VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

Fakulta elektrotechniky
a komunikačních technologií

DIPLOMOVÁ PRÁCE

Brno, 2017 Bc. Zdeněk Skulínek

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ
FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

ÚSTAV TELEKOMUNIKACÍ
DEPARTMENT OF TELECOMMUNICATIONS

PARALELIZACE GOERTZELOVA ALGORITMU
PARALLEL IMPLEMENTATION OF GOERTZEL ALGORITHM

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — **Bc. Zdeněk Skulínek**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. Petr Sysel, Ph.D.**
SUPERVISOR

BRNO 2017

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

Diplomová práce

magisterský navazující studijní obor **Telekomunikační a informační technika**
Ústav telekomunikací

Student: Bc. Zdeněk Skulínek ***ID:*** 16865
Ročník: 2 ***Akademický rok:*** 2016/17

NÁZEV TÉMATU:

Paralelizace Goertzelova algoritmu

POKYNY PRO VYPRACOVÁNÍ:

Seznamte se s možnostmi paralelního zpracování na počítačích typu PC a grafických procesorech. Prostudujte principy a techniky paralelizace používané např. v prostředí Matlab nebo v rámci některé z knihoven pro paralelní zpracování. Vytvořte ukázkové úlohy paralelního zpracování. Porovnejte výpočetní náročnost sekvenčního a paralelního zpracování. Dále se seznamte s Goertzelovým algoritmem a navrhněte, jakým způsobem by bylo možné jej urychlit pomocí paralelního zpracování.

DOPORUČENÁ LITERATURA:

[1] OpenCV 2.4.11 Documentaion. 2014. Dostupné na URL http://docs.opencv.org

[2] SYSEL, P.; RAJMIC, P. Goertzel Algorithm Generalized to Non-integer Multiples of Fundamental Frequency. EURASIP Journal on Advances in Signal Processing. 2012. 2012(56). p. 1 - 20. ISSN 1687-6172.

Termín zadání: 1.2.2017 ***Termín odevzdání:*** 24.5.2017

Vedoucí práce: Ing. Petr Sysel, Ph.D.
Konzultant:

doc. Ing. Jiří Mišurec, CSc.
předseda oborové rady

ABSTRAKT

Technické problémy znemožňují neustále zvyšovat hodinové frekvence procesorů. Jejich výkon tak v současné době roste díky zvyšování počtu jader. To s sebou přináší nutnost nových přístupů pro programování takovýchto paralelních systémů. Tato práce ukazuje, jak využít paralelismus k číslicovému zpracování signálu. Jako příklad zde bude uvedena implementace Geortzelova algoritmu s využitím výpočetního výkonu grafického čipu.

KLÍČOVÁ SLOVA

Goertzelův algoritmus, zpracování signálu, openCL, paralelní výpočet, GPU

ABSTRACT

Technical problems make impossible steadily increase processor's clock frequency. Their power are currently growing due to increasing number of cores. It brings need for new approaches in programming such parallel systems. This thesis shows how to use paralelism in digital signal processing. As an example, it will be presented here implementation of the Geortzel's algorithm using the processing power of the graphics chip.

KEYWORDS

Goertzel's algorithm, signal processing, openCL, parallel computing, GPU

SKULÍNEK, Zdeněk *Paralelizace Goertzelova algoritmu*: diplomová práce. Brno: Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, Ústav telekomunikací, Rok. 80 s. Vedoucí práce byl Ing. Petr Sysel, Ph.D.

PROHLÁŠENÍ

Prohlašuji, že svou diplomovou práci na téma „Paralelizace Goertzelova algoritmu“ jsem vypracoval(a) samostatně pod vedením vedoucího diplomové práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny citovány v práci a uvedeny v seznamu literatury na konci práce.

Jako autor(ka) uvedené diplomové práce dále prohlašuji, že v souvislosti s vytvořením této diplomové práce jsem neporušil(a) autorská práva třetích osob, zejména jsem nezasáhl(a) nedovoleným způsobem do cizích autorských práv osobnostních a/nebo majetkových a jsem si plně vědom(a) následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon), ve znění pozdějších předpisů, včetně možných trestněprávních důsledků vyplývajících z ustanovení části druhé, hlavy VI. díl 4 Trestního zákoníku č. 40/2009 Sb.

Brno ....

podpis autora(-ky)

PODĚKOVÁNÍ

Rád bych poděkoval vedoucímu diplomové práce panu Ing. Petru Syslovi, Ph.D. za odborné vedení, konzultace, trpělivost a podnětné návrhy k práci.

Brno ....

podpis autora(-ky)

OBSAH

SEZNAM OBRÁZKŮ

SEZNAM TABULEK

ÚVOD

Moderní architektury procesorů využívají paralelismus jako důležitou cestu ke zvýšení výpočetního výkonu. Řeší se tím zejména technické problémy při zvyšování hodinových kmitočtů, kde se naráží na fyzikální limity. CPU zvyšují výkon přidáváním jader. GPU se vyvinula z pevně dané renderovací funkcionality do podoby paralelních programovatelných procesorů. Protože dnešní počítače často obsahují vícejádrové CPU a GPU a další procesory, je velmi důležité vzít v úvahu specifika programování pro takovéto paralelní systémy.

Vytváření aplikací pro vícejádrové procesory je oproti tradičním jednojádrovým aplikacím složitější, neboť je potřeba použít zcela jiného přístupu. Více procesorové systémy jsou navíc silně platformně a hardwarově závislé, což činí jejich programování obtížnější.

Mým úkolem je ukázat možnosti dnešních počítačů při zpracování signálu. Jako modelový příklad mám implementovat *Goertzelův algoritmus*, což je číslicový filtr, na jehož vstupu je číslicový signál a parametr k, udávající konkrétní harmonickou složku spektra signálu navzorkovaného s kmitočtem f_{vz}. Jeho výstupem je pak jediná hodnota vyjadřující amplitudu na určitém kmitočtu.

Paralelizovaný Goertzelův algoritmus představím na své aplikaci nazvané *Sound Analyzer*, která počítá libovolná počet složek spektra v reálném čase a zobrazuje výsledné spektrum g grafickém prostředí.

Tato práce zažíná představením *Goertzelova algoritmu* 1. Na jejím základě jsem napsal kapitolu *možnosti řešení*2, ve které uvádím techniky řešení a odvozuji vzorce, které použiji pro sestavení programu *Sound Analyzer*. Kapitola následující je pak malým úvodem do problematiky paralelizace. Řešení diplomové práce, program *Sound Analyzer* 4 včetně jeho součástí, koncepce a licence jsou uvedeny v kapitole 4. Z popisu programu přecházím do popisu použitých knihoven5 a vysvětlení, proč jsem využil zrovna tyto. V poslední kapitole *měření přínosu* 6 odpovídám na otázku, jaký je přínos využití GPU při zpracování signálu.

1 GOERTZELŮV ALGORITMUS

1.1 Diskrétní Fourierova řada

Máme řadu N hodnot libovolné posloupnosti. Fourier ukázal, že je možno převést ji na N hodnot nějaké frekvenční charakteristiky. *Diskrétní Fourierova řada* přiřazuje časové periodické posloupnosti periody N, posloupnost spektra, rovněž periodickou a rovněž periody N. Následující vzorec pochází z knihy *Systémy a signály* od prof. Smékala[7].

$$S_p[k] = \sum_{n=0}^{N-1} s_p[n] \mathrm{e}^{-\mathrm{j}k\frac{2\pi}{N}n}, \tag{1.1}$$

kde k k. hodnota amplitudy ve spektru, nabývá hodnot $0..N-1$, N je délka sekvence časové posloupnosti i délka spektra *DFŘ*. Řada má tedy určitý, omezený počet členů.

1.2 Diskrétní Fourierova transformace - DFT

Na rozdíl od *DFŘ*, není obraz *DFT* periodický. *DFT* přiřazuje časové posloupnosti délky N, posloupnost spektra, také délky N. S pomocí *DFŘ* by se vytvořil asi takto:

- Zperiodizování průběhu s, kde $s[n + lN] = s_p[n]$
- Výpočet DFŘ.
- Oříznutí spektra na jednorázovou posloupnost.

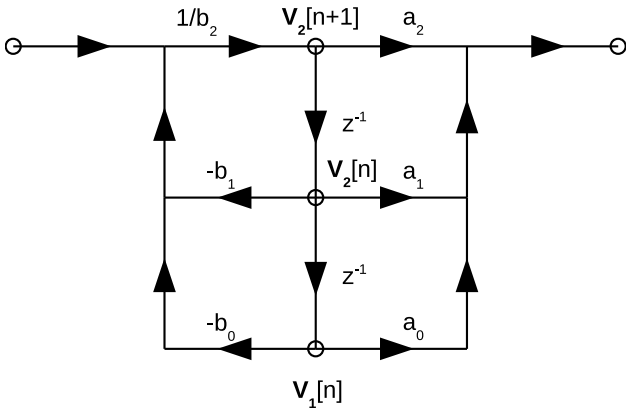
DFT zapisujeme jako

$$S[k] = \sum_{n=0}^{N-1} s[n] \mathrm{e}^{-\mathrm{j}k\frac{2\pi}{N}n}. \tag{1.2}$$

I tento vzorec pochází z knihy prof. Smékala ([7]).

1.3 Goertzelův algoritmus

Často je třeba řešit požadavek na zjištění složky spektra u určitém úzkém pásmu kmitočtů. Šlo by samozřejmě spočítat Fourierovou transformací celé spektrum a vybrat jen ten kmitočet, který je pro nás zajímavý. Další možností je spočítání jedné



Obr. 1.1: Graf signálových toků druhé kanonické struktury.

hodnoty z definice Fourierovy transformace, tak jako ve vzorci pro DFT (1.2). Myšlenkou *Goertzelova algoritmu* je k tomuto účelu použít číslicové filtrace. Goertzel použil druhou kanonickou strukturu (1.3), realizovanou filtrem IIR (na obrázku 1.1).

$$\begin{aligned} v_1[n+1] &= v_2[n] \\ v_2[n+1] &= \frac{1}{b_2}(x[n] - b_1 v_2[n] - b_0 v_1[n]) \\ y[n] &= a_2 v_2[n+1] + a_1 v_2[n] + a_0 v_1[n] \end{aligned} \tag{1.3}$$

1.4 Odvození Goertzelova algoritmu

Následující odvození pochází rovněž z knihy prof. Smékala Systémy a signály [7]. Označme:

$$W_N = \mathrm{e}^{-\mathrm{j}\frac{2\pi}{N}}, \tag{1.4}$$

pak

$$(W_N)^{-kN} = (\mathrm{e}^{-\mathrm{j}\frac{2\pi}{N}})^{-kN} = 1, \tag{1.5}$$

Protože člen (1.5) je jedna, můžeme s ním vynásobit definiční vztah Fourierovy transformace (1.2), aniž by došlo ke změně.

$$S[k] = \sum_{m=0}^{N-1} s[m] W_N^{-kN} = (W_N)^{-kN} \sum_{m=0}^{N-1} s[m] W_N^{-kN} = \\ = \sum_{m=0}^{N-1} s[m] W_N^{-k(N-m)} = \sum_{m=0}^{N-1} s[m] \mathrm{e}^{-\mathrm{j}\frac{2\pi}{N}k(N-m)} \tag{1.6}$$

Konečný tvar vztahu (1.6) zjevně vypadá jako konvoluce vstupní posloupnosti $x[n] = s[n]$ s impulsní charakteristikou číslicového filtru typu IIR

$$h[n] = \mathrm{e}^{-\mathrm{j}k\frac{2\pi}{N}} = W_N^{-kN} = 1, n = 0, 1, 2..\infty \tag{1.7}$$

Impulsní charakteristika má komplexní koeficienty, což je zřejmé i z toho, že spektrum DFT je také komplexní. Výstupní odezva potom je

$$y[n] = x[n] * h[n] = \sum_{m=0}^{\infty} s[m] h[n-m] = \sum_{m=0}^{\infty} s[m] W_N^{-k(n-m)} = \\ \sum_{m=0}^{\infty} s[m] \mathrm{e}^{-\mathrm{j}k\frac{2\pi}{N}(n-m)} \tag{1.8}$$

Tento vzorec (1.8) je ovšem nekonečná řada. Naštěstí není nutné počítat do nekonečna, jelikož v hledané spektrální složce je jediný vzorek v čase N roven

$$y[N] = \sum_{m=0}^{\infty} s[n] \mathrm{e}^{-\mathrm{j}k\frac{2\pi}{N}(N-m)} = \sum_{m=0}^{\infty} s[n] \mathrm{e}^{-\mathrm{j}k\frac{2\pi}{N}N} \mathrm{e}^{-\mathrm{j}k\frac{2\pi}{N}m} = \sum_{m=0}^{N-1} s[n] \mathrm{e}^{-\mathrm{j}k\frac{2\pi}{N}m} = \\ S[k] \tag{1.9}$$

Přenosová charakteristika filtru typu IIR s impulsní charakteristikou (1.7) je prvního řádu

$$H(z) = \sum_{n=0}^{\infty} h[n] z^{-n} = \sum_{n=0}^{\infty} \mathrm{e}^{-\mathrm{j}k\frac{2\pi}{N}n} z^{-n} = \sum_{n=0}^{\infty} (\mathrm{e}^{-\mathrm{j}k\frac{2\pi}{N}} z^{-1})^n = \\ \frac{1}{1 - \mathrm{e}^{-\mathrm{j}k\frac{2\pi}{N}} z^{-1}} = \frac{z}{z - W_N^{-k}} \tag{1.10}$$

Nevýhodou zcela jistě je skutečnost, že se v přenosové funkci vyskytují komplexní koeficienty. Následující úprava vztahu (1.10) tohle řeší. Daň za možnost práce s jen reálnými čísly je přenosová funkce druhého řádu.

$$H(z) = \frac{1}{1 - W_N^{-k} z^{-1}} = \frac{1}{1 - W_N^{-k} z^{-1}} . \frac{1 - W_N^{k} z^{-1}}{1 - W_N^{k} z^{-1}} = \frac{1}{1 - \mathrm{e}^{-\mathrm{j}k\frac{2\pi}{N} z^{-1}}} . \frac{1 - \mathrm{e}^{-\mathrm{j}k\frac{2\pi}{N} z^{-1}}}{1 - \mathrm{e}^{-\mathrm{j}k\frac{2\pi}{N} z^{-1}}} = \frac{1 - \mathrm{e}^{-\mathrm{j}k\frac{2\pi}{N} z^{-1}}}{1 - (\mathrm{e}^{-\mathrm{j}k\frac{2\pi}{N}} + \mathrm{e}^{-\mathrm{j}k\frac{2\pi}{N}}) z^{-1} + z^{-2}} = \frac{1 - \mathrm{e}^{-\mathrm{j}k\frac{2\pi}{N} z^{-1}}}{1 - 2\cos k\frac{2\pi}{N} z^{-1} + z^{-2}} = \frac{z^2 - \mathrm{e}^{-\mathrm{j}k\frac{2\pi}{N} z}}{z^2 - 2\cos k\frac{2\pi}{N} z + 1} = \frac{a_2 z^2 + a_1 z + a_0}{b_2 z^2 + b_1 z + b_0} \tag{1.11}$$

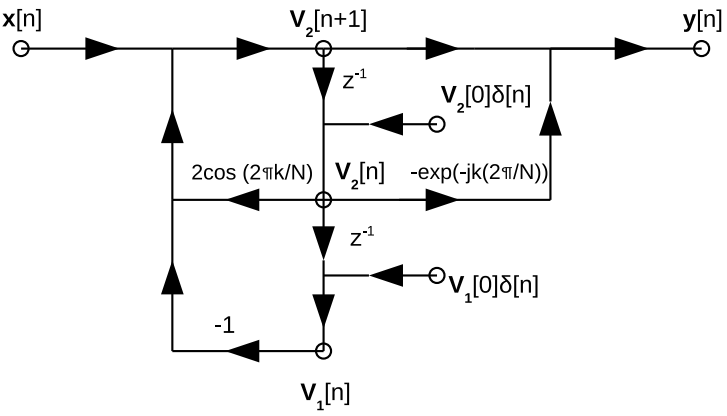
Výsledek porovnáme se stavovými rovnicemi (1.3) A zjistíme tak následující koeficienty.

$$\begin{aligned} b_2 &= 1, \\ b_1 &= -2\cos k\frac{2\pi}{N}, \\ b_0 &= 1, \\ a_2 &= 1, \\ a_1 &= -\mathrm{e}^{-\mathrm{j}k\frac{2\pi}{N}}, \\ a_0 &= 0 \end{aligned} \tag{1.12}$$

Jejich dosazením do (1.3) dostaneme...

$$\begin{aligned} v_1[n+1] &= v_2[n], \\ v_2[n+1] &= x[n] + 2\cos k\frac{2\pi}{N} v_2[n] - v_1[n], \\ y[n] &= v_2[n+1] - (\cos k\frac{2\pi}{N} - \mathrm{j}\sin k\frac{2\pi}{N}) v_2[n], \end{aligned} \tag{1.13}$$

Výhodou této struktury je, že nám stačí ve smyčce počítat jen proměnné v_1 a v_2. Výstup y se počítá jen jednou, při posledním průchodu cyklem, kdy $n = N$. Počáteční hodnoty v_1 a v_2 jsou nulové. Komplexní operace tak bude provedena pouze jednou a to až na závěr.



Obr. 1.2: Graf signálových toků Goertzelova algoritmu ve 2. kanonické struktuře.

2 MOŽNOSTI PARALELIZACE

2.1 Rozdělení na N-částí a jejich průměrování

Jako první nápad je, aby každé jádro procesoru zpracovávalo poměrnou část výpočtu. Dostali bychom N výsledků (kde je N-počet jader procesoru) a z těch by se udělal průměr. Jenže z N-krát méně vzorků pro výpočet by znamenalo také N-krát širší pásmo, které zkoumám. Filtr IIR je totiž pásmová propust. Abych měřil co nejpřesněji, musí být co nejužší. To znamená N-krát menší přesnost. Průměrování to zcela jistě nespraví. Navíc se tím neušetří žádný výpočet. Proto dále nebudu v této úvaze pokračovat.

2.2 Variace stavové proměnné

V našem případě by bylo výhodnější, kdybych mohl počítat s neznámými stavovými proměnnými. Ty totiž počítá jiné jádro. Řekněme, že začínám n-tým vzorkem, počítám pro 4 hodnoty $x[n]$, ve stavových proměnných v_1 a v_2 mám neznámé hodnoty R a S.

$$
\begin{aligned}
v_1[n] &= R, \\
v_2[n] &= S, \\
2\cos k\frac{2\pi}{N} &= C \\
v_1[n+1] &= v_2[n] = S, \\
v_2[n+1] &= x[n] + 2\cos k\frac{2\pi}{N} v_2[n] - v_1[n] = x[n] + CS - R, \\
v_1[n+2] &= v_2[n+1] = x[n] + CS - R, \\
v_2[n+2] &= x[n+1] + 2\cos k\frac{2\pi}{N} v_2[n+1] - v_1[n+1] = \\
&\qquad x[n+1] + C(x[n] + CS - R) - S = \\
&\qquad x[n+1] + Cx[n] + (C^2 - 1)S - CR, \\
v_1[n+3] &= v_2[n+2] = \\
&\qquad x[n+1] + Cx[n] + (C^2 - 1)S - CR,
\end{aligned}
\tag{2.1}
$$

$$
\begin{aligned}
v_2[n+3] &= x[n+2] + 2\cos k\frac{2\pi}{N} v_2[n+2] - v_1[n+2] = \\
&\quad x[n+2] + C(x[n+1] + Cx[n] + (C^2-1)S - CR) \\
&\quad - x[n] - CS + R = \\
&\quad x[n+2] + Cx[n+1] + (C^2-1)x[n] + CS(C^2-2) - \\
&\quad (C^2-1)R \\
v_1[n+4] &= v_2[n+3] = \\
&\quad x[n+2] + Cx[n+1] + (C^2-1)x[n] + CS(C^2-2) - \\
&\quad (C^2-1)R \\
v_2[n+4] &= x[n+3] + 2\cos k\frac{2\pi}{N} v_2[n+3] - v_1[n+3] = \\
&\quad x[n+3] + C(x[n+2] + Cx[n+1] + (C^2-1)x[n] + \\
&\quad CS(C^2-2) - (C^2-1)R) - x[n+1] - Cx[n] - \\
&\quad (C^2-1)S + CR = \\
&\quad x[n+3] + Cx[n+2] + (C^2-1)x[n+1] + C(C^2-2)x[n] + \\
&\quad (C^4 - 2C^2 - C^2 + 1)S - (C^3 - 2C)R
\end{aligned}
$$

Zcela zjevně se dají naše dvě neznámé stavové proměnné nahradit třemi novými, zato známými. Každá rovnice se totiž dá nahradit výrazy:

$$
\begin{aligned}
v_1[n+1] &= v_2[n] = w_0[n] + w_1[n]S - w_2[n]R, \\
v_2[n+1] &= x[n] + Cv_2[n] - v_1[n] = \\
&\qquad w_0[n+1] + w_1[n+1]S + w_2[n+1]R,
\end{aligned} \tag{2.2}
$$

Vyskytuje se tu řada: $1, C, C^2-1, C^3-2C, C^4-2C^2-C^2+1, C^5-3C^3+C-C^3+2C$

Při bližším prozkoumání této řady vidím, že má určitou zákonitost. Založím si

tedy novou stavovou proměnnou, nazvu ji w_3. Začnu...

$$\begin{aligned}
w_3[-1] &= 0 \\
w_3[0] &= 1 \\
w_3[1] &= C \\
&\dots \\
w_3[n+1] &= Cw_3[n] - w_3[n-1] \\
&\dots \\
w_3[2] &= Cw_3[1] - w_3[0] = C^2 - 1 \\
w_3[3] &= Cw_3[2] - w_3[1] = C(C^2-1) - C = C^3 - 2C \\
w_3[4] &= Cw_3[3] - w_3[2] = C(C^3 - 2C) - (C^2 - 1) = \\
&\quad C^4 - 3C^2 + 1 \\
w_3[5] &= Cw_3[4] - w_3[3] = C(C^4 - 3C^2 + 1) - (C^3 - 2C) = \\
&\quad C^5 - 3C^3 + C - C^3 + 2C = C^5 - 4C^3 + 3C
\end{aligned} \tag{2.3}$$

Pokud zkombinuji rovnice (2.1) a (2.2), můžu si vyjádřit nové stavové proměnné w_1, w_2 a w_3. Začnu s w_0:

$$\begin{aligned}
w_0[n] &= 0 \\
w_0[n+1] &= x[n] \\
w_0[n+2] &= x[n+1] + Cx[n] \\
w_0[n+3] &= x[n+2] + Cx[n+1] + (C^2-1)x[n] \\
w_0[n+4] &= x[n+3] + Cx[n+2] + (C^2-1)x[n+1] + C(C^2-2)x[n]
\end{aligned} \tag{2.4}$$

Což mohu vyjádřit stavovou proměnnou w_3

$$
\begin{aligned}
w_0[n+1] &= w_3[0]x[n] \\
w_0[n+2] &= w_3[0]x[n+1] + w_3[1]x[n] \\
w_0[n+3] &= w_3[0]x[n+2] + w_3[1]x[n+1] + w_3[2]x[n] \\
w_0[n+4] &= w_3[0]x[n+3] + w_3[1]x[n+2] + w_3[2]x[n+1] + w_3[3]x[n]
\end{aligned} \tag{2.5}
$$

Zkusím to opět rozložit podle vzorce (2.3),což vede na původní tvar. Rovnice lze ale přeskládat tak, abych se dostal k rekurentní formuli.

$$
\begin{aligned}
w_0[n+1] &= w_3[0]x[n] = x[n] \\
w_0[n+2] &= w_3[0]x[n+1] + (Cw_3[0] - w_3[-1])x[n] = \\
&\quad Cw_3[0]x[n] + w_3[0]x[n+1] - w_3[-1]x[n] = \\
&\quad Cw_0[n+1] + x[n+1] \\
w_0[n+3] &= w_3[0]x[n+2] + w_3[1]x[n+1] + (Cw_3[1] - w_3[0])x[n] = \\
&\quad w_3[0]x[n+2] + (Cw_3[0] - w_3[-1])x[n+1] + (Cw_3[0]- \\
&\quad w_3[-1])Cx[n] - w_3[0]x[n] = \\
&\quad x[n+2] + C(x[n+1] + Cx[n]) - x[n] = \\
&\quad x[n+2] + Cw_0[n+2] - w_0[n+1] \\
w_0[n+4] &= w_3[0]x[n+3] + w_3[1]x[n+2] + w_3[2]x[n+1]+ \\
&\quad w_3[3]x[n] = x[n+3] + (Cw_3[0] - w_3[-1])x[n+2] + (Cw_3[1] \\
&\quad - w_3[0])x[n+1] + (Cw_3[2] - w_3[1])x[n] = \\
&\quad x[n+3] + Cx[n+2] + (C^2 - 1)x[n+1] + (C^3 - 2C)x[n] = \\
&\quad x[n+3] + C(x[n+2] + Cx[n+1] + C^2x[n] - x[n])- \\
&\quad Cx[n] - x[n+1] = x[n+4] + Cw_0[n+3] - w_0[n+2]
\end{aligned} \tag{2.6}
$$

Pokračuji s w_1:

$$\begin{aligned} w_1[n] &= 1 = w_3[0] \\ w_1[n+1] &= C = w_3[1] \\ w_1[n+2] &= C^2 - 1 = w_3[2] \\ w_1[n+3] &= C^3 - 2C = w_3[3] \\ w_1[n+4] &= C^4 - 3C^2 + 1 = w_3[4] \end{aligned} \tag{2.7}$$

což je již odvozené w_3. Proto tedy mohu napsat

$$w_1[n+1] = w_3[1] = Cw_1[0] - w_1[-1] \tag{2.8}$$

A nakonec w_2:

$$\begin{aligned} w_2[n] &= 0 \\ w_2[n+1] &= 1 \\ w_2[n+2] &= C \\ w_2[n+3] &= C^2 - 1 \\ w_2[n+4] &= C^3 - 2C \end{aligned} \tag{2.9}$$

w_2 je posunuté w_3

$$w_2[n+1] = w_3[n] = w_1[n] \tag{2.10}$$

Ještě je tu malý problém. Rekurentní vzorce pro w_0 a w_1 se odkazují nejen na poslední člen historie, ale i předposlední. Proto formálně zavedu nové stavové proměnné. Z pohledu procesoru je totiž stejně jedno, zda nějaké paměťové místo

nazývám jako $w_4[n]$ nebo $w_0[n-1]$.

$$\begin{aligned} w_4[n+1] &= w_0[n] \\ w_0[n+1] &= x[n+1] + Cw_0[n] - w_0[n-1] = \\ &\qquad x[n+1] + Cw_0[n] - w_4[n] \\ w_2[n+1] &= w_1[n] \\ w_1[n+1] &= Cw_1[n] - w_2[n] \end{aligned} \tag{2.11}$$

Vztahy pro w_0 a w_4, které jsem odvodil jsou vlastně vztahy Goertzelova algoritmu (1.13). Přibyly jen rovnice pro w_1 a w_2. Zajímavé je že w_1 a w_2 jsou jen konstanty a vůbec nezávisejí na vstupním signálu. To by ovšem znamenalo zjednodušení spojování těch mezivýsledků, co víc, w_1 a w_2 by se vůbec nemusely počítat v tomto algoritmu, protože jsou již předem známé.

Nyní zkusím malou kontrolu. Zkusím, zda výše uvedené vzorce předpovídají další hodnotu posloupnosti v_2, respektive, zda koeficienty u neznámých R a S jsou z řady w_3.

$$\begin{aligned} v_1[n+5] &= v_2[n+4] = \\ &\qquad x[n+3] + Cx[n+2] + (c^2-1)x[n+1] + C(C^2-2)x[n] + \\ &\qquad (C^4 - 2C^2 - C^2 + 1)S - (C^3 - 2C)R \\ v_2[n+5] &= x[n+4] + 2\cos k\frac{2\pi}{N} v_2[n+4] - v_1[n+4] = \\ &\qquad x[n+4] + C(x[n+3] + Cx[n+2] + (c^2-1)x[n+1] + \\ &\qquad C(C^2-2)x[n] + (C^4 - 2C^2 - C^2 + 1)S - (C^3 - 2C)R) - \\ &\qquad (x[n+2] + Cx[n+1] + (C^2-1)x[n] + CS(C^2-2) - \\ &\qquad (C^2-1)R) = \\ &\qquad [n+4] + Cx[n+3] + (C^2-1)x[n+2] + (c^3-2C)x[n+1] + \\ &\qquad (C^4 - 2C^2 - C^2 + 1)x[n] + (C^5 - 3C^3 + C - C^3 + 2C)S - \\ &\qquad (C^4 - 2C^2 - C^2 + 1)R \end{aligned} \tag{2.12}$$

A na závěr vyzkouším, zda jsem předpověděl dobře novou stavovou proměnnou w_0.

$$\begin{aligned} w_0[n+5] &= w_3[0]x[n+4] + w_3[1]x[n+3] + w_3[2]x[n+2] + \\ &\quad w_3[3]x[n+1] + w_3[4]x[n] = \\ &\quad x[n+4] + Cx[n+3] + (C^2-1)x[n+2] + \\ &\quad (C^3-2C)x[n+1] + (C^4-3C^2+1)x[n] = \\ &\quad x[n+4] + C(x[n+3] + Cx[n+2] + C^2x[n+1] - x[n+1] + \\ &\quad C^3x[n] - 2Cx[n]) - (x[n+2] + Cx[n+1] + C^2x[n] - \\ &\quad x[n]) = x[n+4] + Cw_1[n+4] - w_1[n+3] \end{aligned} \tag{2.13}$$

Zkusím tedy malou rekapitulaci výše uvedeného.

Mám signál o N diskrétních hodnotách a chci jej počítat na P procesorových jádrech. Pro jednoduchost budu předpokládat že N je násobek P. Vlastně je úplně jedno, zda bloky, na které N-prvkový signál záhy rozdělím, jsou stejně dlouhé. Každopádně by měly být co možná nejvíce stejně dlouhé, protože celek bude pracovat efektivněji. Jako prerekvizity si spočítám $C = 2\cos 2\pi/N$ a spočítám si w_1 a w_2, které stačí spočítat jednou v případě, že všech P bloků je stejně dlouhých. Na každém bloku signálu spočítám hodnoty podle vztahu (2.11) w_0 a w_4, tedy klasickým způsobem Goertzelovým algoritmem, bez výpočtu výstupní proměnné y. Provedu tedy právě N jednoduchých výpočtů. Nyní musím mezivýsledky nějak spojit. U každého mezivýsledku znám w_0, w_1 i w_2. Mohu tedy napsat v souladu se vztahem (2.2).

$$\begin{aligned} R[0] &= 0 \\ S[0] &= 0 \end{aligned} \tag{2.14}$$

$$\begin{aligned} R[1] &= w_0[0] \\ S[1] &= w_4[0] \end{aligned} \tag{2.15}$$

$$\begin{aligned} R[2] &= w_0[1] + w_1[1]R[1] - w_2[1]S[1] \\ S[2] &= w_4[1] + w_1[1]R[1] - w_2[1]S[1] \end{aligned} \tag{2.16}$$

$$\begin{aligned} R[3] &= w_0[2] + w_1[2]R[2] - w_2[2]S[2] \\ S[3] &= w_4[2] + w_1[2]R[2] - w_2[2]S[2] \end{aligned} \tag{2.17}$$

$$\begin{aligned} R[n+1] &= w_0[n] + w_1[n]R[n] - w_2[n]S[n] \\ S[n+1] &= w_4[n] + w_1[n]R[n] - w_2[n]S[n] \end{aligned} \tag{2.18}$$

2.3 Maticové operace

Protože budu používat knihovnu openCL, která obvykle využívá výpočetní výkon grafické karty, je vhodné to nějakým způsobem využít. V grafických výpočtech se používají matice, druhého, třetího a čtvrtého řádu. Grafický čip je tedy pro výpočty s těmito maticemi optimalizován. Proto převedu předchozí úvahu do maticové terminologie. Začnu s přepsáním Goertzelova vztahu (1.13).
Pro $v[1]$:

$$v[1] = \begin{pmatrix} v_1[1] \\ v_2[1] \end{pmatrix} = Av[0] + Bx[0] = \begin{pmatrix} 0 & 1 \\ -1 & C \end{pmatrix} \begin{pmatrix} v_1[0] \\ v_2[0] \end{pmatrix} + \begin{pmatrix} 0 \\ 1 \end{pmatrix} \Big(x[0] \Big) \tag{2.19}$$

a pro $v[2]$

$$\begin{aligned} v[2] = \begin{pmatrix} v_1[2] \\ v_2[2] \end{pmatrix} &= A(Av[0] + Bx[0]) + Bx[1] = A^2 \\ &\begin{pmatrix} v_1[0] \\ v_2[0] \end{pmatrix} + A \begin{pmatrix} 0 \\ 1 \end{pmatrix} \Big(x[0] \Big) + \begin{pmatrix} 0 \\ 1 \end{pmatrix} \Big(x[1] \Big) \end{aligned} \tag{2.20}$$

a nakonec zobecním. Vztah (2.21) je Goertzelův algoritmus v maticovém zápisu.

$$v[n+1] = \begin{pmatrix} v_1[n+1] \\ v_2[n+1] \end{pmatrix} = Av[n] + Bx[n] = \\ \begin{pmatrix} 0 & 1 \\ -1 & C \end{pmatrix} \begin{pmatrix} v_1[n] \\ v_2[n] \end{pmatrix} + \begin{pmatrix} 0 \\ 1 \end{pmatrix} \Big(x[n] \Big) \tag{2.21}$$

Ten můžu volně přepsat:

$$v[n+k] = \\ A(A(\dots A(A(A(v[n]) + Bx[n]) + Bx[n+1]) + Bx[n+2]) \dots) + \\ Bx[n+k-1])) + Bx[n+k] \tag{2.22}$$

Ještě to upravím tak, aby vstup x byl jeden dlouhý sloupcový vektor.

$$v[n+k] = \begin{pmatrix} v_1[n+k] \\ v_2[n+k] \end{pmatrix} = A^k v[n] + \\ \begin{pmatrix} A^{k-1} \begin{pmatrix} 0 \\ 1 \end{pmatrix} & A^{k-2} \begin{pmatrix} 0 \\ 1 \end{pmatrix} & A^{k-3} \begin{pmatrix} 0 \\ 1 \end{pmatrix} \dots \begin{pmatrix} 0 \\ 1 \end{pmatrix} \end{pmatrix} \begin{pmatrix} x[n] \\ x[n+1] \\ x[n+2] \\ \vdots \\ x[n+k] \end{pmatrix} = \\ A^k v[n] + Dx[n, n+k]^T \tag{2.23}$$

Tento vzorec, když si představím $k = 3$, dává jasný návod, jak využít matice čtvrtého řádu. Oproti realizaci elementárními operacemi klesne počet maticových operací na čtvrtinu, protože počítám hned se čtyřmi hodnotami. Rovněž popisuje spojovaní mezivýsledků z jednotlivých jader. Strana $Dx[n, n+k]^T$ je *Goertzelův* vztah, a je k němu připočten předchozí mezivýsledek vynásobený maticí A^{k-1}, tedy konstantou známou před výpočtem. Něco takového jsem již odvodil pro případ počítání po jedné hodnotě x (2.11). Samotná matice D je ale také konstanta, která se dá spočítat předem a při konstrukci programu toho využiji.

2.4 Počítání hodnot na více kmitočtech současně

Předchozí výsledky se dají ještě trochu vylepšit. Prvně nepotřebuji počítat celý horní řádek matice A^k. Dostanu z něj předchozí hodnotu a tu vlastně již znám. Z toho druhého řádku mám (podle 2.24) počítám pro druhý řádek matice A $v_2[n+1] = a_{21}v_1 + a_{22}v_2$. To je nyní první řádek matic ve vztahu 2.24 s tím, že první index přes čárkou označuje index počítaného kmitočtu. Z maticových operací se stanou vektorové.

Protože ale mám možnost počítat matice čtvrtého řádu, proč nevzít rovnou čtyři frekvence současně? Začnu s pravou stranou. Protože jsem v minulé větě uvedl, že stačí použít druhý řádek matice D, tak teď jej 3-krát zkopíruji do ostatních řádků. Stejný ale úplně nebude, jelikož konstanta C v sobě zahrnuje kmitočet, takže hodnoty v řádcích budou jiné. Na levé straně u matice A^k provedu to stejné. Vektor $v[n]$ rozšířím také 4-krát, ale do šířky. Výsledkem je matice, říkejme jí E, ze které je důležitý jen vektor na diagonále (kde jsou prováděny stejné operace), ostatní hodnoty jsou nenulové ale nejdou pro nás již potřebné.

$$
\begin{aligned}
v[n+3] = \begin{pmatrix} v_{1,2}[n+3] \\ v_{2,2}[n+3] \\ v_{3,2}[n+3] \\ v_{4,2}[n+3] \end{pmatrix} = \\
\operatorname{diag}\left(\begin{pmatrix} a^k_{1,21} & a^k_{1,22} \\ a^k_{2,21} & a^k_{2,22} \\ a^k_{3,21} & a^k_{3,22} \\ a^k_{4,21} & a^k_{4,22} \end{pmatrix} \cdot \begin{pmatrix} v_{1,1} & v_{2,1} & v_{3,1} & v_{4,1} \\ v_{1,2} & v_{2,2} & v_{3,2} & v_{4,2} \end{pmatrix} \right) + \\
\begin{pmatrix} C_1^3 - 2C_1 & C_1^2 - 1 & C_1 & 1 \\ C_2^3 - 2C_2 & C_2^2 - 1 & C_2 & 1 \\ C_3^3 - 2C_3 & C_3^2 - 1 & C_3 & 1 \\ C_4^3 - 2C_4 & C_4^2 - 1 & C_4 & 1 \end{pmatrix} \cdot \begin{pmatrix} x[n] \\ x[n+1] \\ x[n+2] \\ x[n+3] \end{pmatrix}
\end{aligned}
\tag{2.24}
$$

diag(A^kv[n]) jde ještě vyjádřit jako součet dvou vektorů.

$$v[n+3] = \begin{pmatrix} v_{1,2}[n+3] \\ v_{2,2}[n+3] \\ v_{3,2}[n+3] \\ v_{4,2}[n+3] \end{pmatrix} = \begin{pmatrix} a^k_{1,21} \\ a^k_{2,21} \\ a^k_{3,21} \\ a^k_{4,21} \end{pmatrix} \cdot \begin{pmatrix} v_{1,1} & v_{2,1} & v_{3,1} & v_{4,1} \end{pmatrix} + \begin{pmatrix} a^k_{1,22} \\ a^k_{2,22} \\ a^k_{3,22} \\ a^k_{4,22} \end{pmatrix} \cdot \begin{pmatrix} v_{1,2} & v_{2,2} & v_{3,2} & v_{4,2} \end{pmatrix} + \begin{pmatrix} C_1^3 - 2C_1 & C_1^2 - 1 & C_1 & 1 \\ C_2^3 - 2C_2 & C_2^2 - 1 & C_2 & 1 \\ C_3^3 - 2C_3 & C_3^2 - 1 & C_3 & 1 \\ C_4^3 - 2C_4 & C_4^2 - 1 & C_4 & 1 \end{pmatrix} \cdot \begin{pmatrix} x[n] \\ x[n+1] \\ x[n+2] \\ x[n+3] \end{pmatrix} \tag{2.25}$$

Kde první index je vždy číslo kmitočtu, druhý je index matice původní, neupravené ve vzorci 2.23.

3 PARALELNÍ VÝPOČTY

V této kapitole proberu úvod do využívání paralelních počítačů.

3.1 Sériové vs. paralelní výpočty

Podívejme se například na následující pseudokód.

```
for(i=0;i<N;i++){
          resultA        = task_a(i);
           resultB        = task_b(i);
           resultC        = task_c(i);
           resultD        = task_d(i);
           resultAll        += resultA + resultB + resultC + resultD;
}
```

Provádění výše uvedeného kódu na jednom jednojádrovém procesoru znamená, že budeme muset provést *task_a*, pak *task_b*, následně *task_c* a nakonec *task_d*. To vše je třeba udělat N krát. Pro případ $N = 4$ platí obrázek 3.1. Tomuto případu říkáme *sériový přístup*.

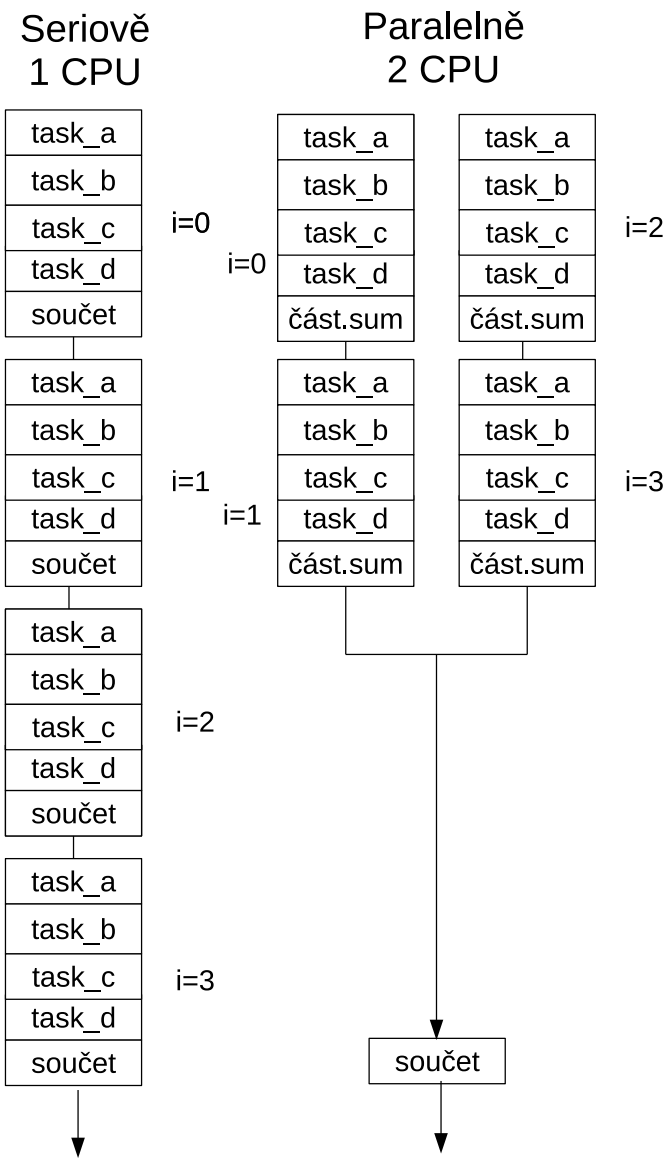
Nyní ale máme k dispozici dvoujádrový procesor. Bez problémů na něm můžeme spustit výše uvedený program. Druhé jádro tohoto nového dvoujádrového procesoru je ale zcela nevytíženo. Výpočet je tak do značné míry neefektivní. Řešením by tedy bylo, aby úkol byl rozdělen na dva poloviční úkoly a na každém jádře by byl prováděn právě jeden. Tomuto říkáme *paralelní přístup* 3.1.

Během plánovací fáze paralelních programů je třeba vzít v úvahu některé stávající zákony. První zákon říká, že pokud program provádí nějaké % časového kódu, který nelze provést paralelně, předpokládá se, že očekávaná rychlost z paralelizace bude v nejlepším zlepšení 1 / y. Zákon lze prokázat následovně. Předpokládejme, že T_s představuje čas potřebný pro spuštění části kódu, kterou nelze paralelizovat, a T_p představuje čas potřebný pro spuštění části kódu, který může mít prospěch z paralelizace. Spuštěním programu s 1 procesorem je doba zpracování:

$$T(1) = T_s + T_p \tag{3.1}$$

Když ale použijeme N procesorů, tak je to méně:

$$T(N) = T_s + T_p/N \tag{3.2}$$



Obr. 3.1: Porovnání sériového a paralelního přístupu

Pokud řekneme, že N jde do nekonečna, zrychlíme výpočet až $S = 1/y$. Tomuto zákonu se říká *Amdahlův*.

3.2 Datový paralelismus vs. úlohový paralelismus

V paralelizaci jsou dva základní přístupy. Jedna instrukce a více dat (SIMD D.51), které představuje datový paralelismus a Více instrukcí(jedna úloha) více dat (SPMD D.52) představující úlohový paralelismus.

3.2.1 Datový paralelismus

Takovým příkladem pro SIMD je sčítání vektorů. Všechny výpočetní jednotky provádějí stejnou instrukcí, každá však nad jiným indexem.

3.2.2 Úlohový paralelismus

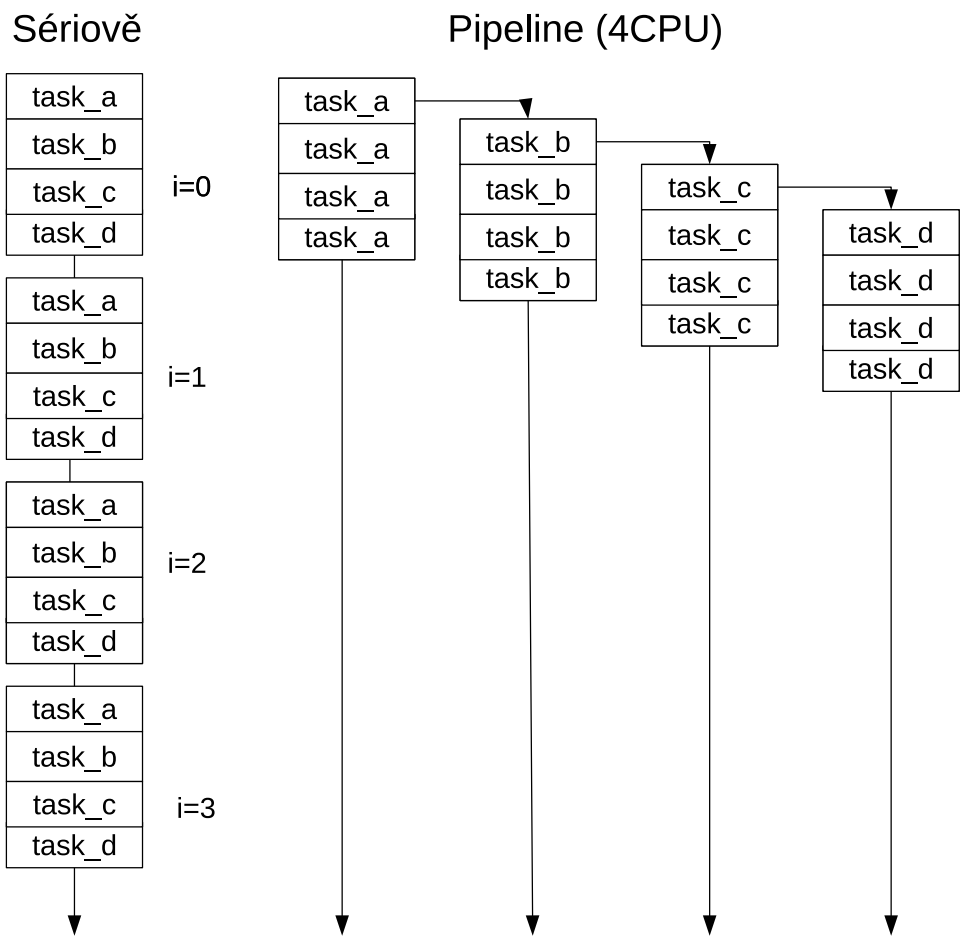
Kdyby se však stejný postup místo na instrukce použil na celou úlohu, efektivita výpočtu by se zjevně snížila. Každá úloha by potřebovala jinak dlouhý časový úsek pro svou práci.

3.2.3 Load balancing

Load balancing je technika, která dokáže využít plně možností víceprocesorového systému. Zde je implementována fronta procesů, ze které se systém postupně odebírá a přiděluje tak procesy volným jádrům.

3.2.4 Pipeline

Obrázek 3.2 ukazuje, že je možný ještě další přístup. Každý procesor je specializovaný na jednu určitou úlohu. Toto schema se s výhodou využívá například u zpracování videa či signálů.



Obr. 3.2: Porovnání sériového přístupu a pipeline

4 PROGRAM SOUNDANALYZER

4.1 Koncepce programu

SoundAnalyzer je grafická aplikace pro sledování signálu a jeho kmitočtové charakteristiky v reálném čase. Vytvořil jsem ji pro demonstrování, ale hlavně vyzkoušení pararelizovaného *Goertzelova algoritmu*. Umí zobrazovat vstup z mikrofonu, výstup počítače (monitor výstupu) a také přehrávat zvukový soubor (formát *wav*), nebo získaná data jako zvukový soubor uložit. Umí také provést jednorázový výpočet spektra ze souboru *matlabu* a uložit jej též ve formátu *matlabu*. Má velké možnosti konfigurace pro každý funkční blok (recorder, Goertzelův algoritmus, obecná nastavení). Ačkoli je v této práci poskytnuta verze pouze pro linux, je program po úpravách multiplatformní.

4.2 Použité knihovny

Grafické uživatelské rozhraní je napsáno v jazyce *QML* a knihovně *Qt*. Tím je dosažena určitá multiplatformnost. Další důvod pro zvolení knihovny *Qt* jsou mé zkušenosti s touto knihovnou a licenční politika.

Pro přehrávání a záznam zvuku ze zařízení je použita knihovna *OpenAL*. Odkazy na funkce z ní jsou výhradně v bloku *recorder*.

Pro zobrazení signálu je použita knihovna *OpenGL*. Její funkční volání je použito výhradně v bloku *barGraphRenderer*.

Vlastní výpočet využívá knihovnu *OpenCL*, které je jako jediná proprietární, ale pro program šířený v rámci *GNU GPLv2* je možné je použít. Jediné použití *OpenCL* funkcí je v bloku *OpenCLGoertzel*.

Všechny bloku programu jsou pak napsány a pospojovány standardní knihovnou jazyka *C++*. Funkce jazyka *C* nejsou nikde v programu použity.

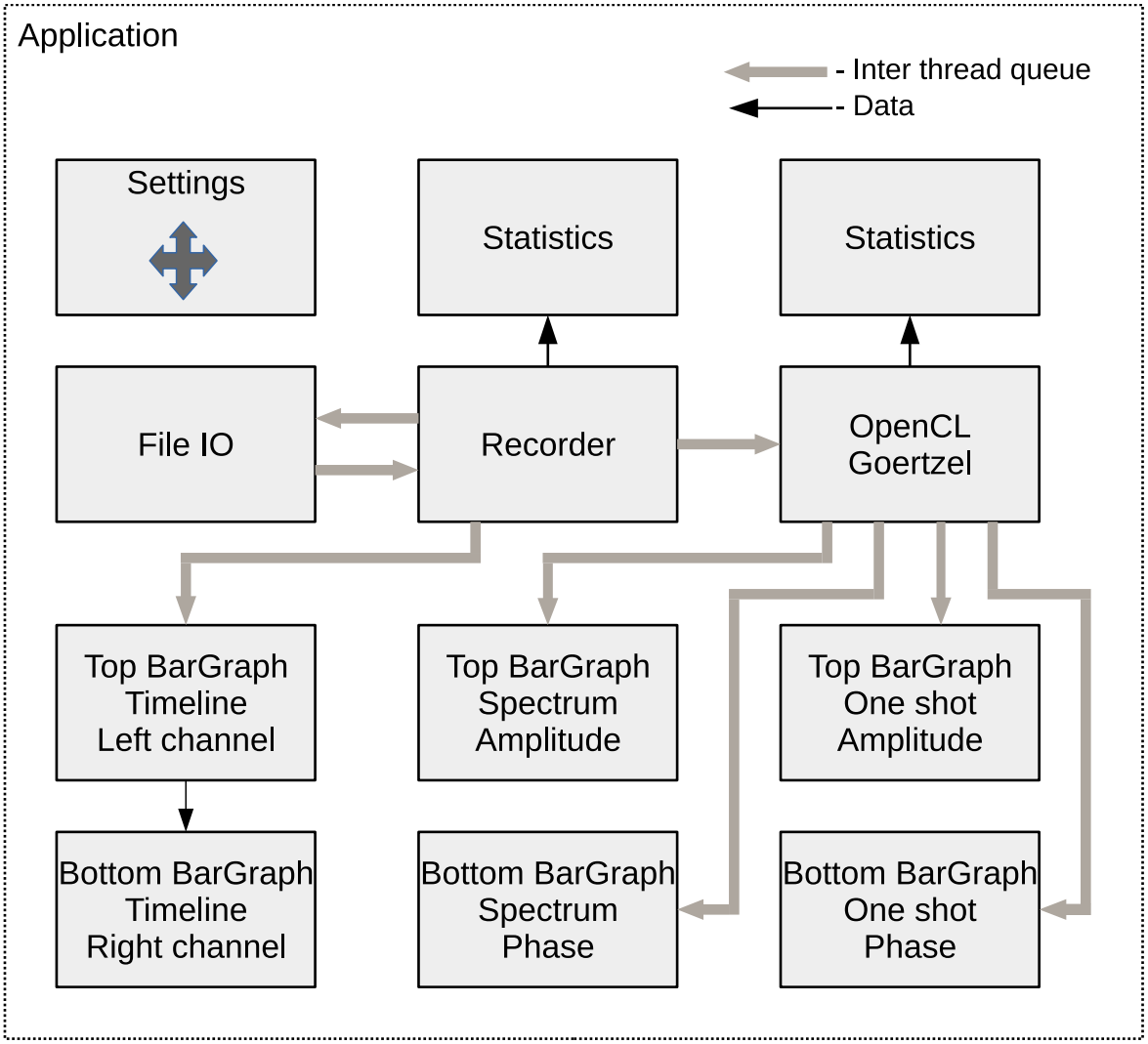
Více o použitých knihovnách najdete v kapitole 5.

4.3 Blokové schéma

Zaměřme se nyní na obrázek 4.1. Vidíme na něm blokové schéma programu *Sound Analyzer*. Nyní proberu jednotlivé části podrobněji.

4.3.1 Inter thread queue

Aplikace jako tato, která funguje jako přehrávač, záznamník, kreslí do okna grafy, musí být nutně vícevláknová. Už jen proto, že závisí na spoustě hardwaru a služeb



Obr. 4.1: Blokové schéma programu Sound Analyzer

současně a synchronizace mezi nimi je téměř nemožná. Procesor totiž střídá vlákna po cca 10ms a pokud by se něco zdrželo ve vymezeném čase pro hlavní vlákno, docházelo by zcela jistě k trhání a zpomalení celé aplikace. Tato aplikace má tedy několik téměř samostatných funkcí, každá běžící ve svém vlákně. Problém souběhu řeší tato *mezivláknová fronta*.

Blok *mezivláknová fronta* tedy realizuje frontu opatřenou zámky. Současně poskytuje zásobu segmentů, které může první vlákno naalokovat, zapsat do nich data a zařadit do fronty. Druhé vlákno pak segment vyjme z fronty a vrátí. První vlákno nad to ještě může označit segment za poslední a když je přečten, další přečtené segmenty jsou již vždy *nullptr*. Pro účely ladění se kontroluje, zda segment zařazený do fronty a vrácený segment, byly vydány touto frontou. Typická velikost zásoby segmentů je spočítána na cca 1 sekundu vzorků, minimálně však na 10 segmentů.

4.3.2 *OpenCL* Goertzel

OpenCLGoertzel je stěžejní komponenta celé aplikace. Vychází ze vztahu 2.23. Než ale začne samotný výpočet, je třeba spočítat matice A a D. Matice A je jednoduchá. Ale i zde lze provést dvě optimalizace. Protože budeme počítat s typem *float – 32 bitů*, lze očekávat možné nepřesnosti dané malým počtem bitů. Jedno řešení je tedy jasné – počítat konstanty v datovém typu *double – 64 bitů* a převést jen před vlastním zapsáním do grafické karty. Další operace jsou výpočty matice A^k, kde se na k díváme jako na binární číslo.

$$\begin{aligned} k &= \ldots k_3.2^3 + k_2.2^2 + k_1.2^1 + k_0.2^0 \\ A^k &= \ldots k_3((AA)(AA))((AA)(AA))k_2(AA)(AA)k_1AAk_0A \end{aligned} \tag{4.1}$$

tak počítám násobení matice nejvýše 6krát a nikoli 15krát jako kdybych počítal prostým cyklem. Pokud mám různé mocniny A, snadno určím matici D.

Pokud bychom řekli, že sčítání mezivýsledků a počítání jednoho mezivýsledku dle vzorce 2.23 trvá stejně dlouho, (jsou tam stejné matematické operace) dojdeme k závěru, že optimální by bylo rozdělit výpočet série N vzorků na $N_M.N_L$ operací, kde N_M a N_L jsou stejné (přibližně odmocniny z N). Takto také tato komponenta počítá kolik cyklů (L) a kolik jader (M) bude použito pro výpočet. Já jsem v programu *Sound Analyzer* ze začátku skutečně proměnné L a a M zavedl. Nyní se to již zdá být zbytečné, nicméně je to v programu stále, protože se ukázalo, že předpoklad ze začátku tohoto odstavce není úplně splněn a možná bude lepší jiný algoritmus pro stanovení L a M. Protože je možné doplnit řadu vzorků nulami, aniž by se ovlivnil výpočet, udělám to a zajistím si tak, že programy jader budou jednodušší, protože

všechna jádra budou počítat stejný počet cyklů. Nakonec ani není třeba řešit, pokud by odmocnina z N bylo iracionální číslo, prostě proměnné L a M zaokrouhlím nahoru. Ještě dodejme, že nuly se musí přidat před vzorky signálu, nikoli za něj.

Tato komponenta umí počítat nejen ze vzorků čerstvě přišlých z *mezivláknové fronty*, ale i několika až 16386 vzorků předchozích. Je tak možno získat větší přesnost, ale za cenu toho, že počítáme z delšího intervalu a výsledky tak nemusejí odpovídat okamžité realitě.

Na závěr sekce se zmíním o velikosti vektorů, se kterými lze počítat v knihovně *OpenCL*. V dokumentaci *OpenCL* je uvedena maximální velikost 16 s tím, že je implementačně závislá a program si musí zjistit skutečnou velikost vektoru. V případě mého grafického čipu *AMD 7850* je to hodnota 8. Ovšem já potřebuji složky vektoru sečíst a funkce, která to provádí, má podle dokumentace další omezení – na velikost 4. Nakonec tedy počítám s velikostí 4. *OpenCL* neumožňuje maticové operace.

Vlastní *jádro*D.30 je velmi jednoduché:

```
__kernel void main(
    __constant float*   streams,   //vstupní segment signálu
    __constant float*   D1,        //horní řádek matice D
    __constant float*   D2,        //dolní řádek matice D
    unsigned int        Channels,  //počet kanálů
    unsigned int        K,         //délka vektoru OpenCL, konstanta 4
    unsigned int        L,         //počet cyklů, respektive délka vektoru
                                   // se kterým pracujeme / 4
    unsigned int        M,         //počet jader spuštěných nad algoritmem
                                   // pro jeden kmitočet, jeden kanál
    unsigned int        Length,    //délka segmentu
    __global float2*    outStreams //výtupní vektor
)
{
    unsigned int indexFrequency = get_global_id(0);
    unsigned int indexChannel = get_global_id(1);
    unsigned int indexItem = get_global_id(2);
    unsigned int i;
    float2 acc={0.0,0.0};
    __constant float4* data = (__constant float4*)&streams[indexChannel
                    * Length + indexItem * L * K];
    __constant float4* mxD1 = (__constant float4*)&D1[indexFrequency
                    * L * K];
```

```
    __constant float4* mxD2 = (__constant float4*)&D2[indexFrequency
                    * L * K];
    for(i=0;i<L;i++)
    {
        acc.x = acc.x + dot(data[i] , mxD1[i]);
        acc.y = acc.y + dot(data[i] , mxD2[i]);
    }
    outStreams[indexFrequency * Channels * M + indexChannel * M
                    + indexItem] = acc;
}
```

Funkci, jež má modifikátor ___*kernel* D.30 umíme jako jedinou spustit z hlavního programu a říkáme jí *jádro*. Naše funkce má 9 parametrů, které mají před svými jmény modifikátory specifikující, ve které paměťové třídě se proměnná nachází. Více o těchto paměťových třídách najdete v kapitole 5.1. Jak je vidět, program pouze realizuje cyklus:

$$outStreams[i_c][n] = \\ (\sum_{n=0}^{L-1} streams[i_c][n].D_1[i_f][n], \sum_{n=0}^{L-1} streams[i_c][n].D_2[i_f][n]), \tag{4.2}$$

kde *stream* je blok vstupního signálu a $D_1[kmitočet], D_2[kmitočet]$ jsou přepočítané řádky matice D. Index kmitočtu je pak i_f a index kanálu je i_c.

4.3.3 Recorder

Recorder plní mezivláknovou frontu segmenty pro výpočet. Načítá je ze vstupního zařízení. Implicitní je určitě mikrofon, dále je možnost vybrat monitor výstupu, tedy provádět výpočet spektra nad signálem poskytovaným jinou aplikací. Volitelně může navíc ještě segmenty zkopírovat k zapsání do souboru komponentou popsanou níže.

Druhý režim *recorderu* je přehrávání. Segmenty pořízené komponentou *FileIO* jsou přehrány na standardním výstupu a zkopírovány do fronty pro výpočet spektra. Tím že jsou do této fronty kopírovány právě v okamžiku po jejich přehrání, je zajištěno časování.

Pokud je zvolen monitor standardního výstupu, nebo monitor HDMI výstupu, je signál zkreslen. Domnívám se, že je to ochrana proti kopírování. Záznam zvuku z těchto monitorů totiž není zatížen ani minimálním šumem.

4.3.4 File IO

Souborové operace v programu *Sound Analyzer* jsou načítání zvukového souboru ve formátu *wav* do *mezivláknové fronty* a samozřejmě i směrem z *mezivláknové fronty* do souboru. V případě, že služby souborového systému jsou pomalejší než přísun segmentů, jsou ve výsledném *wav* souboru nespojitosti v tom smyslu, že je soubor je validní, dá se přehrát, ale jeho obsah je nespojitý. Je také vypisováno hlášení na konzoli. V případě, že souborový systém není schopen dodat dostatek dat pro přehrávání, mezivláknová fronta je prázdná a přehrávání je přerušované. I zde je vypisováno varovné hlášení na konzoli.

4.3.5 Stats

Komponenta statistiky je velmi jednoduchá. Má jen frontu hodnot a vždy zná jejich součet. Umí tak poskytnout kmitočet obnovování, vzorkovací kmitočet a průměrný čas výpočtu spektra. Jednou za 0,5s vrací příznak, že komponenta, která statistiku vlastní, může zobrazit své data v uživatelském prostředí.

4.3.6 Settings

Nastavení je malá množina párů hodnota-klíč, sloužící k uchování nastavení. Její výhoda je přístup jak z *C++*, tak z jazyka *QML*. Po skončení aplikace se její obsah uloží do souboru a po startu je načten zpět.

4.3.7 Application

Aplikace všechny výše uvedené komponenty spojuje. Komponenty v *Sound Analyzeru* jsou striktně odděleny a pokud spolu potřebují komunikovat nebo něco zobrazit, využívají funkce komponenty aplikace. Aplikace je jediná komponenta svázaná s grafickou knihovnou *Qt*. Je ještě jedna funkce, kterou aplikace má. Při zapnutí nastaví všechny komponenty podle nastavení v komponentě settings a při změně pohledu mění propojení mezivláknových front.

4.4 Licence

Všechny použité knihovny jsou šířeny, nebo podporují *GNU licenci*, takže i má práce musí být šířena pod licencí *GNU GPLv2*.

5 KNIHOVNY POUŽITÉ PŘI SESTAVOVÁNÍ PROGRAMU SOUND ANALYZER

V této kapitole vám přiblížím knihovny použité na projektu *Sound Analyzer*. Základem všech je ovšem stadardní knihovna jayka *C++*.

5.1 Knihovna openCL

Tato kapitola je výtahem z dokumentace k openCL ([1]). Úplnou dokumentaci lze najít `kronos.org`.

5.1.1 OpenCL framework

OpenCL *framework* obsahuje tři hlavní části:

- OpenCL platformní vrstva – dovoluje *hostiteli* zjišťovat možnosti *zařízení*, a vytváří *kontext*.
- OpenCL runtime – všechny operace manipulující s kontextem.
- Překladač openCL – vytváří spustitelné objekty obsahující *jádra*. Vychází z ISO C99.

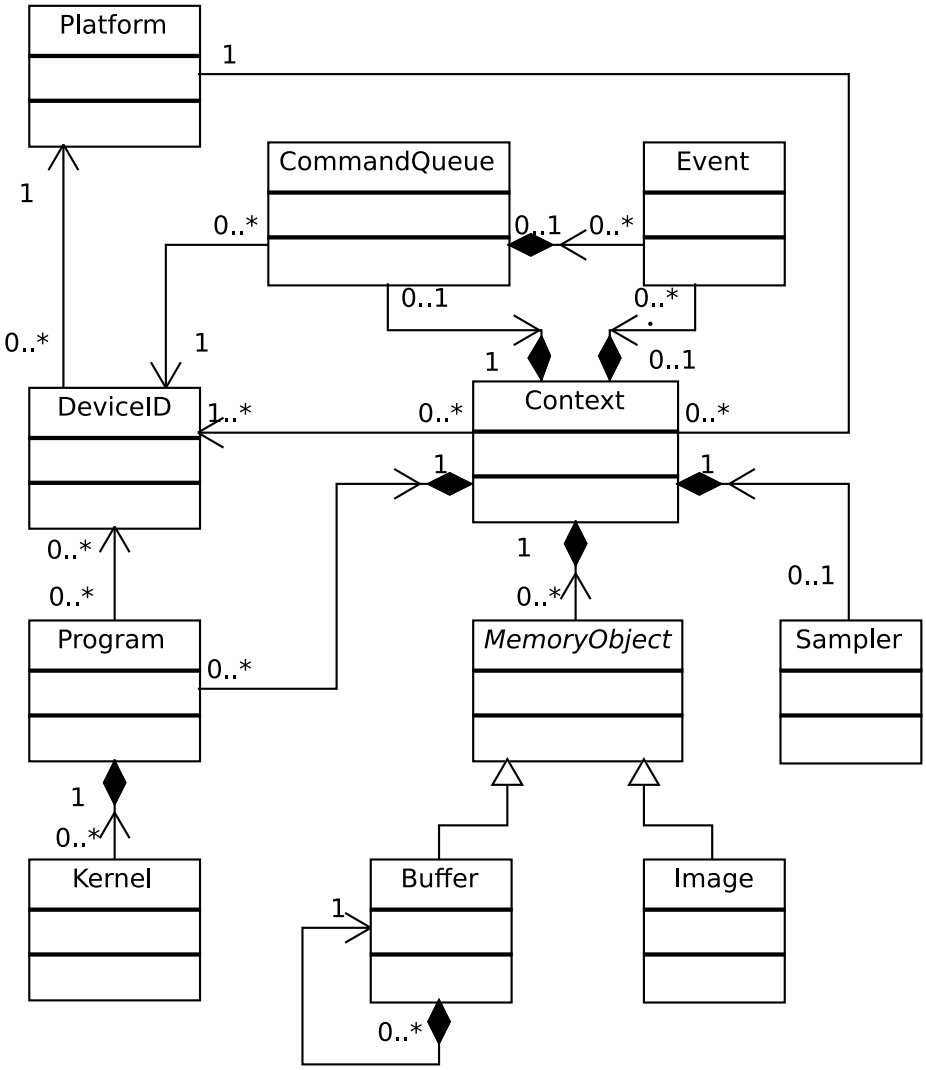
5.1.2 Architektura openCL

OpenCL je průmyslový standard pro programování heterogenních skupin CPU, GPU a dalších zařízení organizovaných do jedné platformy. Je to více než jen jazyk. Je to celý framework který obsahuje programovací jazyk, API, knihovny a runtime systém pro podporu vývoje. OpenCL poskytuje nízkoúrovňovou abstrakci hardware plus framework.
V dalších kapitolách blíže popíši následující modely:

- Model platformy.
- Model paměťový.
- Model prováděcí.
- Model programovací.

5.1.3 Diagram tříd

Obrázek (viz obr. 5.1) popisuje specifikaci openCL jako UML diagram tříd. Jsou na něm znázorněny jen základní třídy, nikoli jejich atributy.



Obr. 5.1: UML Diagram tříd

5.1.4 Model platformy

Model se sestává z *hostitele* připojeného k jednomu nebo více openCL *zařízením* . Toto *zařízení* je dále děleno na jednu nebo více *výpočetních jednotek* a tyto *výpočetní jednotky* jsou dále děleny na jednu nebo více *zpracujících jednotek* .

OpenCL *aplikace* běží na *hostiteli* OpenCL *aplikace* posílá *příkazy* z *hostitele* ke zpracování výpočtů do *zpracujících jednotek* v rámci *zařízení*. *Zpracující jednotky* v rámci *výpočetní jednotky* provádějí jednu sadu instrukcí jako *SIMD* jednotky, nebo jako *SPMD* jednotky, kde každá *zpracující jednotka* má svůj čítač instrukcí.

Podpora různých verzí

OpenCL je navrženo s podporou více *zařízení* s různými možnostmi společně provozovanými pod jedním *hostitelem*. To ovšem znamená, že každé zařízení může splňovat jinou verzi knihovny openCL. Máme tedy verzi runtime prostředí a zvlášť verzi u každého *zařízení*. Dále každé *zařízení* může podporovat svou vlastní verzi jazyka openCL C.

5.1.5 Prováděcí model

Když je *jádro* pověřeno *hostitelem* k provedení, je třeba definovat indexový prostor. Instance *jádra* se nazývá *pracovní položka* a je definována bodem v indexovém prostoru, který definuje *globální ID* pro tuto *pracovní položku*. Pracovní položka je svázánu s jedním, konkrétním jádrem výpočetního zařízení.*Pracovní položky* jsou organizovány v *pracovních skupinách* . *Pracovní skupiny* mají unikání ID *pracovní skupiny*. *Pracovní položky* můžeme v indexovém prostoru identifikovat stejným způsobem, Také mají své *globální ID*. Mimo to je ale můžeme identifikovat dvojicí *globální ID pracovní skupiny* a *lokálním ID* (v rámci této *pracovní skupiny*).
Indexový prostor, který je definován knihovnou openCL, je N-rozměrná pole. Nazývají se *NDRange*. N může být jedna, dvě nebo tři. To znamená, že i indexy jsou N-prvkové. Položky indexu začínají vždy od nuly.
Index tedy definuje určitou *pracovní skupinu* v rámci *výpočetní jednotky*. Stejným způsobem je také definován index *pracovní položky* v rámci *pracovní skupiny*.

Kontext a fronty příkazů

Kontext vytváří a spravuje *hostitel* prostřednicvím funkčních volání openCL API. *Hostitel* v *kontextu* vytvoří zejména *frontu příkazů* , aby mohl zařízení ovládat. Do těchto *front příkazů* zapisuje *příkazy* k provedení. Mezi tyto *příkazy* patří:

- Příkazy k provádění jádra.
- Příkazy pro práci s pamětí.

- Synchronizační příkazy.

V jednom *kontextu* může být více *front příkazů* které jsou prováděny nezávisle na sobě.

5.1.6 Paměťový model

Pracovní položky rozlišují čtyři druhy pamětí:

- Globální paměť.
- Paměť konstant.
- Lokální paměť.
- Soukromá paměť.

Přístup z	**Global**	**Constant**	**Local**	**Private**
Hosta	Dynamická alokace Přístup pro čtení / zápis	Dynamická alokace Přístup pro čtení / zápis	Dynamická alokace Není přístup	Bez alokace Není přístup
Jádra	Není alokace Přístup pro čtení / zápis	Statická alokace Přístup pro jen čtení	Statická alokace Přístup pro čtení / zápis	Statická alokace Přístup pro čtení / zápis

Tab. 5.1: Typy pamětí v openCL

Aplikace běžící na *hostiteli* vytváří *paměťové objekty* v globální paměti.

Paměť *hostitele* a *zařízení* jsou na sobě nezávislé. Nicméně je potřeba aby *zařízení* a *hostitel* spolu nějakým způsobem komunikovaly. Toho je docíleno kopírováním paměti nebo jejím sdílením. Kopírování může býti jak blokující, tak neblokující. Blokující nebo neblokující může být rovněž mapování paměti. *Aplikace* většinou mapuje paměť na nezbytně nutnou dobu a když jsou všechny operace čtení a zápisu dokončeny, opět paměť odmapuje.

5.1.7 Programovací model

OpenCL nabízí dva programovací modely, respektive dva základní přístupy. **Datově paralelní model** a **úlohově paralelní model**.

Datově paralelní model

V datově paralelním programovacím modelu provádějí všechny *pracovní položky* v rámci *pracovní skupiny* současně jednu sadu instrukcí. Model je rozdělen na dvě kategorie. U **explicitní** definuje aplikace, jak bude *pracovní skupina* rozdělena na *pracovní položky*. U **implicitní** aplikace pouze definuje, kolik pracovních položek chce použít a rozdělení *pracovní skupiny* na *pracovní položky* nechá na knihovně openCL.

Úlohově paralelní model

V úlohově paralelním modelu je každé *jádro*D.30 vykonáváno zcela nezávisle na jiných. Znamená to, že je prováděno v *pracovní skupině*, která má jednu jedinou *pracovní položku*. V tomto modelu je zdůrazněna paralelnost:

- používáním vektorových datových typů.
- řízením více úloh.

Synchronizace

Pracovní položky položky mohou být synchronizovány mezi sebou využitím *bariér*. Pro synchronizaci *pracovních skupin* mezi sebou, žádný takový mechanismus neexistuje.

Příkazy mají také možnost používat *bariéry*. Tyto *bariéry* zajistí, že všechny příkazy zařazené před touto *bariérou* ve *frontě příkazů*, budou provedeny dříve, než se začne s prováděním dalších *příkazů*.

Všechny *příkazy hostitele*, které vkládají *příkazy* do *fronty příkazů*, vracejí objekt **události**. Další *příkaz* ve *frontě příkazů* může na tuto událost čekat.

5.1.8 Paměťové objekty

Jsou dvě kategorie *paměťových objektů*: *obrázky* a *buffery* . *Buffer* je řada objektů nějakého typu a můžeme k nim přistupovat pomocí rukojeti (handle D.23). Naproti tomu je struktura *obrázku* skryta a k *obrázku* můžeme přistupovat pouze pomocí speciálních funkcí jádra. *Obrázek* nemusí mít stejný formát u *hostitele* a uvnitř *jádra*.

5.2 OpenGL

5.2.1 Historie OpenGL

Do devadesátých let minulého století panovalo všeobecné nadšení z výpočetního výkonu počítačů dovolujícího kreslit 3D grafiku. Byla řada výrobců hardware. Bohužel byla i velká řada výrobců vytvářejících vlastní grafické knihovny a ovladače.

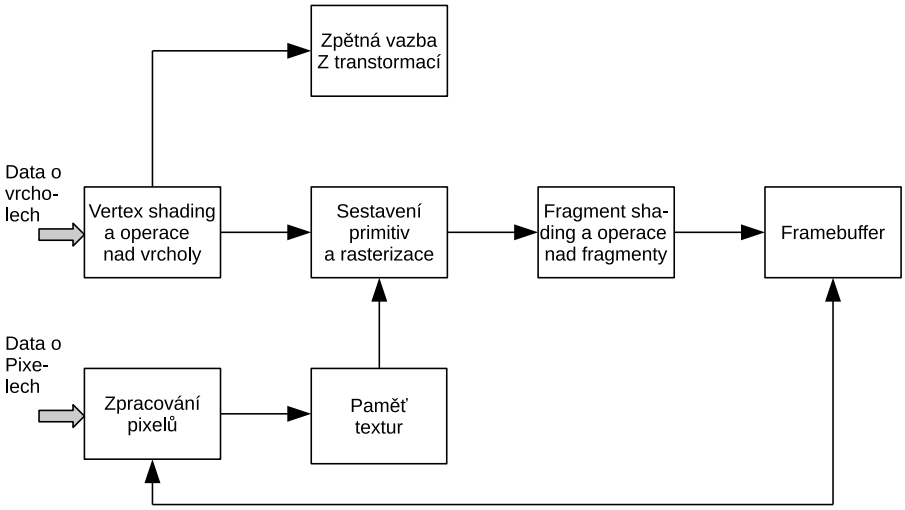
Žádný tak neuměl zacházet s jakýmkoli hardwarem a měl tedy jen malou omezenou množinu aplikací. Vznikla tak nutnost unifikace.

V další dekádě byla špičkou v grafice společnost *SGI*. Vyvinula svůj standard *IrisGL*. Jeho nevýhodou bylo začlenění funkcí pro *X11* a použití patentovaných algoritmů. Proto začali ostatní výrobci pracovat na rozhraní od *IrisGL* odvozeném a nazvali jej *OpenGL*.

V roce 1992 bylo pro účely správy *OpenGL* vytvořeno konsorcium *OpenGL* ARB. V roce 2006 byla *OpenGL* předána konsorciu *Kronos Group*.

5.2.2 Vykreslovací řetězec

OpenGL popíšu jen lehkým úvodem. Začnu s popisem vykreslovacího řetězce, jak jej uvedli Procházka, Koubek a a Andrýsková v [4]. V současných verzích *OpenGL* je ještě složitější, takto odpovídá verzi *OpenGL* 2.0.



Obr. 5.2: *OpenGL* vykreslovací řetězec

Framebuffer

Framebuffer je paměť jejíž obraz je bude zobrazen. Framebuffery jsou většinou dva nebo více, jeden se synchronně posílá na zobrazovací zařízení, ve druhém je tvořen následující snímek.

Zpracování pixelů

Grafický subsystém, jakožto rasterové zařízení umožňuje práci s rasterovými daty. V postatě je schopen je jen zapsat/vyčíst framebufferu a zapsat do paměti textur.

Zpracování vrcholů

Vrchol (vertex) není jen bod v prostoru. V *OpenGL* má vertex kromě své pozice řadu dalších uživatelských vlastností, které lze z programu nastavit a také je při zpracování vrcholu využít. Dříve byla tato jednotka fixní, grafický čip prováděl vždy stejnou operaci nad vrcholy. Nyní, od *OpenGL* 2.0, je vykreslovací řetězec programovatelný a tato jednotka může vykonávat s vrcholem jakoukoli operaci pomocí programu v jazyce *GLSL*. Lze tak realizovat základní operace s vrcholem, jako otočení, změna měřítka, posunutí, ale i třeba efekty jako jsou vlny na hladině, efekt lomení paprsku na hladině a podobně.

Sestavení primitiv a rasterizace

Když předchozí jednotka stanovila již definitivní souřadnice ploch, je třeba z každé jednotlivé plochy získat množinu bodů, které odpovídají pixelům na zobrazovacím zařízení, fragmentům.

Fragment shading a operace nad fragmenty

Stejně jako vertex není jen bod v prostoru, tak i fragment obsahuje více informací než jen polohu a barvu. Tato jednotka je od *OpenGL* 2.0 programovatelná v jazyce *GLSL* a umožňuje nanášet textury, vybarvovat pixel podle jeho polohy (barevný přechod), některé efekty se světlem, míchání barev a podobně.

5.3 OpenAL

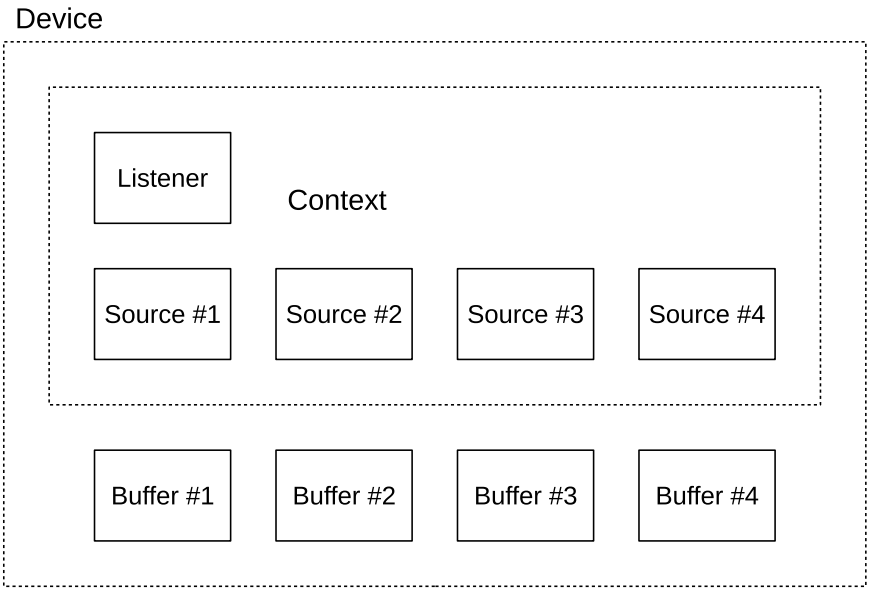
OpenAL („Open Audio Library“) je softwarový interface k audio hardwaru. Interface poskytuje několik funkcí, kterými program modeluje objekty generující ve 3D prostoru a samozřejmě zajišťuje mixování všech zvuků od všech objektů. Nepodporuje přímo stereo ozvučení, naproti tomu prostě propočítá *streamy* pro potřebný počet kanálů. Nabízí také zpracování efektů jako je Doplerův jev, odrazy, překážky, vysílání a dozvuky.

OpenAL je implementováno pro většinu platforem.

5.3.1 Historie *OpenAL*

OpenAL bylo původně vyvinuto společností *Loki Software* jako doplněk ke knihovně *OpenGL* asi okolo roku 1998. Dále jej udržuje Creative Technology, která jako první uvolnila hardwarově akcelerovanou *OpenAL*.

5.3.2 Stručný popis



Obr. 5.3: *OpenAL* struktura API

V *OpenAL* je jeden Posluchač (Listener) který reprezentuje výstup. Ten je tvořen jedním nebo mnoha, klidně stovkami zvuků od Zdrojů (Sources). Každý zdroje má kromě zaznamenaného zvuku také přiřazen vzorkovací kmitočet, polohu a vektor rychlosti (pro výpočet Dopplerova efektu). Zdroj přehrává data z jednoho nebo fronty několika bufferů.

5.4 *Qt*

Qt je multiplatformní framework. Je definován snad na všechny desktopové i mobilní operační systémy. Je vydáván ve dvou licencích. *LGPL* umožňuje volné šíření *Qt*, ale neumožňuje statické linkování, jsou dostupné jen základní částni bez přídavných modulů, programátor musí dát návod jak vyměnit dynamicky linkované knihovny za novější, licence jeho produktu musí být také *LGPL*. Výše uvedené nedostatky řeší licence proprietární.

5.4.1 Historie

Historie *Qt* se začala datovat v roce 1990 v Norsku. V roce 1995 bylo první vydání firmou Trolltech. V roce 2005 bylo *Qt* zcela přepracováno na verzi 4.0. V roce 2008 byl Trolltech koupen Nokií. V roce 2010 Nokia vydává Symbian, operační systém

pro smartphony založený na *Qt*. V roce 2012 byl představen jazyk *QML* a *Qt* 5.0. *Qt* se stalo samostatnou společností v roce 2014.

5.4.2 Koncept signál-slot

V *Qt* je alternativa k callback funkcím, je to koncept signál-slot. Když nastane nějaká událost, je emitován signál. Slot je funkce , která je volána jako odezva na emitovaný signál. V *Qt* je řada předdefinovaných signálů, ale lze jednoduše tvořit signály vlastní.

Signály a sloty jsou typově bezpečné, čímž se míní skutečnost, že parametry, respektive typy a počet parametrů jsou u signálu i slotu stejné.

Je varianta, kde se signál a slot propojí jmény funkcí (string based), překlad jména funkce na volání skutečné funkce se provádí za běhu aplikace.

Signály a sloty jsou vláknově bezpečné, dokonce se ve vícevláknových aplikacích požívají pro mezivláknovou komunikaci.

5.4.3 *QML*

Ve verzi 5 *Qt* představilo jazyk *QML*. Je to deklarativní jazyk, který popisuje layout uživatelského rozhraní. Nad to obsluha událostí je nyní ve stejném *QML* souboru v jazyce javascript. Celý kontroler se tak přesunul na novou platformu.

Přesto je stále možné tvořit ovládací prvky pro *QML* v jazyce C++, volat z *C++ javascriptové* funkce a volat z *javascriptových* funkcí *C++* funkce.

Velkou novinkou je zavedení *vlastností* (properties) objektů v *QML*. Nastavení vlastnosti nemusí být jen konstanta, ale i složitější konstrukce v jazyce *javascript*. Pokud se jedna nebo více proměnných, na kterých *vlastnost* závisí změní, dojde k automatickému přepočítání a je stanovena nová hodnota proměnné-vlastnosti.

Úskalím tohoto konceptu je, že při neopatrném zacházení s vlastnostmi mohou vznikat cyklické závislosti, kde jedna *vlastností* závisí na druhé, ale ta současně na první. Často nejsou tyto závislosti zřejmé.

Předchozí tzv. widgetové aplikace se nedají s tímto novým konceptem kombinovat. Je ale stále možné takovéto widgetové aplikace v *Qt* 5 vytvářet.

Daní za *QML* je nižší výkon aplikace a vysoké paměťové nároky. Výhodou má naopak být rychlost vývoje a aplikace *javasriptu* je velmi stabilní i po chybě programátora.

6 MĚŘENÍ PŘÍNOSU VYUŽITÍ GRAFICKÉHO ČIPU

Aby bylo možno nějakým způsobem prověřit funkčnost programu, jsou do něj přidány některé měřící funkce.

6.1 Počet využívaných jader

První informace která je obecně zajímavá, je kolik jader *GPU* potřebuji, kolik jich mám k dispozici a kolik jsem chopen jich reálně využít. Proto je na liště nástrojů programu *Sound Analyzer* sedm údajů: Globální $N/C/M$, lokální $N/C/M$ a maximální počet jader současně spustitelných. Více o počtu jader najdete v manuálu, sekci *Počet jader použitých při výpočtech* A.8.6.

6.2 Průměrná délka výpočtu za poslední dobu

Program měří čas každého provedení výpočtu v mikrosekundách. Využívá přitom komponentu *Stats*, které implementuje frontu 250 hodnot. Z těchto 250 hodnot je počítán aritmetický průměr.

6.3 Metodika měření

Abychom změřili uspořený čas *CPU*, bylo by dobré nejdříve provést měření které nezatěžuje *GPU*, tedy nějaký jednoduchý výpočet. Potom provedu měření zatížení všech jader, ale tak, aby globální počet jader se rovnal lokálnímu počtu jader, tedy jeden výpočetní cyklus. Jako třetí měření zatížím *GPU* tak, abych maximálně využil paměť. Při velkých požadavcích *OpenCL* jednoduše vrátí chybu.

Při všech těchto měřeních by bylo jistě zajímavé vědět, jak je zatížen počítač jako takový. Ideální na to je program *mstats* z balíku *sysstat*. Budu počítat zatížení procesoru za periodu 10s.

Ještě připomenu, že v programu *Sound Analyzer* se délkou segmentu myslí délka, kterou *recorder* načte ze vstupního zařízení. K se přičítá *overlap* vzorků z předchozích segmentů.

Tab. 6.1: Parametry testovacího počítače

Operační systém	Linux Mint 17.3 64bit
CPU model	AMD FX-8350
CPU jader	8
CPU kmitočet	4000MHz
Paměť	32GB
Chipset	AMD 990FX
GPU model	AMD Radeon HD7850
GPU paměť	2GB
GPU jader	1200
GPU kmitočet	1000MHz

6.4 Použité počítače

6.4.1 PC1

6.4.2 Notebook1

Tab. 6.2: Parametry testovacího notebooku

Operační systém	Linux Mint 18.1 64bit
CPU model	Intel i7-4850HQ
CPU jader	4 (8 vláken)
CPU kmitočet	2300MHz
Paměť	16GB
Chipset	-
GPU model	NVidia GeForce GT 750M Mac edition / Intel IRIS 5200
GPU paměť	2GB / 128MB
GPU jader	384 /-
GPU kmitočet	926MHz/-

6.4.3 Další podmínky měření

$f_{vz} = 8000Hz$
Na počítačích byl spuštěn pouze program *Sound Analyzer* a terminál s programem *mstats*.

6.5 Naměřené hodnoty

6.5.1 PC1

Tab. 6.3: Naměžené hodnoty pro pro testovací počítač

Kmit. počet [−]	Segm. délka [−]	Over-lap [−]	Globální NDRange [−/ − /−]	Lokální NDRange [−/ − /−]	CPU doba [μs]	CPU využití [%]	GPU doba [μs]	GPU využití [%]
0	0	0	0/0/0	0/0/0	-	4,83	-	-
1	4	0	1/1/1	1/1/1	105	6,54	160	7,13
1	512	15872	1/1/64	1/1/64	230	5,3	410	5,0
16384	4	0	16384/1/1	1024/1/1	300	24,4	-	-
16384	512	15872	16384/1/64	16/1/64	61090	82,64	-	-
64	512	15872	64/1/64	16/1/64	690	5,25	-	-
64	512	15872	64/1/64	4/1/64	-	-	430	5,20

6.5.2 PC1 - měření využití jader

Tab. 6.4: Naměřené hodnoty pro měření využití jader pro testovací počítač

Lokální počet jader [μs]	Globální počet jader [μs]								
	1	2	4	8	16	32	64	128	256
1	154	-	-	-	-	-	-	-	-
2	154	154	-	-	-	-	-	-	-
4	154	154	154	-	-	-	-	-	-
8	155	155	155	155	-	-	-	-	-
16	155	155	155	155	155	-	-	-	-
32	156	156	156	156	156	156	-	-	-
64	157	157	157	157	157	157	157	-	-
128	157	157	157	157	157	157	157	157	-
256	158	158	158	158	158	158	158	158	158

Tab. 6.5: Naměžené hodnoty pro pro testovací notebook

Kmit. počet [−]	Segm. délka [−]	Over-lap [−]	Globální NDRange [−/ − /−]	Lokální NDRange [−/ − /−]	CPU doba [μs]	CPU využití [%]	GPU doba [μs]	GPU využití [%]
0	0	0	0/0/0	0/0/0	-	4,11	-	-
1	4	0	1/1/1	1/1/1	-	-	105	6,41
1	512	15872	1/1/64	1/1/64	-	-	610	4,50
16384	4	0	16384/1/1	1024/1/1	-	-	190	8,85
16384	512	15872	16384/1/64	16/1/64	-	-	161410	10,12
64	512	15872	64/1/64	16/1/64	-	-	1265	4,62

6.5.3 Notebook1

6.6 Shrnutí

Prvně je třeba vysvětlit nenaměřené hodnoty. U *NVidie* vrátí inicializační funkce chybu vždy, když se pokusíme použít výpočet procesorem. U *AMD* je nějaký limit na paměť okolo 4000 jader, po jeho překročení *OpenCL* vrátí chybu „Failed to execute kernel! (2) err=0xfffffffb“, což znamená „Nemohu naalokovat paměť“.

Z prvních řádků obou měření je vidět, kolik času zabírá hlavní smyčka programu, která po cca 100ms překresluje okno. Lehce silnějším se zdá být procesor *i7*.

Poté jsem spustil jedno jediné jádro a vidím, že požadavky na *CPU* vzrostly asi dvojnásobně, protože počítáme cca 2000krát za jednu sekundu. Je též patrné, že funkční volání, které spouští *OpenCL*, trvá asi 105μs na *NVidii* a 160μs na *AMD*. Pod tyto časy už se zřejmě nepůjde dostat.

Pak se dá zkusit počítat jednu dlouhou řadu s 64 mezivýsledky, tedy s využitím 64 jader. Jak patrno, potřebný výkon *CPU* značně klesl, protože počítáme s nižším obnovovacím kmitočtem, asi jen 15,6 Hz, oproti 2000Hz v případě předchozím. U *AMD* to zvedlo potřebu *CPU* o pouhých 0,17% a doba strávená výpočtem je větší asi jen čtyřnásobně. U *NVidie* stoupla potřeba *CPU* o 0,39% a výpočet je delší méně než šestinásobně.

Ještě se podívejme na poslední řádky tabulek naměřených hodnot. Zde počítáme hodnoty pro 64 kmitočtů a 16384 vzorků dlouhé řady. Zatížení *CPU* je pro oba grafické čipy minimálních 0,39%. Grafický desktopový čip *AMD* je ale výrazně rychlejší než mobilní *NVidie*. Z předposledního řádku první tabulky je vidět, že počítání CPU je asi o polovinu pomalejší.

Nakonec jsem ještě změřil, jak moc pomáhá grafický čip procesoru. Měřil jsem tak, že jsem upravil program *SoundAnalyzer* na možnost ovlivnění počtu lokálních

jader. Zastavil jsem $1, 2, 4, 8, 16, 32, 64, 128$ *a* 256 jader jako požadavek (globální) tak jako lokální. Výsledek je celkem překvapující a je v tabulce 6.5.2. Ať jsem nastavil jakýkoli počet jader, doba strávená výpočtem byla vždy 154 *až* $158\mu s$, tedy jen lehce úměrná počtu spouštěných výpočetních jednotek. Knihovna *OpenCL* tedy nebere parametr $lokálníNDRange$ jako počet procesorů, které chceme použít, ale použije vždy veškeré možné. Čím více mezivýsledků sčítám, tím je provádění delší, ačkoliv ten rozdíl na 256ti výpočetních jednotkách byl pouze $4\mu s$. Protože ale není možné spustit pouze jedno jádro, nejsem schopen s přesností porovnat pararelní a seriový přístup. Navíc doba provádění je tak malá, že ji lze jen těžko měřit prostředky PC. Navíc tato doba hodně kolísá díky tomu, že v počítači běží řada různých procesů. Přes to lze říci, že paralelizace se vyplatila. Výpočet 256krát delší zátěže trvá jen o cca 0,25% déle.

Závěrem sekce ještě uvedu srovnání počtu jader ve vztahu k předpokladům daných hardwarem. Ačkoli *GPU AMD* má 1200 jader, *OpenCL* nabízela k použití vždy jen 256 jader. Na proti tomu 384jádrová *NVidia* nabízela k výpočtu vždy 1024 jader. Softwarové zpracování poskytovala pouze *GPU AMD* a nabízela vždy 1024 jader.

6.7 Závěr měření

Z výsledků je patrné, že využití GPU ke zpracování signálu skutečně může výrazně odlehčit hlavnímu procesoru. Je ale třeba brát v úvahu několik předpokladů. Zatížení *GPU* nesmí být nesmyslně malé. Je tu totiž velká režie s voláním *OpenCL* funkcí a jednoduše se to nemusí vyplatit. Zejména mám na mysli počítat v každé výpočetní jednotce dostatečně dlouhý blok signálu. Rozdělení segmentu signálu na bloky se tedy vyplatí jen u segmentů delších něž cca 2048 vzorků.

Je to patrné z prvních řádků tabulek, kde přenesení výpočtu na*CPU* výpočet výrazně urychluje.

Aplikace by také měla být vícevláknová s jedním vyhrazeným vláknem pro výpočet na *GPU*, zejména pro případ realtimeové aplikace.

7 ZÁVĚR

V kapitole „Možnosti paralelizace“ (2) bylo ukázáno, že výpočet se dá za cenu mírného zesložitění rozdělit na více procesorových jednotek a následně mezivýsledky spojovat. Byly vybrány knihovny pro vývoj aplikace. Po bližším zkoumání několika možných knihoven byly použity knihovny *OpenCL*, *OpenAL*, *OpenGL*, *Standardní C++* a *Qt*.

Vytvořená aplikace *Sound Analyzer* zobrazuje spektrum ze vstupního zvukového zařízení nebo přehrávaného zvukového signálu. Je realtimeová a multiplatformní. Její funkčnost je možné zkontrolovat zejména proti funkci *FFT matlabu* tak, že se libovolný signál v *matlabu* uloží do souboru, nad tímto souborem se provede funkce „process mat file“ programem *Sound Analyzaror* a výsledný soubor se načte zpět do prostředí *matlab*.

Lze říci, že paralelizace se vyplatila. V kapitole 6 jsem ukázal, že výpočet 256krát delší zátěže trvá jen o cca 0,25% déle.

Praktickým ověřením funkcionality programu bylo měření výkonu aplikace využívající *GPU* ve srovnání se stejnými výpočty na hlavním procesoru. Bylo zjištěno, že to funguje, respektive, že *GPU* hlavnímu procesoru skutečně ulehčí. Je ale třeba *GPU* hodně zatížit, což prakticky znamená rozdělit úlohu na méně výpočetních jednotek s nějakou komplexnější funkcionalitou nebo prováděním nějakého cyklu. Jinak převáží vyšší režie, kterou outsourcing z *CPU* na *GPU* přináší. Větší počet výpočetních jednotek lze s výhodou využít například pro počítání většího objemu dat, tedy více kmitočtu ve více kanálech.

LITERATURA

[1] Munshi A.; The OpenCL specification: *Kronos Group online documentation*, 2012, veze 1.2, revize 19, dostupné na `kronos.org`.

[2] Kolektiv autorů; The OpenAL specification: *OpenAL Specification*, 2005, veze 1.1, LOKI software , dostupné na `https://www.openal.org/documentation/openal-1.1-specification.pdf`.

[3] OpenCV.org: *OpenCV 2.4.11 Documentaion*, 2014, dostupné na `opencv.org`.

[4] PROCHÁZKA, D., KOUBEK, T., ANDRÝSKOVÁ, J.:*Programování grafických aplikací s využitím OpenGL a OpenCV*, Mendelova Iniverzita Brno, dostupné z `https://is.mendelu.cz/eknihovna/opory/index.pl?cast=28313`

[5] OpenGL.org: *OpenGL Documentaion*, 2017, dostupné na `opengl.org`.

[6] Qt.io: *Qt Documentaion*, 2017, dostupné na `qt.io`.

[7] SMÉKAL, Z.: Systémy a signály. *Sdělovací technika* , 2013, Praha, ISBN 978-80-86645-23-0.

[8] SMÉKAL, Z., SYSEL, P.: Signálové procesory. *Sdělovací technika* , 2006, Praha

[9] SYSEL, P.; RAJMIC, P.: Goertzel Algorithm Generalized to Non-integer Multiples of Fundamental Frequency. *EURASIP Journal on Advances in Signal Processing*, 2012. 2012(56). p. 1 - 20. ISSN 1687-6172.

[10] VICH, R., SMÉKAL, Z.: Číslicové filtry. *Academia*, 200, Praha

SEZNAM SYMBOLŮ, VELIČIN A ZKRATEK

A, D	Matice, v tomto případě jsou to konstanty v Goertzelově algoritmu
$a_{3,12}$	Prvek matice A maticového Goertzelova vztahu pro více kmitočtů– 3. kmitočet, 1. řádek, 2. sloupec matice maticového Goertzelova vztahu
CPU	Central Processing Unit – centrální procesorová jednotka
DFŘ	Disktrétní Fourierova Řada
DFT	Disktrétní Fourierova Transformace
DSP	číslicové zpracování signálů – Digital Signal Processing
FFT	Fast Fourier Transform – rychlá Fourierova transformace
f_{vz}	vzorkovací kmitočet
GPU	Graphics Processing Unit – grafická procesorová jednotka
Jádro	Program vykonávaný na jedné pracovní jednotce v knihovně OpenCL
Load balancing	Rozložení zátěže
NDRange	Rozměry N-rozměrné krychle v knihovně OpenCL
Overlap	Přesah - segment délky n načtený ze vstupního zařízení je rozšířen o tuto délku na velikost $n + overlap$ a nad segmentem této délky je prováděn výpočet Goertzelovým algoritmem
Pipeline	Zřetězené zpracování
Pracovní jednotka	Fyzické jádro CPU nebo GPU jak je značeno v knihovně OpenCL
SIMD	Single Instruction Multiple Data– jedna instrukce, více dat
SPMD	Single Process Multiple Data – jeden proces, více dat

A UŽIVATELSKÝ MANUÁL PROGRAMU SOUND ANALYZER

A.1 Slovníček pojmů užitých v této kapitole

A.1.1 Frequency indexes

Indexy kmitočtů které budou počítány. Jsou to čísla n ze vzorce 1.2.

A.1.2 Sampling frequency

Vzorkovací kmitočet.

A.1.3 Scale

Měřítko specifikuje zobrazený rozsah grafů a je znázorněno na pravítkách.

A.1.4 Segment length

Segmentem je myšlen vektor určité délky, reprezentující část signálu. Komponenta *Recorder* čte ze vstupu (nebo přehrává) právě po úsecích této délky.

A.1.5 Segment overlap

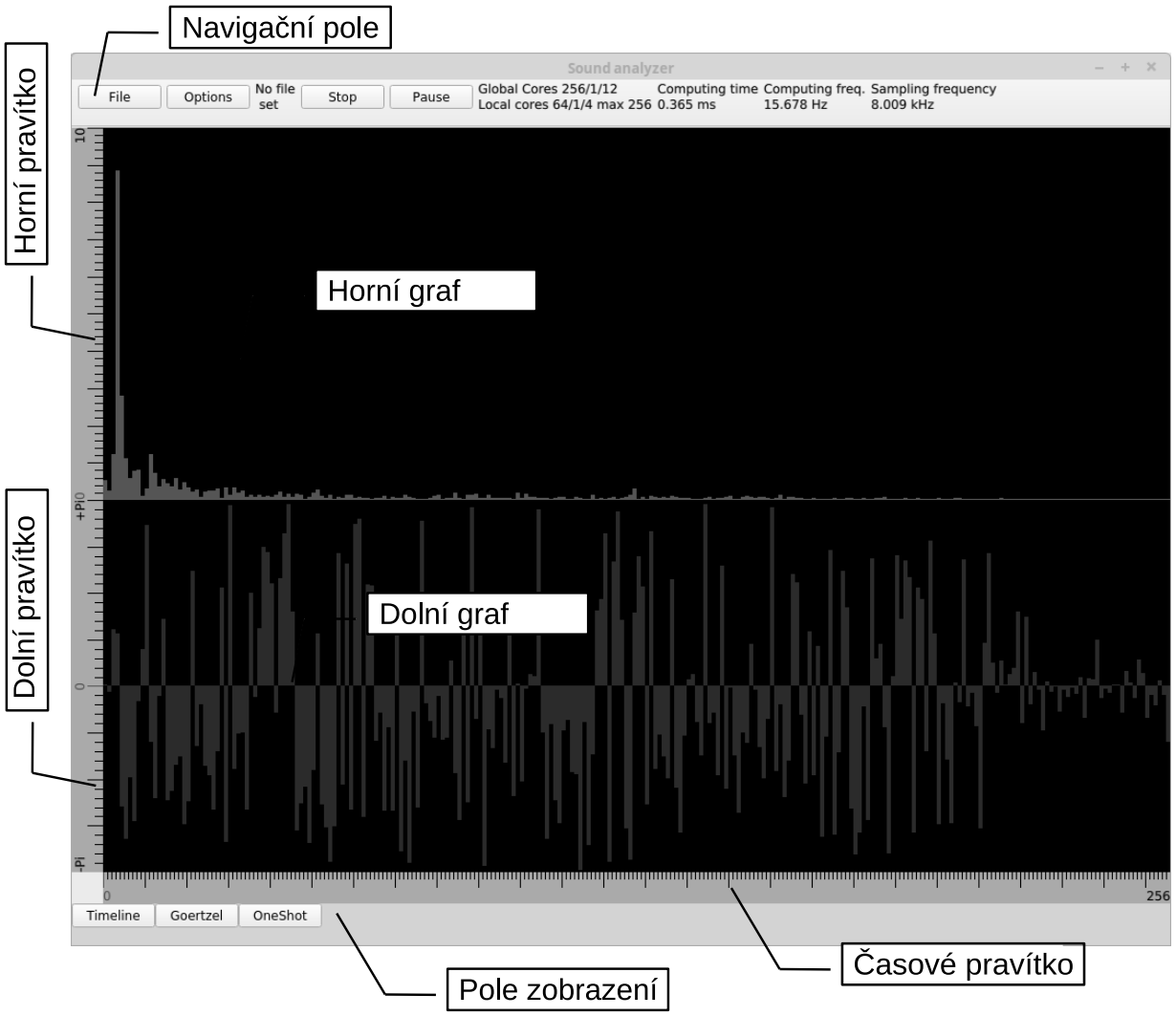
Komponenta *Goertzel* umí pracovat s větší délkou segmentu než s tou, kterou načte ze vstupu. *Segment overlap* je délka z předchozích vzorků, o kterou se rozšíří segment (vstupní vektor).

A.2 Okno aplikace

Na obrázku A.1 vidíme základní obrazovku programu *Sound Analyzer*. Pravítka slouží k odečítání časů a velikostí amplitud jednoho nebo dvou grafů. Spodní lišta slouží k přepínání pohledů na signál. Konečně navigační pole slouží ke konfiguraci a zobrazuje důležité statistické informace o signálu.

A.3 Horní graf

Horní graf má dva režimy zobrazení. V zobrazení typu *Goertzel* a *OneShot* reprezentuje amplitudu. V režimu zobrazení *Timeline* zobrazuje průběh signálu levého



Obr. A.1: Pohled na aplikaci Sound Analyzer

kanálu. Ten se neposouvá, jinak řečeno, vždy když se načte segment, je zobrazen místo segmentu předcházejícího. Pokud je rychlost změn větší než obnovovací rychlost GUI knihovny *Qt* zobrazený segment je zahozen a zobrazuje se ten poslední.

Zobrazení *Timeline* může být dvougrafové(stereo) nebo grafem přes celou velikost okna(Mono). To je možno nastavit přepínačem *Channels* a kartě *Recorder* z dialogu *Options*.

Měřítko v horizontálním směru je popsané v sekci časové pravítko A.6.

Měřítko ve vertikálním směru je popsané v sekci Dolní a horní pravítko A.5.

A.4 Dolní graf

I dolní graf má dva režimy zobrazení. V zobrazení typu *Goertzel* a *OneShot* reprezentuje fázi. V režimu zobrazení *Timeline* zobrazuje průběh signálu pravého kanálu. Ten se cyklicky překresluje, jak je blíže popsáno o horního grafu A.3.

Měřítka, stejně jako u grafu horního jsou popsány v sekcích o pravítkách A.6, A.5.

A.5 Horní a dolní pravítko

V zobrazení typu *Goertzel* a *OneShot* reprezentuje amplitudu v rozsahu od 0 do y_{max}. V režimu zobrazení *Timeline* zobrazuje průběh signálu v rozsahu $-y_{max}$,y_{max}. Rozsah y_{max} lze měnit jednak klávesovými zkratkami, jednak rozbalovacím seznamem *Scale* na kartě *General* z dialogu *Options*.

A.6 Časové pravítko

V režimu *Timeline* je na vodorovné ose vynesen počet vzorků které jsou najednou načteny ze vstupu. Je to současně nastavená hodnota *Segment length* na kartě *Recorder* v dialogu *Options*. V režimu *Goertzel* je na této ose počet počítaných harmonických, uvedených v seznamu *Frequency indexes* na kartě *Goertzel* v dialogu *Options*. V režimu *OneShot* se počítají všechny možné kmitočty signálu popsaném v souboru z *Matlabu*. Na vodorovné ose je tedy počet kmitočtů, číselně roven délce zkoumaného signálu.

A.7 Pole zobrazení

V *poli zobrazení* jsou tři tlačítka reprezentující tři režimy. Jsou to jediné akce, které můžete provádět za běhu.

Kromě změny zobrazení, tyto tlačítka přestavují toky dat mezi komponentami, jak je patrno z obrázku 4.3. Proto se při přepnutí z jednoho režimu do druhého a zpět zobrazí implicitní sinus. Při přepnutí fronty se totiž segment se zobrazovanými daty musí vrátit.

A.7.1 Timeline

Režim *Timeline* zobrazuje syrová data ze vstupu tak jak jdou za sebou. Jeden graf zobrazuje jeden segment. Zobrazení se s přibývajícím čase neposouvá, ale jen se překreslí aktuální segment.

Pokud je frekvence příchozích segmentů vyšší než je zobrazovací frekvence knihovny *Qt*, tak se segmenty jednoduše zahodí (a vidět je ten poslední).

A.7.2 Goertzel

Režim *Goertzel* zobrazuje spektrum signálu, respektive jen jeho požadované harmonické. Na horizontální ose najdeme počet počítaných harmonických.

A.7.3 OneShot

Režim *OneShot* je kompletní spektrum signálu dodaného ze souboru *matlabu*. Pokud je v tomto souboru více vektorů se signály, zobrazen se jen ten poslední.

A.8 Navigační pole

Zde uvedu informační a ovládací prvky navigačního pole zleva doprava.

A.8.1 Tlačítko File

Open for reading

Klasickým *File dialogem* si můžeme vybrat *wav* soubor, který bude přehráván po stisknutí tlačítka *Start*. Protože přehráváním se současně udává takt, bude program i nadále fungovat v reálném čase. Je možné mít otevřený soubor buď jen pro čtení nebo jen pro zápis, nikoli oba současně.

Open for writing

Klasickým *File dialogem* si můžeme vybrat *wav* soubor, do kterého budou ukládány načtené vzorky ze vstupu. V případě že výstupní zařízení není schopno ukládat požadované množství dat, bude výstupní soubor poškozený.

Close file

Tímto zrušíte obě předchozí volby. Nyní se bude počítat spektrum jen ze vstupu.

Process mat file

Tato funkce je v programu *Sound Analyzer* nejen pro ladící účely. Provede spočtení všech harmonických signálu dodaného v souboru typu *mat* vytvořeném v prostředí *matlab*.

Soubor typu *mat* může obsahovat více vektorů. V takovém případě se provede výpočet spektra všech vektorů, ale vidět v pohledu *OneShot* je pouze poslední.

Výstupní soubor je pojmenovaný jako $< vstupnísoubor > \_ap$. V tomto souboru jsou dvě, respektive dvakrát více vektorů než v souboru vstupním (amplituda a fáze). Jméno vektoru s amplitudami je $< vstupnívektor > \_am$ a vektory s fázemi mají jména $< vstupnívektor > \_ph$.

Quit

Konec programu.

A.8.2 Tlačítko Options

Spustí dialog nastavení. Více v sekci *Dialog nastavení* A.10

A.8.3 Jméno souboru pro zápis/čtení

Zde je jméno souboru pro zápis/čtení a informace o tom, v jakém režimu souborové operace jsou (zápis/čtení/nic).

A.8.4 Tlačítko Start

Tlačítko *Start* slouží ke startu jak skenování vstupu, tak počítání spektra. Aby se počítalo spektrum, je nutné zobrazit režim *Goertzel*, jinak načtená data nejdou přes Goertzelův algoritmus, ale pouze se zobrazují.

Teprve začátkem zaznamenávání nebo přehrávání se aplikují změny v dialogu nastavení.

Kliknutím se text na tlačítku změní na *Stop*.

A.8.5 Tlačítko Pause

Tlačítko *Pause-Continue* funguje i když zrovna není zapnuto skenování. To znamená že když je stav *zapauzovaný* a následně kliknete na tlačítko *Start*, skenování se zinicializuje, ale vlastní skenování nepoběží.

A.8.6 Počet jader použitých při výpočtech

V tomto okénku je sedm údajů: Globální $N/C/M$, lokální $N/C/M$ a maximální počet jader současně spustitelných. N je počet počítaných kmitočtů, C je počet kanálů a konečně M je počet mezisoučtů. Globální počet jader je tedy celkový požadavek na výpočet a jeho velikost je $N \times C \times M$. Tři souřadnice udávají počet jader se v terminologii *OpenCL* nazývají $NDrange$. Naproti tomu lokální $NDRange$ je skutečná počet jader použitých současně pro výpočet. Určitě tedy platí, že $globálníNDRange \geq lokálníNDRange$. Je tu ještě jedno omezení. Globální souřadnice $NDRange$ musí být celočíselný násobek lokální souřadnice $NDRange$.

A.8.7 Průměrný čas výpočtu spektra

Vlákno programu *Sound Analyzer*, které v cyklu počítá některé harmonické složky, obsahuje jednoduchou počítací smyčku. Jedna její iterace trvá právě tento zobrazený čas.

A.8.8 Průměrná frekvence výpočtu spektra

Jedná se o frekvenci, se kterou se provádí výpočty. Měla by být někde okolo $f_{vz}/ < délkasegmentu >$.

A.8.9 Průměrná vzorkovací frekvence

Je spočtená hodnota braná z komponenty *recorder*. Měla by být podobná nastavenému *vzorkovacímu kmitočtu*.

A.9 Klávesové zkratky

+	Zvýšení rozsahu vertiálního měřítka
-	Snížení rozsahu vertikálního měřítka
S	Start, Stop
space	Pause, Continue

A.10 Dialog nastavení

A.10.1 Karta General

Automatic hide

Automatické schovávání je jen hříčka pro „hezčí“ zobrazení, bez okraje okna, *navigačního pole* a *pole zobrazení*.

Scale

Vertikální měřítko grafu se aplikuje okamžitě. Je to jedna z mála výjimek v dialogu *Nastavení*. Pro rychlejší manipulaci s měřítkem jsou v programu klávesové zkratky $+, -$.

A.10.2 Karta Recorder

Recorder je komponenta zodpovědná za načítání vzorků ze zvukového zařízení, případně přehrávání zvukového souboru. V případě že je to povoleno, posílá ještě data k uložení do souboru.

Bits per sample

Knihovna *OpenAL* nabízí jen dvě možnosti přesnosti každého vzorku, 8bitovou a 16bitovou. Oba jsou to celočíselné typy.

Channels

Program *Sound Analyzer* podporuje jak mono, tak stereo záznam, tak i přehrávání i výpočet spektrálních čar. Modrou barvou je značen levý kanál, červenou pak pravý.

Device

Zde jde nastavit jen zařízení *OpenAL Soft*. Knihovna *OpenAL* nikdy žádné jiné zařízení nenabídla.

Capture device

Zařízení pro záznam zvuku jsou obvykle tři. Výchozí je vždy „vnitřní zvukový systém“, což je vždy mikrofon. Další zařízení, které tam vždy najdeme, je „monitor vnitřního zvukového systému“, což je standardní zvukový výstup. Když ale zkusíme tento vstup nahrávat, zjistíme, že je nějak poškozen. Protože výchozí vstupní zařízení zvuk nedeformuje, jsem názoru, že jde o ochranu proti kopírování. Tento vstup

totiž není zatížen ani minimálním šumem. Jako poslední je obvykle vstup, „monitor HDMI výstupu“. Ten je nějak zkreslen podobně jako vstup předchozí.

Sampling frequency

Tato vzorkovací frekvence je použita pro inicializaci *recorderu*. Pokud se podíváme na kmitočet spočtený A.8.9, zjistíme že pro *zařízení-monitory*, nejsou tyto kmitočty totožné. V tom je to zkreslení.

Segment length

Délka segmentu (vektor vstupního signálu) v počtu vzorků. Více o něm je v sekci *segment length* ve slovníčku pojmů A.1.4.

A.10.3 Karta Goertzel

Tato karta se týká výhradně výpočtu spektrálních čar a knihovny *OpenCL*.

Segment overlap

Počet vzorků, o které se rozšíří načtený segment. Více je ve slovníčku *segment overlap* A.1.5.

Frequency indexes

V tomto textovém poli je seznam indexů spektrálních čar počítaného spektra. Nic nebrání tomu počítat jen stejnosměrnou složku (index 0) nebo několik málo kmitočtů, které nejdou po sobě. Jednotlivé celočíselné indexy jsou odděleny „;“ a pro lepší přehlednost mohou obsahovat libovolný počet přechodů na nové řádky.

Tlačítko Set All

Snazšího vyplnění předcházejícího pole dosáhneme tlačítkem *Set All*. To nahradí jakýkoli obsah pole *frequency indexes* indexy 0 až (*segment_length*+*segment_overlap*).

Přepínač Device

Tímto říkáme knihovne *OpenCL*, zda požadujeme provádět výpočty hlavním procesorem nebo grafickým čipem.

A.11 Příklad použití

Předpokládejme, že bude chtít počítat stejnosměrnou složky signálu a současně amplitudu na kmitočtu 1kHz. Vzorkovací kmitočet použiji 8kHz.

Délku segmentu si zvolím. Například délka 200 vzorků by se načítala 200/8000s = 25ms. To je tak maximální doba, při které může třeba řečový signál považovat za stacionární.

Nejnižší, základní kmitočet, který jsem takto schopen měřit je 8000/200 = 40Hz. Nás ale zajímají indexy 0 - stejnosměrná složka a 1000Hz = 25 × 40Hz. Druhý index je tedy 25.

Ve shodě s předchozím nastavíme v dialogu *nastavení* toto:

Bits per sample	16bit
Channels	Mono
Device	OpenAl Soft
Capture device	Vnitřní zvukový systém analogové stereo
Sampling frequency	8000
Segment length	200
Segment overlap	0
Frequency indexes	0;25;
Device	GPU

B INSTALACE PROJEKTU - SPUŠTĚNÍ APLIKACE

K aplikaci není přiložen žádný instalační program. Není potřeba. Spouští se jen spustitelným souborem *SoundAnalyzer* v adresáři téhož jména. Zabalený archiv s programem naleznete na přiloženém CD disku. Stačí jej jen rozbalit.

Nicméně pro správnou funkci je třeba mít správně nainstalované knihovny *OpenCL* a *OpenGL*. Knihovny *OpenGL* a *OpenCL* dodává výrobce grafické karty, tudíž je problematické napsat nějaký návod na její nainstalování, nebo dokonce nějaký instalační skript. Obecně lze říci, že v linuxu je lepší používat proprietární ovladače od výrobců grafických čipů něž ovladače volně šiřitelné. Jejich instalace ale může přinést potíže, mně třeba pokus o instalaci ovladače od NVidie způsobil nefunkčnost operačního systému a musel jsem tento znovu nainstalovat.

B.1 GPU AMD

Na `http://support.amd.com/en-us/download/linux` lze stáhnout ovladač *fglrx*. S jeho instalací jsem neměl žádné problémy. Možná bude potřeba nainstalovat i *OpenCL* z adresy developer.amd.com a je rovněž na přiloženém CD nosiči. Instalace je bez problémů.

B.2 GPU NVidia

Ovladač Nvidia je nyní ve standardním repozitáři Ubuntu a Mint a to v poslední verzi *375*. Když instalujete nové Ubuntu nebo Mint, je třeba před instalací spustit správce ovladačů a vybrat „používat NVidia 340“. Když Linux nainstalujete a připojíte k síti a opět spustíte výše jmenovaný program, nabídne vám již aktuální verzi *375*. OpenCl je součátí CUDA a je také ve standardním repozítáři. Nainstalujete zadáním:

```
sudo apt-get install libcuda1-375
```

C INSTALACE PROJEKTU - POKRAČOVÁNÍ VE VÝVOJI

C.1 Instalace GNU C++

Přestože knihovny *OpenCV* a *OpenCL* jsou multiplatformní, rozhodl jsem se pro operační systém Linux, distribuce Mint 17. Vyhnul jsem se konstrukcím platformně závislým, takže je možná přenositelnost na jiné operační systémy. Jako překladač jsem zvolil *GCC*, a to ze stejného důvodu jako předchozí, je platformně nezávislý a je v každé distribuci Linuxu jako jeden ze základních balíčků. Mně stačilo k instalaci napsat:

```
sudo apt-get install build-essential
```

a potom...

```
sudo apt-get install g++
```

Pro Windows existuje disrtibuce GNU cygwin, kterou je možno stáhnout a nainstalovat z webu `sourceware.org/cygwin`.

Důležité je nezapomenout zaškrtnout *devel* a *debug* v seznamu balíčků k instalaci.

C.2 Verzovací systém Git

Verzovací systém Git používám nejen v zaměstnání, ale i v domácích projektech a má své místo i v této práci. Může být důležité moci se v případě nutnosti vrátit k předchozím verzím projektu.

Git je standardní balíček snad všech distribucí. Instaluje je jako obvykle napsáním

```
sudo apt-get install git
```

S klíči se zachází standardním způsobem, měly by být uloženy v adresáři `~/.ssh`.

Vytvořil jsem speciální repozitář na svém testovacím serveru. Přihlašovat se jde výhradně pomocí klíče, který najdete na CD–příloze této práce. Projekt stačí už jen naklonovat příkazem,

```
git clone virtualbox@46.167.245.175:dp
```

přesněji, pomocí klíče na CD

```
ssh-agent sh -c 'ssh-add <cesta>/virtualbox_rsa;
git clone virtualbox@46.167.245.175:dp'
```

C.3 Qt Creator

Qt Creator je nedílnou součástí vývojového kitu knihovny Qt. Nyní je již verze 5.8, já jsem ale použil verzi 5.5.1, která je podporována ve standardních repozitářích linuxu, konkrétně jsem vyzkoušel Mint 18.1 a Ubuntu 16.04.

Instalační baliček je součástí přiloženého CD a je možné ho rovněž stáhnout ze stránek `qt.io`.

Celá aplikace je napsaná v prostředí *Qt Creator*. *Qt creator* má problémy s vizuálním návrhářem, a to takové, že je zcela nepoužitelný a layout dialogů je nutné psát textově v jazyce /emphqml. Stále je však možno použít prostředí *QtCreatoru* pro psaní textu a překlad programu.

C.4 OpenGL

Instalaci proprietárního ovladače a knihovny *OpenGL* jsem popsal v kapitole *instalace programu* B.

Tím ale máme v systému knihovny, nikoli už ale hlavičkové soubory pro jejich použití. Já jsem napsal pro začátek vývoje v OpenGL toto:

```
sudo apt-get install freeglut3
sudo apt-get install freeglut3-dev
sudo apt-get install binutils-gold
sudo apt-get install mesa-common-dev
sudo apt-get install libglew1.5-dev libglm-dev
```

C.5 OpenAL

Instalace vývojových balíčků pro knihovnu OpenAl je velmi jednoduchá, prakticky nikdy se nestalo že by něco nefungovalo. V Ubuntu jsou to standardní balíčky:

```
sudo apt-get install libopenal-dev
sudo apt-get install libalut-dev
```

C.6 OpenCL

Tak jako *OpenGL* i *OpenCL* je poskytována výrobcem grafického čipu. Je tedy nutné mít dobře nainstalovanou grafickou kartu s proprietárními ovladači.

Ve standardních repozitářích Ubuntu je následující balíček...

```
sudo apt-get install ocl-icd-libopencl1,
```

ten nainstaluje nějakou *OpenC*L knihovnu. Program potom jde spustit, ale inicializace vrátí chybu, není platforma, takže není možno provádět výpočet ani softwarově.

SDK pro *OpenCL* na grafických čipech *AMD* je možno stáhnout z adresy developer.amd.com a je rovněž na přiloženém CD nosiči. Instalace je bez problémů. Jediné co potom musíte udělat, je nastavení cesty ke hlavičkovému souboru a knihovně v případě, že chcete použít její vývojovou verzi. U mně to bylo například: */opt/AMDAPPSDK-2.9-1/include*.

SDK pro *OpenCL* na grafických čipech *NVidia* je v Ubuntu ve standardních repozitářích a je spojena s implementací knihovny *CUDA*. Pro instalaci stačí napsat:

```
sudo apt-get install libcuda1-375
sudo apt-get install nvidia-cuda-dev
```

kde 375 je jméno verze ovladače grafického čipu NVidia.

D SLOVNÍČEK POJMŮ KNIHOVNY OPENCL

Tato kapitola je výtahem z dokumentace k openCL ([1]). Úplnou dokumentaci lze najít `kronos.org`.

D.1 Application – Aplikace

Kombinace programů běžících, jak na *hostitelském zařízení (host device)* (viz termín D.24), tak na openCL *zařízení (device)* (viz termín D.16).

D.2 Blocking a Non-Blocking Enqueue API calls – Blokující a neblokující příkazy

Neblokující příkaz (command) (viz termín D.6) ve frontě volání vloží *příkaz* (viz termín D.6) do *příkazové fronty* (viz termín D.7) a okamžitě vrátí řízení *hostiteli* (viz termín D.24). *Blokující* příkaz ve frontě nevrátí řízení hostiteli, dokud není *příkaz* (viz termín D.6) proveden.

D.3 Barrier – Bariéra

Jsou dva druhy *bariér (barriers)* (viz termín D.3) – bariéra *fronty zpráv* (viz termín D.7) a bariéra *pracovní skupiny (work-group)* (viz termín D.57).

- *OpenCL API* poskytuje funkci zařazení *bariéra fronty zpráv* (viz termín D.8). Tento příkaz zajistí, že předchozí příkazy jsou provedeny před tím, než začne příkaz následující.
- *OpenCL C* programovací jazyk poskytuje vestavěnou funkci *bariéra pracovní skupiny* (viz termín D.58). Tato bariéra může býti volána *jádrem (kernel)* (viz termín D.30) pro zajištění synchronizace mezi *pracovními členy (work-items)* (viz termín D.59) v *pracovních skupinách* (viz termín D.57) provádějících *jádra* (viz termín D.30). Všechny *pracovní členy* v *pracovní skupině* musí provést tuto synchronizaci, než bude možno pokračovat v provádění následujících příkazů.

D.4 Buffer object – Buffer

Buffer je paměťový objekt, ve kterém je uložena souvislá řada bytů. Tento buffer je přístupný použitím ukazatele z *jádra* prováděném na *zařízení*. S těmito buffery může

býti manipulováno použitím OpenCL API funkcí. Buffer zapouzdřuje následující informace:

- Velikost v bytech.
- Parametry popisující uložení v paměti a ve které oblasti je alokován.
- Buffer data.

D.5 Built-in kernel – Vestavěné jádro

Vestavěné jádro je je *jádro* (viz termín D.30) prováděné na openCL *zařízení* (viz termín D.16) nebo *speciálním zařízení (custom device)* (viz termín D.14) pevně daným hardwarem nebo firmwarem. Aplikace může používat *vestavěná jádra* nabízená *zařízeními* nebo *speciálními zařízeními*. *Objekty programů (program objects)* (viz termín D.44) mohou obsahovat jen *jádra* napsané v jazyce openCL C nebo *vestavěná jádra*, ale nikdy ne obojí. Více viz *jádra* (viz termín D.30) a *programy* (viz termín D.43).

D.6 Command – Příkaz

Operace v openCL jsou posílány k provedení do *fronty příkazů (command queue)* (viz termín D.7). Například openCL *příkazy* (viz termín D.6) pověřují *jádra (kernels)* (viz termín D.30) k provedení na *výpočetním zařízení* (viz termín D.16), manipulaci s paměťovými objekty atd.

D.7 Command Queue – Fronta příkazů

Fronta příkazů (command queue) (viz termín D.7) je objekt ve kterém jsou uloženy příkazy, které budou vykonány na určitém *zařízení* (viz termín D.16). *Fronta příkazů* (viz termín D.7) je vytvořena na určitém *zařízení* s určitým *kontextem (context)* (viz termín D.13). *Příkazy* (viz termín D.6) jsou zařazeny do *fronty příkazů* (viz termín D.7) v pořadí, ale vykonat se mohou v tomto pořadí nebo v pořadí jiném. Více viz *provádění v pořadí (in-order execution)* (viz termín D.29) a *provádění mimo pořadí (out-order execution)* (viz termín D.38).

D.8 Command Queue Barrier – Bariéra fronty příkazů

Více na *bariéra (barrier)* (viz termín D.3).

D.9 Compute device memory – Paměť výpočetního zařízení

Jedno *zařízení* (viz termín D.16) může mít připojeno jednu nebo více pamětí.

D.10 Compute unit – Výpočetního jednotka

OpenCL *zařízení* (viz termín D.16) může mít jednu nebo více *výpočetních jednotek (compute units)* (viz termín D.10). *Pracovní skupina (work-group)* (viz termín D.57) je provedena na jedné výpočetní jednotce. Výpočetní jednotka je tvořena jedním nebo více *procesorovým prvkem (processing element)* (viz termín D.42) a *lokální pamětí (local memory)* (viz termín D.33). Výpočetní jednotka také může obsahovat filtr textur, který může být přístupný z *procesorových prvků* (viz termín D.42).

D.11 Concurrency – Souběžnost

Vlastnost systému, ve kterém je nějaká skupina úloh současně aktivní a provádí nějakou akci. Pro využití *souběžného provádění* (viz termín D.11) programů, musí programátor identifikovat možné problémy *souběžného* zpracování, zahrnout je do svých zdrojových kódů a využít synchronizačních možností zařízení.

D.12 Constant memory – Paměť konstant

Oblast v paměti přístupná *jádru* (viz termín D.30), která je za běhu *jádra* konstantní. Tuto paměť alokuje i inicializuje *hostitel* (viz termín D.24).

D.13 Context – Kontext

Prostředí, ve kterém se provádí *jádra* (viz termín D.30) a doména, ke které se váží synchronizační a *paměťové objekty* (viz termín D.35). Kontext zahrnuje množinu *zařízení*, příslušnou paměť k těmto *zařízením*, proměnné vázající se k těmto pamětem a jednu nebo více *front příkazů* (viz termín D.7).

D.14 Custom device – Speciální zařízení

Speciální zařízení má podporu openCL runtime, ale není možné pro něj psát programy v openCL C. Tyto zařízení jsou obvykle vysoce efektivní pro určité typy úloh.

Mohou mít vlastní překladač. Pokud ho nemají, je možné spouštět pouze *vestavěná jádra* (viz termín D.5).

D.15 Data parallel programming model – Datový paralelní programovací model

Tradičně je tím myšlen model, kde je jeden program spuštěn souběžně na řadě stejných objektů.

D.16 Device – Zařízení

Zařízení je množina *výpočetních jednotek* (viz termín D.10). Každá jednotka má *fronty příkazů* (viz termín D.7). Příkazy mohou například spouštět *jádra* (viz termín D.30) nebo zapisovat a číst *paměťové objekty* (viz termín D.35). Příkladem takovéhoto *zařízení* je *GPU*, vícejádrové *CPU* nebo například *DSP*.

D.17 Event object – Objekt události

Objekt události zapouzdřuje status operace, jako například nějakého *příkazu* (viz termín D.6). Může být využit k synchronizaci operací v rámci *kontextu* (viz termín D.13).

D.18 Event wait list – Seznam čekajících událostí

Je seznam *objektů událostí* (viz termín D.17) který může být využit k řízení začátku provádění dílčího *příkazu* (viz termín D.6).

D.19 Framework – Framework

Je několik softwarových balíků umožňujících vývoj software. Obvykle zahrnuje knihovny, API, *runtime* prostředí, překladače a podobně.

D.20 Global ID – Globální ID

Global ID jednoznačně specifikuje *pracovní položka* (viz termín D.59) a je založen na počtu globálních *pracovních položek* a je dán před spuštěním *jader* (viz termín

D.30). Global ID je N-rozměrný vektor začínající (0,0,...0). Více na *Lokální ID* (viz termín D.32).

D.21 Global memory – Globální paměť

Tato paměť je přístupná *pracovním položkám* (viz termín D.59) prováděných v rámci *kontextu* (viz termín D.13). Z *hostitele* může být přístupný pomocí *příkazů* jako číst, zapisovat a mapovat.

D.22 GL share group – Sdílená GL skupina

Sdílená GL skupina je objekt pro správu sdílených prostředků s knihovnami openGL a openGL ES. Například tam patří textury, buffery, framebuffery a je asociována s jedním nebo více openGL kontexty. Sdílený GL objekt je obvykle zástupný objekt na není přímo přístupný.

D.23 Handle – Rukojeť

Zástupný typ ukazující na objekty ve správě openCL. Každá operace na nějakém objektu je určena *rukojetí* na tento objekt.

D.24 Host – Hostitel

Hostitel komunikuje s openCL API prostřednicvím *kontextu* (viz termín D.13).

D.25 Host pointer – Ukazatel hostitele

Ukazatel na paměť, která je ve virtuálním adresovém prostoru *hostitele*.

D.26 Illegal – Nedovolený

Chování systému, které není výslovně dovoleno, bude nahlášeno jako chyba systému openCL.

D.27 Image object – Obrázek

Obrázek (viz termín D.27) má dvou nebo tří dimenzionální pole, rozměry atd. Data lze modifikovat pomocí funkcí čtení a zápisu. Operace čtení využívá *vzorkovač (sampler)* (viz termín D.50).

D.28 Implementation defined – Implementačně závislé

Chování, kde je výslovně povolena nějaká odlišnost od openCL rozhraní. V těchto případech je povinná dobrá dokumentace výrobce.

D.29 In-order execution – Provádění v pořadí

Model provádění, kde *příkazy* ve *frontě příkazů* (viz termín D.7) jsou prováděny jeden za druhým, *příkaz* (viz termín D.6) je vždy dokončen před započetím *příkazu* následujícího.

D.30 Kernel – Jádro

Kernel (jádro) je funkce definovaná v programu a označená identifikátorem `kernel` nebo `__kernel`.

D.31 Kernel object – Objekt jádra

Objekt jádra (kernel object) zapouzdřuje *jádro* (viz termín D.30) (funkci definovanou s kvalifikátorem `__kernel`) a vstupní argument.

D.32 Local ID – Lokální ID

Lokální ID jednoznačně specifikuje *pracovní položka* (viz termín D.59) v rámci *pracovní skupiny* (viz termín D.57). Lokální ID je N-rozměrný vektor začínající (0,0,...0). Více na *Globální ID* (viz termín D.20).

D.33 Local memory – Lokální paměť

Tato paměť je přístupná *pracovním položkám* (viz termín D.59) prováděných v rámci *pracovní skupiny* (viz termín D.57).

D.34 Marker – Značka

Je *příkaz* zařazený ve *frontě příkazů* (viz termín D.7), který označí všechny předchozí *příkazy* (viz termín D.6) ve *frontě příkazů* a jeho výsledkem je *událost* (viz termín D.17), kterou může poslouchat aplikace, a tak počkat na všechny *příkazy* zařazené před *značkou* ve *frontě zpráv*.

D.35 Memory objects – Paměťové objekty

Paměťový objekt je *rukojeť* – ukazatel na nějakou oblast v globální paměti. Používá se *počítadlo odkazů (reference counting)* (viz termín D.45). Více na *buffer* (viz termín D.4) nebo *obrázek* (viz termín D.27).

D.36 Memory regions (pools) – Paměťové oblasti

V openCL jsou různé adresové prostory. Paměťové oblasti mohou překrývat ve fyzické paměti, ale openCL je bere jako logicky různé. Paměťové oblasti mohou být označeny jako `private`, `local`, `constant` a `global`.

D.37 Object – Objekt

Objekty jsou abstraktní reprezentace *zdrojů (resources)* (viz termín D.47) a může s nimi býti manipulováno pomoci openCL API. Například lze uvést *objekt programu* (viz termín D.44), *objekt jádra* (viz termín D.31) a *paměťový objekt* (viz termín D.35).

D.38 Out-of-order execution – Provádění mimo pořadí

Model provádění, kde *příkazy* (viz termín D.6) vložené ve *frontě příkazů* (viz termín D.7) jsou prováděny v libovolném pořadí. Je ovšem respektován *seznam čekajících*

událostí (viz termín D.18) a *bariéra fronty zpráv* (viz termín D.8). Viz *provádění v pořadí* (viz termín D.29).

D.39 Parent device – Rodičovské zařízení

OpenCL *zařízení* (viz termín D.16) může být rozděleno na další *subzařízení* (viz termín D.53). Protože i *subzařízení* může být dále děleno na *subzařízení*, nemusí být *rodičovské zařízení* vždy *kořenové zařízení (root device)* (viz termín D.49).

D.40 Platform – Platforma

Je *hostitel* (viz termín D.24) a jedno nebo několik openCL *zařízení* (viz termín D.16) které mohou být ovládány openCL *frameworkem* (viz termín D.19). Tento *framework* umí mezi zařízeními sdílet zdroje a provádět *jádra* (viz termín D.30) na *zařízení* v rámci *platformy*.

D.41 Private memory – Soukromá paměť

Oblast v paměti vyhrazená výhradně pro *pracovní položku* (viz termín D.59). Proměnné zde definované nemohou býti viděny z jiné *pracovní položky*.

D.42 Processing element – Zpracující jednotka

Virtuální skalární procesor. *Pracovní položka* (viz termín D.59) může být prováděna na jedné nebo více *zpracujících jednotkách* (viz termín D.10).

D.43 Program – Program

OpenCL *program* (viz termín D.43) se skládá z množiny *jader* (viz termín D.30). *Programy* mohou také obsahovat pomocné funkce volané z `__kernel` funkcí a konstantní data.

D.44 Program object – Programový objekt

Programový objekt zapouzdřuje následující informace:

- Ukazatel na odpovídající *kontext* (viz termín D.13).
- Zdrojový text nebo binární kód programu.

- Poslední úspěšně přeložený proveditelný program, seznam *zařízení* (viz termín D.16), pro které byl přeložen, nastavení překladače a log.
- Několik připojených *objektů jader* (viz termín D.31).

D.45 Reference count – Počítadlo odkazů

Životní cyklus objektů v openCL je dán *počítadlem odkazů* (viz termín D.45) na tento objekt. Když vytvoříte openCL objekt, *počítadlo odkazů* se nastaví na 1. Příslušný *retain (přivlastni)* (viz termín D.48) jako `clRetainContext`, `clRetainCommandQueue` zvyšují hodnotu tohoto počítadla. Volání *release (uvolni)* (viz termín D.48) jako například `clReleaseContext` nebo `clReleaseCommandQueue` toto počítadlo snižují o 1. Když *počítadlo odkazů* klesne na nulu, objekt je dealokován.

D.46 Relaxed consistency – Rozvolněná soudržnost

Model soudržnosti paměti, ve kterém je viditelnost pro jiné *pracovní položky* (viz termín D.59) nebo *příkazy* (viz termín D.6) různá. Výjimkou jsou synchronizační objekty, jako třeba *bariéry* (viz termín D.3).

D.47 Resource – Zdroj

Je třída *objektů* (viz termín D.37) definovaná v openCL. Instance *zdroje* je *objekt*. Zdroji obyčejně rozumíme *kontexty* (viz termín D.13), *fronty příkazů* (viz termín D.7), *programové objekty* (viz termín D.44), *objekty jádra* (viz termín D.31) a *paměťové objekty* (viz termín D.35). Výpočetní *zdroje* jsou hardwarové součásti využívající čítač instrukcí. Jako příklad lze uvést *hostitele* (viz termín D.24), *zařízení* (viz termín D.16), *výpočetní jednotky (compute units)* (viz termín D.10) nebo *zpracující jednotky (processing elements)* (viz termín D.42).

D.48 Retain, release – Přivlastni, uvolni

Pojmenování akce inkrementace(*retain*) (viz termín D.48), nebo dekrementace(*release*) (viz termín D.48) *počítadla odkazů* (viz termín D.45). Zajišťuje, že objekt nebude smazán, dokud nejsou hotovy všechny procesy, které jej používají. Viz *počítadlo odkazů* (viz termín D.45).

D.49 Root device – Kořenové zařízení

Kořenové zařízení je openCL *zařízení* (viz termín D.16), které není rozdělená část jiného *zařízení*. Více viz *zařízení* a *rodičovské zařízení* (viz termín D.39).

D.50 Sampler – Vzorkovač

Objekt (viz termín D.37), který popisuje jak se má navzorkovat obrázek, když je čten *jádrem* (viz termín D.30). Funkce čtoucí obrázky mají *vzorkovač* jako svůj argument. *Vzorkovačem* se definuje adresní mód, což je chování v případě, že se souřadnice v obrázku nacházejí mimo jeho rozměry. Jako další jsou případy, kdy jsou souřadnice obrázku normalizované a nenormalizované hodnoty a filtrační mód.

D.51 SIMD:Single instruction multiple data – SIMD

Programovací model, kde *jádro* (viz termín D.30) je prováděno současně na více *zpracujících jednotkách* (viz termín D.10), každá z nich má svá vlastní data a společně sdílejí jeden čítač instrukcí. Všechny *zpracující jednotky* provádějí naprosto stejnou množinu instrukcí.

D.52 SPMD: Single program multiple data – SPMD

Programovací model, kde *jádro* (viz termín D.30) je prováděno současně na více *zpracujících jednotkách* (viz termín D.10). Každá z nich má svůj vlastní čítač instrukcí. *Zpracující jednotky* mohou mít množinu instrukcí různou.

D.53 Sub-device – Subzařízení

OpenCL *zařízení* (viz termín D.16) může být rozděleno do více *subzařízení* podle rozdělovacího plánu. Nová *subzařízení* alias specifické skupiny *výpočetních jednotek* (viz termín D.10) v rámci *rodičovského zařízení* (viz termín D.39) mohou býti použita stejně jako jejich *rodičovská zařízení*. Rozdělení *rodičovského zařízení* na *subzařízení* nezničí toto *rodičovské zařízení*, které může být průběžně dále používáno současně se *subzařízeními*. Viz také *zařízení* (viz termín D.16), *rodičovské zařízení* (viz termín D.39) a *kořenové zařízení* (viz termín D.49).

D.54 Task parallel programming model – Paralelně úlohový programovací model

Programovací model, kde jsou výpočty vyjádřeny podmínkami více *souběžných* (viz termín D.11) úloh. Úloha je zde *jádro* (viz termín D.30) v jedné *pracovní skupině* (viz termín D.57) o velikosti jedna. Souběžné úlohy mohou provádět různá *jádra*.

D.55 Thread-safe – Vláknově bezpečné

OpenCL je považováno za *vláknově bezpečné*, pokud interní stav, který je spravován knihovnou openCL, zůstává konzistentní i v případě, že *hostitel* (viz termín D.24) volá z více vláken. OpenCL API, které jsou *vláknově bezpečné*, dovolují aplikaci volat tyto funkce z více současně probíhajících vláken bez použití *mutexu*. Říkáme pak, že jsou i *reentratní (re-entrant-safe)*.

D.56 Undefined – Nedefinováno

Chování volání openCL API, kde *vestavěná funkce* (viz termín D.5) uvnitř *jádra* (viz termín D.30) není výslovně definována. Po implementaci openCL není požadováno definovat, co se stane, když je tato funkce použita.

D.57 Work group – Pracovní skupina

Skupina *pracovních položek* (viz termín D.59), které jsou vykonávány na jedné *výpočetní jednotce* (viz termín D.10). *Pracovní položky* ve skupině provádějí stejné *jádro* (viz termín D.30) a sdílejí *lokální paměť* (viz termín D.33) a *bariéry pracovní skupiny* (viz termín D.58).

D.58 Work group barrier – Bariéra pracovní skupiny

Viz *bariéra* (viz termín D.3).

D.59 Work-Item – Pracovní položka

Jedna z množiny paralelního provádění *jádra* (viz termín D.30) spuštěná na *zařízení* (viz termín D.16) *příkazem* (viz termín D.6). *Pracovní položka* je prováděna

na jedné nebo více *zpracujících jednotkách* (viz termín D.10) jako část provádění *pracovní skupiny* (viz termín D.57) prováděné na *výpočetní jednotce*. *Pracovní položka* je rozeznávána mezi jinými ve skupině svým *lokálním ID* (viz termín D.32) a *globálním ID* (viz termín D.20).