého pole vzorků je načítaný počet vrácen jako výsledek. Rozvinutí cyklů funkci nijak neovlivňuje.
- `civ_zero_crossings_f4s` zpracovává čtyři prokládané rámce najednou. Funkce využívá kódování datového typu `float`, při kterém je znaménko uloženo v nejvyšším bitu. V jednom kroku algoritmu je získán exkluzivní součet (XOR) aktuálně zkoumaného vzorku a vzorku předchozího. Pokud se znaménka liší, dostaneme v nejvyšších bitech složek vektoru bit 1, při shodných znaménkách bit 0. Abychom mohli tuto hodnotu sečíst, je třeba tento bit posunout na pozici 0 a přetypovat vektor na celočíselný. Celočíselný typ se stejnou bitovou šířkou, jako má `float` je `unsigned long`. Provedeme tedy logický posun o 31 bitů vpravo, čímž vynulujeme všechny bity složky vektoru krom nejvyššího, který se dostane na bit nejméně významný. Dostáváme tak hodnotu 0 či hodnotu 1. Výsledek přičteme do počítadla změn znaménka. Funkce rozlišuje kladnou a zápornou nulu a přechod mezi nimi vyhodnotí jako změnu znaménka. To však není na škodu, pokud signál osciluje blízko nuly, jedná se nejspíše o sykavku a u takové je vyšší počet průchodů signálu nulou příznačný.

Během vývoje jsme otestovali ještě jinou implementaci vyhodnocení znaménka, pomocí celočíselného porovnání a operace logického součinu namísto posunu vpravo. Tyto dvě instrukce trvají dva taktů, na rozdíl od posunu, který trvá čtyři. Vzájemná nezávislost dat je však dostatečná, takže delší trvání výpočtu rotace se neprojeví negativně, naopak se ušetří taktů díky nahrazení dvou instrukcí instrukcí jednou. Konkrétně statická analýza dává pro implementaci s posunem při úrovni rozvinutí cyklů rovné osmi trvání 85 taktů, zatímco implementace se součinem 102 taktů. Vnitřní cyklus trvá v prvním případě 44 a ve druhém 51 taktů. Pro délku rámce 512 vzorků dostáváme úsporu cca 460 taktů.

4.6 Rychlá Fourierova transformace

Implementace FFT vychází z implementace, která je dodávána spolu s Cell SDK. Minimální velikost transformované posloupnosti je 32 komplexních čísel. Taková implementace se hodí pro dopřednou FFT při spektrální analýze. Pro výpočet zpětné FFT pomocí DCT již vhodná není, protože jde pouze o cca 24 čísel.

4.6.1 Generování twiddle faktorů

Funkce vyhodnocující FFT jsou navrženy tak, aby akceptovaly stejný formát twiddle faktorů a aby byly schopny využít jejich symetrie (viz odst. 3.3.1). Dále je umožněno používat pole twiddle faktorů větší, než je pro danou FFT potřeba, což přináší úsporu paměti v případě výpočtu FFT nad různě velkými daty (stačí vygenerovat pouze jedno, největší potřebné pole). Pro vygenerování twiddle faktorů je k dispozici funkce `civ_twiddle_gener`. Funkce požaduje coby vstupní parametr délku FFT, pro kterou budou twiddle faktory použity. Výsledkem je pole komplexních twiddle faktorů uložených prokládaně, tzn. v každém vektoru dvě čísla, střídavě reálná a imaginární složka. Pro výpočet FFT délky N je zapotřebí $\frac{N}{2}$ komplexních twiddle faktorů. Díky jejich symetrii můžeme dále ušetřit výpočet goniometrických funkcí s některými argumenty. Ve výsledku stačí pro FFT délky N vyhodnotit $\frac{N}{4}$ goniometrických funkcí. Funkce `civ_twiddle_gener` není paralelizována na úrovni instrukcí. Předpokládá se její využití na PPU, které předgeneruje koeficienty a potřebným SPU je zašle. Kromě symetrie twiddle faktorů využívá také vlastnost lineárního růstu argumentu kosinu. Díky tomu je možno namísto násobení využívat součtu. V každém kroku algoritmu je vygenerována reálná složka jednoho twiddle faktoru a imaginární složka jiného. Rozvinutím cyklu je toto opakováno tolikrát, kolik je úroveň rozvinutí. Tomu také musí odpovídat velikost FFT, pro kterou se twiddle faktory generují - musí být soudělná s dvojnásobkem úrovně rozvinutí cyklů.

4.6.2 Jednoduchá komplexní Fourierova transformace

Z implementace FFT dodávané v rámci knihoven Cell SDK bylo třeba oddělit poslední krok. Počítáme-li transformaci reálných posloupností, potřebujeme ho totiž provést samostatně (viz kapitolu 3.3.2). Z této implementace nevychází pouze výpočet reálné FFT, ale i výpočet dvojitě komplexní FFT, používané pro vyhodnocení čtyř reálných FFT (viz kapitolu 3.3.2.2). Protože jiná dokumentace, než komentáře ve zdrojovém kódu k principu fungování tohoto algoritmu neexistuje, je vhodné zde tento princip uvést. Jsou na něm založeny naše modifikace tohoto algoritmu.

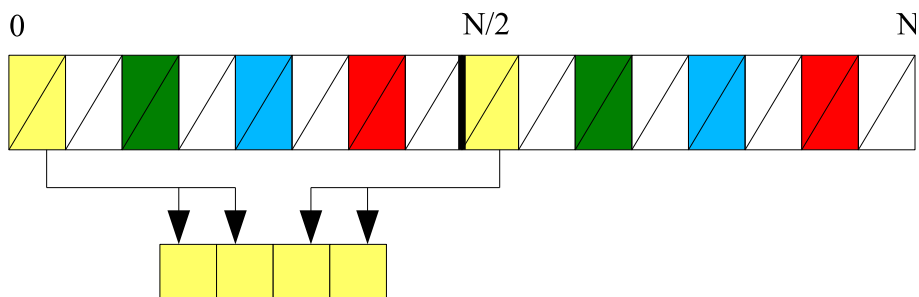
Výpočet FFT je rozdělen do několika fází. V úvodní fázi jsou v iteraci vykonávány první tři kroky FFT. V následující fázi algoritmus v iteraci provádí další kroky FFT, počínaje čtvrtým a konče krokem $N - 2$. Poslední dva kroky jsou tedy provedeny mimo tuto iteraci. Předposlední krok je vypočten prakticky stejným způsobem, jako kroky v iteraci. Rozdílem je drobná výkonná optimalizace. V poslední fázi je prováděn poslední krok spolu s formátováním výsledku. Vstupem FFT je pole typu `vector float`, kdy v každém vektoru jsou uložena dvě komplexní čísla, a to prokládaně (viz obr. 4.4).



Obrázek 4.4: Dvě komplexní čísla uložena prokládaně v jednom vektoru

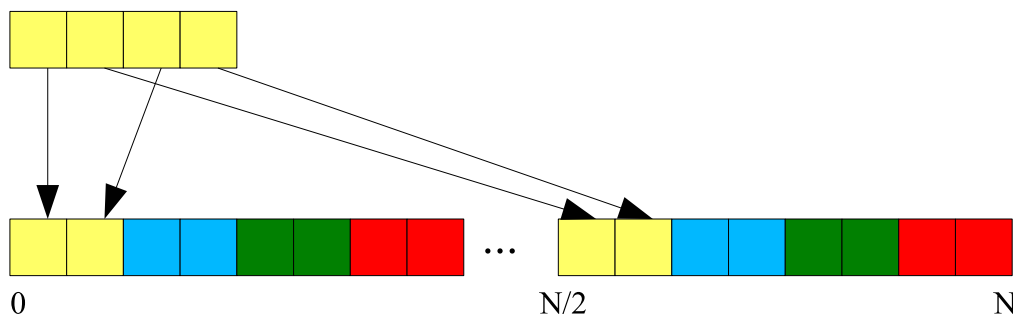
Před provedením prvního kroku FFT je třeba ze vstupní posloupnosti načíst data a setřídít je způsobem bit-reversing (viz [9]). Proto data nenačítáme postupně, ale tak, abychom třídění

maximálně usnadnili. Na obrázku 4.5 jsou barevně označeny vektory, které ze vstupního pole použijeme při první iteraci první fáze.



Obrázek 4.5: Výběr vektorů ze vstupních polí při prvním kroku FFT

Další operací je bit-reversing v rámci načtených vektorů. Položky ve vektorech označených stejnou barvou se po setřídění bit-reversingem stávají sousedy. Z každé dvojice stejnou barvou označených vektorů vzejde dvojice komplexních čísel patřících do první poloviny setříděné posloupnosti a druhá dvojice patří do druhé poloviny. Toto třídění je naznačeno na obrázku 4.6.



Obrázek 4.6: Bit-reversing aplikovaný na jednotlivá komplexní čísla

Je vidět, že vhodným výběrem vektorů ze vstupního pole jsme získali dvě posloupnosti po sobě následujících setříděných čísel. Tyto dvě posloupnosti budeme zpracovávat paralelně, čímž zvýšíme datovou nezávislost po sobě jdoucích operací. Můžeme přistoupit k motýlku mezi sousedícími komplexními čísly.

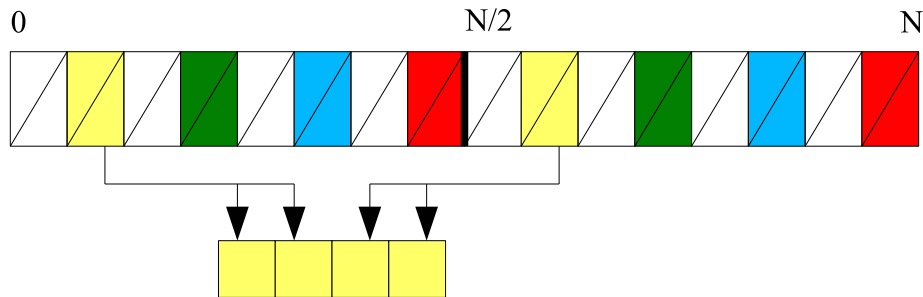
V celé první fázi FFT jsou používané twiddle faktory jednoduché. Pro první krok jde o komplexní číslo $1 + 0j$. V rámci jedné iterace první fáze provedeme celkem 8 motýlků, čtyři v první polovině setříděné posloupnosti a čtyři v druhé.

Ve druhém kroku provádíme motýlek dvojic umístěných ob jedno číslo a oddělujeme imaginární a reálnou složku do separátních polí. Dvojice komplexních čísel je uložena v jednom vektoru, jde tedy o operaci mezi dvěma vektory. Výsledkem jsou čtyři reálná a čtyři imaginární čísla, tedy opět dva vektory. Pole reálných a imaginárních složek během iterací prvních tří kroků FFT průběžně naplňujeme jednak od indexu 0 a také od poloviny. Snižuje se tím datová závislost po sobě jdoucích operací. Twiddle faktory pro tento krok jsou opět triviální, jde o čísla $1 + 0j$ a $0 - j$.

Ve třetím kroku jde již o základní verzi motýlka, která je používána v dalších krocích. Tedy motýlek čtyř a čtyř komplexních čísel, jejichž reálné a imaginární složky jsou odděleny. Toto

oddělení opět zvyšuje nezávislost po sobě jdoucích operací. Pro motýlka čtyř dvojic komplexních čísel potřebujeme již čtyři twiddle faktory. Stále jde o relativně jednoduchá čísla. Schematické znázornění prvních tří kroků FFT je na obrázku 3.9.

Po provedení tří kroků FFT nad šestnácti komplexními čísly následuje další iterace, kdy je zpracováno dalších šestnáct čísel. Čísla jsou ze vstupního pole vybírána tak, aby po setřídění bit-reversingem navazovala na čísla již zpracovaná. Výběr ilustruje obrázek 4.7.



Obrázek 4.7: Výběr vektorů ze vstupního pole při druhé iteraci první fáze FFT

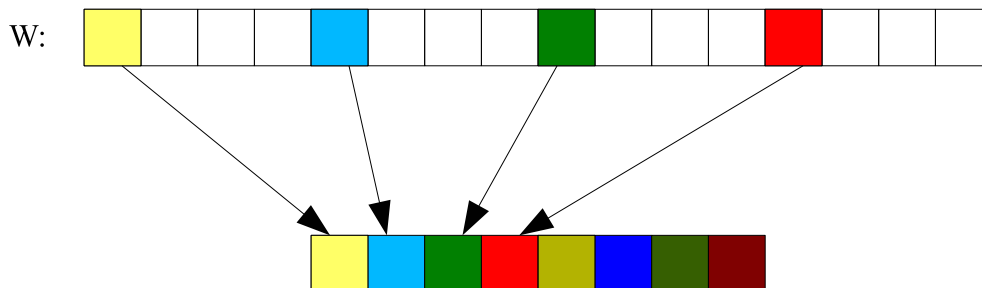
Ve druhé fázi jde o motýlky osmic, šestnáctic a dále. Je zde problém s uložením twiddle faktorů, který se v první fázi nevyskytoval. Prozatím byly tyto faktory buď jednoduché nebo se alespoň vešly do jednoho vektorového registru. To není případ druhé fáze, kde je potřeba osm a více komplexních twiddle faktorů.

Princip práce druhé fáze spočívá ve výběru čtyř po sobě jdoucích twiddle faktorů. Následně je procházeno pole s reálnými a imaginárními složkami mezivýsledků, přičemž krokem (ve zdrojovém kódu identifikovaném proměnnou `stride`) je šestnáct čísel pro motýlky osmic, 32 pro motýlky šestnáctic atd. Z každé dvojice n -tic, nad kterými se operuje jsou spočteny první čtyři složky – známe totiž pouze čtyři twiddle faktory. Navíc, díky symetrii twiddle faktorů, je zároveň zpracovávána druhá polovina n -tice. Stejně jako v první fázi pracujeme najednou nad první i druhou polovinou prvků. Následuje načtení dalších čtyř twiddle faktorů a opakování tohoto výpočtu nad další částí motýlků.

Které twiddle faktory se z pole vyberou je určenou mohutností n -tic, které se účastní motýlka. Cekově je zapotřebí $\frac{n}{2}$ twiddle faktorů, díky symetrii jich z pole stačí vybrat $\frac{n}{4}$. Twiddle faktory jsou z pole vybírány s druhou mocninou, tzn. pokud vybíráme k twiddle faktorů, pak pole rozdělíme na $2^{(k-1)}$ segmentů a použijeme první twiddle faktor v každém segmentu. V posledním kroku, kdy provádíme motýlka nad celou transformovanou posloupností a tedy $k = N$, použijeme z pole twiddle faktorů všechny twiddle faktory. Výběr je ilustrován na obrázku 4.8. Tmavšími odstíny barev jsou vyobrazeny odvozené twiddle faktory. Pro odvození jsou využívány vztahy uvedené v kapitole 3.3.1.

Do tohoto algoritmu výběru jsme doplnili princip vstupního parametru `log2_twiddle_skip`. Jde o snížení dělicího koeficientu pole twiddle faktorů. Na vstupu je pak možné předat více twiddle faktorů, než je pro daný konkrétní případ potřeba. Tím se ušetří inicializační čas a především paměť, protože pak stačí předpočítat pouze jedno pole twiddle faktorů, takové, které je zapotřebí pro nejdelší algoritmem FFT transformovanou posloupnost.

V rámci závěrečného kroku je možno výsledky buď ponechat v oddělených polích pro jejich reálnou a imaginární složku, nebo tyto složky opět sloučit do stejného formátu, jaký je na vstupu, čili prokládaně reálná a imaginární složka. Zásahem do původního algoritmu jsme pomocí `inline` funkcí umožnili obě možnosti. Uživatel použije nejprve funkci pro výpočet všech kroků, kromě posledního (`_civ_fft_part1`). Následným voláním funkce `_civ_fft_last_step_inter` resp. `_civ_fft_last_step_2` obdrží promíchané resp. separované výsledky.



Obrázek 4.8: Výběr twiddle faktorů pro motýlky šestnáctic

Komplexní FFT lze vyhodnotit voláním funkce `civ_fft_cmplx_inter_f4`, jejímž vstupem i výstupem je pole prokládaných komplexních čísel.

4.6.3 Výpočet reálné FFT pomocí jednoduché komplexní FFT

Fourierovu transformaci reálných čísel je možné počítat pomocí komplexní Fourierovy transformace. Je to vhodné již proto, že ačkoliv je vstupní posloupnost reálná, výsledek je komplexní. Můžeme buď do imaginárních složek vstupních komplexních čísel vložit 0, nebo využít možnosti vypočítat reálnou FFT pomocí poloviční FFT komplexní. První způsob je jednak méně optimální, ale hlavně je na vektorovém procesoru vkládání nul do pole reálných čísel neefektivní. Druhý způsob je výhodnější, protože není třeba vstupní data nikterak transformovat a vyhodnocovaná FFT má poloviční velikost. Po takové FFT musí následovat sudo-lichá dekompozice (viz odst. 4.6.5) a ještě jeden poslední krok FFT.

Pro výpočet reálné FFT jsou k dispozici funkce `civ_fft_real_inter_f4`, která vrací výsledek ve formě prokládaného pole reálných a imaginárních složek komplexních čísel, a funkce `civ_fft_real_2_f4`, která vrací reálnou a imaginární složku odděleně. Obě funkce pracují na principu výpočtu reálné FFT pomocí poloviční FFT. Protože pro závěrečný krok je zapotřebí dvakrát více twiddle faktorů, než pro poloviční FFT (závěrečný krok je transformace délky vstupní reálné posloupnosti), využívá se zde parametr `log2_twiddle_skip`, který nepotřebné twiddle faktory přeskočí.

4.6.4 Dvojitá komplexní Fourierova transformace

Mnoho algoritmů výše pracovalo s prokládaným formátem čtyř posloupností, případně čtyř rámců. Většina těchto algoritmů se jevila jako neoptimálnější. Aby bylo možné zpracovávat vstupní data tímto způsobem, je zapotřebí implementovat FFT, která bude schopna počítat 4 reálné FFT najednou. Protože dvě reálné FFT je možno vyhodnotit pomocí jedné komplexní FFT, je také možné pomocí komplexní FFT pro dvě nezávislé komplexní posloupnosti vyhodnotit 4 reálné FFT.

Vstupním parametrem dvojitě komplexní FFT je pole typu `vector float`. Každý vektor v tomto poli obsahuje dvě komplexní čísla, přičemž každé z nich reprezentuje jednu položku dvou nezávislých komplexních posloupností. Ve vektoru je uložena reálná a imaginární složka položky první posloupnosti následovaná reálnou a imaginární složkou posloupnosti druhé. Formát ilustruje obrázek 4.9.

Hlavní myšlenkou výpočtu dvojitě komplexní FFT je vypuštění prvního kroku jednoduché FFT, kdy probíhá motýlek mezi složkami vektoru. Prvním krokem bude původně druhý, motýlek mezi jedním a jedním vektorem. Protože každý vektor obsahuje jednu položku jedné posloupnosti, jedná se vlastně o zdvojený první krok. Tento princip je uplatňován i nadále. V první fázi FFT je tedy vyhodnocován první až třetí krok, který je prakticky shodný s druhým



Obrázek 4.9: Formát vstupního pole pro dvojitou komplexní FFT

až čtvrtým krokem jednoduché posloupnosti. Protože je v první fázi třeba dospět až k původně čtvrtému kroku, kdy se motýlka účastní dvě osmice vektorů, je třeba pracovat od počátku s šestnácti vektory. (Namísto osmi u jednoduché FFT.)

Další změnou oproti jednoduché komplexní FFT jsou používané twiddle faktory. Protože v každém kroku počítáme vlastně s dvojnásobným počtem čísel, je třeba twiddle faktory taktéž zdvojit. Funkce využívá stejné pole twiddle faktorů, jako jednoduchá komplexní FFT, zdvojení se provádí v rámci algoritmu.

Druhá fáze této implementace FFT se od jednoduché FFT liší ve výběru vektorů pro zpracování. Zatímco jednoduchá FFT pracuje nad dvěma n -ticema najednou, přičemž každá je z jedné poloviny vyhodnocované posloupnosti, zde je, díky zdvojení twiddle faktorů, vhodnější pracovat na jedné n -tici, ale pokrýt její větší část. Z paměti jsou totiž načteny minimálně čtyři twiddle faktory. Jejich zdvojením získáváme osm twiddle faktorů. Je tedy třeba těchto osm twiddle faktorů uplatnit. Zpracovávání více n -tic osmi twiddle faktory je sice možné, ale znamenalo by to vyšší minimální délku zpracovávané posloupnosti, nebo zesložení kódu (kdyby se délka musela nějak ověřovat a dle zjištěného provádět či neprovádět druhý výpočet).

4.6.5 Sudo-lichá dekompozice

Sudo-lichá dekompozice složí k rozkladu posloupnosti na sudou a lichou posloupnost, jejichž sečtením získáme posloupnost původní. Protože je třeba zpracovávat jednotlivá čísla indexovaná od začátku a od konce pole, navíc bez položky s nulovým indexem, jde o obtížně SIMDizovatelnou funkci. Následuje popis funkcí realizujících sudo-lichou dekompozici:

- `civ_eodec_2io` je referenční skalární funkce. Na svém vstupu očekává dvě pole čísel typu `float` každé z poloviny naplněné posloupnostmi, která se bude rozkládat. Funkce tato pole naplní sudou a lichou složkou zadaných posloupností, konkrétně
 - první polovinu prvního pole sudou složkou prvního pole
 - druhou polovinu prvního pole sudou složkou druhého pole
 - první polovinu druhého pole lichou složkou druhého pole
 - druhou polovinu druhého pole lichou složkou prvního pole s opačným znaménkem

Toto rozložení je vhodné pro další aplikaci FFT pro dokončení kalkulace reálné FFT pomocí komplexní (viz. kapitola 3.3.2). Funkce vybírá položky pro zpracování z polí jednoduchým indexováním. Pokud jsou rozvinuty cykly, je v rámci jedné iterace algoritmu dekomponováno více položek pole. Délka výstupního pole musí být soudělná s úrovní rozvinutí cyklů.

- `civ_eodec_2io_f4` realizuje výše vysvětlený rozklad vektorově. Největším problémem vektorizace tohoto výpočtu je, že do výpočtu vstupují čísla ze tří různých vektorů. Číslo na první pozici pole totiž do výpočtu vůbec nevstupuje, protože jde o stejnosměrnou složku. Vezmeme-li poslední vektor, tedy poslední čtyři čísla v posloupnosti, potřebujeme k nim získat tři čísla z prvního vektoru a jedno číslo z vektoru druhého (viz obr. 4.10). Z čísel získaných ze dvou vektorů z počátku pole potřebujeme sestavit vektor s opačným pořadím

prvků, aby souhlasil se čtyřmi čísly z konce pole. To není problém, protože vektory beztak skládáme instrukcí `shuffle`. Po provedení matematické operace musíme opět instrukcí `shuffle` výsledek transformovat zpět.



Obrázek 4.10: Vektory vybírané pro sudo-lichou dekompozici

Lichá pipeline je tak značně vytížená a vlastní výpočet je obestřen mnoha operacemi sloužící pouze pro správné uspořádání dat. Ačkoliv se nejspíše jedná o nejméně optimalizovaný algoritmus v rámci této implementace frontendu rozpoznávače řeči vůbec, je podle experimentálních měření až 3x rychlejší, než jeho skalární ekvivalent běžící na Pentiu 4.

- `civ_eodec_f4s` provádí dekompozici dvou posloupností na čtyři, z čehož dvě jsou liché a dvě sudé. Funkce je využívána pro výpočet čtyř nezávislých reálných FFT. Princip práce funkce je obdobný referenční verzi tohoto algoritmu. Rozdíl je v jednotce zpracování – zatímco funkce `civ_eodec_2io` pracuje s jednoduchými čísly typu `float`, tato funkce pracuje s celými vektory, jejichž položky jsou nezávislé. Jsou-li rozvinuty cykly, je zpracováno více vektorů najednou.
- `civ_eodec_power_f4s` provádí totéž, jako `civ_eodec_f4s`, navíc však z dekomponovaných složek přímo počítá výkonové spektrum, které je jejím výstupním parametrem. tato kalkulace je v tomto místě výhodná, protože vytěžuje především sudou pipeline, která při sudo-liché dekompozici není příliš vytížená. Dalším důvodem je ušetření čtení a zápisu do paměti – výpočet výkonového spektra po FFT běžně následuje. Takto jej můžeme vypočítat, dokud máme hodnoty v registrech. V neposlední řadě ušetříme paměťové místo – výstupní pole je pouze jedno, na rozdíl od dvou výstupů funkce předešlé. Funkce je však zatím ve vývojovém stádiu, je třeba ji odladit.

4.6.6 Výkonové spektrum

Výpočet výkonového spektra ze spektra reálného a imaginárního, které získáme z FFT, probíhá dle vzorce

$$P_i = \sqrt{Re_i^2 + Im_i^2}$$

Funkce, vyhodnocující tento vzorec jsou k dispozici dvě – referenční verze `civ_power_spec` a `civ_power_spec_f4s`. První je funkcí psanou ve standardním C, druhá je optimalizována pro SPU a zpracovává čtyři nezávislá spektra uložená prokládaně najednou.

Vzhledem k těsné závislosti dat ve vzorci je třeba počítat několik výpočtů paralelně. Provedeme tedy nejprve načtení několika vektorů z paměti (v závislosti na úrovni rozvinutí cyklů), dále zahájíme výpočet druhých mocnin, poté součtů a nakonec odmocnin. Použijeme-li dostatečné množství registrů, překladač při nejvyšší úrovni optimalizace instrukce vhodně promíchá. Nedochozí pak k jedinému čekacímu stavu. Pro zvýšení výkonu můžeme sloučit operaci umocnění se součtem díky instrukci `madd`.

Protože se při výpočtu výkonového spektra uplatňuje hlavně sudá pipeline (vykonávající matematické operace) a naopak při sudo-liché dekompozici, která této operaci předchází, se uplatňuje převážně lichá pipeline (jde o přemisťování dat v paměti), nabízí se možnost obě operace sloučit do jediné funkce. Z toho důvodu existuje funkce `civ_eodec_power_f4s`, která je však zatím ve vývojovém stádiu (není odladěna).

4.7 Logaritmus

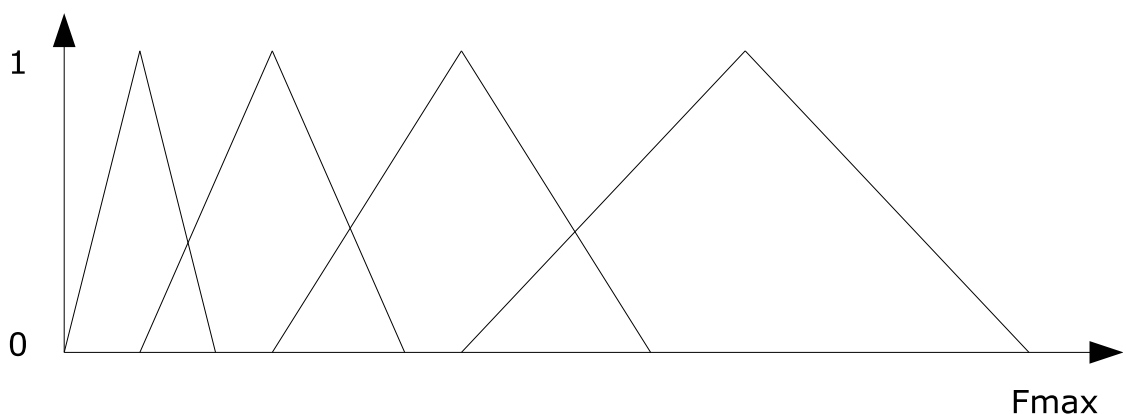
Pro výpočet logaritmu jsme použili standardní knihovny, tedy pro skalární výpočty knihovnu `m` a pro SIMDizované knihovnu `simdmath`, která je dodávána jako součást SDK. Knihovna `simdmath` provádí aproximaci logaritmu pomocí polynomu šestého řádu. Analytický nástroj `asm-visualiser` ukazuje, že jde o funkci, kde na sobě následující operace relativně těsně navazují. Proto je v rámci optimalizace vhodné provádět několik logaritmů najednou. Jelikož se jedná o `inline` funkci, není pro překladač problém instrukce po sobě jdoucích logaritmů promístit.

Dalšího zrychlení je možno dosáhnout použitím polynomu nižšího stupně, případně použitím nějaké lineární aproximace. To jsme neprovedli, protože nemáme k dispozici nástroj, kterým bychom experimentálně vyhodnotili vliv změny výpočtu logaritmu na kvalitu rozpoznávání. Analytické vyhodnocení by bylo možné provést srovnáním velikosti chyby s ostatními články zpracovávajícího řetězce.

4.8 Filtrace trojúhelníkovými filtry

Filtraci trojúhelníkovými filtry jsme rozdělili do dvou fází – nejprve je třeba předpočítat tzv. banku filtrů, což je relativně zdlouhavá operace, která je jen velmi těžko SIMDizovatelná. Ve druhé fázi pak stačí tuto banku filtrů pouze opakovaně aplikovat. Zde se již jedná prakticky pouze o násobení.

Vytvoření banky filtrů zajišťuje funkce `civ_mel_filterbank_gener`. Funkce očekává na vstupu alokované pole struktur typu `FILTER`, maximální frekvenci a počet filtrů. Filtry jsou generovány na Melovské, tudíž logaritmické ose (viz obr. 4.11). Počet struktur typu `FILTER` v předaném poli musí být roven počtu požadovaných filtrů k vygenerování. Struktury pak obsahují údaje o počáteční a středové frekvenci filtru a jeho délce. Maximální frekvence odpovídá celkové délce použité frekvenční osy. Pokud filtrujeme rámce např. 512 vzorků dlouhé, je odpovídající maximální frekvencí 512.



Obrázek 4.11: Banka logaritmicky umístěných trojúhelníkových filtrů

Funkce `civ_mel_filterbank_gener` kromě vyplněného pole struktur typu `FILTER` také alokuje nové pole, do kterého umístí parametry jednotlivých trojúhelníků, čili násobící koeficienty. Ačkoliv se trojúhelníky překrývají, v tomto alokovaném poli jsou uloženy za sebou. Překryv je řešen až při aplikaci filtru. Pro snadnější SIMDizaci jsou délky jednotlivých trojúhelníků zaokrouhlovány dolů na násobek úrovně rozvinutí cyklů. Na to je třeba dát především pozor,

pokud je F_{\max} nízká - mohlo by dojít k vytvoření filtru o nulové délce. V takovém případě funkce `civ_mel_filterbank_gener` navrátí místo vytvořené banky filtrů hodnotu `NULL`.

Aplikace filtru je prováděna funkcemi `civ_filterbank_apply` či `civ_filterbank_apply_s`. První z těchto dvou funkcí je psána ve standardním C, druhá je optimalizovaná pro SPU. Funkce očekávají na vstupu předpočítanou banku filtrů. Vlastní aplikace spočívá ve sčítání součinů části vstupního vektoru s koeficienty filtru. Pokud jsou rozvinuty cykly, probíhá sčítání paralelně v několika registrech. Po dokončení sčítání jednoho filtru je nutno provést sčítací paralelní redukci těchto registrů a výsledek uložit do výstupního pole. Funkce `civ_filterbank_apply_s` aplikuje v tomto případě optimalizaci – zatímco je prováděna silně datově závislá paralelní redukce, je zároveň aplikován následující filtr a vykonána první iterace aplikačního cyklu.

Tuto optimalizaci nelze aplikovat na poslední filtr, kdy už žádný další nebude následovat. Z tohoto důvodu obsahuje funkce podmíněný příkaz `if` s explicitně nastavenou predikcí na vykonání – čili předpokládáme, že se nejedná o poslední filtr. Nesprávná predikce nastane vždy pouze jednou, pro poslední filtr. Filtrů je tradičně 24, proto je tato predikce oprávněná.

4.9 Diskrétní kosinová transformace

Diskrétní kosinovou transformaci (DCT) je možno implementovat dvěma různými způsoby. Buď je možné vyjít z definice a pak je časová složitost výpočtu $O(N^2)$, nebo můžeme k výpočtu využít FFT s časovou složitostí $O(N \log N)$. Aplikace Fourierovy transformace však není přímočará, vstupní posloupnost je třeba nejprve modifikovat, poté aplikovat Fourierovu transformaci a výsledek opět modifikovat. Pro kratší posloupnosti je, vzhledem k optimalizaci FFT na výkon pro delší posloupnosti, vhodnější využít algoritmu o kvadratické složitosti s předpočítanou maticí koeficientů. Modul pro výpočet DCT obsahuje podpůrné funkce a funkce počítající DCT, které buď podpůrné funkce samy volají, nebo požadují jejich výstup jako jeden ze svých vstupních parametrů.

4.9.1 Výpočet DCT z definice

- `civ_dct_quad` na svém vstupu požaduje pouze vstupní posloupnost, její velikost a ukazatel na paměť, kam budou uloženy výsledky. Vstupní posloupnost totiž nelze přepsat výstupní z toho důvodu, že pro každou vstupní hodnotu je zapotřebí znát všechny vstupní. Koeficienty získané z goniometrické funkce jsou počítány za běhu podle potřeby. Každý výsledný koeficient vzniká jako součet součinů. Pokud nastavíme určitou úroveň rozvinutí cyklů, je v rámci vnitřního cyklu vypočteno více součinů. Délka vstupní a výstupní posloupnosti tedy musí být soudělná s úrovní rozvinutí cyklů. Jde o referenční verzi, která používá standardní operátory jazyka C.
- `civ_dct_quad_f4` vychází také z definice a vyhodnocuje jednu vstupní posloupnost. Ta je však tentokrát typu `vector float`. Vstupní posloupnost je tedy zpracovávána po čtyřech hodnotách. Goniometrické koeficienty jsou opět počítány průběžně. Růst argumentu goniometrické funkce je lineární, čehož se dá využít k eliminaci násobení. Stačí po spuštění funkce určit, o kolik argument roste a toto číslo přičítat. Součet je vykonáván rychleji, než součin, navíc tím eliminujeme podstatně složitější vzorec. Vnitřní cyklus lze rozvinout nastavením úrovně rozvinutí cyklů, počet vektorů ve vstupní posloupnosti musí pak být soudělný s úrovní rozvinutí. Po vykonání vnitřního cyklu sečteme položky sčítacího vektoru pomocí paralelní redukce a výsledek typu `float` uložíme do výstupního pole. Je vidět, že zde vektorové SPU ztrácí výkon, proto je lepší využít funkci `civ_dct_matrix_f4s`, pokud nám to počítaný případ umožňuje.

4.9.2 Výpočet DCT prokládaných posloupností

V této kategorii je implementována pouze jedna výpočetní funkce. Používá předpočítanou matici koeficientů. Jejím účelem je vykonávat inverzní Fourierovu transformaci krátké reálné posloupnosti, která je řešitelná transformací DCT. Krátká reálná posloupnost je v tomto případě nejčastěji kepsstrum před zpětnou Fourierovou transformací. Typicky jde o 24 čísel, takže nelze použít Cooley-Tukey algoritmus pro výpočet FFT.

- `civ_dct_matrix_gener_s` generuje matici koeficientů, které slouží jako vstupní parametr funkce `civ_dct_matrix_f4s`. Výsledná matice má velikost N^2 vektorů typu `vector float`, kde N je délka posloupnosti, která se bude pomocí DCT transformovat. Každý vektor obsahuje čtyři stejné hodnoty, protože je matice určena k výpočtu čtyř nezávislých posloupností. Vektory v matici jsou v paměti uloženy po řádcích, což napomáhá větší datové nezávislosti při násobení matice vektorem. Předpokládá se, že funkce bude spuštěna pouze jednou na PPU, koeficienty je možno používat opakovaně. Proto tato funkce používá klasické operátory jazyka C (není tedy SIMDizována) a úroveň rozvíjení cyklů na ní nemá žádný vliv.
- `civ_dct_matrix_f4s` počítá DCT čtyř prokládaných posloupností. Vychází v podstatě z definice, přičemž goniometrické koeficienty očekává na vstupu ve formě matice vygenerované funkcí `civ_dct_matrix_gener_s`. Velikost DCT a matice musí být v souladu, tzn. nelze použít matici generovanou pro jinou velikost DCT, výsledek pak není správný. Funkce provádí násobení matice vektorem, který obdrží na vstupu. Vektor je pole typu `vector float`, kde každý `vector float` obsahuje čtyři nezávislé položky čtyř nezávislých vektorů typu `float`. V první fázi výpočtu je spočten součin vektoru s prvním řádkem a výsledky jsou uloženy do paměti. Následuje cyklus, ve kterém jsou s vektorem násobeny další řádky matice. Násobení má tři fáze
 1. Načtení hodnot vektoru a matice z paměti
 2. Součin dat s vektorem a přičtení hodnoty k dosavadnímu součtu – provádí se instrukcí `madd`.
 3. Uložení vypočtených hodnot.

Součty je nutno průběžně ukládat do paměti a opět je načítat, protože v čase překladač nevíme, jak dlouhý vektor bude. Provedeme-li rozvinutí cyklů, pak je každá z výše uvedených fází rozvinuta zvlášť, čímž se zvyšuje datová nezávislost. Optimalizující překladač má pak možnost výše uvedené tři části promíchat dohromady. První a třetí krok totiž vytěžují lichou pipeline, zatímco prostřední krok sudou. Po načtení dat pro první součet je možno jej začít okamžitě provádět. Stav pipeline SPU jsou během výpočtu podobné jako při okénkovém výběru vyobrazeném na obrázku 4.3. Jako u ostatních algoritmů musí být při rozvinutí cyklů délka vstupní posloupnosti soudělná s úrovní rozvinutí. Pro kepsstrálních 24 čísel je tedy maximální úroveň rozvinutí, která je rovna osmi, vyhovující.

4.9.3 Výpočet DCT pomocí FFT

Výpočet DCT pomocí FFT je výhodný, pokud je délka transformované posloupnosti relativně velká (alespoň 128 položek).

- `civ_dct_scramble` přeuspořádává předanou posloupnost tak, aby se pro výpočet DCT dala použít FFT. Tedy položky se sudými indexy uspořádá na začátku pole, zatímco položky s lichými indexy uspořádá sestupně (vzhledem k původnímu indexu) od konce pole. Jedná se o referenční verzi funkce, jsou používány operátory jazyka C. Nastavíme-li

rozvinutí cyklu, pak je v rámci jednoho cyklu algoritmu zpracován dvojnásobek položek, než je úroveň rozvinutí cyklů. Položky jsou totiž zpracovávány po dvou, vždy sudá a lichá. Délka vstupního pole tedy musí být soudělná s dvojnásobkem úrovně rozvinutí cyklů.

- `civ_dct_scramble_f4` provádí totéž, co funkce `civ_dct_scramble`, pracuje však s vektory čtyř čísel. Rozvíjení cyklů má na tuto funkci také stejný vliv. Jelikož funkce provádí načítání a ukládání do paměti spolu s instrukcí `shuffle`, je značně vytížena lichá pipeline, zatímco sudá pouze počítá celočíselné indexy a proto je značně nevytížená. Funkce je však spíše reliktem hledání optimálního výpočtu DCT kepstra, proto není příliš optimalizovaná. Budoucí optimalizace by mohla spočívat v přidání počtu používaných registrů při větším stupni rozvinutí cyklů.
- `civ_dct_coef_gener` generuje komplexní goniometrické koeficienty, jejichž násobením získáváme z výsledku FFT požadovanou DCT. Jde o předpis $\cos(\frac{i\pi}{2N}) - j \sin(\frac{i\pi}{2N})$. Výsledky jsou tedy komplexní, čemuž odpovídají i dva výstupní parametry funkce pro reálnou a imaginární složku. Na rozdíl od `twiddle` faktorů je výhodnější mít tyto složky oddělené. Ačkoliv jde o podobný vztah jako pro `twiddle` faktory používané při výpočtu FFT, nemůžeme je využít. Argumenty goniometrických funkcí v tomto případě rostou mnohem pomaleji, využití `twiddle` faktorů by díky nepřesnosti zcela znehodnotilo výsledek. Proto je nutné mít pro tyto koeficienty zvláštní funkci. Rozvinutí cyklu má za následek zpracování více koeficientů v rámci jednoho cyklu algoritmu. Požadovaná délka DCT, pro kterou jsou koeficienty počítány musí být samozřejmě soudělná s úrovní rozvinutí cyklů.
- `civ_dct_coef_gener_f4` je pouze makrem, které vyvolá funkci `civ_dct_coef_gener`. SIMDizace výpočtu těchto koeficientů je obtížná, nepříliš efektivní a vzhledem k faktu, že se koeficienty generují většinou pouze jednou během spuštění aplikace, je SIMDizace i zbytečná. Vhodné je vygenerovat koeficienty na PPU.
- `civ_dct_log_f4` realizuje výpočet DCT pomocí FFT. Protože používá FFT, požaduje na vstupu pole `twiddle` faktorů pro výpočet FFT o délce transformované posloupnosti. Dalšími vstupy jsou transformovaná posloupnost, předpočítané goniometrické koeficienty pro DCT (funkce `civ_dct_coef_gener`) a dvojkový logaritmus velikosti transformované posloupnosti. Protože je používána implementace FFT, která očekává délku vstupní posloupnosti, která bude mocninou dvou, je nutné, aby i DCT byla délky mocniny dvou. Proto je vstupem dvojkový logaritmus velikosti.

Funkce nejprve provede přeuspořádání dat, pomocí `civ_dct_scramble_f4`. Následuje reálná FFT. Posledním krokem je vynásobení výsledku vygenerovanými koeficienty, to již provádí přímo tato funkce. Reálná a imaginární složka položek je násobena reálnou a imaginární složkou koeficientů. Míra rozvinutí cyklů ovlivňuje, kolik položek se v rámci jednoho cyklu vynásobí. Zde je prostor pro optimalizaci, bylo by vhodné v případě rozvinutí cyklů používat větší množství registrů.

4.9.4 Další funkce pro DCT

V rámci vývoje a testování vznikly dvě další funkce, které je možno využít při případné implementaci inverzní DCT pomocí FFT. Transformovaná posloupnost se nejprve vynásobí jistými goniometrickými koeficienty, čímž z reálné posloupnosti vznikne posloupnost v komplexním oboru. Dále aplikujeme inverzní FFT. Po inverzní FFT nastupuje závěrečný krok, inverzní operace k `civ_dct_scramble`. Tu implementují funkce `civ_dct_unscramble` a `civ_dct_unscramble_f4`. Jejich použití a princip jsou zcela ve shodě s `civ_dct_scramble` resp. `civ_dct_scramble_f4`, proto je zde popisovat nebudeme.

5 Testování

5.1 Principy testování

5.1.1 Testy správnosti algoritmů

Ověření, že algoritmy dávají správné výsledky jsme prováděli ručně. Jako referenční hodnoty pro základní neoptimalizované verze byly používány především výsledky obdržené použitím programu `scilab`. Po získání funkčních referenčních implementací jsme prováděli srovnávání výsledků mezi těmito funkcemi a funkcemi optimalizovanými. Bylo-li to potřeba, vrátili jsme se ke `scilabu`.

5.1.2 Výkonnostní testy

Pomocí shellového skriptu a testovací funkce implementované v souboru `civ_speech_measure.c` jsme provedli výkonnostní test každé implementované funkce, která bude při zpracování řeči používána opakovaně. Tzn. funkce sloužící pro generování `twiddle` faktorů, trojúhelníkových filtrů ap., které se volají pouze při startu programu, měřeny nebyly.

Pro vlastní měření rychlosti algoritmů na SPU jsme použili čítač SPU decrementer, ke kterému může programátor přistupovat pomocí knihovních funkcí SDK `spu_write_decrementer` a `spu_read_decrementer`, případně přes kanál SPU. Vzhledem k tomu, že v rámci jednoho měření měříme vždy jedinou funkci a nepoužíváme události SPU, nemusíme řešit zastavování a znovuspouštění čítače. Čítač po spuštění odečítá hodnotu v něm uloženou. Rychlost odečítání není pro Cell obecně pevně dána, závisí na hardwarových okolnostech. U Sony Playstation 3 se frekvence čtení blíží 80 MHz. Tento údaj je možno zjistit buď z dokumentace, nebo přímo čtením souboru `/proc/cpuinfo`, kde se jedná o parametr `timebase`.

V případě funkcí typu `f4s`, tedy funkcí pracujících s více poli najednou, bylo měřeno na čtyřnásobném počtu čísel. Měření stejného počtu dat by znamenalo čtyřikrát menší počet čísel v každém paralelně zpracovávaném poli, což je vzhledem k omezením danými rozvíjením cyklů nepřijatelné. Úroveň rozvinutí cyklů má vliv pouze na některé funkce. Jaký a na které je popsáno v kapitole 4.

5.2 Výsledky testů

V této kapitole rozebereme některé naměřené výsledky, prezentované především formou grafů. Vlastní hodnoty naměřených údajů je možno nalézt v příloze XXX.

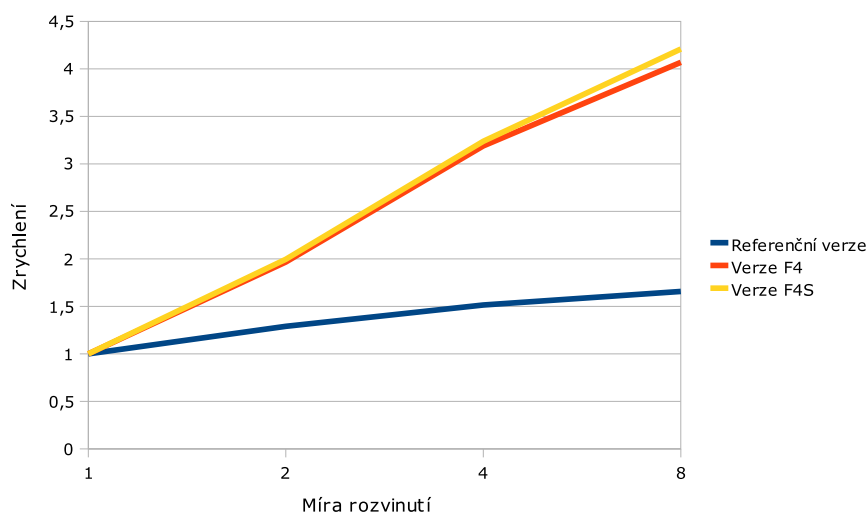
5.2.1 Vliv rozvíjení cyklů na výkon

Vliv rozvíjení cyklů závisí na implementaci algoritmu, jakým způsobem toto rozvinutí využije. My jsme nejčastěji používali dva způsoby – výpočet používající více nezávislých registrů, které nakonec spojíme paralelní redukcí a registrovů `double buffering`, kdy střídáme operace nad dvěma sadami registrů.

5.2.1.1 Větší počet registrů

Vliv rozvíjení cyklů na algoritmus výpočtu střední hodnoty je viditelný na obrázku 5.1.

Ve své referenční verzi využívá tento algoritmus pouhou redukcí počtu iterací, více registrů nevyužívá. Ve verzi používající vektorové operace dochází k výraznějšímu urychlení, protože je využíván větší počet nezávislých registrů. Verze `F4S`, která slouží pro výpočet prokládaných rámců, zaznamenává ještě o málo vyšší zrychlení, než funkce `F4`, zpracovávající jeden rámeček po

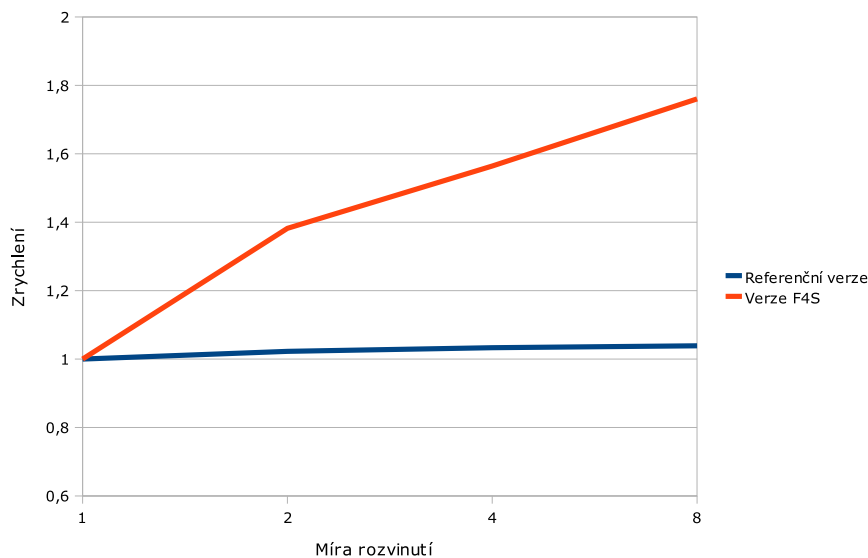


Obrázek 5.1: Vliv rozvinutí cyklů na algoritmus výpočtu střední hodnoty

čtyřech vzorcích. Důvodem je větší počet paralelních redukcí prováděných funkcí typu F4. Na tyto redukce totiž rozvinutí nemá příliš vliv a proto se celkově méně uplatní.

5.2.1.2 Dvě sady registrů

Vliv využití dvou sad registrů si ukážeme na algoritmu preemfáze. Výsledky jsou na obrázku 5.2.



Obrázek 5.2: Vliv rozvinutí cyklů na algoritmus preemfáze

Referenční verze používá rozvinutí cyklů podobně, jako referenční verze výpočtu střední hodnoty – pouze je zvýšen počet operací v jednom cyklu a tím dochází k menšímu počtu skoků. Díky datovým závislostem, které u tohoto algoritmu hrají významnou roli je výsledné zrychlení velice nízké. Naopak použitím dvou sad registrů se zrychlení zvyšuje mnohem výrazněji. Z grafu

je vidět, že nejvyšší nárůst zrychlení algoritmu F4S nastává mezi stupněm rozvinutí 1 a 2. Dvě sady registrů jsou totiž zapojeny až při stupni rozvinutí 2.

Algoritmus preemfáze trpí oproti střední hodnotě vyššími datovými závislostmi, proto není zrychlení tolik výrazné. Zvýšit účinnost rozvíjení cyklů aplikací optimalizace použité ve výpočtu střední hodnoty na preemfázi však není možné. Jedná se o principy aplikovatelné na odlišné případy algoritmů.

5.2.2 Porovnání výkonu algoritmů na různých platformách

Výkon jednotlivých algoritmů byl měřen nejen na SPU, ale také na PPU a na systému s procesorem Pentium 4, taktovaným na 2,8 GHz. Na PPU i na Pentiu 4 byly spouštěny pouze referenční verze algoritmů, psané ve standardním C, tedy bez použití speciálních sad instrukcí. Výsledné zdrojové kódy byly kompilovány překladačem `gcc` s volbami `-fomit-frame-pointer` a `-O3`. V případě Pentia 4 ještě `-march=pentium4`. (Pro PPU není tato volba potřeba, protože jsme přímo použili upravenou verzi `gcc`.)

K měření času na Pentiu 4 a PPU jsme využili funkce operačního systému (v obou případech Linux). Tyto funkce však pracují s mikrosekundami, jakožto s nejmenší jednotkou. Na SPU jsme čas měřili v nanosekundách, což se ukázalo být nezbytné, běh některých funkcí dobu jedné mikrosekundy nepřekonal. Proto jsme na Pentiu 4 i na PPU spouštěli funkci vždy několikrát, výsledný čas jsme pak vydělili počtem spuštění. Nevýhodou tohoto přístupu je, že na dobu trvání opakovaného výpočtu může mít do značné míry vliv cache a její velikost. Vzhledem k tomu, že na procesoru běží mimo jiné i systém a další procesy, rozhodli jsme se tuto chybu i se všemi ostatními akceptovat.

Vzhledem k podobnosti některých algoritmů dostáváme i podobné výsledky. Takové algoritmy nám poslouží k porovnání detailních odlišností mezi jednotlivými platformami. Velikostí dat je v následujících odstavcích míněn počet čísel typu `float`, která funkce dostala na vstup.

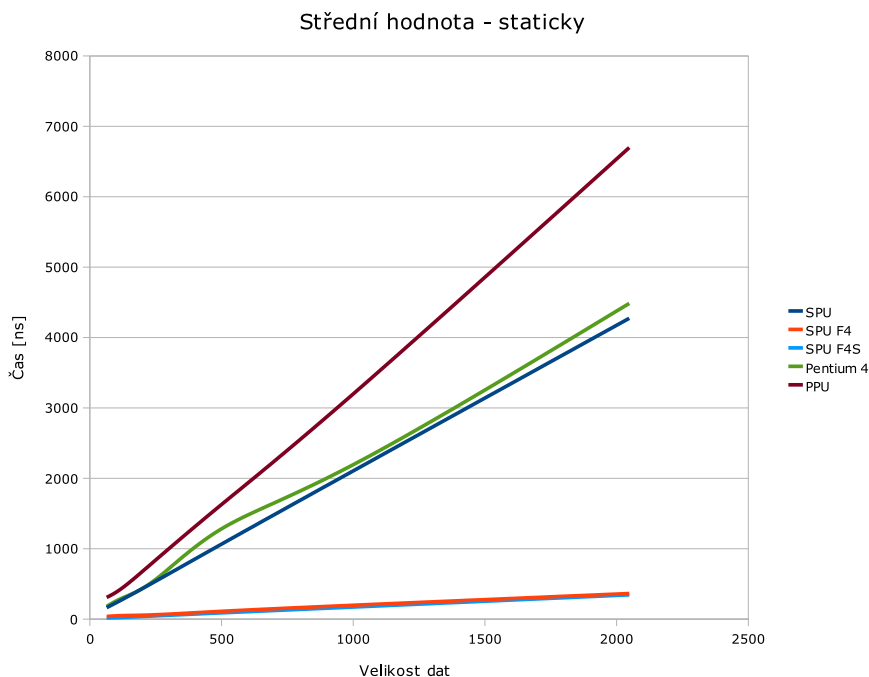
5.2.2.1 Střední hodnota a krátkodobá energie

Prvním případem podobných algoritmů jsou výpočet střední hodnoty pole implementovaný funkcemi `civ_mean_static` (viz obr. 5.3) a algoritmus výpočtu střední krátkodobé energie `civ_energy` (viz obr. 5.4). V obou případech procházíme pole, přičemž buď přímo sčítáme jeho prvky, nebo prvky nejprve umocňujeme na druhou a až poté sčítáme. Průběhy se liší pouze málo, projevuje se zde i dále patrný trend, že výkon Pentia negativně ovlivňuje násobení, zatímco u PPU není pokles výkonu v takovém případě tolik markantní. Umocňování na druhou při výpočtu střední krátkodobé energie se u PPU totiž projeví mnohem méně, než u Pentia. SPU počítá krátkodobou energii pomaleji díky hluboké pipeline. Jednoznačně nejrychlejší je výpočet optimalizovanými algoritmy, přičemž algoritmus typu `f4s` nemusí provádět závěrečnou paralelní redukci, proto je o málo rychlejší (pro 2048 prvků v poli činí rozdíl cca 25 ns).

5.2.2.2 Přičítání konstanty a okénkový výběr

Dalšími algoritmy s podobnými výsledky jsou přičítání konstanty k prvkům pole, implementované funkcemi `civ_addconst` (viz obr. 5.5) a okénkový výběr `civ_win_sel` (viz obr. 5.6). Je grafů je vidět, že násobení potřebné u okénkového výběru snižuje výkon Pentia, zatímco výkon PPU oproti přičítání konstanty naopak vzrostl. Jde však nejspíše o chybu měření způsobenou během ostatních procesů na PPU, násobení není rychlejší než sčítání. Pohledem na křivky týkající se výkonu SPU vidíme, že okénkový výběr trvá na SPU delší dobu. Větší zpomalení oproti sčítání je způsobeno potřebou načítání koeficientů okénka z paměti, což je tradičně největší brzdou SPU programů.

Srovnáme-li časy běhu těchto a předešlých dvou algoritmů (kapitola 5.2.2.1), zjistíme, že druhé



Obrázek 5.3: Algoritmus výpočtu střední hodnoty pole

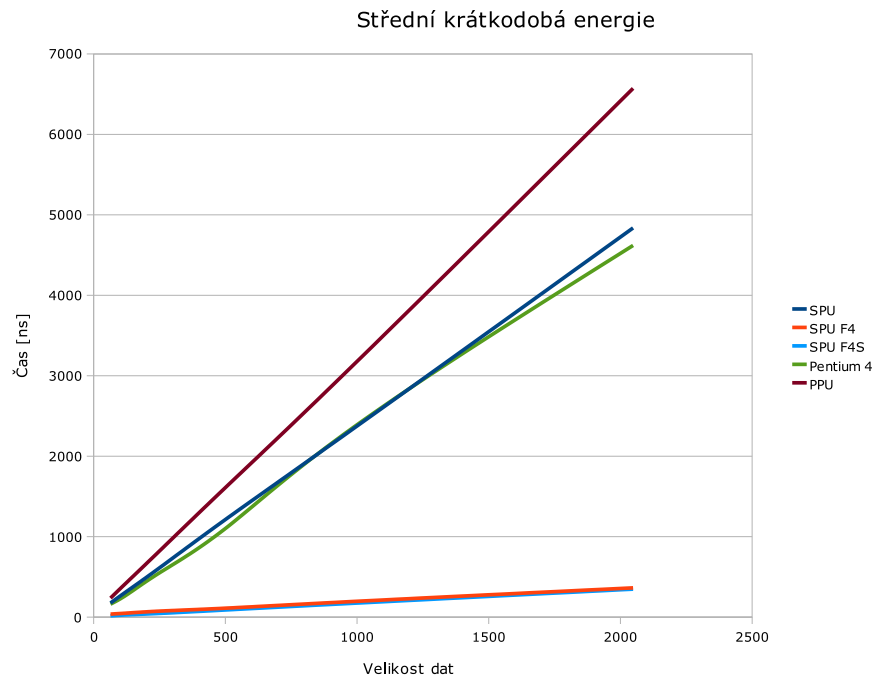
dva algoritmy jsou pomalejší, ačkoliv provádějí stejné matematické operace. Zpomalení je způsobeno tím, že zatímco výsledkem střední hodnoty a střední krátkodobé energie je jedno číslo, výsledkem přičítání konstanty a okénkového výběru je celé pole, které je nutno uložit do paměti. Zde je také příčina, proč je u druhých dvou algoritmů neoptimalizovaný běh na SPU horší, než běh na PPU.

5.2.2.3 Dynamická střední hodnota a preemfáze

Algoritmy výpočtu střední hodnoty pomocí derivačního článku a preemfáze jsou zdánlivě podobné. Preemfáze však pracuje pouze se vstupními vzorky, zatímco výpočet střední hodnoty potřebuje k další práci předcházející výsledek. Srovnáme-li grafy rychlosti výpočtu pro různá data pro dynamickou střední hodnotu (obr. 5.7) a pro preemfázi (obr. 5.8) vidíme, že nejpomalejší je výpočet preemfáze na PPU a na SPU bez patřičných optimalizací. Důvodem tohoto výsledku je nutnost načítání jednotlivých čísel z paměti (u SPU) nebo pomalá práce s pamětí vůbec (PPU).

Naopak optimalizovaný algoritmus pro SPU je mnohem rychlejší, než optimalizovaný algoritmus pro SPU při výpočtu dynamické střední hodnoty. Optimalizace použitá v tomto algoritmu spočívá v používání dvou sad registrů, registrový double-buffering. Jeho výsledkem je, že nevzniká žádné čekání na načtení dat z paměti. Protože počítáme čtyři prokládané proudy dat, nejsou zde ani závislosti mezi položkami vektorů, pracujeme s celými vektory, což je pro SPU optimální. Algoritmus pro dynamickou střední hodnotu potřebuje ke své činnosti předcházející výsledky, čímž vznikají prodlevy.

Nejpomalejší platforma pro výpočet dynamické střední hodnoty je, trochu překvapivě, Pentium 4. Nejspíše je na vině úzká datová závislost mezi jednotlivými výpočty, která vyřadí použitelnost velice dlouhé pipeline Pentia 4. PPU má nejspíše pipeline kratší a proto závislostmi tolik netrpí. Navíc běží na vyšší hodinové frekvenci. Rychlost neoptimalizovaného algoritmu na SPU je překvapením. Analýzou pomocí `asm-visualiseru` jsme objevili velké množství čekacích stavů



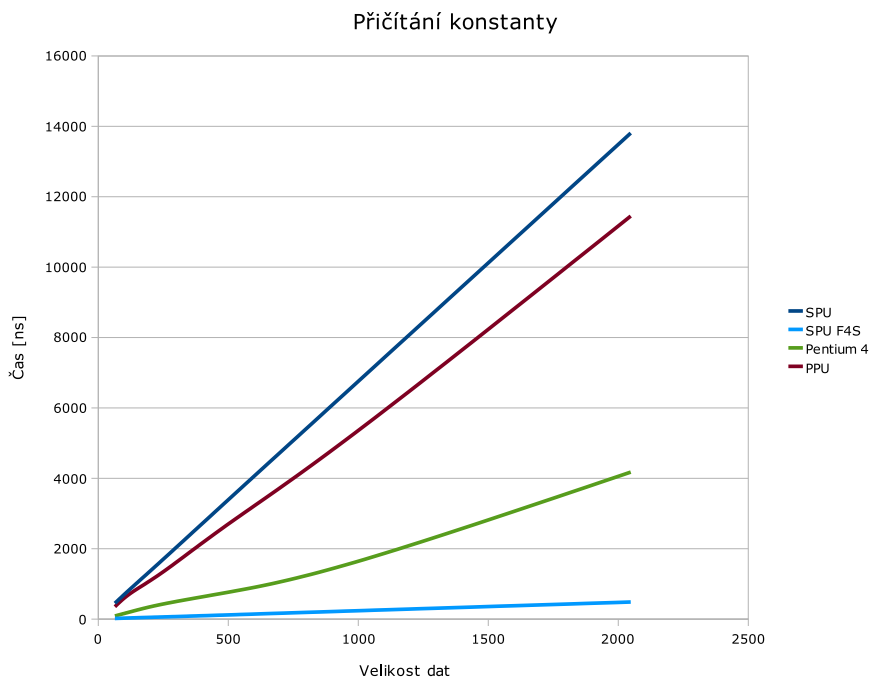
Obrázek 5.4: Algoritmus výpočtu krátkodobé energie

obou pipeline, která nenasvědčují tomu, že by SPU mohlo být takto rychlé.

5.2.2.4 Diskrétní kosinová transformace

Diskrétní kosinovou transformaci jsme implementovali třemi různými způsoby. Jednak referenční verzi, která vychází přímo z definice a potřebné koeficienty počítá za běhu, dále verzi používající FFT, vhodnou pro transformaci většího množství dat a nakonec verzi využívající násobení maticí předpočítaných koeficientů. Poslední uvedená verze je vhodná pro transformaci menších rozměrů, jednak z důvodů paměťových, kdy je třeba uložit předpočítanou matici a dále z důvodů rychlostních. Asymptotická složitost maticového násobení je vyšší, než výpočet FFT. Závislost času výpočtu na velikosti dat pro DCT využívající FFT je natolik podobná průběhu naměřenému na FFT, že není potřebné ji zde zvlášť uvádět. Na obrázku 5.9 je možno pozorovat závislost času výpočtu na velikosti dat pro implementaci z definice. Na první pohled je vidět, že výsledné křivky očekávané paraboly nepřipomínají ani pro nižší velikosti transformovaných polí. Je to způsobeno velkou lineární složkou složitosti, která vychází z výpočtu kosinů. DCT pomocí matice bylo vzhledem k paměťovým nárokům nutno počítat na menších instancích. Grafy 5.9 a 5.10 proto nelze srovnávat z hlediska času. Na druhém z grafů je však vidět kvadratický trend složitosti násobení vektoru maticí.

Porovnáme-li mezi sebou různé platformy, vidíme, že v násobení maticí vektorem dopadlo nejhůře PPU. Jde nejspíše o tradiční pomalou práci s pamětí. Výkon Pentia 4 přičítáme použití integrovaného koprocesoru. V grafu 5.10 je uvedena křivka naměřená na Pentiu bez použití rozvíjení cyklů. Ačkoliv u většiny algoritmů bylo rozvíjení cyklů na této platformě spíše ke škodě věci, zde je Pentium při úrovni rozvinutí rovné osmi až o třetinu rychlejší. Nepříliš vysoký výkon SPU viditelný na grafu 5.9 je dán nepříznivým načítáním jednotlivých čísel z paměti, ale hlavně výpočtem kosinů, které jsou aproximovány polynomem počítaným pomocí těsně závislých instrukcí.



Obrázek 5.5: Algoritmus přičítání konstanty k prvkům pole

5.2.2.5 Pomocné funkce pro výpočet DCT pomocí FFT

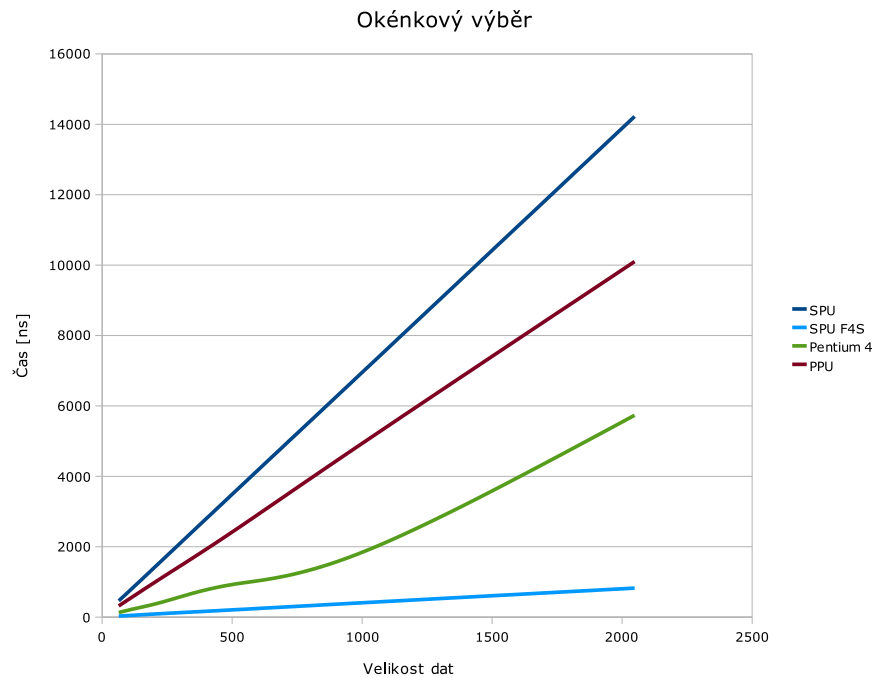
Funkce `civ_dct_scramble` a `civ_dct_unscramble` mají vzájemně inverzní význam. Neprovádějí žádné výpočty, pouze reorganizují data v paměti, aby nebylo třeba psát novou FFT rutinu, která by data v potřebném pořadí přímo načítala. Naměřené průběhy jsou zcela shodné co se týče vzájemného poměru výkonů platform i časů jednotlivých výpočtů. Proto zde uvádíme pouze jediný graf, zastupující oba (viz obr. 5.11).

Projevují se zde obvyklé trendy - práce SPU s jednotlivými čísly je zcela neefektivní. Pentium 4 je značně výkonnější než PPU. Na vině může být v tomto případě rychlejší paměťová sběrnice v PC. Prováděné operace jsou na SPU vykonávány především lichou pipeline, která zajišťuje načítání a ukládání z/do paměti a instrukci `shuffle`. SIMDizace zde tedy také není tolik účinným řešením, jako jinde protože je využita jen jedna pipeline. To je vidět na malé rychlostní ztrátě Pentia oproti SIMD verzi běžící na SPU. Implementace pro zpracování několika proudů dat najednou není k dispozici. Pro výpočet DCT takovýchto dat slouží DCT založená na násobení maticí.

5.2.2.6 Výpočet logaritmu

Ačkoliv se jedná o podobný scénář, jako u algoritmů v kapitole 5.2.2.2, dominuje v tomto případě výpočet logaritmu a vlastní práce s pamětí takový vliv nemá. Jedná se totiž o mnohem složitější operaci. Ačkoliv výpočet neoptimalizované verze logaritmu na SPU trpí značnými meziinstrukčními závislostmi, nedopadlo SPU nejhůře. Důvod pomalosti PPU nám není znám. Může se, vzhledem k délce výpočtu, jednat o vliv běhu operačního systému. Tuto teorii však příliš nepotvrzuje výsledek Pentia 4, které je cca pětikrát rychlejší než PPU. Je možné, že má Pentium k dispozici nějaké převodní tabulky, na rozdíl od PPU.

Operace nad čtyřmi nezávislými proudy dat je již tradičně nejrychlejší, logaritmus nebyl výjimkou. Analýza assembleru SPU ukazuje, že v rámci výpočtu nejsou prakticky žádné meziin-



Obrázek 5.6: Algoritmus okénkového výběru

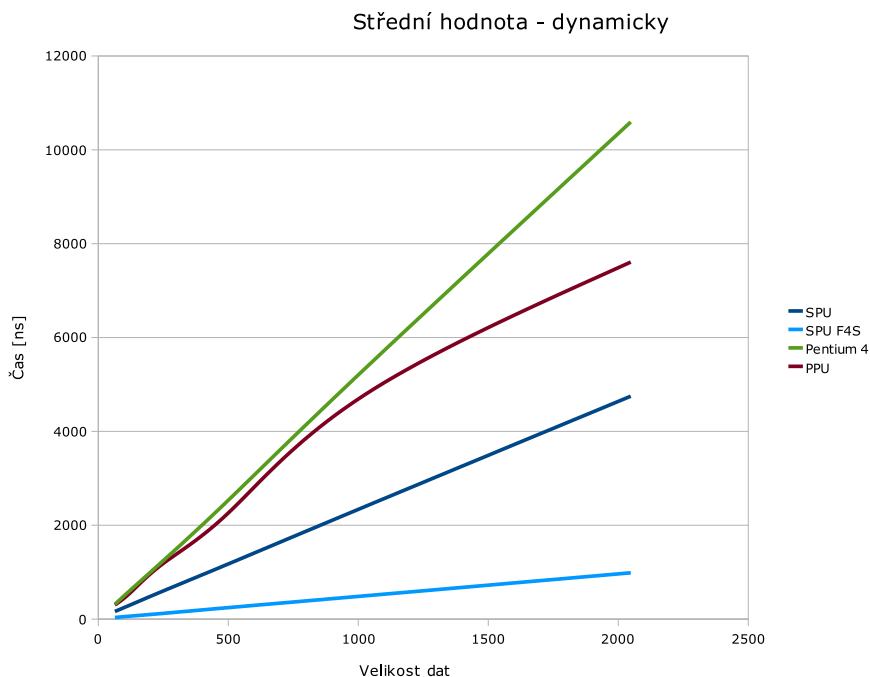
strukční závislosti, ačkoliv SPU není plně vytíženo, v některých okamžicích pracuje pouze jedna ze dvou pipeline.

5.2.2.7 Počet průchodů signálu nulou

Na datech naměřených na algoritmu výpočtu počtu průchodů signálu nulou, implementovaným funkcemi `civ_zerocross` (viz obr. 5.13) je vidět, jak je důležitá optimalizace pro SPU. Výpočet neoptimalizovaným algoritmem na SPU je totiž suverénně nejhorší, zatímco optimalizovaný algoritmus je až šedesátkrát rychlejší. Neoptimalizovaný algoritmus totiž spočívá v postupném průchodu pole čísel typu `float` a porovnávání jejich znamének. To je postup pro SPU zcela nevhodný - SPU musí extrahovat z vektoru jednotlivá čísla. Díky nerozvinutému cyklu se toto navíc děje opakovaně pro každé číslo. Následuje podmínka lišících se znamének, jejíž výsledek nelze u náhodného šumu (kterým bylo testování provedeno) předvídat. Špatná predikce je na SPU značně penalizována. Pravdou je, že se tento problém v praxi vyskytne pouze u sykavek. Dále je znát, že oproti dříve zmíněným algoritmům dopadlo velice dobře Pentium, na kterém běžel stejný kód, jako na PPU. Jendoznačně nejrychlejší je však optimalizovaná verze na SPU, která nepoužívá podmíněný příkaz `if`, ale počítání průchodů nulou řeší pomocí logických operací nad čtyřmi čísly najednou. Logické operace trvají na SPU dva až tři taktů (oproti 12 taktům penalizace za špatnou predikci a cca 8 taktům při načítání každého jednoho čísla z paměti).

5.2.2.8 Filtrování trojúhelníkovými filtry

Změřili jsme časové závislosti algoritmu pro filtrování trojúhelníkovými filtry. Měřili jsme jednak závislost na velikosti dat a také závislost na počtu filtrů pro velikost dat 512 čísel typu `float`. Protože jsou koeficienty filtru předpočítané, jde prakticky pouze o násobení a práci s pamětí. Se vzrůstajícím počtem dat narůstá lineárně čas výpočtu. Rychlost nárůstu odpovídala především schopnostem platformy při práci s pamětí, nejpomalejší v této disciplíně bylo tradičně



Obrázek 5.7: Výpočet střední hodnoty dynamicky

PPU. Pentium 4 a běh neoptimalizovaného algoritmu na SPU byly srovnatelně rychlé, což je u Pentia 4 poněkud překvapivé. Víme však, že násobení reálných čísel nepatří na této platformě k nejrychlejším operacím. Naměřené lineární průběhy zde neuvádíme.

Vztah mezi platformami se zachoval ve velmi podobné míře i při měření časové závislosti na počtu filtrů. Výsledné průběhy (obr. 5.14) jsou logaritmické. To je způsobeno logaritmickou délkou Melovské osy, na které jsou filtry umístěny. Výsledek je tedy co do průběhu shodný s očekávaným.

5.2.2.9 FFT

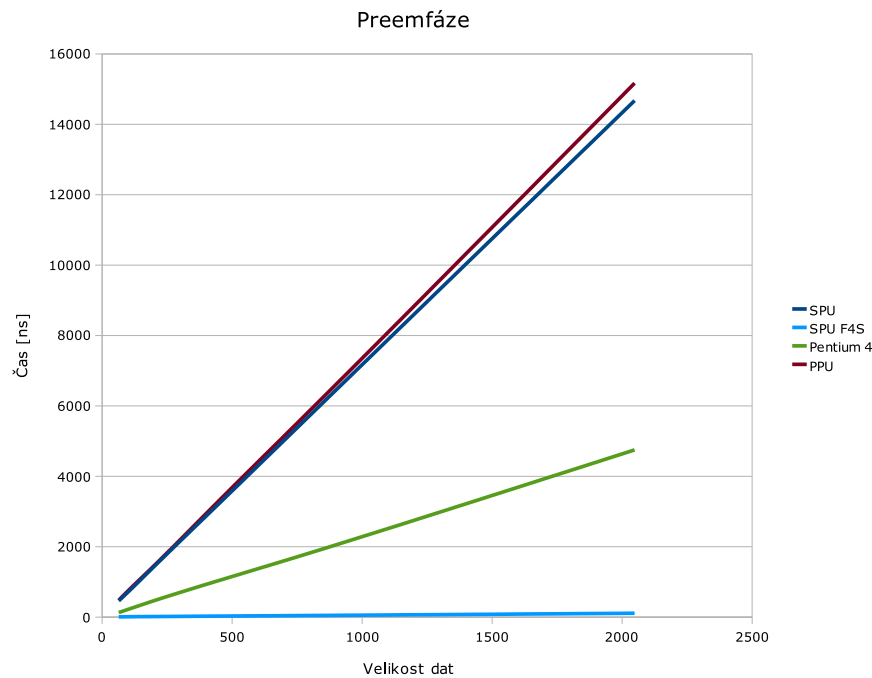
Na grafu 3.9 jsou naměřené výkony jednotlivých platform při výpočtu FFT. Ačkoliv je FFT algoritmem s časovou složitostí $O(N \log N)$, jsou průběhy spíše lineární. V případě naší implementace to je způsobené sudo-lichou dekompozicí, jejíž složitost je lineární a nejspíše má na výsledný průběh značný vliv.

Verze F4 reprezentuje výpočet FFT pomocí funkce `civ_fft_real_2_f4`, zatímco verze F4S, která je výkonnější reprezentuje výpočet čtyř reálných FFT pomocí dvojitě komplexní FFT. Výkonnostní rozdíl je s největší pravděpodobností dán tím, že u verze F4S nemusíme provádět závěrečný krok FFT, výsledek získáváme ihned po dokončení sudo-liché dekompozice.

Na Pentiu 4 jsme narozdíl od ostatních měření použili optimalizovanou reálnou FFT z knihovny FFTW (viz [3]). Knihovnu FFTW jsme přeložili s podporou SSE2, proto Pentium nebylo tolik handicapováno, jako u ostatních měření. I přesto je však námi implementovaný algoritmus více než dvakrát rychlejší.

5.2.3 Celkový výkon

Na obrázku 5.16 je srovnání Pentia 4 a optimalizované verze algoritmu pro SPU při výpočtu keprstra. Měření byly všechny navazující algoritmy od počátku frontendu, tedy výpočet a ode-



Obrázek 5.8: Algoritmus pro preemfázi

čtení střední hodnoty, preemfáze, okénkování, FFT, logaritmus, banka trojúhelníkových filtrů a DCT.

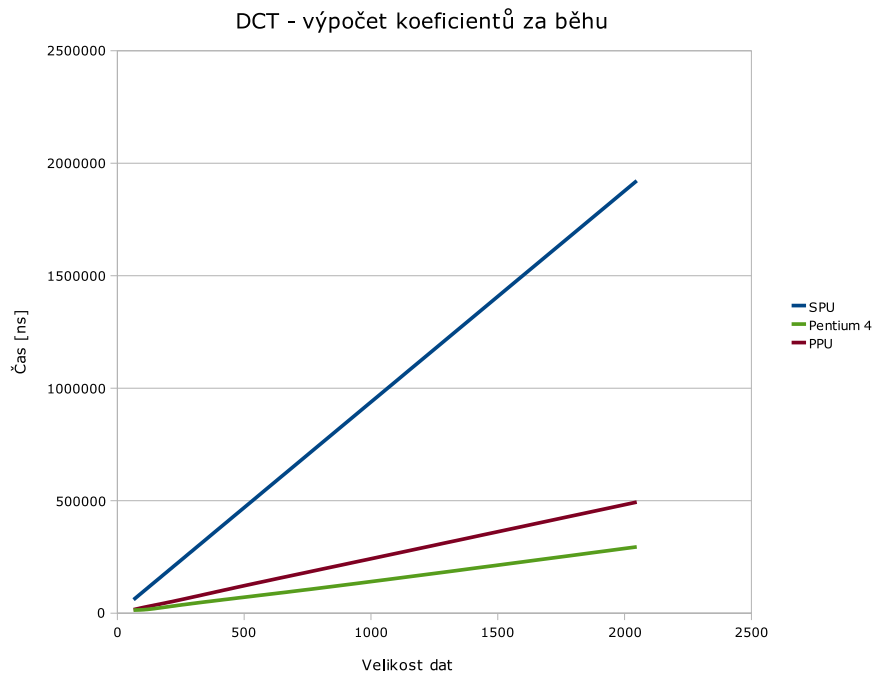
Při měření rychlosti na Pentiu 4 se vyskytl problém s výpočtem logaritmu. Pokud jsme použili knihovnu FFTW, která byla přeložena s podporou instrukcí SSE2 (jedná se o sadu vektorových instrukcí dostupných na Pentiu), byl výpočet logaritmu mnohem pomalejší, než v případě samostatného měření.

Nejspíše se jedná o zablokování koprocesoru, který se pro výpočet logaritmu běžně používá, instrukcemi SSE2. Překladač neumožnil využít v jedné z funkcí tyto instrukce a ve druhé koprocesor. Pentium 4 však není předmětem této práce, proto zde ani neuvádíme důkladný rozbor této situace.

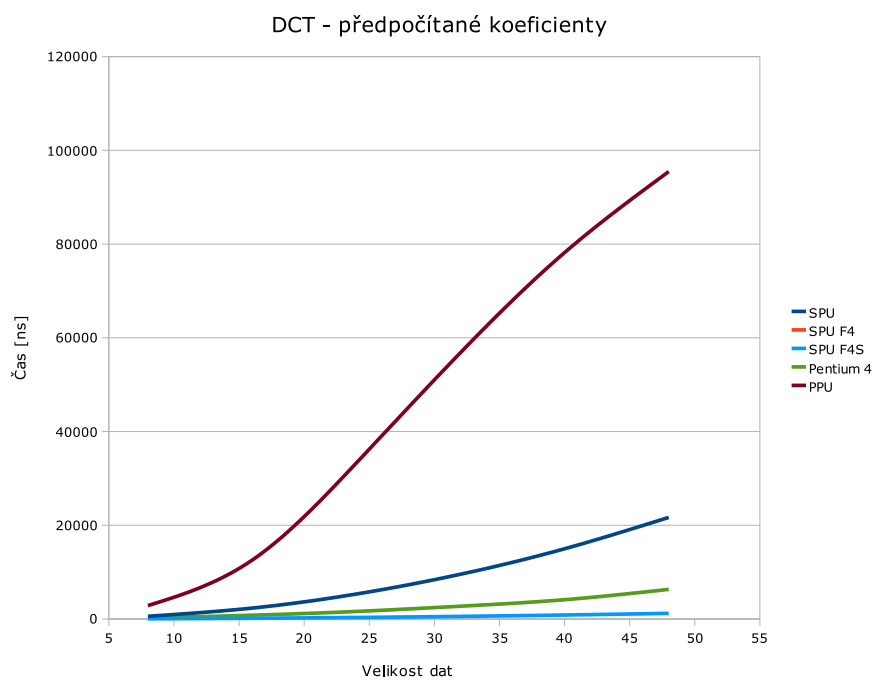
Naměřená data jsme získali změřením řetězce algoritmů frontendu bez logaritmu a následným přičtením času naměřeného při samostatném měření logaritmu.

Z naměřených údajů plyne, že naše implementace na SPU je až 3x rychlejší, než implementace na Pentiu 4. Pravdou je, že algoritmy pro Pentium 4 nebyly nijak optimalizovány (kromě FFT). Podíváme-li se na výše uvedené srovnání jednotlivých algoritmů, je zřejmé, že hlavní roli co do výkonu hraje právě FFT. Ostatní algoritmy jsou totiž často až 30x rychlejší, než je Pentium 4, jediné FFT vykazuje zrychlení 2,3x. Je to způsobeno právě využitím knihovny FFTW.

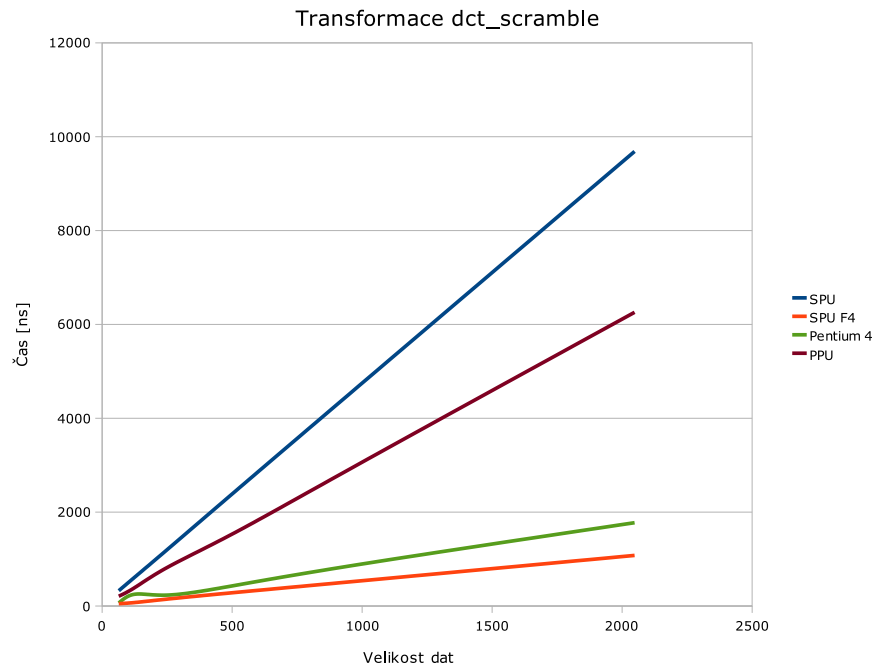
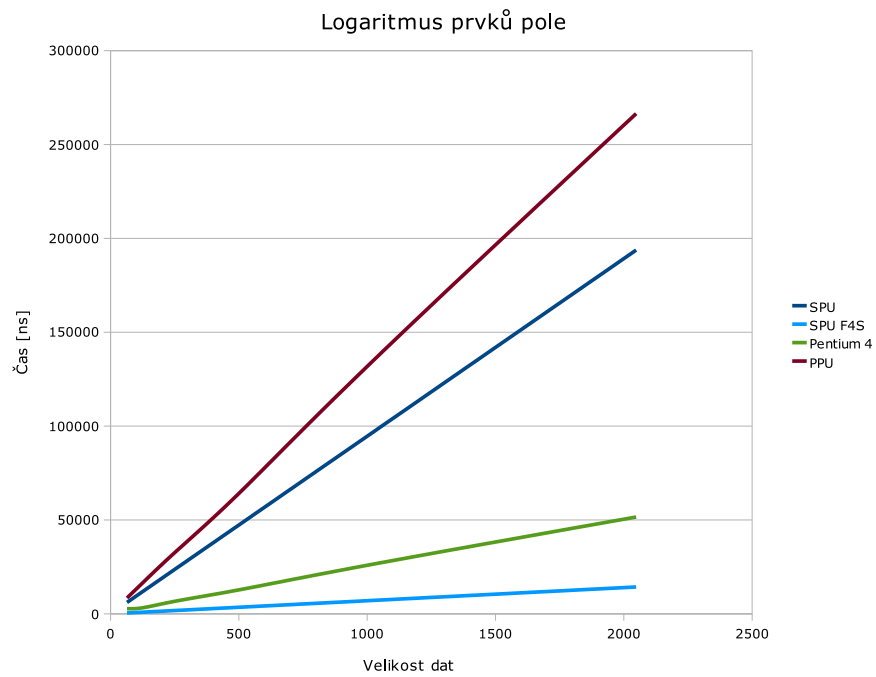
Uvážíme-li vzorkování 8 kHz, které je dle Nyquistova teoremu dostatečné pro telefonní kvalitu hovoru, jejíž šířka pásma je 4 kHz, může SPU zpracovávat cca 2700 nezávislých řečových kanálů na jednom SPU najednou.



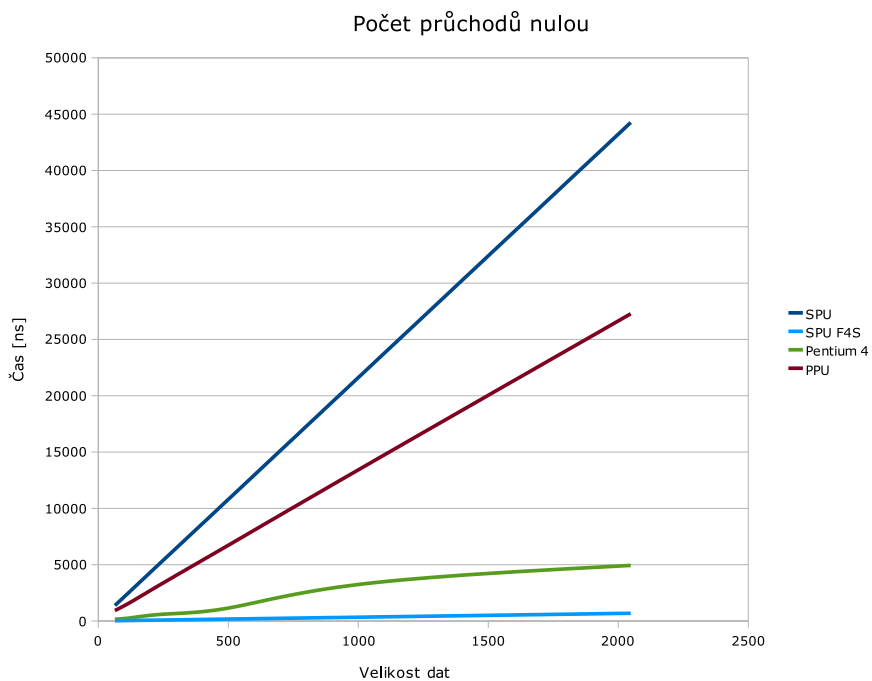
Obrázek 5.9: Diskrétní kosinová transformace s koeficienty počítanými za běhu



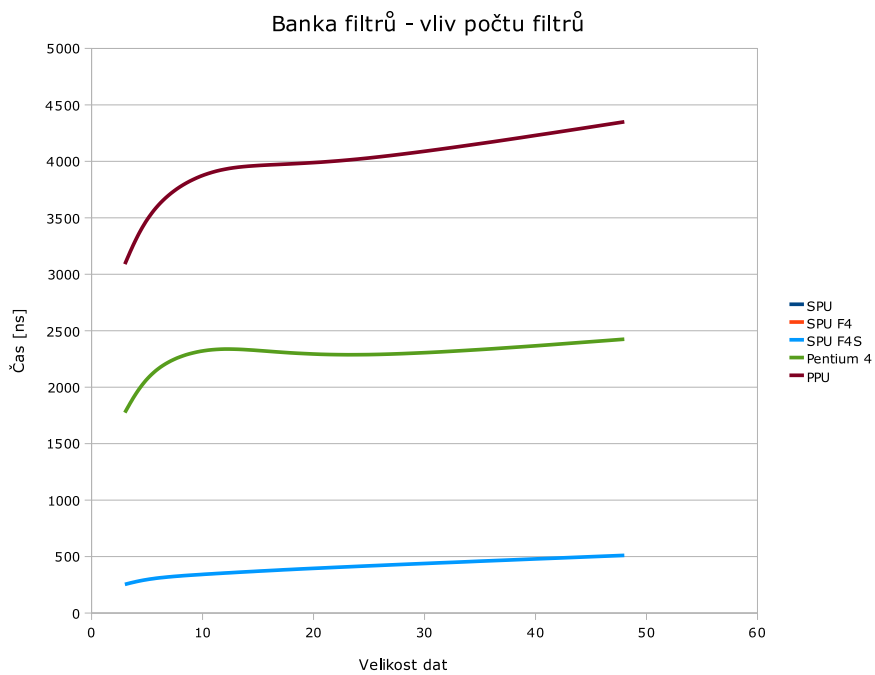
Obrázek 5.10: Diskrétní kosinová transformace počítaná maticovým násobením

Obrázek 5.11: Algoritmus `dct_scramble` užívaný pro výpočet DCT pomocí FFT

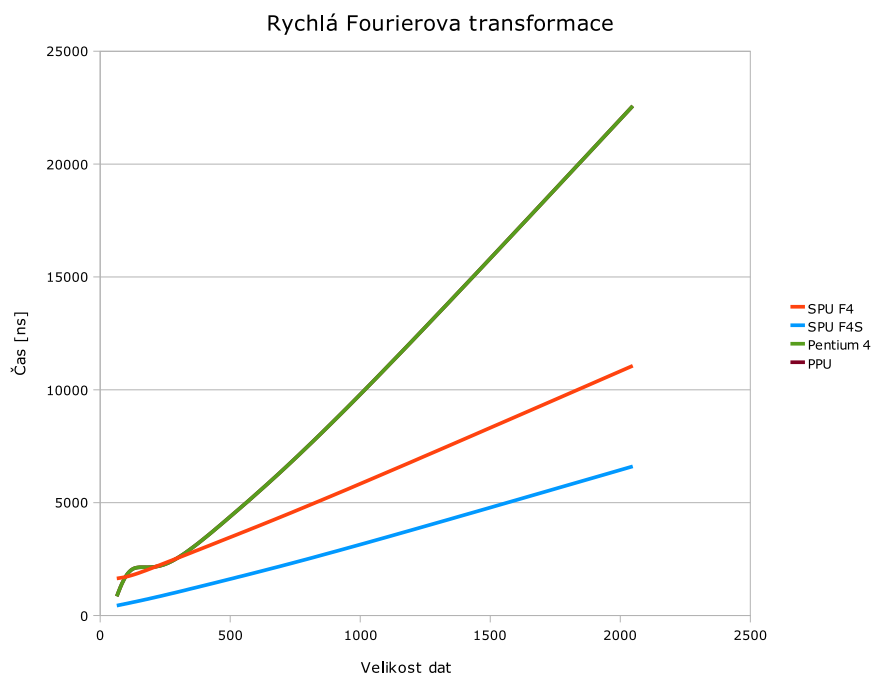
Obrázek 5.12: Algoritmus výpočtu logaritmu



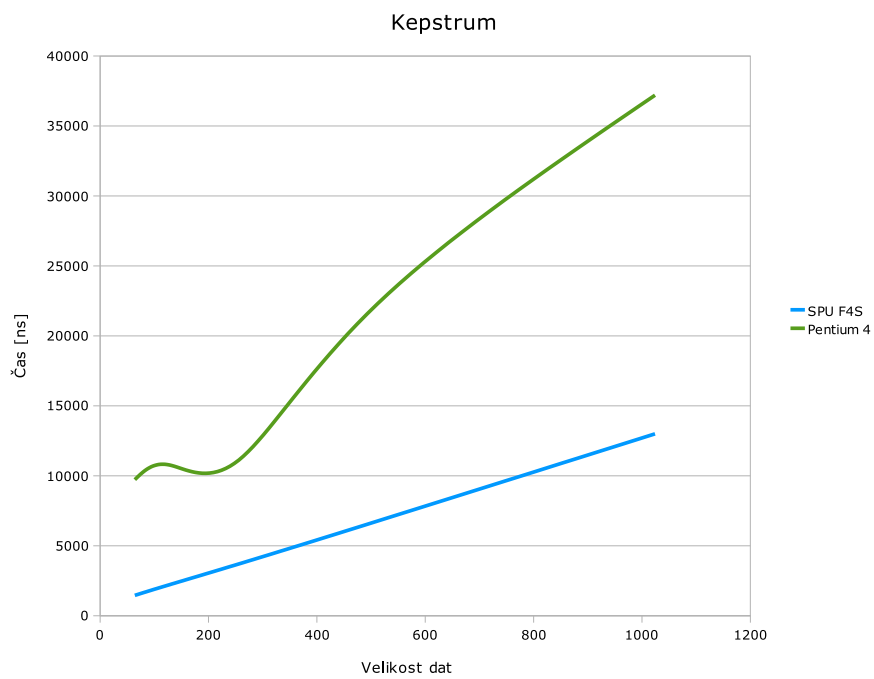
Obrázek 5.13: Algoritmus výpočtu počtu průchodů signálu nulou



Obrázek 5.14: Filtrování trojúhelníkovými filtry, závislost na počtu filtrů



Obrázek 5.15: Porovnání výkonu platforem při FFT



Obrázek 5.16: Srovnání rychlosti výpočtu kepstra

6 Závěr

Implementace rozpoznávače řeči na procesoru Cell broadband engine je při použití vhodných optimalizací velice efektivní, efektivnější, než na běžně rozšířených platformách. Porovnání s Pentiem 4 taktovaným na 2,8 GHz ukázalo, že i při optimalizovaných algoritmech je SPU i 2,5x rychlejší. Srovnáme-li běh neoptimalizovaného algoritmu na Pentiu 4, který vznikne například vykonáváním programu přeloženého pro procesor 80386, a optimalizované verze na SPU, pohybuje se zrychlení často okolo třicetinásobku. Výkon, který procesor poskytuje se tak spíše hodí na zpracování archivu nahrávek, kdy můžeme zpracovávat mnoho kanálů najednou, než pro rozpoznávání řeči jednoho řečníka v reálném čase.

Rozsah přímo implementovaných algoritmů jsme po dohodě s vedoucím práce stanovili na algoritmy používané ve frontendu rozpoznávače řeči. Výkon implementovaných algoritmů umožňuje zpracovávat až 66,5 milionu vzorků za vteřinu, což při telefonní kvalitě hovoru odpovídá 2700 řečovým kanálům v reálném čase na jedné jednotce SPU. Při využití všech SPU na plnohodnotném procesoru Cell s osmi jádry SPU to znamená možnost zpracovat přes 20.000 řečových kanálů v reálném čase.

Vzhledem k naměřenému výkonu nemá v tuto chvíli význam implementovat přenosy dat mezi hlavní pamětí a lokální pamětí jednotlivých SPU, případně rozkládat výpočet na více SPU a organizovat komunikaci mezi nimi. Jedno SPU počítající kepstrum vyžaduje při plném výkonu přísun dat rychlostí 266 MB/s. Už tato hodnota je na hranici možností dnešních disků, odkud by data byla nejspíše čtena. Norma Sata-II poskytuje reálnou přenosovou rychlost maximálně 300 MB/s (viz [12]), ta se však uplatní pouze, pokud nebude docházet k dalším zdržením, daným především mechanickou podstatou disků. Přenos po síti či přes jiné rozhraní naráží na podobné limity.

Pokud jde o implementaci celého rozpoznávače řeči, je vhodné algoritmy implementované v rámci této práce doplnit porovnávací jednotkou, která tvoří další stupeň rozpoznávače, a celý tento blok provozovat na jednom SPU. Mimo jiné tím uspoříme kapacitu mezijádrové propojovací sběrnice EIB. Tu pak můžeme využít například k dynamickému načítání dat ze slovníku, protože 256 KB paměti, které má SPU k dispozici, pro celý slovník nestačí. To však již závisí na návrháři celkové aplikace.

Budoucí práce na rozpoznávači řeči pro procesor Cell tedy spočívají především v propojení vstupní vrstvy, ať už půjde o mikrofon, či dekodér načítající data z disku a implementaci celého zpracovávajícího řetězce na SPU. Tím bude dán datový formát, který bude nutno přenášet mezi hlavní pamětí a lokálními pamětmi SPU. K tomuto účelu je možné použít akcelerační knihovnu ALF, dodávanou v rámci SDK, pravdou však je, že dosavadní verze ještě zcela transparentně neřeší problém efektivního přenosu překrývajících se rámců.

Platforma Cell broadband engine nabízí vysoký výpočetní výkon a díky většímu počtu jader a vysoké propustnosti vnitřní sběrnice (až 307 GB/s) disponuje také vysokou datovou propustností. Ideální úlohou pro procesor Cell jsou operace nepřekračující rámec hlavní paměti. Takovými aplikacemi mohou být hry nebo například neuronové sítě. Zpracováváme-li velké množství dat, narážíme na nízkou datovou propustnost současných periférií (vzhledem k vnitřní propustnosti procesoru). V takových případech je Cell vhodný spíše pro složitější aplikace. Kvalitní rozpoznávač řeči může být jednou z nich.

7 Literatura

- [1] D. A. Brokenshire. Maximizing the power of the cell broadband engine processor: 25 tips to optimal application performance, 2006. <http://www.ibm.com/developerworks/library/pa-celltips1/>.
- [2] N. Drakos and R. Moore. Fast DCT algorithm. <http://fourier.eng.hmc.edu/e161/lectures/dct/node2.html>.
- [3] FFTW library. <http://www.fftw.org/>.
- [4] H. Hermansky and N. Morgan. Rasta processing of speech. *IEEE transaction on Speech and Audio Processing*, 2(4):587–589, 10 1994. <http://lectures.idiap.ch/winter2005-2006/ic-48/courseslide/rasta.pdf>.
- [5] HTK book. <http://htk.eng.cam.ac.uk/docs/docs.shtml>.
- [6] *Data-driven design of Front-End Filter Bank for Lombard Speech Recognition*, 2006. http://noel.feld.cvut.cz/speechlab/publications/046_icslp06.pdf.
- [7] J. Psutka. *Komunikace s počítačem mluvenou řečí*. Academia, 1995.
- [8] S. W. Smith. *The scientist's and engineer guide to digital signal processing*, chapter 12, pages 240–241. California Technical Publishing, 1997. <http://www.dspguide.com>.
- [9] S. W. Smith. *The scientist's and engineer guide to digital signal processing*, chapter 12, pages 228–230. California Technical Publishing, 1997. <http://www.dspguide.com>.
- [10] S. W. Smith. *The scientist's and engineer guide to digital signal processing*, chapter 12, pages 238–242. California Technical Publishing, 1997. <http://www.dspguide.com>.
- [11] Wikipedia – IEEE-754, 1985. http://en.wikipedia.org/wiki/IEEE_floating-point_standard.
- [12] Wikipedia – Serial ATA. http://en.wikipedia.org/wiki/Serial_ATA.
- [13] Wikipedia – přehled okénkových funkcí. http://en.wikipedia.org/wiki/Window_function.
- [14] J. Černocký. *Zpracování řečových signálů – studijní opora*. FIT VUT Brno, 12 2006. http://www.fit.vutbr.cz/~cernocky/speech/opora/zre_opora.pdf.
- [15] J. Černocký. *Zpracování řečových signálů – studijní opora*, chapter 5.8, page 46. FIT VUT Brno, 2006. http://www.fit.vutbr.cz/~cernocky/speech/opora/zre_opora.pdf.

A Přiložené CD

Vytvořené zdrojové kódy, testovací skripty či tento text v elektronické podobě jsou dodávány na příloženém CD, které je nedílnou součástí této práce.

A.1 Obsah příloženého CD

Adresář	Popis obsahu
doc/	Doxygenem vygenerovaná dokumentace k optimalizované implementaci frontendu rozpoznávače řeči
frontend/speech_ppu/	Implementace generátorů pomocných tabulek na PPU + funkce <code>main</code> jednoduše spouštějící kód na SPU
frontend/speech_spu/	Implementace frontendu na SPU, referenční i optimalizovaná
frontend/speech_ppu_test/	Skalární implementace frontendu na PPU pro testovací účely
frontend/speech_x86/	Skalární implementace frontendu pro platformu x86 pro testovací účely
tests/	Testy a výsledky předešlých testování
text/	Elektronická verze tohoto textu
unfinished/alf/	Nedokončené funkce využívající pro přenos dat knihovnu ALF dodávanou v rámci Cell SDK
unfinished/ppu_simd/	Zastaralé a neodladěné SIMD verze algoritmů frontendu rozpoznávače řeči používající instrukce AltiVec
utils/docgener/	Skript pro vygenerování dokumentace implementace frontendu rozpoznávače řeči na SPU a konfigurační soubory doxygenu
utils/wavgen/	Malý program pro generování jednoduchých průběhů, vhodný na testování

A.2 Instalační příručka

Instalace spočívá ve zkopírování obsahu CD do libovolného adresáře na cílový stroj.

Požadavky:

- UNIXový operační systém cílového stroje nebo prostředí, které jej emuluje
- Prohlížeč HTML souborů
- Bash či kompatibilní shell pro spuštění skriptů
- Nainstalovaný gcc toolchain pro překlad programů
- Knihovna FFTW [3] – pro úspěšné měření výkonu na platformě x86

Některým souborům bud třeba nastavit práva pro spuštění. Jejich seznam a další podrobnější informace naleznete v souboru `index.html`, který je umístěn v kořenovém adresáři CD.