VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

AKCELERACE ZPRACOVÁNÍ 3D OBRAZOVÝCH DAT NA GPU

ACCELERATION OF 3D IMAGE PROCESSING USING GPU

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — **JAKUB JOCHLÍK**
AUTHOR

VEDOUCÍ PRÁCE — **MICHAL ŠPANĚL, Ing., Ph.D.**
SUPERVISOR

BRNO 2017

Zadání bakalářské práce/20216/2016/xjochl00

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav počítačové grafiky a multimédií — Akademický rok 2016/2017

Zadání bakalářské práce

Řešitel: **Jochlík Jakub**
Obor: Informační technologie
Téma: **Akcelerace zpracování 3D obrazových dat na GPU**
Acceleration of 3D Image Processing Using GPU
Kategorie: Zpracování obrazu

Pokyny:

1. Prostudujte základy zpracování obrazu, zaměřte se na problematiku filtrace obrazu a zpracování volumetrických dat.
2. Vyberte sadu často používaných filtrů a experimentálně implementujte jejich výpočet na GPU. Pracujte se 2D obrazem.
3. Zvolené filtry rozšiřte do 3D a otestujte jejich vlastnosti na objemových datech.
4. Diskutujte dosažené výsledky a možnosti budoucího vývoje.
5. Vytvořte stručný plakát nebo video prezentující vaši práci, její cíle a výsledky.

Literatura:

- Dle pokynů vedoucího.

Pro udělení zápočtu za první semestr je požadováno:

- Splnění 1-2 bodů zadání.

Podrobné závazné pokyny pro vypracování bakalářské práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Španěl Michal, Ing., Ph.D.**, UPGM FIT VUT
Datum zadání: 1. listopadu 2016
Datum odevzdání: 17. května 2017

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav počítačové grafiky a multimédií
612 66 Brno, Božetěchova 2
L.S.

doc. Dr. Ing. Jan Černocký
vedoucí ústavu

Abstrakt

Tato práce navrhuje řešení pro aplikaci konvolučních filtrů na velké množství 3D obrazových dat za využití výpočetního výkonu grafických karet. Popisované řešení využívá platformu OpenCL, jenž umožňuje akcelerovat veškeré výpočty na grafickém jádře, a příslušnou optimalizaci s využitím lokální paměti, která je na GPU dostupná. Návrh a implementace se zaměřuje primárně na Sobelův filtr.

Abstract

This thesis proposes a solution for acceleration of large 3D image data filtering by using the compute power of graphic cards. This solution uses OpenCL platform to dedicate all necessary computation onto graphical core of the graphic card and furthermore optimize this process using on-board local memory. Design and implementation of the mentioned approach is mainly focused on Sobel Operator.

Klíčová slova

Konvoluce, 3D, OpenCL, GPU, Konvoluční kernely, Akcelerace, Separabilita filtrů

Keywords

Convolution, 3D, OpenCL, GPU, Convolution kernels, Acceleration, Separable filters

Citace

JOCHLÍK, Jakub. *Akcelerace zpracování 3D obrazových dat na GPU*. Brno, 2017. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Španěl Michal.

Akcelerace zpracování 3D obrazových dat na GPU

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Michala Španěla, Ing.,Ph.D. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

....
Jakub Jochlík
11. května 2017

Poděkování

Rád bych poděkoval mému vedoucímu práce panu Michalu Španělovi, Ing.,Ph.D za odborné vedení a konzultace.

Obsah

1 Úvod **3**

2 Zpracování 3D obrazových dat **4**
2.1 Konvoluce . . . 5
2.2 Separabilní vs neseparabilní filtry . . . 6
2.3 Konvoluční filtry . . . 7

3 OpenCL **9**
3.1 Model platformy . . . 9
3.2 Paměťový model . . . 10
3.3 Exekuční model . . . 12

4 Principy akcelerace metod zpracování obrazu **13**
4.1 Přehled existujících řešení . . . 13
4.2 Problémy 3D konvoluce při využití GPU . . . 14

5 Návrh řešení pro výpočet 3D obrazových filtrů na GPGPU **15**
5.1 2D obrazová data . . . 15
5.2 Cílový hardware . . . 16
5.3 Příprava vstupních 3D dat . . . 17
5.4 Návrh konvolučního kernelu . . . 18
5.5 Výsledek konvoluce . . . 21

6 Implementace formou modulu pro knihovnu VPL **22**
6.1 Použitá knihovna . . . 22
6.2 Inicializace OpenCL a její parametry . . . 23
6.3 Výsledné moduly . . . 25
6.4 Měření rychlosti aplikace . . . 26

7 Experimentování a výsledky **27**
7.1 Testovací metodologie . . . 27
7.2 CPU vs GPU . . . 28
7.3 Separabilní vs Neseparabilní implementace . . . 29
7.4 Optimalizace pomocí lokální paměti . . . 30
7.5 Zhodnocení výsledků . . . 31

8 Závěr **32**

Literatura **33**

Přílohy **34**

A Obsah CD **35**

Kapitola 1

Úvod

Tato práce se zabývá návrhem a implementací knihovny, která by umožnila zrychlit zpracování a aplikaci filtrů na velké množství 3D obrazových dat, za využití výpočetního výkonu grafických karet. Použití tohoto druhu zpracování obrazu lze využít ve více oblastech, jako je například lékařství. Existuje velké množství filtrů a postupů, jak data zpracovávat. V této práci je však popsán návrh a implementace algoritmu pro aplikaci Sobelova filtru na 2D a 3D obrazová data, za pomocí konvoluce, využívající globální paměti grafické karty. Následně je tento algoritmus optimalizován pomocí lokální sdílené paměti, jenž je dostupná na grafické kartě. Varianta filtru s lokální pamětí obsahuje separabilní i neseparabilní řešení.

Existuje několik podobných postupů, jenž jsou více přiblíženy v kapitole 4. Přínosem této práce, oproti již existujícímu řešení, je však to, že popisované řešení využívá vlastnosti silného paralelismu konvolučního zpracování dat, a k jeho implementaci využívá platformu OpenCL, která je blíže popsána v kapitole 3. Výhoda zmíněné knihovny je především přenositelnost mezi operačními systémy a schopnost běhu aplikace na všech grafických kartách, které OpenCL podporují. Vlastnosti zvolené platformy a jejich zařízení ovšem znatelně ovlivňují návrh algoritmu a jeho implementaci. Toto je více popsáno v podkapitole 5.2.

Výsledkem této práce je aplikace, jenž umožní aplikovat (separabilní i neseparabilní) Sobelův filtr na velké množství 3D obrazových dat za pomocí GPU. Toto řešení je o 107% rychlejší (při zpracovávání dat o velikosti 1024x1024x309) než existující řešení využívající jako hlavní výpočetní jednotku CPU.

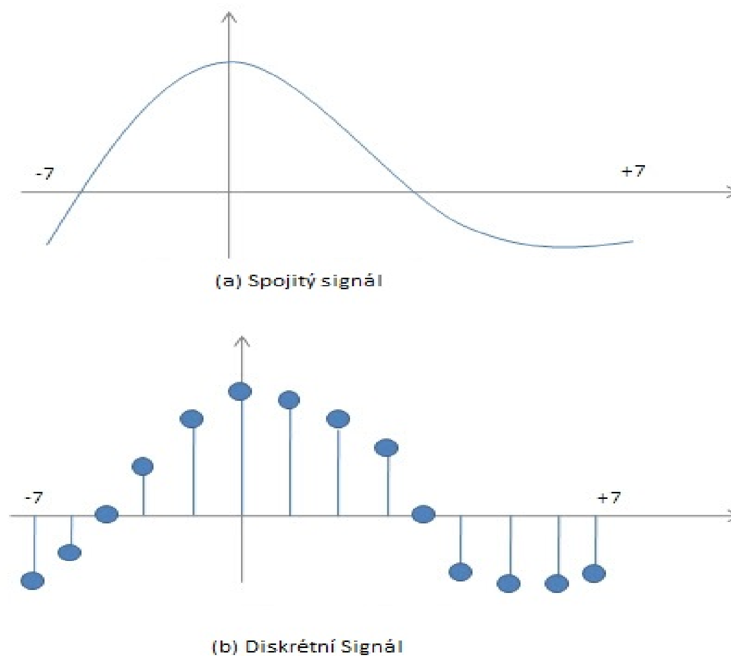
V první části tohoto dokumentu je shrnuta teorie týkající se problematiky zpracování 3D dat za pomoci konvoluce, a to v kapitole 2. Teorie v následujících kapitolách je vykládána se zaměřením na oblasti potřebné k řešení daného problému. Konkrétní řešení dále popisují kapitoly 5 a 6. Výsledky experimentů jsou poté zveřejněny v kapitole 7.

Kapitola 2

Zpracování 3D obrazových dat

V této kapitole zmíním a popíši některé základní pojmy z teorie zpracování signálů, a v podkapitolách 2.1 a 2.3 popíši konvoluci a některé konvoluční filtry. V kapitole 2.2, se zaměřím na vlastnosti separabilních a neseparabilních filtrů.

Vzhledem k tomu, že zpracovávat 3D data znamená zpracovávat 3D signál, zde stručně zmíním vlastnosti a typy signálů. Signál lze definovat jako veličinu nesoucí určitý typ informace, jenž je možno přenést, upravit nebo zobrazit [3]. Signál lze reprezentovat digitálně (pomocí proměnných) nebo analogově (jako elektrický signál). Analogový systém tedy pracuje se spojitými veličinami ve spojitém čase, kdežto digitální systém využívá konečný počet veličin v diskrétním čase. Na obrázku 2.1 jsou graficky znázorněny oba typy signálů. Spojitý



Obrázek 2.1: Příklad dvou typů signálů[1].

signál lze označit jako $x(t)$, kdežto diskrétní signál můžeme označit jako $x[n]$.

[1]Zdroj: http://durofy.com/properties-classification-of-signals/

Jednotlivé typy signálů mohou být zpracovávány různými způsoby, například konvolucí nebo diskrétní furierovou transformací (DFT). Návrh tohoto projektu využívá vlastností konvoluce při paralelním návrhu a implementaci pro akceleraci zpracovávání různých velikostí 3D obrazových dat na grafických kartách.

2.1 Konvoluce

V matematice je konvoluce definována jako matematická operace nad dvěma funkcemi (typicky ***f(x)*** a ***g(x)***)[5]. Výsledkem je třetí funkce - modifikovaná verze předchozích dvou funkcí. Na poli informačních technologií se používá konvoluce nad dvěma signály, kde ***f(x)*** většinou znázorňuje zdrojový, upravovaný signál, ***g(x)*** značí použitý filtr a α označuje integrační proměnnou. Stejně jako signály, i konvoluce může být spojitá

$$(f * g)(x) = \int_{-\infty}^{\infty} f(\alpha) g(x - \alpha)$$

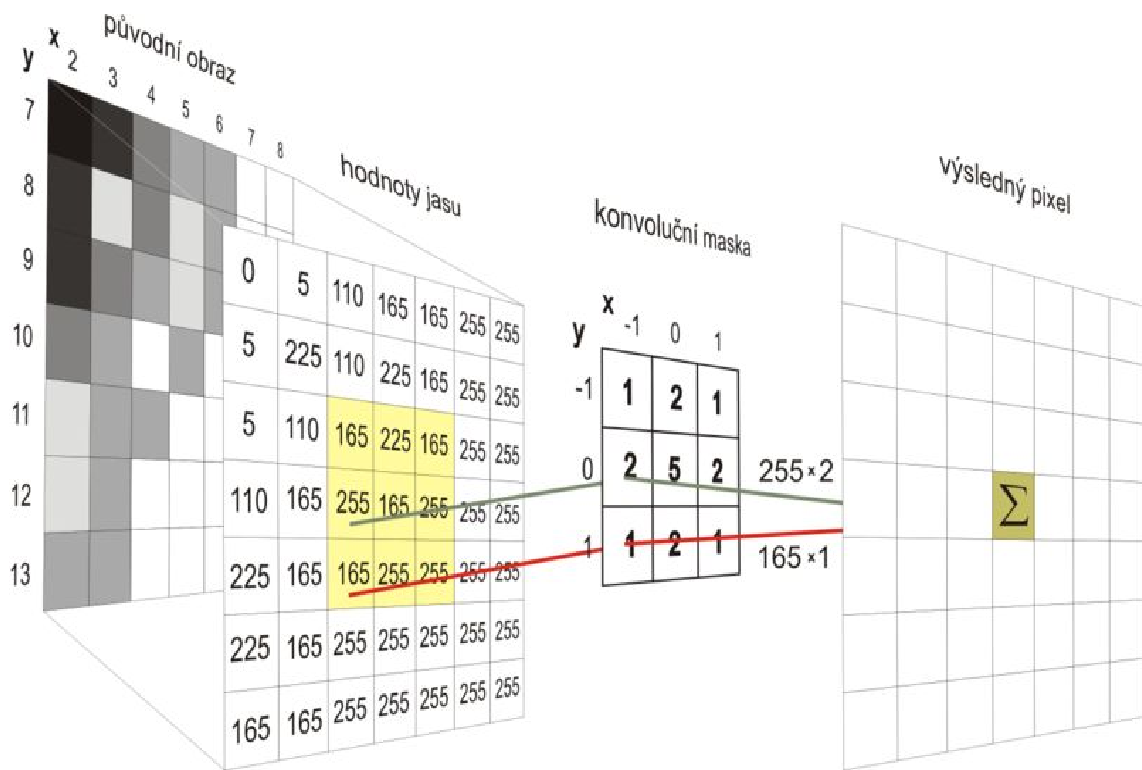
nebo diskrétní (zobrazen matematický zápis pro 2D konvoluci).

$$f(x, y) * h(x, y) = \sum_{i=-k}^{k} \sum_{j=-k}^{k} f(x - i, y - j) \cdot h(i, j)$$

Konvoluce není omezená na jednu dimenzi, je tedy multidimenzionální:

- jedno-dimenzionální: zpracování například audio signálu.
- dvou-dimenzionální: při aplikaci konvolučních filtrů na 2D obrázky.
- tří-dimenzionální: například při aplikaci filtrů na Velká, 3D obrazová data. Z pohledu návrhu je tedy tato varianta nejzajímavější.
- n-dimenzionální

V této práci se tedy zaměřím na diskrétní, tří-dimenzionální konvoluci, jenž se od dvou-dimenzionální konvoluce liší pouze hloubkou dat.



Obrázek 2.2: Aplikace konvoluční masky (kernelu) na 2D obrázek[2].

2.2 Separabilní vs neseparabilní filtry

Filtr je separabilní tehdy, pokud lze jeho konvoluční jádro rozložit do několika menších složek, jenž opětovným složením vytvoří původní konvoluční matici. Zjednodušeně lze říct, že filtr je separabilní právě tehdy, když výsledek konvoluce nerozloženého jádra je stejný, jako výsledek konvoluce jeho rozložených složek. Vzhledem k tomu, že Sobelův filtr je separabilní, si můžeme ukázat, jak takový rozklad vypadá. Využiji k tomu formuli číslo 2.3 pro detekci hran podle osy X z kapitoly 2.3.

$$F_x = A * \begin{pmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{pmatrix} = A * \begin{pmatrix} 1 & 0 & -1 \end{pmatrix} * \begin{pmatrix} 1 \\ 2 \\ 1 \end{pmatrix} \tag{2.1}$$

Tato vlastnost filtrů umožňuje separabilní implementaci daného filtru. Výhodou takto implementovaného kernelu je rychlejší zpracování obrazových dat, na který je filtr použit. Ačkoliv pro 2D obrazy je zrychlení zanedbatelné, u velkého množství 3D dat dochází ke značnému, nezanedbatelnému urychlení prováděného výpočtu. Tento fakt je velmi zajímavý pro tuto práci, neboť navrhované řešení umožní akcelerovat výpočty například pro velké množství dat ve formě CT skenů lidského těla, ale i dat z jiných profesních okruhů. Vliv separabilní implementace na dobu výpočtu bude znázorněn v kapitole 7 sekce 7.3.

[2]Zdroj: https://is.mendelu.cz/eknihovna/opory/zobraz_cast.pl?cast=18431

2.3 Konvoluční filtry

Konvolučních filtrů a jejich modifikací je mnoho, a každý slouží jinému účelu. V této sekci se už budu zaobírat filtry pouze ve spojením s obrazovými daty. Konvoluční filtry nemodifikují zdrojový obrázek, výsledek konvoluce je ukládán do před-připravené matice. Rozměry konvoluční masky jsou typicky čtvercového tvaru o velikosti N x N, kde N je liché číslo. Mimo to se ale lze setkat i s konvoluční maskou, jenž má tvar kruhu, kříže, aj.[7]. Ačkoliv je můj návrh zaměřen pouze na Sobelův filtr, uvedu zde i Gauss filtr, neboť by případná implementace tohoto filtru vyžadovala jen velmi drobné úpravy v návrhu.

- **Sobelův filtr** – V oblasti digitálních filtrů, tento filtr paří do skupiny konvolučních filtrů sloužících pro detekci hran. Mimo jiné v této skupině nalézt například:
  - Robertsův operátor $\begin{pmatrix} -1 & 0 \\ 0 & 1 \end{pmatrix}$ a $\begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix}$
  - Prewitt 3x3 operátor $\begin{pmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{pmatrix}$
  - Sobelův 3x3 operátor $\begin{pmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{pmatrix}$

  V této práci se tedy zaměřím na Sobelův operátor, jenž dává větší váhu na střední pixely, což má za následek spolehlivější a lepší detekci hran na filtrovaném obrázku. Jedná se o separabilní filtr. Separabilita filtrů a je popsána v sekci 2.2. Zápis takovéto 2D konvoluce pro detekci vertikálních hran by pak vypadal následovně

$$F_y = A * \begin{pmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{pmatrix} \tag{2.2}$$

  kde A je vstupní obrázek a F_y značí výsledek konvoluce. Obdobně lze provést i horizontální detekci hran

$$F_x = A * \begin{pmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{pmatrix} \tag{2.3}$$

  kde A je vstupní obrázek a F_x značí výsledek konvoluce. Kombinací obou výsledků z rovnic 2.2 a 2.3 lze vytvořit kompletní 2D detekci hran $F = \sqrt{F_x^2 + F_y^2}$. Výsledek těchto operací lze pozorovat na obrázku 2.3, kde levý obrázek je použit jako zdrojový, a pravý obrázek je výsledek po filtraci Sobelovým filtrem.

- **Gauss-blur filtr** – Poslední filtr, který tu velmi krátce zmíním, je vyhlazovací Gauss-blur filtr (také známý jako "Gauss-Smoothing"). Jak jeho název napovídá, slouží k rozmazání rysů obrázku. Matematický zápis takového 2D filtru vypadá následovně

$$G_{2D}(x, y, \sigma) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

kde $\sigma > 0$ stanovuje šířku použitého kernelu. Při nutnosti použít N-tou dimenzi zde ještě zmíním matematický zápis právě pro N-dimenzí

$$G_{ND}(\vec{x};\sigma) = \frac{1}{(\sqrt{2\pi}\sigma)^N} e^{-\frac{|\vec{x}|^2}{2\sigma^2}}$$

Výsledek této filtrace můžeme pozorovat na obrázku 2.4

Obrázek 2.3: 2D detekce hran[3].

Obrázek 2.4: 2D Gauss filter[4].

[3]Zdroj: http://www.programming-techniques.com/2013/03/sobel-and-prewitt-edge-detector-in-c.html
[4]Zdroj: http://forums.ni.com/t5/Machine-Vision/gaussian-filter/td-p/2441104

Kapitola 3

OpenCL

Open computing language (dále jen OpenCL) je otevřená výpočetní platforma [6] zaměřená na standardizování paralelního programování heterogenních systémů. Nejrozšířenější využití nachází jako platforma umožňující akceleraci výpočtu za využití grafických zařízení. Tyto zařízení mohou být například:

- GPGPU
- CPU
- FPGA

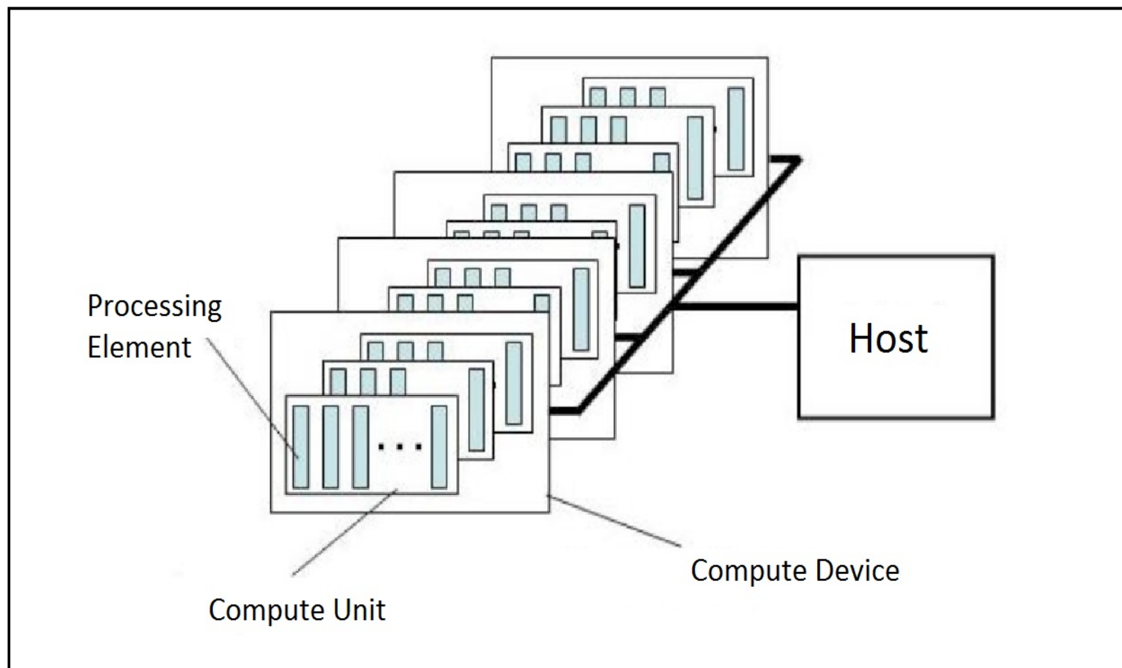
Tato platforma se skládá ze samostatného programovacího jazyka a samotné knihovny. Programovací jazyk OpenCL C je podmnožinou C99, nicméně od verze 2.1 byl nahrazen novým OpenCL C++, který je podmnožinou C++14. I přesto je ovšem OpenCL C stále hojně používán. Tuto knihovnu je možné využít jak v případě akcelerace úkolově paralelních programovacích modelů, tak i datově paralelních modelech. Nicméně při akceleraci 3D konvoluce zde popisovaného řešení je využíván pouze model datově paralelní. Toto je způsobeno především faktem, že povaha problému spočívá v nutnosti zpracovat velké množství obrazových dat pomocí konvolučních filtrů.

Tato sekce je věnována stručnému popisu knihovny OpenCL, kterou jsem si zvolil pro implementaci mnou navrhovaného řešení. Zaměřím se především na ty vlastnosti OpenCL, které měly největší vliv na návrh akceleračních algoritmů. Ze stejného důvodu je v následující sekci 5.2 zmíněn i vliv cílového hardwaru na implementaci.
Architekturu OpenCL je možné popsat několika modely:

1. Model platformy
2. Exekuční model
3. Paměťový model

3.1 Model platformy

Prvním pohledem je model platformy. Jak lze pozorovat na obrázku 3.1 model platformy se skládá z hosta, k jemuž je připojeno jedno a více zařízení podporujících OpenCL. Roli hosta zde zastává CPU. Zařízení podporující OpenCL může být například grafická karta. Toto zařízení se dále dělí na výpočetní jednotky (Compute units - CUs), které se dále dělí na



Obrázek 3.1: OpenCL - Model platformy[1].

procesorové prvky (Processing Elements - PEs). Tyto elementy pak vykonávají samotný výpočet nad daty. Procesorové prvky, které tvoří právě jednu výpočetní jednotku, vykonávají program jako SIMD jednotky (tedy v konkrétním časovém okamžiku zpracovávají všechny procesorové prvky stejnou instrukci spuštěného programu), nebo SPMD, kde všechny jednotky vykonávají stejný program, nicméně nemusí provádět stejnou instrukci. Během obou situacích mohou výpočetní jednotky pracovat s odlišnými daty.

3.2 Paměťový model

Dalším možným pohledem je paměťový model, jenž specifikuje čtyři paměťové prostory. Každý jeden z těchto paměťových prostorů má jiné vlastnosti a práva přístupu. Zařízení se tedy skládá z následujících typů pamětí:

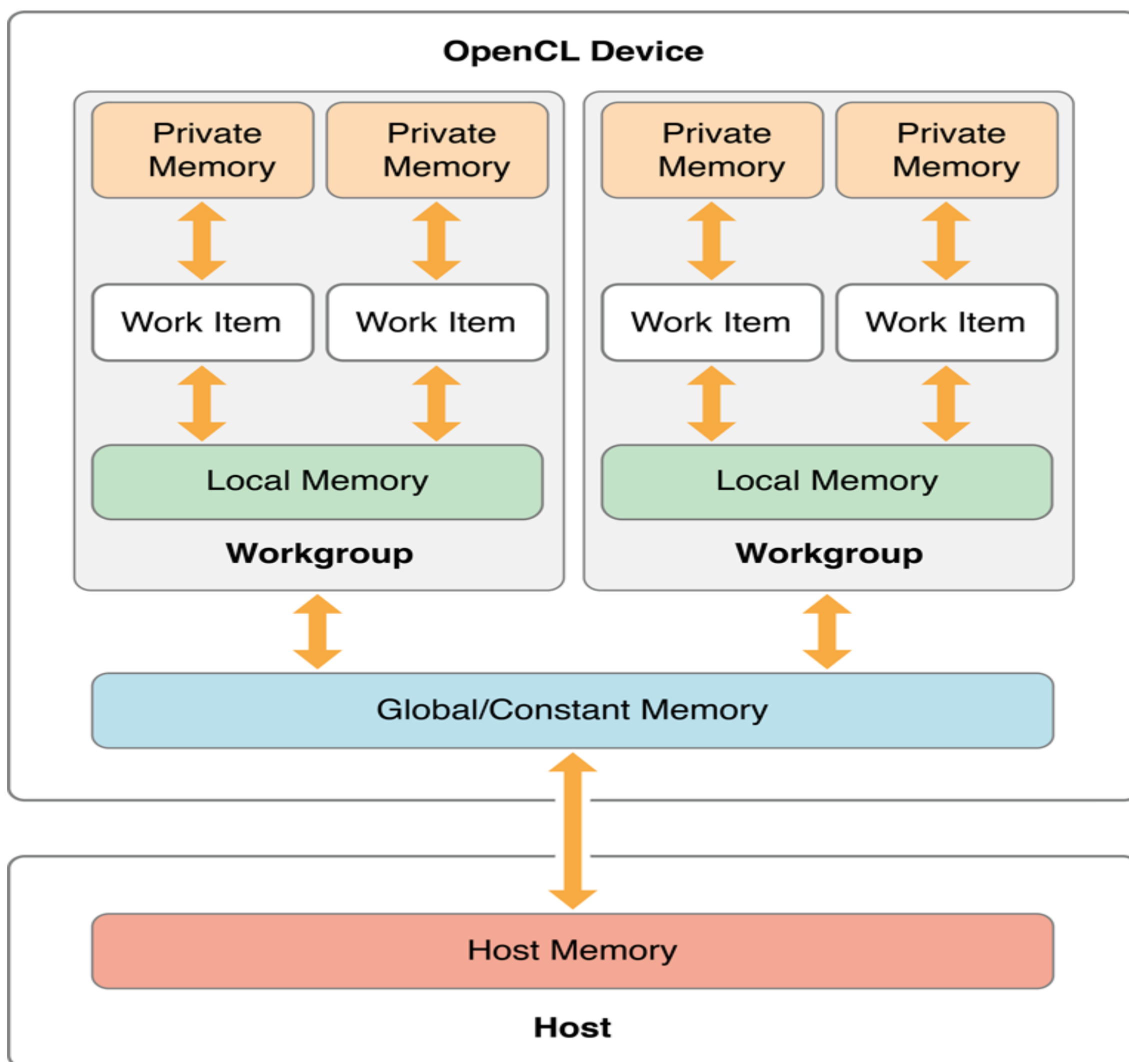
- **Globální (Global Memory)** – Jedná se o paměť nejvyšší úrovně. Zde mohou číst i zapisovat všechny právě běžící vlákna kernelu. Společně s konstantní pamětí je to jediné místo, kam má přístup také host, a to jen pomocí předdefinovaných paměťových prostředků, které poskytuje OpenCL. Toto je největší, ale zároveň nejpomalejší paměť na grafické kartě.
- **Konstantní (Constant Memory)** – Vlastnosti této paměti jsou téměř shodné s vlastnostmi paměti globální. Liší se pouze faktem, že spuštěné kernely nemohou do této paměti zasahovat, a modifikovat ji, ale pouze načítat její obsah.
- **Lokální (Local Memory)** – Tento typ paměti je přístupný pouze vláknům (work-item) uvnitř jedné výpočetní skupiny (work-group). Je mnohonásobně rychlejší a

[1]Zdroj: http://www.comp.leeds.ac.uk/viznet/reports/GPU_report/GPUSTARReport_html.html

menší než paměť globální (veliká pouze v řádech desítek kilobajtů). Vlákna mají právo číst i psát.

- **Privátní (Private Memory)** – Poslední paměť, která je zároveň nejmenší a nejrychlejší, je paměť privátní. Ta je přístupná právě jednomu vláknu, jemuž slouží například jako paměť proměnných.

Z obrázku 3.2 je tedy patrné, že při optimalizaci na lokální popřípadě privátní paměť musí být obrazová data překopírována z paměti hosta (paměť RAM) do globální paměti grafické karty, a odtud následně rozděleny do lokálních, popřípadě privátních pamětí. O tom ale detailně až v kapitole 5.



Obrázek 3.2: OpenCL - Paměťový model.

3.3 Exekuční model

Exekuční model je třetí a finální popis platformy OpenCL. Tento model je rozdělen do dvou částí:

1. Aplikace – Aplikace běží na klasickém procesoru a stará se o veškerou režii, která je nutná pro spouštění kernelů na grafické kartě.

2. Kernel – Kernel obsahuje kód programu, který je vykonáván na grafickém adaptéru. Jednotlivé kernely jsou odesílány a řazeny do fronty ke zpracování. Každá instance kernelu obsahuje hostem definovaný počet výpočetních skupin o přesně stanovené velikosti.

Při návrhu aplikace využívající OpenCL je také potřeba vzít v úvahu cílový hardware, neboť různé výrobní architektury mají určité odlišnosti, které mají vliv na samotný návrh knihovny. Toto je blíže popsáno v sekci 5.2.

Kapitola 4

Principy akcelerace metod zpracování obrazu

Tato kapitola obsahuje přehled již existujících řešení pro aplikaci filtrů na 3D obrazová data. Jejich letmé přiblížení lze nalézt v sekci 4.1. Sekce 4.2 dále popisuje problémy, které jsou spojené s akcelerací výpočtů nad velkém množství 3D obrazových dat, a jednotlivá řešení těchto problémů.

Při akceleraci zpracování obrazu lze využívat několik nástrojů, které platforma OpenCL poskytuje. Nyní zmíním některé prvky, které lze použít při optimalizaci výsledného algoritmu:

- **Lokální paměť** – Do této paměti lze načíst zpracovávaná data z globální paměti grafické karty. Tato paměť je mnohonásobně rychlejší než paměť globální, a proto při jejím použití dochází k zásadnímu urychlení výpočtů. Detailnější popis této paměti lze nalézt v sekci 3.2.

- **Konstantní paměť** – Některá data není potřeba měnit, slouží pouze ke čtení. Taková data je vhodné umístit do konstantní paměti grafické karty. Tato paměť je vytvořena ještě před samotným spuštěním kernelu a dochází tak zrychlení výpočtů. Detailnější popis této paměti lze nalézt v sekci 3.2.

- **Float4** – Použitím datového typu `float4` lze z paměti načíst zaráz 128 bitů dat namísto pouhých 32 bitů (oproti použití běžného `float`). Tímto dochází k dalšímu zrychlení výpočetního kernelu.

- **Loop unrolling** – Ačkoliv ve většině případů je rozvinutí programové smyčky nevýhodné, zmíním ji zde proto, že při práci s velmi malými filtry (například 3x3) dochází k nezanedbatelnému zrychlení.

Výše zmíněné metody optimalizace blíže popisuje studie „Optimalizace OpenCL obrazové konvoluce“[1].

4.1 Přehled existujících řešení

Jedno z existujících řešení [2] popisuje konvoluci objemného množství 3D dat na GPU a její dekompozici. Práce popisuje konvoluční algoritmy s využitím Furierovy transformace. Navrhované řešení využívá DIF (decimation in frequency) algoritmu. Práce umožňuje použití

[1] Zdroj: https://www.evl.uic.edu/kreda/gpu/image-convolution/

více grafických zařízení a porovnává výsledky mezi multi-GPU a multi-CPU systémy. Výsledné řešení je navrženo pro jazyk CUDA, což způsobuje omezení tohoto návrhu pouze na grafická zařízení od společnosti Nvidia.

Druhé existující řešení [1], které zde zmíním, se zaměřuje na rekonstrukci okolní půdy, pomocí zaznamenaných seismických vln, za účelem nalezení ropných a plynových ložisek. K vytvoření tohoto obrazu využívá 3D konvoluci, jejímž výsledkem je 3D model okolní půdy. Popisované řešení se zaměřuje na implementaci algoritmu pro moderní CPU a GPU, a porovnává dosažené výsledky mezi jednotlivými implementacemi. Zvolený filtr je implementován neseparabilně. Navrhované řešení je zaměřeno na karty Nvidia, neboť také využívá jazyk CUDA.

4.2 Problémy 3D konvoluce při využití GPU

Pro správně navržení knihovny pro provádění konvoluce nad 3D obrazovými daty za pomoci grafických zařízení je potřeba nejprve vyřešit několik problémů, které jsou s tímto tématem spojeny.

První problém, který je potřeba vyřešit jsou krajní pixely u zpracovávaných obrázků. Vzhledem k tomu, že šířka, výška a hloubka konvolučních kernelů je nejméně tři pixely, a faktem, že zpracovávaný pixel se nachází uprostřed zpracovávaného prostoru, je nutné nastalou situaci správně ošetřit. Pokud by se na řadu totiž dostal okrajový pixel, aplikace by se pokusila zpracovávat hodnotu mimo vyhrazený paměťový prostor matice, což je nežádoucí situace. Toto lze řešit několika způsoby:

- **Oříznutí** – Prvním možným řešením je prosté oříznutí obrázku. Toto je nejjednodušší varianta, kdy se provede konvoluce nad celým obrázkem, kromě jeho krajních hodnot. Ty jsou pak následně odstraněny, přičemž dochází ke zmenšení obrázku, což ne vždy může být přípustné.

- **Zaobalení** – Druhou možností je použití protějších krajních hodnot. Pokud tedy dojde ke zpracovávaní krajních pixelů, jednoduše se použije hodnota z druhého konce řádku, nebo opačného rohu obrazu. Velikost obrazu je zachována, ale krajní pixely jsou tímto mírně zkresleny.

- **Rozšíření** – Poslední variantu řešení tohoto problému je obalení celého obrazu další vrstvou krajních hodnot. Tyto hodnoty jsou zvoleny tak, aby neovlivnily výsledek konvoluce. Po dokončení konvoluce jsou krajní přidané hodnoty odstraněny. Použití rozšíření nemodifikuje velikost výsledného obrazu.

Další problém představuje samotná velikost zpracovávaných dat. Ty totiž mohou dosahovat řádek gigabajtů, což výrazně přesahuje velikost lokálních pamětí. Lokální paměť je z hlediska optimalizace a rychlosti běhu velmi důležitá, neboť čtení a zápis v této paměti je několikanásobně rychlejší než u paměti globální. Toto je detailněji popsáno v sekci 3.2. Proto je potřeba vytvořit logiku, která by umožnila efektivní využití rychlejší, lokální paměti. Toto je blíže popsáno v kapitole 5.

Poslední problém nastává při stanovení velikosti jednotlivých oddílů, do kterých bude zpracovávaný obrázek rozdělen. Tento problém je úzce spojen s velikostí obrázku a velikostí lokální paměti (tedy je spojen s dvěma výše popsanými problémy). Je totiž potřeba zajistit, aby se daný blok vešel do lokální paměti a zároveň byl dělitelný zvolenou velikostí bloku.

Kapitola 5

Návrh řešení pro výpočet 3D obrazových filtrů na GPGPU

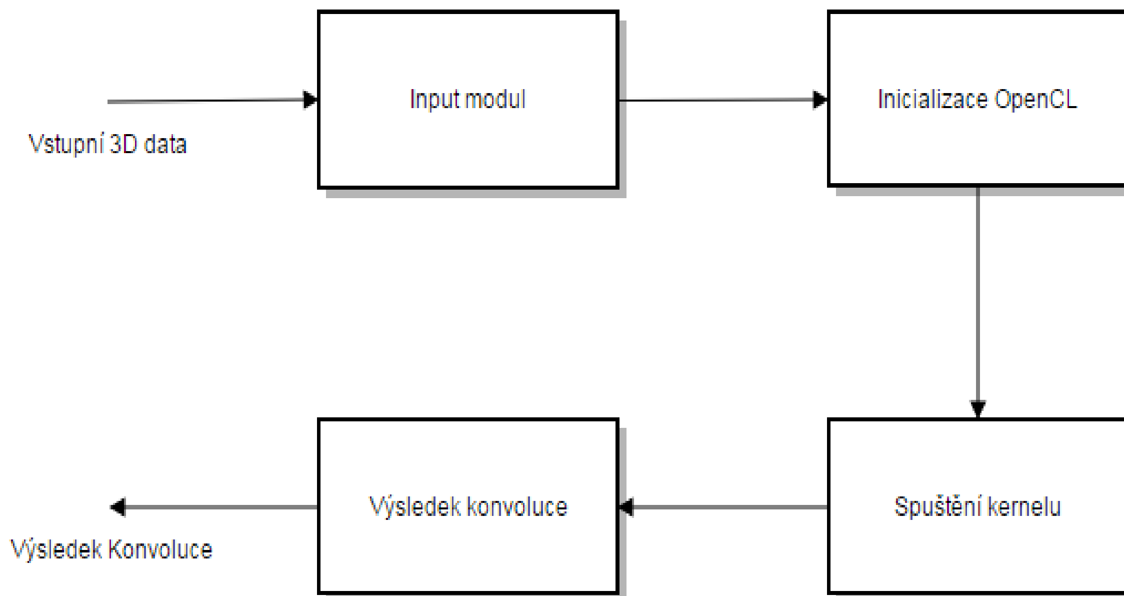
Jak jsem již naznačil v úvodu, a pomocí existujících prací [2], cílem mojí práce je vytvořit přenositelnou knihovnu, která by za využití platformy OpenCL umožnila akcelerovat zpracovávání velkého množství 3D obrazových dat pomocí grafických zařízení. Jedná se například o aplikaci konvolučních filtrů na zmíněná data. Tato práce se zaměřila především na Sobelův operátor. Mým úkolem je takovou knihovnu navrhnout a implementovat.

Nejprve zde zmíním vliv 2D provedení na výsledný návrh knihovny. Poté vysvětlím řešení zajišťující správné načtení a přípravu obrazových dat na straně hosta, jenž budou dále zpracovávána. Následovat bude inicializace a rozvrhnutí běhu OpenCL kernelů, aby bylo zajištěno správné fungování těchto kernelů. Po dokončení popisu těchto příprav přejdu k samotného návrh konvolučního kernelu. Ten musí být navržen tak, aby fungoval na všech kartách podporujících OpenCL platformu. Jako poslední přiblížím zpracování výsledku a jeho prezentaci, popřípadě další zpracování.

5.1 2D obrazová data

Jako základ mého návrhu pro aplikaci filtru na 3D obrazová data posloužil 2D model řešící stejný problém, pouze v dvou dimenzích. Jedná se tedy o „Proof Of Concept“ model, který postupným experimentováním ověřil, zda-li je program schopný správně aplikovat filtr na 2D obraz. Výsledkem těchto experimentů byl modul, který dokázal naivně, pouze za pomoci globální paměti grafické karty, aplikovat zvolený filtr na požadovaný 2D obraz.

Jakmile jsem potvrdil funkčnost modulu, jenž využíval globální paměť, jsem provedl následnou optimalizaci výpočetního kernelu pomocí lokálních pamětí, dostupných na grafickém adaptéru. Výsledkem byl funkční model, který dokázal rychleji a efektivněji řešit zadaný problém ve dvou dimenzích. Výsledek tohoto experimentování posloužil při tvorbě finálního návrhu, který je znázorněn na obrázku 5.1.



Obrázek 5.1: Výsledný 3D model.

5.2 Cílový hardware

Můj návrh je cílený na grafické karty od společnosti Nvidia, konkrétně karta série 600, s výrobní architekturou nesoucí název *Fermi*, Nvidia GeForce GTX 670M [4]. Jádrem této desky je pak čip GF114. I přes to, že je OpenCL multiplatformní, a kernely, které jsou na GPU spuštěny, jsou téměř hardwarově nezávislé, je potřeba zohlednit vlastnosti cílové architektury, na kterou je tento návrh zaměřen, a zároveň umožnit co nejvyšší využití výpočetního výkonu na ostatních grafických čipech. Z těchto důvodů níže uvádím informace využité grafické karty.

Jádro GF114 obsahuje 336 CUDA jader, pracujících na frekvenci 598 MHz, o teoretickém výkonu 803.7 GFLOPS (jednoduchá přesnost). Jednotlivé CUDA jádra (často nazývané *streaming procesory* - SP) jsou rovnoměrně rozděleny do sedmi Streaming Multiprocesorů - SM. Tyto SM bloky také kromě CUDA jader obsahují vlastní sdílené paměti a registry, které jsou popsané v sekci 3.2. Výše zmíněné bloky jsou typu SIMD(Single Instruction Multiple Data). Konkrétně popisovaný čip disponuje 16KB sdílené paměti. Zároveň má na desce k dispozici 3GB DDR5 globální paměti, připojené na 192-bitové sběrnici. Paměť pracuje na frekvenci 1500MHz, a tedy je možné dosáhnout propustnosti až 72 GB/sec.

Výše zmíněné informace jsou pro tento návrh důležité z několika důvodů. Aby bylo dosaženo co nejlepších výsledků, je zapotřebí zajistit, aby výpočetní bloky byly alespoň o velikosti 32 vláken. Dále je nutné si uvědomit, že jednotlivé kernely budou spouštěny na výpočetních blocích typu SIMD. Pokud tedy v programu kernelu dojde k větvení programu (například za využití *IF/ELSE* logiky), bude nutné některá vlákna pozastavit, dokud oddělený blok kódu nebude dokončen. To způsobí nevyužití plného potenciálu jádra a efektivita

kernelu se snižuje. Navíc je také nutno zajistit optimální čtení z paměti pomocí lokální sdílené paměti tak, aby co nejméně výpočetních jednotek čekalo na data, která by mohla zpracovat.

Vzhledem k velikosti zpracovávaného 3D obrazu není možné celý obraz nahrát do lokální paměti naráz. Je tedy potřeba tento obraz rozdělit do menších bloků. Tyto bloky musí být dostatečně velké, aby docházelo k efektivnímu využití výpočetních jednotek, a zároveň dostatečně malé, aby se do takto malé paměti vešly. Zároveň je nutné brát v úvahu proměnlivou velikost lokálních pamětí na různém typu hardwaru. Proto jsem se rozhodl rozdělit zpracovávaný obraz do bloků o velikosti 32x32x1 pixelů. Každý jeden z těchto bloků bude nahrán a zpracováván v lokální paměti právě jedné výpočetní skupiny.

5.3 Příprava vstupních 3D dat

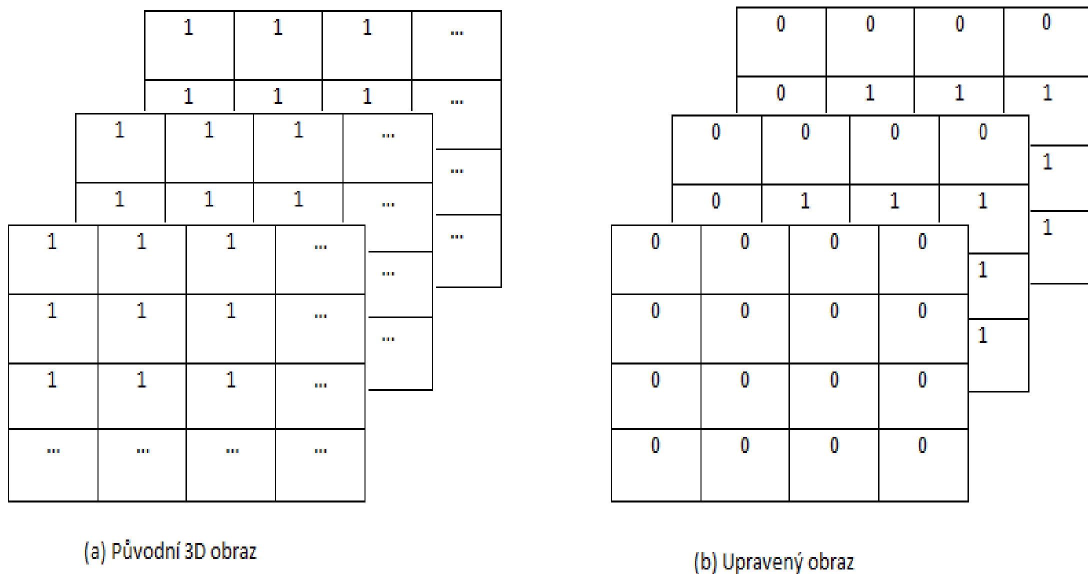
Prvním krokem je načtení a zpracování vstupního 3D obrazu, jenž má na starost první fáze, znázorněná na obrázku 5.1. Vstupní modul tedy nejprve zjistí rozměry zpracovávaného obrazu. Vzhledem k tomu, že jedním z cílů této práce je umožnit zpracovávání 3D dat různých rozměrů, je potřeba získané hodnoty upravit tak, aby ostatní moduly dokázaly správně fungovat nezávisle na rozměrech vstupního 3D obrazu. Tyto zaokrouhlené hodnoty musí odpovídat velikosti jednoho bloku , do kterých bude později zpracovávaný obraz rozdělen. Velikost těchto bloků je blíže zdokumentována v sekci 5.4. Dále je potřeba počítat se situací, kdy je velikost obrazu beze zbytku dělitelná velikostí jednotlivých bloků. Tuto situaci lze elegantně vyřešit pomocí operátoru modulo.

Listing 5.1: Získání zaokrouhlených rozměrů 3D obrazu

```
XSize = XSize + (TILEXSIZE − (XSize % TILEXSIZE));
YSize = YSize + (TILEYSIZE − (YSize % TILEYSIZE));
ZSize = ZSize + 2;
```

Jednotlivé bloky („Tiles“) mají definovanou šířku a výšku 32 pixelů. Při bližším prozkoumání zmíněných tří řádků v útržku kódu 5.1 lze pozorovat, že v případě, kdy obrázek nebude dělitelný velkostí jednotlivých bloků, dojde k jeho rozšíření o určitý počet pixelů, který stanoví výsledek výrazu v závorce. Tato rovnice zároveň řeší i obrázky, které jsou beze zbytku dělitelné velikostí jednotlivých bloků. V tuto chvíli jsem tedy spočítal zaokrouhlenou hodnotu pro výšku a šířku (osy X a Y) zpracovávaného obrázku. Dále je nutné rozšířit obraz po ose Z (tedy do hloubky). Vzhledem k tomu že 3D data lze prezentovat jako velké množství 2D obrázků jdoucích za sebou, stačí rozšířit velikost obrazu o dvě vrstvy − jednu vloženou nad obraz a jednu pod obraz.

V tuto chvíli tedy mám získané správně upravené rozměry zpracovávaného obrazu. Tyto rozměry následně využijí pro přípravu dvou datových polí. Velikost těchto polí se bude rovnat násobku zaokrouhlených hodnot získaných dříve. Jedno pole bude sloužit jako zdrojový obraz, do druhého pak uložím výsledek konvoluce. Ze vstupního kanálu poté načtu 3D obraz, který budu zpracovávat, a uložím jej do alokovaného pole. Zde nastává problém popsaný v kapitole 4 sekce 4.2. Ve chvíli, kdy by kernel začal zpracovávat jeden z krajních pixelů, by se pokusil načíst hodnotu mimo paměť, což způsobí nedefinované chování. V tomto návrhu jsem se rozhodl použít řešení formou rozšíření obrazu. Toho jsem dosáhl prostým posunutím pixelů v matici ve všech dimenzích o jednu pozici. Pokud tedy původní pixel ležel na pozici (0,0,0), jeho pozice byla upravena na (1,1,1). Obecně řečeno pixely na pozicích $p[x, y, z]$ byly přesunuty na pozice $p[x + 1, y + 1, z + 1]$. Grafické znázornění tohoto procesu je na obrázku 5.2.



Obrázek 5.2: Obalení obrazu neutrální hodnotou.

V tuto chvíli je již obraz připravený na konvoluci. Před samotným spuštěním kernelu je ještě zapotřebí inicializovat OpenCL. Tento krok je blíže popsán v sekci 6.2. Následující sekce vysvětlí samotný konvoluční kernel.

5.4 Návrh konvolučního kernelu

Základním úkolem navrženého kernelu je provést filtraci obrazových dat za využití konvoluce. Návrh se cíleně zaměřuje na detekci hran v 3D obrazu. Při navrhování tohoto kernelu jsem dbal především na jeho jednoduchou implementaci a popřípadnou modifikaci v generalizovaný algoritmus, který by dále umožnil aplikovat filtry s odlišnými rozměry konvolučního jádra. Dosáhnout tohoto cíle mi značně usnadnila dimensionalita (možnost indexovat pixely/vlákna pomocí jejich prostorových souřadnic), která je v OpenCL podpořena. Při tvorbě tohoto kernelu jsem použil model, kde jedno vlákno zpracovává právě jeden pixel.

Návrh mého kernelu se skládá ze tří hlavních kroků:

1. Vypnutí výpočtů pro okrajové pixely
2. Na-alokování lokální sdílené paměti a její naplnění daty, která jsou dostupné v globální paměti karty
3. Aplikace filtru pro konkrétní pixel a nahrání výsledné hodnoty do výstupního globálního bufferu

Nyní jednotlivé kroky popíši a vysvětlím jejich logiku

Vzhledem k tomu, že kernel využívá modelu, kdy jedno vlákno programu OpenCL zpracovává právě jeden pixel obrazu, je zapotřebí vypnout vlákna pro krajní pixely. I přesto, že načtený obraz je obalen vrstvou neutrálních pixelů, aby se zamezilo čtení hodnot mimo paměť při zpracovávání krajních pixelů, je stále potřeba zamezit výpočtům na těchto krajních, rozšířením přidaných pixelech (tento proces rozšíření obrazu je blíže popsán v sekci

5.3). I tyto přidané pixely totiž mají svoje vlákna, která by je zpracovávala. Díky faktu, že celý obraz je obalený ve vrstvě neutrálních pixelů, které do obrazu přidal vstupní modul (obrázek 5 sekce 2.2), můžeme lehce určit přesně jednotlivá vlákna, které je nutno zastavit, popřípadě ukončit. Tento krok lze velmi usnadnit pomocí využití identifikačních čísel (`ID`), jenž má při startu kernelu každé vlákno přiděleno. Počet těchto identifikátorů je roven počtu dimenzí, ve kterých byl kernel spuštěn. V tomto případě tedy každé vlákno obsahuje skupinu tří identifikátorů. Následující útržek kódu 5.2 demonstruje lokalizaci vlákna pomocí jeho souřadnic v prostoru za pomocí funkce `get_global_id()`. Grafické znázornění těchto souřadnic lze pozorovat na obrázku 5.3.

Listing 5.2: Získání 3D souřadnic vlákna

```
x = get_global_id(0);
y = get_global_id(1);
z = get_global_id(2);
```

Nyní je potřeba si uvědomit, že 3D data jsou v paměti uložena jako 1D pole, a tedy i vlákna jsou indexována za sebou v 1D poli. Proto je potřeba vypočítat výsledný index pomocí získaných souřadnic (čímž zároveň získáme i index zpracovávaného pixelu). Tento index lze získat dosazením získaných souřadnic do rovnice $index = x + y * sirka + z * sirka * vyska$, kde šířka a výška jsou odpovídající rozměry zpracovávaného obrazu. Tímto způsobem je možné vyhledat a vypnout jakékoliv vlákno podle zadaných souřadnic. Pro zamezení přístupu mimo paměť je potřeba vypnout všechna vlákna, jejichž pozice na ose `X` je nulová nebo maximální (maximální znamená, že vlákno/pixel mají index odpovídající výšce obrázku). Stejně tak je potřeba vypnout vlákna jejichž souřadnice `Y` nebo `Z` jsou rovny nule nebo hodnotě odpovídající šířce nebo hloubce. Vlákna jsou příslušnou podmínkou nalezeny a jejich činnost je v příslušném bloku ukončena. Vypnutím těchto vláken tedy zabráníme nežádoucím přístupům mimo paměť. Tímto je první krok kernelu dokončen.

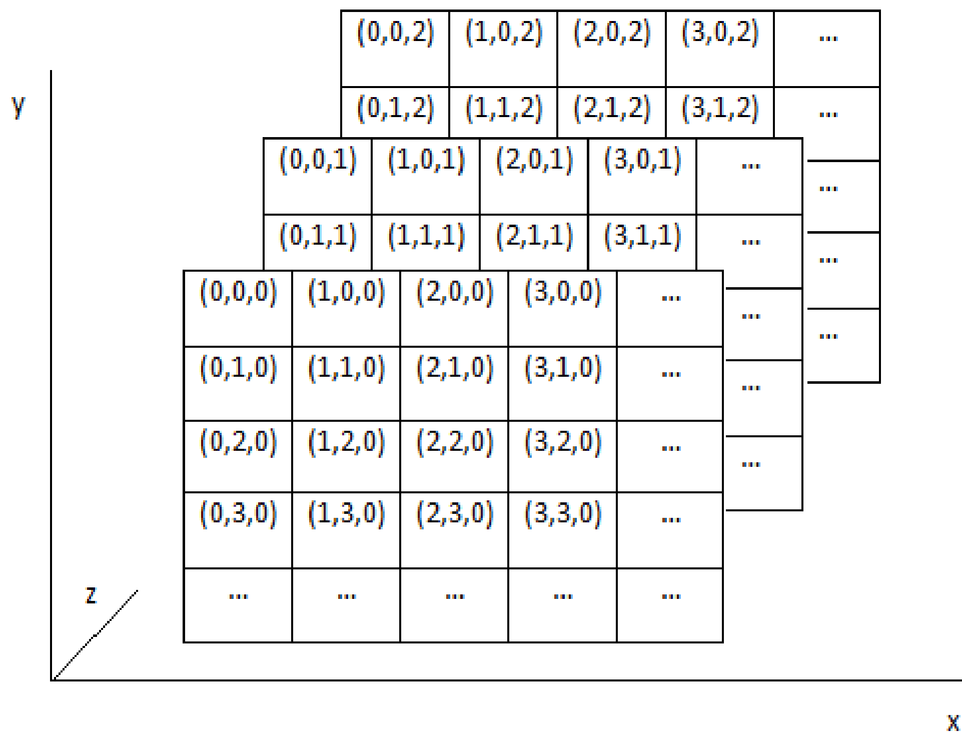
Druhým důležitým krokem je naplnění lokálních pamětí jednotlivých pracovních skupin z paměti globální, kde je momentálně uložen zdrojový obraz. Pole na-alokované do lokální paměti musí být stejného typu jako je pole globální, v tomto případě je použit datový typ *float*. Velikost lokálního pole se odvíjí od velikosti obrazového bloku (obrazové bloky jsou podrobněji popsány v sekci 5.2). Zároveň je potřeba do tohoto pole nahrát i okolní pixely zpracovávaného obrazového bloku, aby bylo možné zpracovat krajní pixely v tomto bloku. Praktický příklad je demonstrován v útržku 5.3.

Listing 5.3: Alokace lokálního pole

```
__local float blok[34 * 34 * 3];
```

Nyní je potřeba vytvořit logiku, která by toto na-alokované lokální pole naplnila příslušným obrazovým blokem a jeho okolím. Toto pole nejdříve naplním základním blokem bez jeho okrajů. Jakmile je základní blok načten, nastává detekce krajních pixelů. Je tedy zapotřebí nalézt všechna vlákna, jenž mají za úkol zpracovat krajní pixely v daném bloku. K tomu jsem využil funkci `get_local_id()`, kterou poskytuje OpenCL. Její použití je podobné jako v útržku 5.2. Jediným rozdílem je, že tato funkce vrací souřadnice relativně k pozici vlákna v dané pracovní skupině. Pokud je tedy globální pozice vlákna (32,32,1) jeho lokální pozice je (0,0,0)(pro bloky velikosti 32x32x1).

Nejdříve pomocí funkce `get_local_id(0)` naleznu všechny pixely na levém a pravém okraji bloku, a uložím je do lokálního pole. Poté načtu jejich sousední pixely (příslušně zleva a zprava) a také je uložím do lokálního pole, vedle nachystaných sousedních pixelů. Následně stejným způsobem za využití `get_local_id(1)` identifikuji pixely na horní a spodní hraně



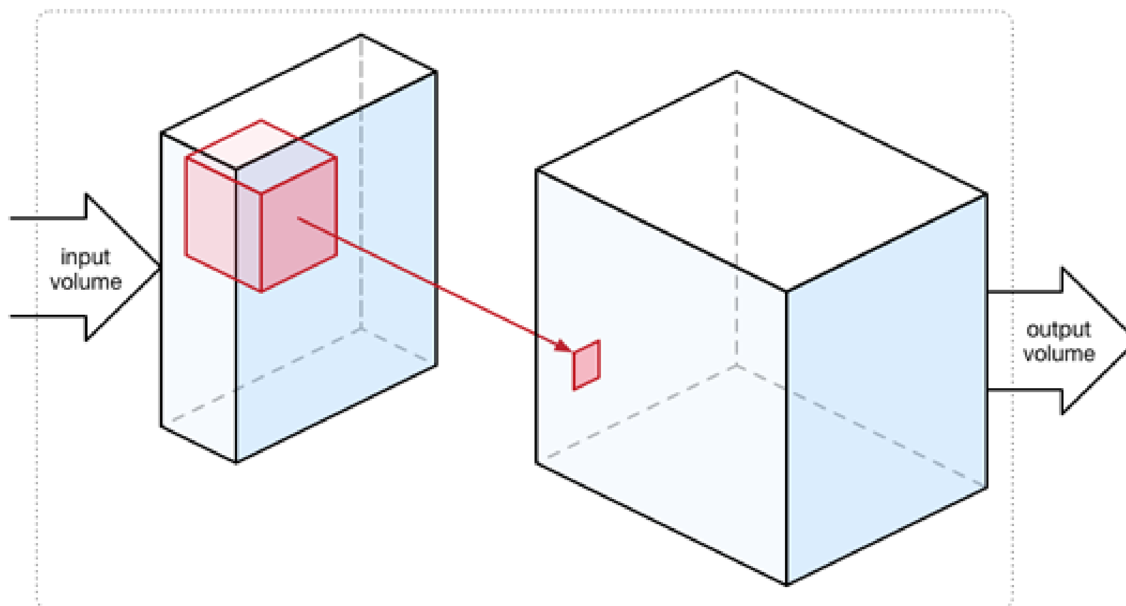
Obrázek 5.3: Grafické znázornění souřadnic.

zpracovávaného bloku a opakuji výše zmíněný postup. Následuje nalezení všech rohových pixelů. Pomocí těchto pixelů dále načítám jejich bezprostřední okolí, aby i tyto pixely bylo možné zpracovat. Jako poslední nahraji pixely ze sousední horní a spodní vrstvy 3D obrazu.

Všechna vlákna, která již skončila s načítáním svých pixelů, je potřeba zastavit, aby nedošlo ke zpracovávání některých pixelů, zatímco ostatní pixely jsou teprve ve fázi načítání. K synchronizaci vláken jsem proto využil paměťové bariéry, kterou OpenCL poskytuje. Tato bariéra zajistí, aby všechna vlákna pokračovala ve výpočtech až poté, co všechna ostatní vlákna skončí s načítáním pixelů z globální paměti do lokální paměti.

Touto logikou tedy lze zajistit, aby všechna potřebná data byla správně nahrána do lokálních pamětí a připravena ke zpracování.

Posledním krokem, který je potřeba provést, je samotná konvoluce. Kernel tedy aplikuje konvoluční masku na obrazové bloky v lokální paměti, které poté poskládá zpátky dohromady. Výsledkem je celistvý filtrovaný obraz, který je zapsán zpátky do globální paměti, kde je k dispozici host systému. Po aplikaci konvoluční masky a nahrání obrázku do výstupního bufferu se kernel ukončí.



Obrázek 5.4: Zpracování jednoho pixelu v obrazovém bloku[1].

5.5 Výsledek konvoluce

Konečnou fází mého návrhu je zpracování výsledku konvoluce. Toto provádí poslední modul z obrázku 5.1. Tento modul provádí následující kroky:

1. Načtení zpracovaného obrazu z výstupního bufferu
2. Úprava velikosti obrazu do originální velikosti
3. Nahrání obrazu na výstupní kanál
4. Uvolnění všech vytvořených zdrojů

V této fázi tedy dojde k načtení zpracovaného obrazu z výstupního bufferu do před-připraveného pole, alokovaného v sekci 5.3. Z obrazu je odstraněna vrstva neutrálních pixelů, které byly do obrazu přidány kvůli zamezení čtení mimo na-alokované pole (toto je blíže popsáno v sekci 5.3). Dále je oříznut o rozšířené pixely, které bylo nutné přidat pro zarovnání obrazu na jednotlivé výpočetní bloky, čímž se zpracovávaný obraz vrátí na své původní rozměry. Filtrovaný obraz je následně nahrán na výstupní kanál. Jako poslední, modul uvolní všechny na-alokované zdroje a aplikace končí.

[1]Zdroj: http://machinethink.net/blog/convolutional-neural-networks-on-the-iphone-with-vggnet/

Kapitola 6

Implementace formou modulu pro knihovnu VPL

V této kapitole popíši implementaci mého návrhu, při které jsem využil již existující knihovny VPL[1]. Také zde zmíním výsledné moduly. Nakonec okomentuji, jakým způsobem jsem implementoval měření rychlosti vytvořeného algoritmu. Výsledky tohoto měření jsou znázorněny v kapitole 7.

6.1 Použitá knihovna

Jako základ mojí práce jsem si vybral knihovnu VPL, jejímž autorem je Michal Španěl. Tato medicínsky orientovaná knihovna umožňuje načítat a zpracovávat 2D a 3D obrazy (využívá pouze CPU), především obrazová data získána pomocí CT skenů. Jedná se o Open Source přenositelnou knihovnu implementovanou v C++, která dokáže načítat data typů jako jsou například:

- JPEG
- PNG
- DICOM

Dále obsahuje implementaci, díky které je možné prezentovat zpracovaný obraz, což ulehčilo při tvorbě mého návrhu a postupném odstraňování logických chyb. Knihovna VPL je složena z desítek jednotlivých modulů, které si mezi sebou data podle potřeby přeposílají (s využitím rour – takzvané „Pipelines“). Jednotlivé moduly jsou implementovány pomocí C++ šablon – jejichž výhodou je rychlost a škálovatelnost, což umožňuje načítat a zpracovávat více datových typů (některé byly zmíněny výše). Použití těchto modulů je demonstrované v útržku 6.1.

Listing 6.1: Použití modulů obsažených v knihovně VPL

```
vplLoadDicom <data/dicom/80.dcm \
| vplSliceFilter -filter gauss -sigma 1.0 \
| vplSliceRange -auto
| vplSliceView
```

[1]Zdroj: https://bitbucket.org/3dimlab/vpl

Aby bylo možné použít knihovnu VPL, je potřeba obstarat a zkompilovat externí knihovny. Pro základní funkčnost VPL knihovny jsou potřeba následující externí knihovny:

- Zlib – Knihovna pro kompresi a dekompresi dat.
- Libpng – Tato knihovna umožňuje načítat obrazy formátu PNG.
- JPEG knihovna – Obdobně jako Libpng akorát pro formát JPEG.
- Eigen – Knihovna zaměřená na lineární algebru, maticové a vektorové operace.
- OpenGL + GLUT – Knihovny umožňující vizualizaci výsledného obrazu.

Knihovna VPL využívá CMake pro udržení jednoduchého, částečně automatizovaného překladu programu podle stanovených parametrů. Program CMake pomocí všech externích knihoven vygeneruje (například) Visual Studio projekt (nebo také „Solution“), který lze poté snadno pomocí Visual Studia přeložit.

Knihovnu VPL jsem si vybral pro její schopnost načítat 2D a 3D obrazová data a následně prezentovat filtrovaný obraz.

6.2 Inicializace OpenCL a její parametry

Po zpracování a přípravě vstupního obrazu (sekce 5.3) následuje inicializace OpenCL. Toto obstarává v pořadí druhý modul, podle obrázku 5.1. Tento modul se stará o kompletní inicializaci a veškerý CPU overhead, který ve spojením s používání OpenCL vzniká.

Před samotným spuštěním kernelu je zapotřebí ověřit, zdali hostující systém má k dispozici požadované zařízení. Pokud aplikace zjistí, že takové zařízení nelze nalézt, vypíše příslušnou chybovou hlášku a program se ukončí. Následně dojde k vytvoření kontextu programu, a fronty příkazů, do kterých poté budou umístěny kernely nachystané ke zpracování. Podle zvoleného přepínače (tedy podle filtru, který chceme aplikovat) se načte příslušný kód daného kernelu do připravené proměnné. Poté dojde k vytvoření samotného OpenCL programu, který načte svůj kód právě z nachystané proměnné obsahující zvolený algoritmus. Tento algoritmus je podrobně popsán v sekci 5.4. Aplikace se pokusí nově sestavený program přeložit. V případě neúspěchu vrátí příslušnou chybovou hodnotu. Vzhledem ke skutečnosti, že OpenCL program se překládá až za běhu aplikace, je potřeba zkontrolovat navrácenou hodnotu funkce, která program překládá. V případě neúspěchu je nutné reagovat na nastalou situaci a zastavit běh programu. Po smazání všech na-alokovaných zdrojů se aplikace ukončí, a vrátí příslušný chybový kód.

Dalším krokem je příprava všech potřebných zdrojů na grafické kartě. V sekci 5.3 jsem popsal zpracování vstupního obrazu a jeho zapsání do alokovaných polí. Tyto pole jsou uloženy v paměti RAM (Random Access Memory), tedy data jsou stále na straně hosta. Aby bylo možné data převést na grafickou kartu, je potřeba vytvořit datový buffer, který poskytuje knihovna OpenCL. Stejně jako bylo zapotřebí na-alokovat dvě pole – jedno pro vstupní data a druhé pro výsledek konvoluce, je také potřeba vytvořit dva datové buffery, do kterých budou později přenesena obrazová data z paměti hosta. Velikost těchto datových bufferů bude shodná se zmíněnými maticemi, tedy hodnota rovná násobku všech rozměrů rozšířeného obrazu. Také je potřeba nachystat tyto buffery i pro všechny ostatní hodnoty, které je potřeba předat na grafické zařízení, aby byla zajištěna správnost běhu aplikace.

Do těchto bufferů, které jsem si připravil výše, nyní nakopíruji všechna data, která budu předávat na grafickou kartu ke zpracování, včetně všech dalších nezbytných hodnot.

Tyto hodnoty uchovávají informace o původní, nezměněné velikosti obrázku, a také hodnoty obsahující rozměry rozšířeného obrazu.

V tuto chvíli poslední co zbývá, je nastavit zvolenému kernelu argumenty a zařadit jej do fronty s požadovaným nastavením. Jako argumenty jsou kernelu předány:

- Ukazatel na naplněný datový buffer obsahující 3D obrazová data nachystaná ke zpracování
- Ukazatel na prázdný, před-chystaný datový buffer o stejné velikosti jako buffer zdrojový, do kterého bude uložen výsledek konvoluce
- Informaci o původní, nezměněné velikosti zpracovávaného obrazu
- Informaci o upravené, rozšířené velikosti zpracovávaného obrazu

Jako poslední krok je potřeba zařadit požadovaný kernel do pracovní fronty, čímž jej předáme grafickému čipu ke zpracování. Aby bylo možné kernel zařadit do fronty ke zpracování, je zapotřebí stanovit, za jakých podmínek se má kernel zpracovat. K tomuto účelu lze využít funkci poskytovanou knihovnou OpenCL nesoucí název `clEnqueueNDRangeKernel()`. Pro tento projekt je zajímavých prvních šest možností, zbylé tři jsou nepodstatné, a tedy se jimi zde nebudu zabývat. Zmíněných šest možností, které budou specifikovány jsou:

1. Příkazová (nebo také pracovní) fronta, do které chceme kernel zařadit ke zpracování
2. Kernel, který jsme si zvolili pro vykonání na základě zvoleného filtru
3. Počet dimenzí, ve kterých se kernel spustí
4. Offset (česky odsazení), se kterým se výpočet spustí – tento ofset říká, kolik prvních položek v poli kernel při svém spuštění přeskočí
5. Počet vláken (work items), kterých bude společně s kernelem spuštěno
6. Velikost pracovních skupin (work groups), do kterých budou spuštěná vlákna rozdělena

Položky číslo 3, 5 a 6 z tohoto seznamu budou podrobněji popsány níže. Praktická ukázka kódu, který odesílá kernel ke zpracování je demonstrována na útržku algoritmu 1.

Algorithm 1: Útržek formou pseudokódu pro spuštění kernelu

```
1 pocetVlaken3D[3] = XSize, YSize, ZSize ;
2 velikost3DSkupin[3] = TILEXSIZE, TILEYSIZE, TILEZSIZE ;
3 dimenze = 3;
4 errCode = clEnqueueNDRangeKernel(prikazovaFronta, kernel, dimenze,
    KERNELOFFSET, pocetVlaken3D, velikost3DSkupin, 0, NULL, NULL);
```

Pro jednoduchost implementace můj návrh využívá možnosti spouštět kernely s dimenzím parametrem. Tento parametr umožní lokalizovat jednotlivé vlákna pomocí jejich pozic v prostoru. Vzhledem k faktu, že se práce zaobírá zpracováváním 3D dat, je tento parametr nastaven na trojku (tedy tři dimenze). Tato výhoda a její využití je blíže popsáno v sekci 5.4.

Posledním dvěma důležitým parametrů, kterým se budu věnovat jsou parametry, kterými lze stanovit počet vláken, která se spustí společně s kernelem a jejich rozdělení do pracovních skupin. Tento návrh pracuje s modelem, kdy jedno vlákno zpracovává právě jeden pixel. Počet spuštěných vláken tedy odpovídá násobku všech rozměrů rozšířeného obrázku – tedy hodnota rovná velikosti alokované matice. Velikost jednoho pracovního bloku je pak rovna velikosti bloků, do kterých je obrázek rozdělen. Tato logika je detailně popsána v sekci 5.4.

V tento okamžik jsou všechny přípravy dokončeny a je možné spustit požadovaný kernel, jenž je popsán v sekci 5.4.

6.3 Výsledné moduly

Výsledkem mojí práce, která využívá knihovnu VPL, jsou dva nové moduly přidané do této knihovny. Ty jsem úspěšně zakomponoval do již existujícího řešení pomocí programu CMake. Výsledné moduly jsou:

- **VPLSliceFilterGPU** – Tento modul vznikl za účelem experimentování s 2D řešením problému s využitím GPU, a posloužil jako vzor pro následující tvorbu 3D návrhu mého modelu. V rámci experimentování, tento modul obsahuje různé varianty Sobelova operátoru, jenž jsou implementovány pomocí široké škály metod. Jmenovitě modul obsahuje implementaci:

  - Detekce hran podle osy X
  - Detekce hran podle osy Y
  - Kombinovaná detekce os X a Y zároveň

  Při implementaci výše zmíněných filtrů jsem také experimentoval s provedením samotné implementace a její postupné optimalizace. V rámci tohoto procesu jsem dané filtry implementoval:

  - Pomocí prostého „naivního“ modelu. Tato implementace využívá pouze globální paměť a s obrazem pracuje jako s jedním velkým blokem dat – výpočet pouze v rámci jedné výpočetní skupiny. Filtr je zde implementován neseparabilně.
  - Optimalizované řešení naivní implementace. Tato implementace obraz rozděluje to jednotlivých obrazových bloků. Jednotlivé bloky zpracovává právě jedna výpočetní skupina. Obrazové bloky jsou nahrány do lokálních pamětí těchto skupin.
  - Variantu využívající separabilní vlastnosti Sobelova filtru – konvoluční kernel je implementován separabilně.

- **VPLVolumeFilterGPU** – Jak jsem již několikrát zmínil, hlavním cílem mé práce byla implementace Sobelova filtru pro 3D data pomocí GPU. Výsledek práce je implementován právě v tomto modulu. Konkrétně jsem tedy implementoval tyto konvoluční kernely:

  - Detekci hran podle osy X
  - Detekci hran podle osy Y
  - Detekci hran podle osy Z

Všechny výše zmíněné kernely jsou optimalizované a rozdělují obraz do bloků, nad kterými pak pracují jednotlivé výpočetní skupiny. Bloky jsou uloženy do lokálních pamětí jednotlivých výpočetních skupin. Tento modul také obsahuje experimentální, separabilní implementaci Sobelova filtru optimalizovaného pomocí lokální paměti.

Vzhledem k tomu, že modul pracuje s medicínskými daty (konkrétně CT sken), jsem se rozhodl využívat datová pole typu `float`. Typ `float` jsem zvolil proto, že umožňuje načítat širokou škálu různých obrazových dat (Obraz CT skenu například využívá Hounsfieldovu stupnici − její hodnoty se pohybují v rozsahu od mínus do plus několika tisíc [2]).

Během experimentální a testovací fáze jsem v tomto modulu objevil jeden drobný nedostatek. Několik málo výsledných pixelů (převážné v rohových oblastech jednotlivých obrazových bloků) obsahuje mírně zkreslené hodnoty (odchylka je v řádech setin procenta).

6.4 Měření rychlosti aplikace

Pro experimentování a získávání časových údajů jsem na potřebné pozice implementovat časové měření za využití C++ knihovny *chrono*. Díky tomu jsem byl schopný získat a zpracovat potřebná časová data, znázorňující:

- Celkový CPU overhead potřebný ke připravě a spuštění jednotlivých kernelů
- Doba potřebná k naplnění pamětí daty
- Samotný čas potřebný k provedení konvoluce − čitě OpenCL kernel
- Celkový čas běhu OpenCL modulu

Výsledky tohoto měření jsou prezentovány v kapitole 7.

[2] Zdroj: https://en.wikipedia.org/wiki/Hounsfield_scale

Kapitola 7

Experimentování a výsledky

V této kapitole jsou prezentovány všechny časové hodnoty získané pomocí jednotlivých měření a experimentů. Tyto hodnoty jsou znázorněny v grafech tak, aby bylo vidět případné zlepšení/zhoršení výpočetního času mezi jednotlivými implementacemi. Testovací metodologie použitá pro získání těchto časových údajů je popsána v sekci 7.1. Shrnutí a detailnější zhodnocení dosažených výsledků lze nalézt v sekci 7.5.

7.1 Testovací metodologie

Pro získání naměřených časů byla použita implementace Sobelova filtru podle osy X. Testovací data byla 2D i 3D rozměrů, aby bylo možné znázornit účinnost této implementace právě pro 3D data v porovnání s 2D daty. Všechny testy proběhly desetkrát, a nad touto množinou výsledků byl proveden aritmetický průměr. Pro 2D testování byly použity obrazy o velikostech (Šířka x Výška obrazu v pixelech):

– 100 x 45

– 1920 x 1080

– 3840 x 2140

Pro 3D testování byla použita data o rozměrech (Šířka x Výška x Hloubka obrazu v pixelech):

– 256 x 256 x 361

– 512 x 512 x 361

– 1024 x 1024 x 309

Jednotlivé testy byly provedeny na notebooku, jehož parametry jsou následující:

CPU: Intel Core i7-3610QM

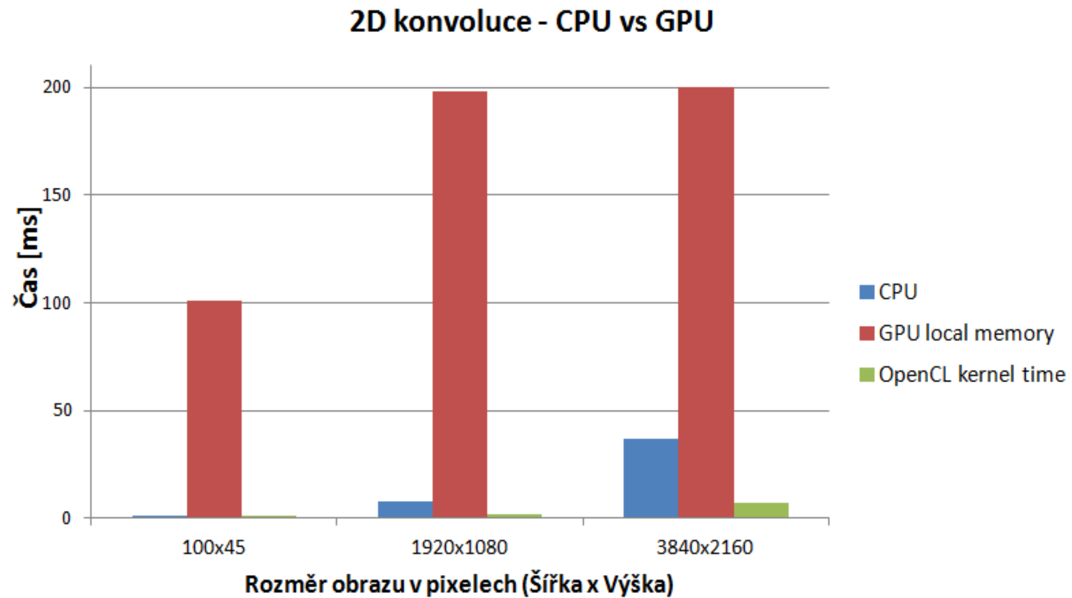
GPU: Nvidia GeForce GTX 670M/3GB GDDR5

RAM: 8GB DDR3

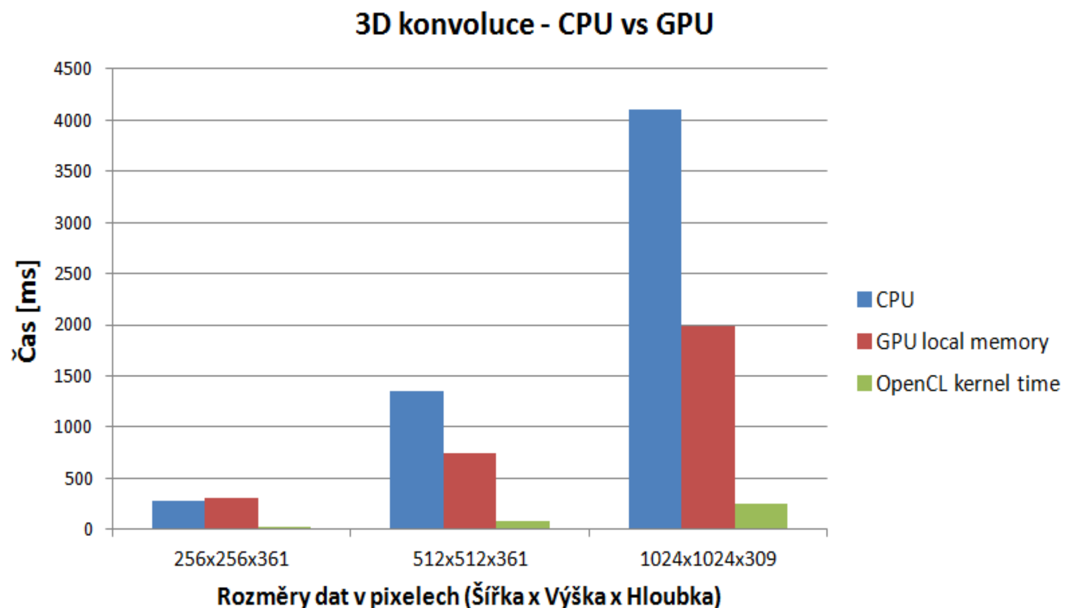
OS: 64-bit Windows 7 SP1

7.2 CPU vs GPU

V této sekci jsou znázorněny výsledky porovnávající 2D a 3D implementace mezi CPU a neseparabilní variantou Sobelova filtru optimalizované pomocí lokálních pamětí za využití GPU. Na obrázku 7.1 je vidět, že implementace 2D konvoluce za využití OpenCL se nevyplatí, neboť CPU overhead spojený s tímto procesem je několikanásobně větší, než samotná doba výpočtu na CPU (i přesto, že samotná doba výpočtu OpenCL kernelu je velmi malá). Akcelerace těchto výpočtů na GPU se vyplatí až u většího množství 3D dat, což znázorněno na obrázku 7.2.



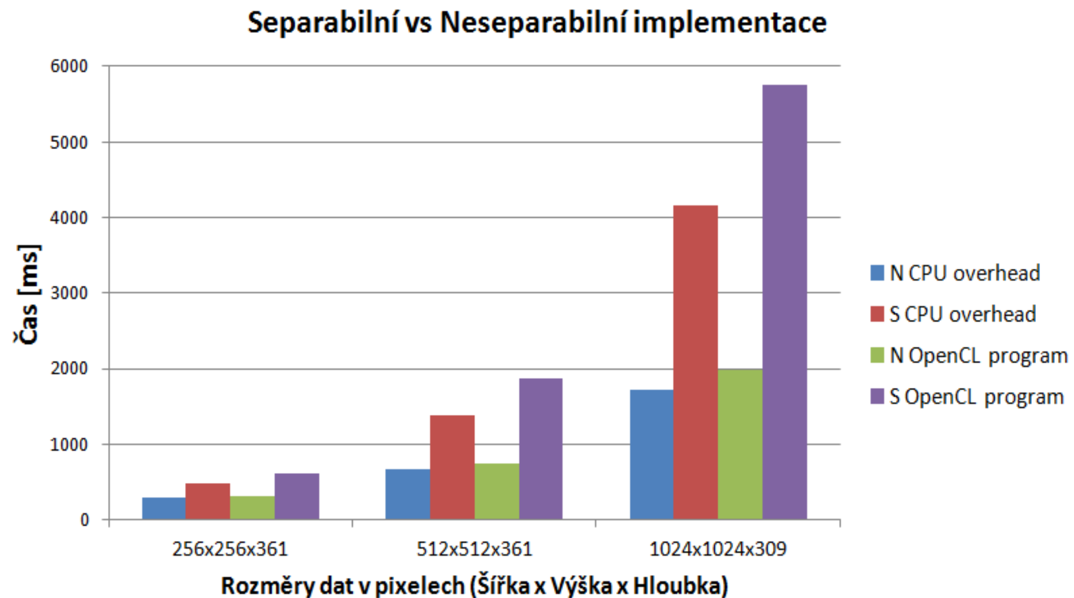
Obrázek 7.1: 2D konvoluce - CPU vs GPU.



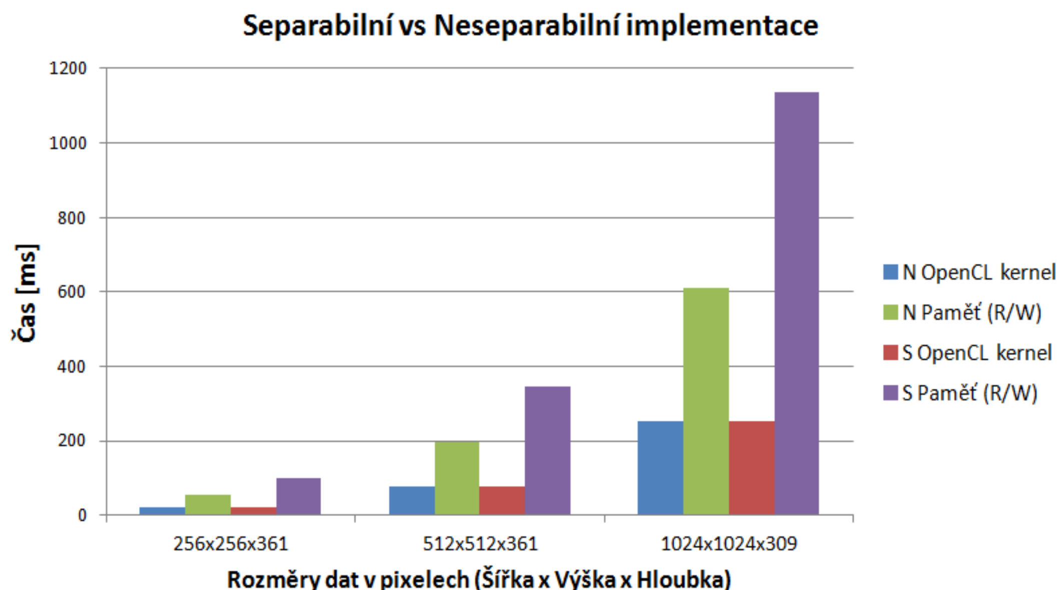
Obrázek 7.2: 3D konvoluce - CPU vs GPU.

7.3 Separabilní vs Neseparabilní implementace

Tato sekce obsahuje výsledky, jenž porovnávají separabilní (S) a neseparabilní (N) implementaci 3x3x3 filtru. Obě varianty využívají lokální paměť. Jak je vidět na obrázku 7.3 využití separabilní implementace takto malého filtru není zrovna výhodné (OpenCL program - celková doba běhu OpenCL části programu). Proto je v tomto případě lepší použít neseparabilní implementaci a separabilní variantu použít při větším rozměru konvolučního filtru, například 11x11x11. Na obrázku 7.4 je znázorněn celkový čas potřebný pro provedení výpočtů a doba nezbytná pro práci s pamětí.



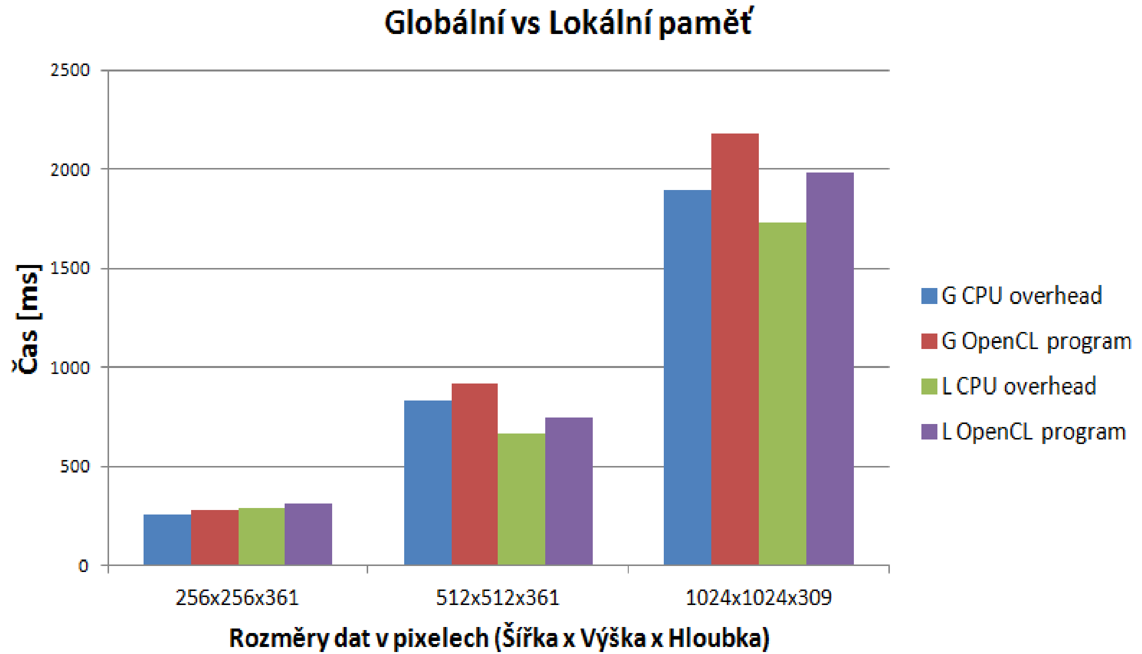
Obrázek 7.3: Porovnání časů pro celkový běh programu a CPU overhead.



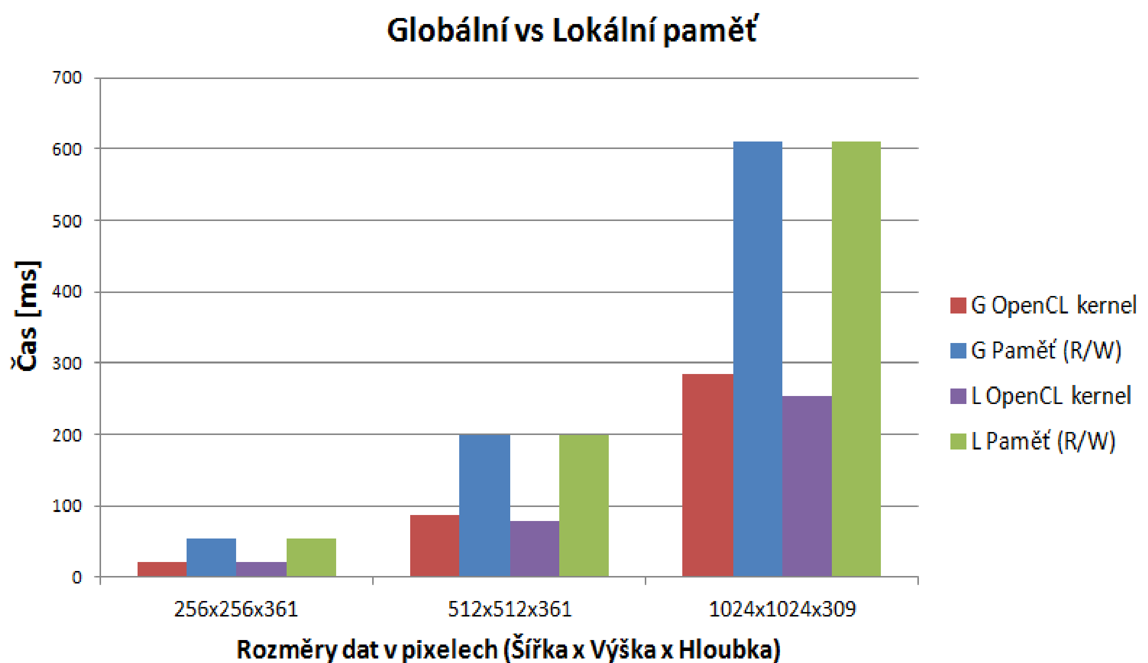
Obrázek 7.4: Porovnání časů mezi kernely a čtením a zápisem do pamětí grafické karty.

7.4 Optimalizace pomocí lokální paměti

V následujících obrázcích 7.5 a 7.6 je znázorněn časový rozdíl, získaný optimalizací „naivní“ implementace (která využívá globální paměť), pomocí lokální paměti.



Obrázek 7.5: Implementace pomocí globální vs lokální paměti.



Obrázek 7.6: Implementace pomocí globální vs lokální paměti.

7.5 Zhodnocení výsledků

V sekci 7.2 jsem znázornil na grafech časový rozdíl mezi implementací na CPU a GPU pro 2D i 3D obrazová data. Na obrázku 7.1 je vidět, že akcelerace výpočtů na GPU pro 2D data není ideální neboť CPU řešení je v takovémto případě rychlejší. Samotný Výpočet na GPU je sice v řádech několika milisekund, nicméně přítomný CPU overhead (neboli režije procesoru při inicializaci a řízení OpenCL části aplikace) značně přesahuje dobu potřebnou pro dokončení výpočtů pouze pomocí CPU. Tím je tedy OpenCL varianta nevhodná pro použití na 2D obrazová data. Síla tohoto návrhu nastává až při zpracovávání většího množství 3D obrazových dat (obrázek 7.2), kdy exponenciálně vrůstá časová náročnost těchto výpočtů. CPU overhead společně se samotným výpočtem (který je hotov za několik stovek milisekund) je výrazně rychlejší, než stejná implementace Sobelova filtru pouze za využití CPU.

Dále jsem v sekci 7.3 porovnal rozdíl mezi separabilní a neseparabilní implementací Sobelova filtru o velikosti konvolučního jádra 3x3x3. Ačkoliv separabilní implementace je pouze experimentální, na grafech v obrázcích 7.3 a 7.4 lze pozorovat značné prodloužení běhu programu (OpenCL program) při použití separabilní implementace zvoleného filtru. To je způsobeno dodatečnou režií, která vznikla rozdělením konvolučního jádra do jednotlivých 1D složek. Ačkoliv doba běhu kernelu zůstává stejná, doba potřebná pro manipulaci s daty v paměti se zvýšila. Separabilní implementace je tedy vhodná pouze u konvolučních kernelu vyšších rozměrů. Zároveň by bylo vhodné provést dodatečné optimalizace této varianty.

Jako poslední jsem porovnal naměřené časy při použití lokální paměti vůči globální paměti grafické karty (sekce 7.4). Naměřená data jsou zobrazena na obrazu 7.5, který porovnává CPU overhead a celkovou dobu běhu OpenCL programu, a 7.6, jenž znázorňuje časové rozdíly mezi dobou výpočtu a časem stráveným paměťovými operacemi. Je možné pozorovat mírné zrychlení výpočtu (OpenCL kernel), kdežto paměťové operace vyžadují stejné množství času. Zároveň při použití lokálních pamětí klesá CPU overhead čímž dochází k dalšímu zrychlení běhu programu (OpenCL program).

Kapitola 8

Závěr

V této práci jsem prezentoval možný přístup pro návrh a implementaci knihovny, která by s pomocí platformy OpenCL umožnila akcelerovat zpracování velkého množství 3D obrazových dat pomocí výpočetního výkonu grafické karety. Cíl práce byl tedy úspěšně splněn, neboť výsledné řešení urychlilo výpočet o více než 100% pro data o rozměru 1024x1024x309. Zrychlení tohoto algoritmu by se dále projevilo při stoupajícím rozlišení a objemu zpracovávaných dat, kdy by výpočet za použití CPU dosahoval několika hodin. Návrh používá model „jeden pixel – jedno vlákno“, kdy jedno vlákno programu OpenCL zpracovává právě jeden pixel obrazových dat.

Tento návrh je zaměřen primárně na Sobelův filtr o velikosti 3x3x3, a proto jedním z dalších možných směrů rozvoje by mohla být implementace generického řešení pro výpočet různých konvolučních filtrů, kdy by bylo možné specifikovat rozměry zadaného filtru.

Jako poslední dvě možnosti budoucí práce zde uvedu implementaci algoritmu rychlé furierovy transformace pro diskrétní furierovu transformaci, která by umožnila další možnosti zpracovávání 3D obrazových dat. Také je možné vytvořit řešení, které by umožnilo zpracovávat data, jejichž velikost přesahuje velikost globální paměti použitého grafického adaptéru.

Literatura

[1] Aqrawi, A. A.: *Three Dimensional Convolution of Large Data Sets on Modern GPUs*. Diplomová práce, Norwegian University of Science and Technology, Trondheim, 2009.

[2] David Svoboda, P. K.: Convolution of Large 3D Images on GPU and its Decomposition. *EURASIP Journal on Advances in Signal Processing*, 2011, [Online; cit. 2017-04-22].
URL https://is.muni.cz/th/106808/fi_r/eurasip-double.pdf

[3] Doc. Ing. Zdeněk Hrdina, C.: *Signály a soustavy*. Praha : ČVUT., 2003, kapitola 1, s. 7-28.

[4] Hinum, K.: NVIDIA GeForce GTX 670M. [Online; cit. 2017-04-21].
URL https://www.notebookcheck.net/NVIDIA-GeForce-GTX-670M.72197.0.html

[5] Konvoluce: Convolution. 2017, [Online; cit. 2017-04-21].
URL https://en.wikipedia.org/wiki/Convolution

[6] Munshi, A.: Technická zpráva. [Online; cit. 2017-04-20].
URL https://www.khronos.org/registry/OpenCL/specs/opencl-1.2.pdf

[7] Youlian Zhu, C. H.: An Improved Median Filtering Algorithm for Image Noise Reduction. *Physics Procedia*, ročník 25, č. 1, 2012: s. 609–616, [Online; cit. 2017-04-22].
URL http://www.sciencedirect.com/science/article/pii/S1875389212005494

Přílohy

Příloha A

Obsah CD

- Zdrojové kódy včetně upravených Cmake souborů lze nalézt ve složce `\VPLimplementation\`
- Potřebné externí knihovny pro Windows 7 - Visual Studio 2015 v adresáři `\3rdParty\`
- Přeložené spustitelné soubory (včetně potřebných knihoven) pro demonstraci funkčnosti společně s demonstračním skriptem v adresáři `\demo\`
- Tato práce ve formátu PDF v adresáři `\thesis\`
- Zdrojové soubory pro pro tuto práci ve formátu LaTeXv adresáři `\thesis\latex\`
- Plakát k práci ve formátu PDF v adresáři `\poster\`
- README – Soubor obsahující návod na zprovoznění požadované knihovny a této práce + popis obsahu přiloženého CD