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 **RADOSLAV KOZOVSKÝ**
AUTHOR

VEDOUCÍ PRÁCE **ING. MICHAL ŠPANĚL, PH.D**
SUPERVISOR

BRNO 2017

Abstrakt

Cílem této práce je akcelerace výpočtu zvolených 2D a 3D obrazových filtrů pomocí GPU s využitím OpenCL. Konkrétně se zabývá implementací a porovnáním různých variant Sobelova hranového detektoru a Gaussova filtru s využitím globální nebo lokální paměti.
Podařilo se dosáhnout zrychlení na 3D variantách filtrů. Na 2D variantách filtrů byla režie pro přenos dat do a z GPU příliš vysoká.

Abstract

The aim of this work is to accelerate the calculation of selected 2D and 3D image filters using GPU using OpenCL. Specifically, it deals with the implementation and comparison of various variants of Sobel's edge detector and Gauss filter using global or local memory.
Acceleration has been achieved on 3D filter variants. On 2D filter variants, the overhead for data transfer to and from the GPU was too high.

Klíčová slova

NVIDIA CUDA, OpenCL, Sobelův hranový filtr, Gaussův vyhlazovací filtr, konvoluce, DICOM, volumetrická data

Keywords

NVIDIA CUDA, OpenCL, Sobel edge filter, Gaussian blur filter, convolution, DICOM, volumetric data

Citace

KOZOVSKÝ, Radoslav. *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
Ing. Michala Španěla, Ph.D
Další informace mi poskytl Ing. Matúš Kozovský
Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……………………
Radoslav Kozovský
11.4.2017

Poděkování

Ďakujem vedúcemu práce Ing. Michalovi Španělovi, Ph.D za vedenie, rady a korekcie mojej bakalárksej práce. Ďakujem svojej rodine a priateľom za podporu počas vypracovávania tejto práce.

© Radoslav Kozovský, 2017
Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Obsah

1 Úvod.....5
2 Konvolúcia a obrazové filtre.....6
2.1 Obrazové filtre.....6
2.2 Konvolúcia.....6
2.3 Separabilná konvolúcia.....7
2.4 Objemové dáta.....7
2.5 Použité filtre.....9
3 Prístupy k akcelerácii obrazových filtrov na GPU.....13
3.1 Naivná implementácia.....13
3.2 Použitie konštantnej pamäte.....13
3.3 Použitie vektorových dátových typov.....13
3.4 Rozvinuté cykly.....13
3.5 Použitie lokálnej pamäte.....14
4 Akcelerácia výpočtov pomocou GPGPU a OpenCL.....15
4.1 NVIDIA CUDA.....16
4.2 OpenCL.....24
5 Návrh riešenia filtrácie 3D skenov s pomocou GPU.....28
6 Implementácia.....29
6.1 2D grafické filtre.....29
6.2 3D grafické filtre.....32
6.3 Použitie lokálnej pamäte.....34
7 Praktické testy grafických filtrov a dosiahnuté výsledky.....35
7.1 Testovanie a meranie.....35
7.2 Dosiahnuté výsledky.....37
7.3 Testovanie 2D Gaussovho filtra.....39
7.4 Testovanie 3D Gaussovho filtru.....41
7.5 Testovanie hranového filtru Sobel.....44
8 Záver.....46
Literatúra.....47
Zoznam príloh.....48
Príloha 1. DVD.....49

1 Úvod

Medicínske CT dáta sa od klasického röntgenového snímku líšia tým, že namerané hodnoty je možné reprezentovať v 3D priestore, zatiaľ čo röntgenová snímka vytvorí 2D obrázok. Keďže ide o meranie tak ako v každom meraní dochádza k šumu vo výsledných meraných dátach. Pre potlačenie šumu je možné použiť grafické filtre, avšak tieto sú pomerne náročné na spracovanie z dôvodu veľkého objemu spracovávaných dát. Klasický procesor má síce veľký výpočtový výkon, ale pomerne malý počet procesorov. GPU grafickej karty má veľké množstvo procesorov s inštrukciami na výpočet s vysokou presnosťou určenými hlavne na spracovanie obrazu. Preto sa zameriavam na filtráciu 3D medicínskych dát a 2D obrázkových dát za pomoci surového výkonu grafickej karty s mnohými výpočtovými jadrami. To pri správnej implementácii filtrov umožňuje vyfiltrovať vstupné dáta za zlomok času potrebného pri filtrácii cez CPU.
Spracovanie medicínskych obrazov je jedným s prvých odvetví ktoré využili výhody GPU výpočtu na urýchlenie.

Cieľom práce bolo naštudovať a overiť možnosti akcelerácie spracovania 2D a 3D obrazových dát s použitím GPGPU. Počas riešenia bolo overených viacero prístupov ktoré sú štandardne používané. Na testovanie boli použité známe filtre s jednoduchým postupom spracovania.

Namerané výsledky boli uložené do tabuliek a grafov. Z grafov je vidieť nárast času výpočtu podľa objemu spracovávaných dát.

Na základe zistených údajov je vhodné používať GPU akceleráciu pre zložité filtre a obrázky s veľkým množstvom dát. Pri jednoduchších filtroch, alebo malých obrázkoch s malým rozlíšením môže byť celkový čas spracovania spolu s potrebnou réžiou na GPU dlhší.

2 Konvolúcia a obrazové filtre

V dnešnej dobe sú často spracovávané rôzne obrazové dáta, nie len pre medicínske účely, ale taktiež sa s týmto spracovaním stretávame v grafických alebo kancelárskych programoch, ktoré sú bežne používané, či už ide o rotáciu obrázku (násobenie maticou rotácie), zvýraznenie farieb alebo potlačenie šumu vo fotografii.

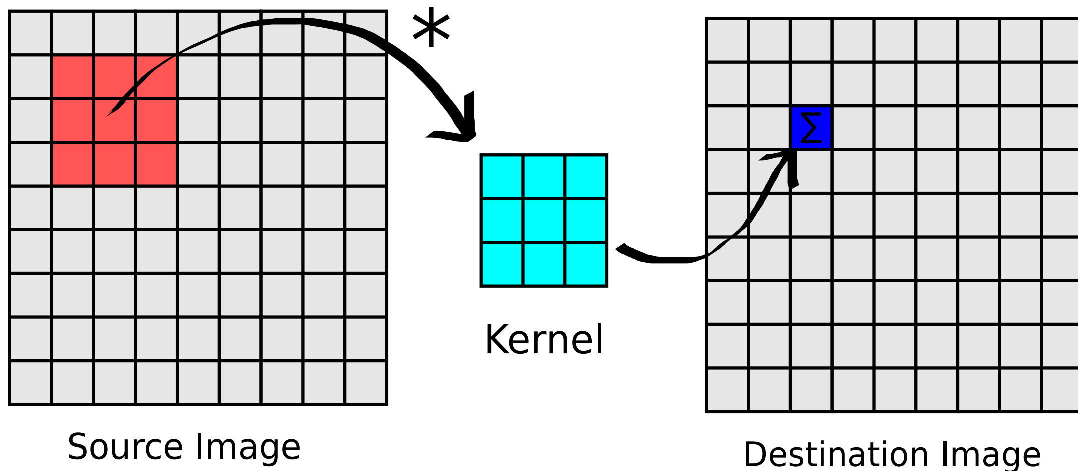
2.1 Obrazové filtre

Spracovanie obrazu: Rastrový obraz možno spracovávať mnohými rôznymi metódami s ohľadom na výsledný požadovaný efekt. Techniky pomocou ktorých sú v počítači upravované obrázky sa nazývajú spracovanie obrazu (image processing). Pri spracovaní obrazu sú existujúce obrazy prevedené do číslicovej podoby, a následne sa zobrazujú v grafickej forme na displeji. Po prevode obrazu do číslicovej reprezentácie je možné dáta ďalej spracovávať. Napríklad vyhľadávať informácie v obraze, farebne odlíšiť segmenty obrazu na základe vstupných podmienok, zmeniť jas, ostrosť, farebnosť a mnoho ďalšieho. V lekárstve sa spracovávanie obrazu používa pri skvalitnení a zvýraznení fotografických snímkov. Tiež sa využíva v tomografii, v ultrazvukových a rádioizotopových vyšetreniach [1].

Pri filtrovaní sa s každým bodom obrazu vykonáva pomerne jednoduchá matematická operácia. Náročnosť vzniká veľkým množstvom údajov pri vysokom rozlíšení obrazu a veľkom počte spracovávaných bodov. Výhodou je jednoduché rozdelenie na paralelné spracovanie, keď výpočty môžu prebiehať súčasne na všetkých procesoroch grafickej karty. Nevýhodou sú časy potrebné na presun údajov do grafickej karty a späť.

2.2 Konvolúcia

Konvolúcia je operácia s maticou pomocou inej matice, takzvaného jadra (kernel). Ako prvá matica sa použije obraz určený na úpravu. Obraz je dvojrozmerná pravouhlá súradnicová sústava pixlov. Použité jadro závisí na požadovanom efekte. Filter postupne spracováva všetky body obrázku. Každý z nich vynásobí hodnotu aktuálneho pixlu a jeho osem susedných pixelov (pre kernel 3x3) odpovedajúcimi hodnotami z jadra. Výsledné hodnoty sa sčítajú a výsledok je potom hodnotou priradenou aktuálnemu pixlu. Konvolúcia sa môže aplikovať aj na väčšie okolie bodov. Pomocou dvojrozmerných konvolučných filtrov je možné vykonávať zaostrenie obrazu, rozmazávanie, zvýrazňovanie hrán alebo tvorbu reliéfu, a iné [1]. Princíp konvolúcie je zobrazený na obrázku 1.



Obrázok 1: princíp konvolúcie.

2.3 Separabilná konvolúcia

Pri separabilnej konvolúcii je možné výpočty rozdeliť a realizovať postupne. Napríklad 2D filter je možné prerátať najprv po riadkoch a následne po stĺpcoch (Vzorec 1.1). Takto je možné spracovať napríklad Gaussov filter, Sobelov filter a iné.

Pri neseparabilnej konvolúcii je potrebné dodržať plošný výpočet bez možnosti rozdelenia.

$$g(x,y)=g_1(x)g_2(y) \tag{1.1}$$

2.4 Objemové dáta

Objemové dáta bývajú uložené v rôznych formátoch. Štandardné formáty používajú kartézsku mriežku, kde každý bod mriežky popisuje vlastnosti dát. Tento bod sa označuje voxel (volumetric pixel). Ďalšími možnými usporiadaniami dát sú regulárne a rovnobežné mriežky. Špeciálnym prípadom sú neštrukturované mriežky, ktoré sú opísané štvorstenmi pomocou špeciálnych dátových

štruktúr. Vlastnosti, ktoré objemové dáta reprezentujú závisia hlavne od spôsobu ich získavania. Najčastejšie sa objemové dáta získavajú v medicíne, kde ich produkujú snímacie zariadenia ako počítačová tomografia (CT), magnetická rezonancia (MRI), 3D ultrazvuk, kofokálna mikroskopia a iné.

Ďalším veľkým producentom objemových dát sú rôzne odvetvia priemyslu ktoré skúmajú prúdenia kvapalín a plynov. Pomocou 3D skenu sa dajú odhaliť rôzne mikroskopické trhliny a nedostatky v materiáloch bez ich deštrukcie. Niektoré dáta môžu byť aj umelo vygenerované z rôznych simulácií a výpočtov. Rôzne odvetvia produkujú dáta s rôznymi vlastnosťami, najčastejšie ale reprezentujú hodnoty intenzít pomocou odtieňov sivej. Objemové dáta môžu reprezentovať aj viacrozmerné informácie, ako napríklad farbu vo forme RGB informácie, smerový vektor prúdenia kvapaliny a podobne. Dáta získané z dnešných medicínskych zariadení zaberajú stovky MB a niekedy až GB (snímka v čase, snímka celého tela pacienta). Štandardné rozmery tomografického rezu sú 512x512 pixlov a ich počet závisí od skúmanej časti pacienta a môže presahovať aj 1000 rezov [2].

2.4.1 3D Konvolúcia

Konvolúcia je možná aj v 3D súradniciach. Spracovávaný obrázok je 3D obrázok, a matica jadra je rozšírená na kocku.

$$(f*g)(x,y,z)=f'(x,y,z)=\sum_{0\le i,j,k<d} f(x+i,y+j,z+k)\,g(i,j,k) \tag{2.1}$$

Kde *f* je definovaná na intervale {0, .. n -1} x {0, .. n-1} x {0, .. n-1} a jadro g na intervale {0, .. d -1} x {0, .. d-1} x {0, .. d-1}. Ak funkciu g môžeme rozložiť na postupnosť 3 samostatných funkcií aplikovaných postupne potom sa jedná o separabilnú konvolúciu, pre ktorú platia vzorce 2.2, 2.3 a 2.4:

$$f_1(x,y,z)=\sum_{0\le i<d} f(x+i,y,z)\,g_1(i) \tag{2.2}$$

$$f_2(x,y,z)=\sum_{0\le j<d} f(x,y+1,z)\,g_2(j) \tag{2.3}$$

$$f_3(x,y,z)=\sum_{0\le k<d} f(x,y,z+1)\,g_3(k) \tag{2.4}$$

2.5 Použité filtre

V tejto práci sa zaoberám hlavne hranovým Sobelovým filtrom a vyhladzovacím Gaussovým filtrom.

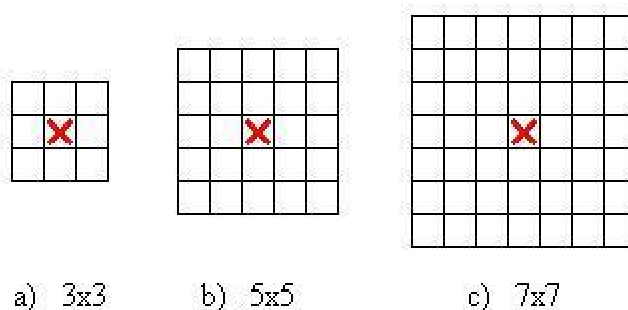
2.5.1 Rozbor Sobelovho hranového filtru

Ako ostatné filtre aj tento je potrebné zrátať bod po bode. Pre výpočet sa používajú aj okolité body. Oblasť ktorá sa analyzuje je vynásobená konvolučnou maticou ktorá má tvar štvorca. Matica jadra má nasledovnú formu (Vzorec 3.1):

$$K=\begin{bmatrix}-1 & 0 & 1\\ -2 & 0 & 2\\ -1 & 0 & 1\end{bmatrix} \qquad (3.1)$$

tento výpočet je potrebné spraviť pre každý jeden bod. Väčšinou nie je možné postupné medzivýsledky používať viac krát. Podľa hodnôt v matici sú aj výsledky tohoto filtru rôzne, tiež je možné povedať že hodnotami v matici je daný samotný typ filtru, napríklad sobelov, prewittov, Robinsonov alebo Kirschov operátor. Taktiež je možné nastaviť týmito hodnotami citlivosť v ktorom smere je daný filter citlivý. Niekedy sa používajú filtre ktoré sú citlivé len v jednej osi.

Možné veľkosti oblastí ktoré sa najbežnejšie používajú pre výpočet je možné vidieť na obrázku 2.



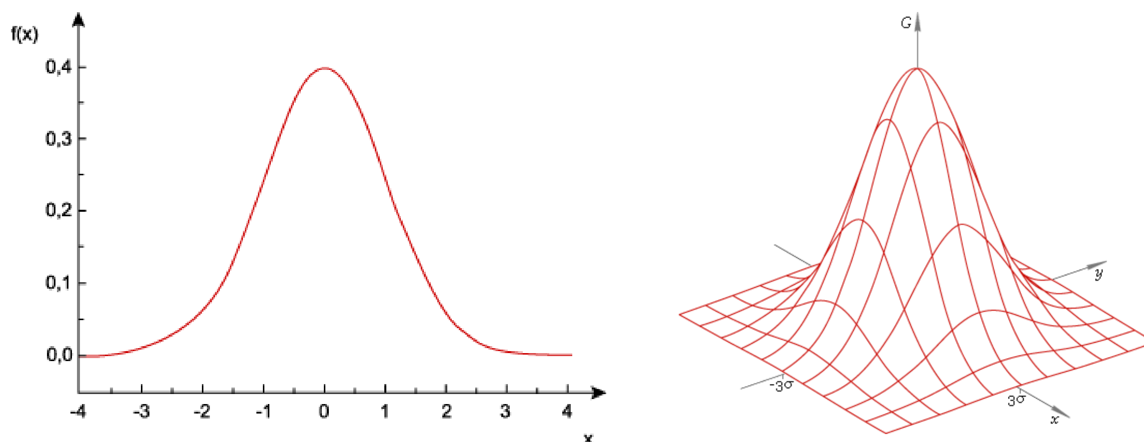
Obrázok 2: Možné oblasti spracovania [8].

Pre demonštráciu správania filtra je na nasledovnom obrázku uvedený príklad obrázku pred spracovaním a po filtrovaní. Ako filter bol v našom prípade použitý filter sobel. ktorého implementácia je popísaná ďalej.

Obrázok 3: Ukážka výsledku hranového grafického filtru.

2.5.2 Rozbor Gaussovho vyhladzovacieho filtru

Ďalším z filtrov ktoré sa v dnešnej dobe bežne používajú sú vyhladzovacie filtre, na rozdiel od predchádzajúceho filtru tento dokáže napríklad odstrániť šum z obrazu, samozrejme za cenu mierneho rozostrenia. Princíp výpočtu je pomerne jednoduchý, ide o spriemerovanie práve rátaného bodu a okolitých bodov za využitia presne stanovenej krivky. V prípade že by išlo o jednorozmerný obrázok bolo by možné povedať že je každý prvok a jeho okolné prvky sú prenásobené danou krivkou. Aby obrázok nemenil svoju intenzitu mal by byť po výpočte vydelený objemom pod krivkou. Pre dvojrozmerný obrázok potom ide o násobenie plochou (Obrázok 4). Pre gaussov filter ide o nasledovné krivky, samozrejme s využitím iných filtrov budú aj krivky iné.



Obrázok 4: Gausovské rozloženie.

Matematicky je možné Gaussov filter pre dvojrozmerný priestor (klasický 2D obrázok) vyjadriť nasledovne (Vzorec 4.1).

$$g(x,y)=\frac{1}{2*\pi*\sigma^2}*e^{-\frac{x^2+y^2}{2*\sigma^2}} \quad (4.1)$$

- *g(x,y)* je funkcia prepočtu zo vstupného obrázku do výstupného obrázku
- *x* je index riadku príslušného bodu v pravouhlej súradnicovej sústave
- *y* je index stĺpca príslušného bodu v pravouhlej súradnicovej sústave
- *σ* je parameter gausovského rozloženia.

Ako vidíme z prvého pohľadu, zdá sa že ide o pomerne náročný filter, avšak je možné tento výraz modifikovať. Touto modifikáciou dostávame násobok dvoch jednorozmerných filtrov (Vzorec 4.2). A je možné ich rátať postupne ako jednorozmerné

$$g(x,y)=\frac{1}{\sqrt{2*\pi}*\sigma}*e^{\frac{-x^2}{2*\sigma^2}}*\frac{1}{\sqrt{2*\pi}*\sigma}*e^{\frac{-y^2}{2*\sigma^2}}=g(x)*g(y) \quad (4.2)$$

potom jednorozmerný filter má nasledovný tvar (Vzorec 4.3):

$$g(x)=\frac{1}{\sqrt{2*\pi}*\sigma}*e^{\frac{-x^2}{2*\sigma^2}} \quad (4.3)$$

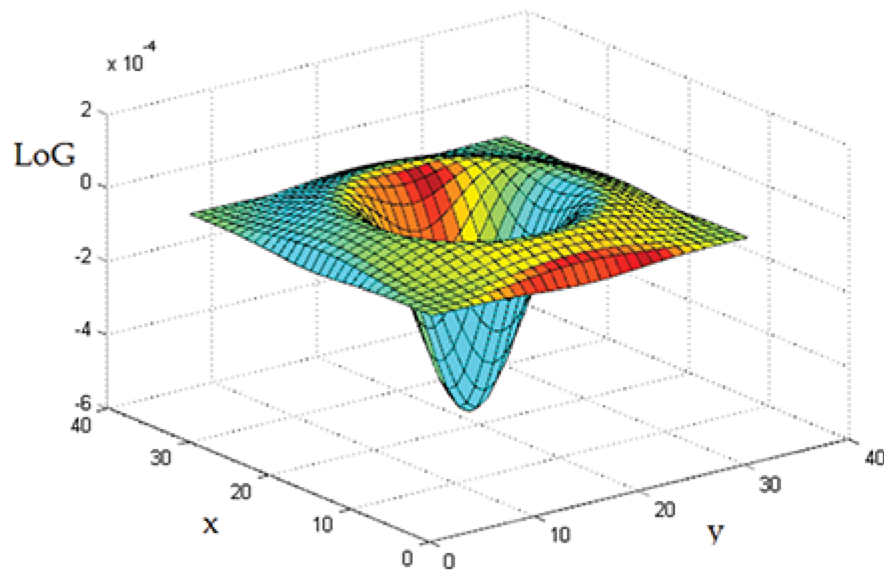
Ak si predstavíme že filter bude rátaný do vzdialenosti 2 od práve rátaného bodu (súradnice x-2 až x+2) je možné vyjadriť tieto filtre pomocou maticového zápisu (Vzorec 4.4).

$$K_g=\begin{bmatrix} g(-2,-2) & g(-1,-2) & g(0,-2) & g(1,-2) & g(2,-2) \\ g(-2,-1) & g(-1,-1) & g(0,-1) & g(1,-1) & g(2,-1) \\ g(-2,0) & g(-1,0) & g(0,0) & g(1,0) & g(2,0) \\ g(-2,1) & g(-1,1) & g(0,1) & g(1,1) & g(2,1) \\ g(-2,2) & g(-1,2) & g(0,2) & g(1,2) & g(2,2) \end{bmatrix} \quad (4.4)$$

túto maticu je možné vyrátať buď to ako klasický hranový filter ktorý bol zmienený vyššie, alebo po zjednodušení je ho možné vyrátať samostatne v dvoch osiach ako dva jednorozmerné filtre. Tento postup výrazne obmedzí výpočtový čas. Inými slovami maticu K_g je možné získať nasledovným výpočtom (Vzorec 4.5):

$$V_g = [g(-2), g(-1), g(0), g(1), g(2)] \qquad V_g^T * V_g = K_g \qquad (4.5)$$

Týmto postupom je možné implementovať viaceré filtere, nie len Gaussov filter, ale napríklad aj Laplasov filter (Obrázok 5). Dôležitý predpoklad je aby boli súradnice rátaného bodu v exponente funkcie.



Obrázok 5: Rátaná plocha pre Laplacian of Gaussian.

3 Prístupy k akcelerácii obrazových filtrov na GPU

Pri prenose implementácie z jednoduchého procesora na paralelnú architektúru vhodnú pre GPU je možné použiť niekoľko techník, a optimalizácií [3].

3.1 Naivná implementácia

Naivná implementácia priamo presunie výpočet z CPU na GPU bez optimalizácie spracovania.
Využíva sa iba globálna pamäť GPU. Nevýhodou môže byť nízky prínos v rýchlosti spracovania výpočtu, ktorý je závislý na spracovávanom probléme, a tiež plne nevyužitý potenciál GPU.

3.2 Použitie konštantnej pamäte

Premenné, ktoré sa počas výpočtu nemenia (konštanty) budú uložené do konštantnej pamäte. Skráti sa síce čas potrebný na prenosy údajov medzi CPU a GPU, ale výrazné skrátenie času celkového výpočtu to neprinesie.

3.3 Použitie vektorových dátových typov

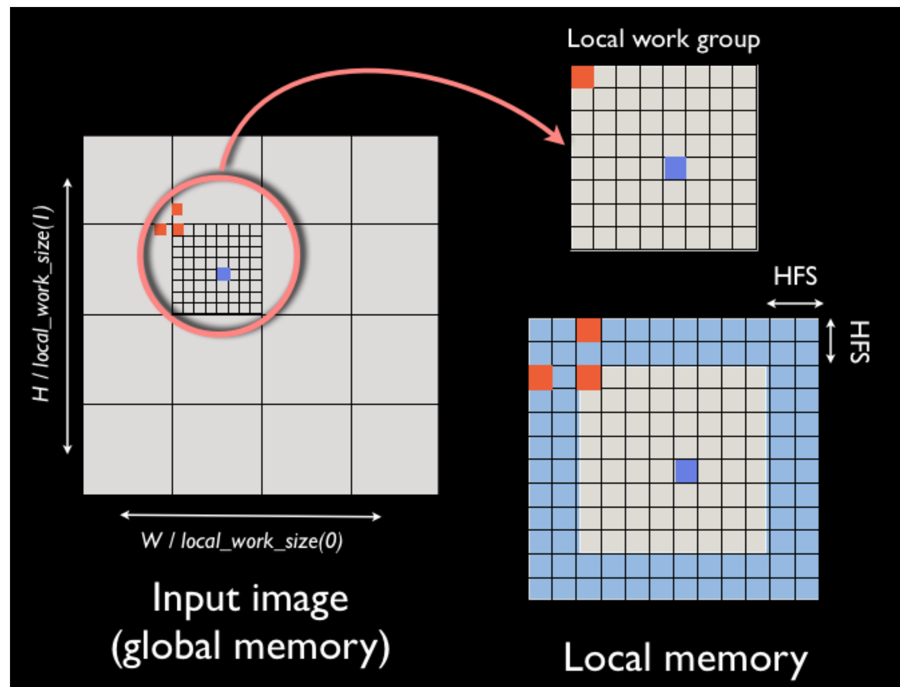
Oproti predchádzajúcemu spracovaniu je spracovanie vektoru premenných vykonávané paralelne. Na spracovanie sa používa napríklad dátový typ float4. Týmto spôsobom sa zníži počet prenosových inštrukcií. Pri jednom čítaní sa prenesie celý vektor RGBA bodu obrazu alebo viacero pixelov volumetrických dát. Tento formát nie je podporovaný všetkými grafickými kartami.

3.4 Rozvinuté cykly

V grafickom procesore je niekedy rýchlejšie vykonať n rovnakých inštrukcií ako pripraviť a vykonať cyklus n opakovaní. Je síce dlhší zdrojový kód, ale celkový čas spracovania bez vyhodnocovania podmienok skončenia cyklu môže byť kratší.

3.5 Použitie lokálnej pamäte

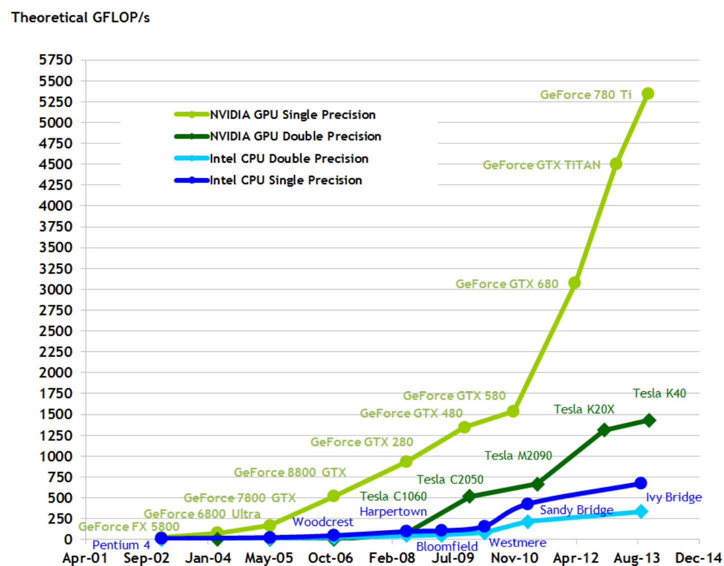
Pri použití lokálnej pamäte sa minimalizuje počet prístupov do pomalej globálnej pamäte. Z globálnej pamäte sa presunie časť informácií do lokálnej pamäte. Do tejto pamäte potom pristupuje celá pracovná skupina. Práca s touto pamäťou je podstatne rýchlejšia ako s globálnou pamäťou.
Výhoda tohoto riešenia sa prejaví až keď je potrebné mnohonásobné čítanie uložených informácií. Obrázok sa rozseká na bloky (napríklad štvorce) určitej veľkosti, vypočítajú sa požadované operácie pre každý blok (Obrázok 6).



Obrázok 6: Segmentácia obrázku na bloky, prevzaté z [3].

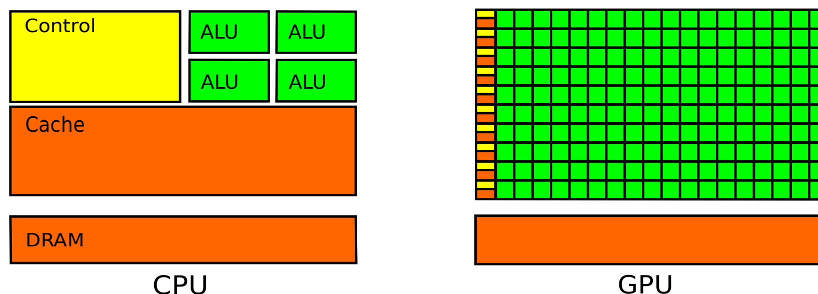
4 Akcelerácia výpočtov pomocou GPGPU a OpenCL

Schopnosť grafickej karty počítať väčšie množstvo výpočtov v plávajúcej rádovej čiarke ako procesor je dané tým, že GPU je špecializovaná na výpočet náročných vysoko paralelných výpočtov – renderovanie obrazu (Obrázok 7).



Obrázok 7: Výkon GPU vs. výkon CPU, prevzaté z [7].

A preto väčšie množstvo tranzistorov je využité na spracovanie dát ako na ich ukladanie a kontrolu toku (Obrázok 8).

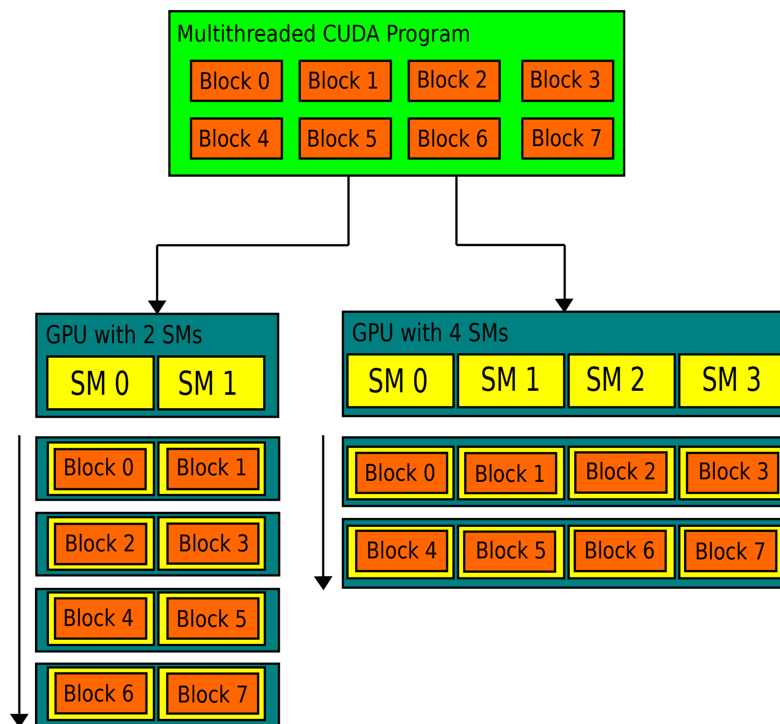


Obrázok 8: CPU vs. GPU štruktúra, prevzaté z [7].

GPU je vhodná na počítanie algoritmov pracujúcich paralelne. Ideálna na spracovanie obrazu, čo je aj náš problém filtrácie 2D a 3D medicínskych obrazových dát. Výkonné grafické karty vznikli najmä kvôli dopytu herných nadšencov po akcelerácií 3D scén v hrách. GPU je paralelný, viacjadrový a viacvláknový hardware s vysokým výkonom, pamäťovou a dátovou priepustnosťou.

4.1 NVIDIA CUDA

Nvidia CUDA (Compute Unified Device Architecture) je platforma pre paralelné výpočty pre všeobecné použitie za pomoci grafickej karty (GPGPU) od spoločnosti Nvidia. Dá sa využiť na výpočet komplexných problémov efektívnejšou cestou než výpočet na klasickom CPU. Podporuje začlenenie do rôznych programovacích jazykov (C, Fortran, Python …). Využíva sa škálovateľný programovací model. Systém pozná fyzickú GPU (počet výpočetných jadier, ...) až za chodu programu a naplánuje potrebné úlohy na jednotlivé jadrá paralelne alebo sériovo (Obrázok 9).



Obrázok 9: Vykonávanie CUDA programu, prevzaté z [7].

Využívajú sa vlákna, zdielaná pamäť, bariéry… Časť programu pre GPU môže byť skompilovaná offline alebo kompilovaná za behu programu. To má za následok dlhšie spúšťanie ale podporuje aj zariadenia ktoré v čase písania programu a jeho offline kompilácie neexistovala, a môže profitovať s nových rozšírení a úprav v CUDA toolkite. Pamäť je rozdelená na pamäť hostiteľskú

(CPU), a pamäť zariadenia (GPU). CUDA zabezpečuje kopírovanie dát medzi pamäťami, a ich alokáciu a dealokáciu.

V roku 2006 predstavila firma NVIDIA technológiu CUDA. Táto hardwarová a softwarová architektúra umožnila ako prvá spúšťať na grafických kartách užívateľské programy. Podporuje programy v programovacích jazykoch C, Fortran, Python. Podpora tejto platformy je zabezpečená vo všetkých hlavných operačných systémoch (MS Windows, Mac OS X, Linux). Avšak podporuje iba grafické akcelerátory od firmy NVIDIA.

CUDA je platforma pre paralelné programovanie, programové aplikačné rozhranie (API), a zároveň tiež programovací model vyvinutý firmou NVIDIA. CUDA umožňuje, pri správnej analýze a aplikácii, dramatické navýšenie výpočtového výkonu s využitím všeobecne použiteľných grafických kariet GPGPU. GPGPU označuje možnosť použitia GPU čipov, ktoré majú bežne na starosti špecializované grafické výpočty, na výpočty, ktoré sa väčšinou realizujú v CPU. Zvýšenie výkonu je možné dosiahnuť aj paralelným využitím viacerých grafických kariet na jednom počítači alebo väčšieho počtu grafických čipov. Okrem toho, aj jednoduchý GPU-CPU framework poskytuje výhody, ktoré riešenie s viacerými CPU neposkytuje, čo vyplýva zo špecializácie jednotlivých čipov. GPU čipy umožňujú, pri správnom použití, značne skrátiť dobu výpočtu.

Akého urýchlenia sa dosiahne však závisí okrem konkrétnej aplikácie na výkone použitého GPU čipu. Dôvodom nezrovnalosti vo výpočtoch s plávajúcou rádovou čiarkou medzi CPU a GPU je špecializácia pre náročné, vysoko paralelné výpočty – presne to čo je potrebné v zobrazovaní – a preto sú GPU navrhnuté tak, že viac tranzistorov je venovaných na spracovávanie dát naproti ukladaniu dát do pamäte cache. GPU sú navrhnuté aby dobre zvládali problémy, ktoré môžu byť vyjadriteľné ako výpočty s dátovým paralelizmom (Jeden program vykonávaný na väčšom množstve dátových prvkov paralelne). Zároveň sa využíva vysoký pomer aritmetických ku pamäťovým operáciám. Keďže sa pre každý prvok vykonáva rovnaký program, nie sú také požiadavky na reguláciu toku riadenia a je možné skryť latenciu prístupu do pamäti cez veľké množstvo aritmetických operácií, naproti veľkým pamätiam cache. Paralelné spracovanie dát mapuje dátové prvky na paralelne pracujúce vlákna. V 3D zobrazení sa tento prístup využíva na spracovanie súborov pixlov a bodov.

4.1.1 CUDA programovací model

Keďže paralelizmus dnešných GPU rastie s Moorovým zákonom, je nutné aby sme vyvíjali software, ktorý transparentne škáluje svoj paralelizmus aby využil narastajúci počet jadier procesorov. Paralelný programovací model, ktorý CUDA prináša, je navrhnutý tak, aby prekonal túto výzvu,

zatiaľ čo si zachováva nízku krivku učenia pre programátorov oboznámených so štandardnými programovacími jazykmi ako napríklad jazyk C.

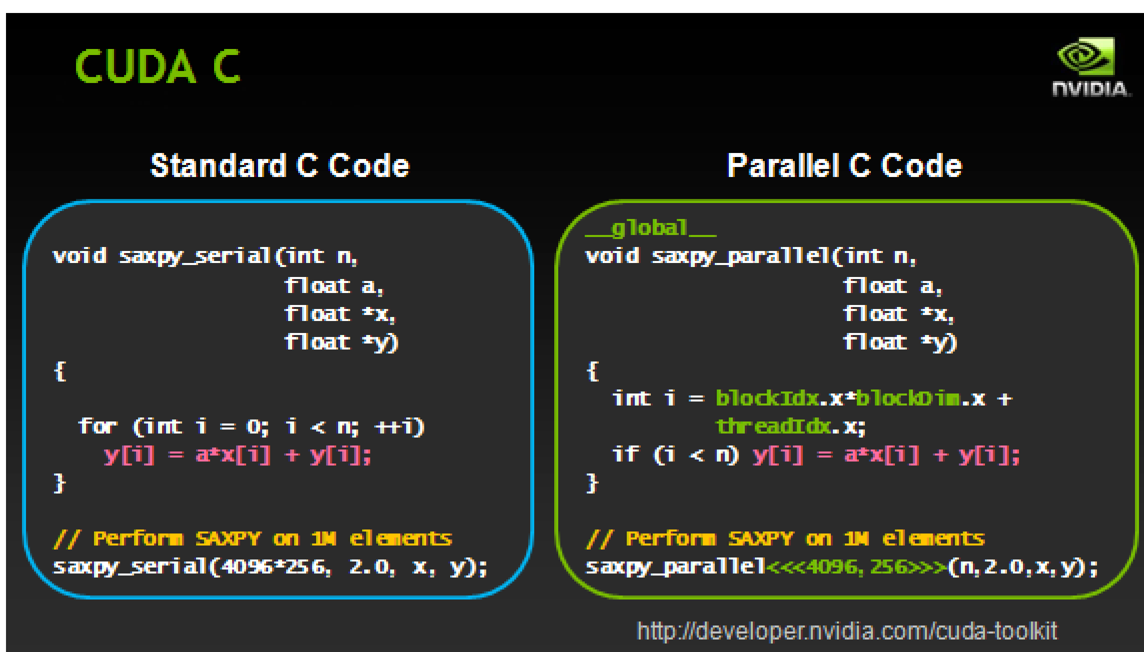
Jadrom tohoto programovacieho modelu sú tri kľúčové abstrakcie:

- Hierarchia skupín vlákien
- Zdieľaná pamäť
- Bariérová synchronizácia

Tieto abstrakcie sú programátorovi sprístupnené ako minimálna kolekcia rozšírení jazyka, čo umožňuje ich jednoducho integrovať do existujúceho ale aj nového kódu. Spomínané abstrakcie prinášajú dátový paralelizmus a paralelizmus vlákien. Vedú programátora k tomu aby rozdelil problém na pod-problémy, ktoré môžu byť riešené nezávisle v paralelných blokoch vlákien. Každý pod-problém je potom ešte delený na menšie časti tak, aby mohol byť riešený spoluprácou paralelne bežiacich vlákien v jednom bloku. Toto rozdelenie zachováva vyjadrovacie možnosti jazyka tým, že dovoľuje vláknam spolupracovať na riešení pod-problémov zatiaľ čo umožňuje automatickú škálovatelnosť. Každý blok vlákien môže byť skutočne naplánovaný na ktoromkoľvek dostupnom multiprocesore v rámci GPU, v akomkoľvek poradí, paralelne alebo sekvenčne, tak aby skompilovaný CUDA program mohol byť spustený na akomkoľvek počte multiprocesorov a len systém spravujúci zariadenie musí vedieť fyzický počet multiprocesorov.

4.1.2 CUDA dôležité princípy

CUDA C rozširuje jazyk C a čiastočne C++ tým, že pridáva programátorovi možnosť definovať funkcie, vo formáte jazyka C, zvané kernely. Keď je kernel funkcia spustená, tak dôjde k jej vykonaniu N-krát paralelne, naproti bežnému vykonaniu jeden krát. Každá inštancia kernelu je spracovávaná v rámci jedného z N spustených CUDA vlákien. Kernel funkciu je pri jej vytváraní nutné špecificky zadefinovať. Pri volaní funkcie kernelu je nutné určiť počet spustení kernelu definované pomocou veľkosti bloku a veľkosti mriežky. Každé vlákno, ktoré spúšťa kernel, dostane pridelené unikátne identifikačné číslo vlákna (threadIdx). Nepodobne všetky vlákna v rámci jedného bloku zdieľajú identifikačné číslo bloku (blockIdx), ktoré je medzi blokmi unikátne. ThreadIdx a blockIdx sú pre vlákna prístupné počas celého behu kernelu. Sú využívané algoritmami pre definíciu prístupu do pamäti a delenie práce medzi vlákna.



Obrázok 10: Porovnanie kódu v jazyku C a paralelnom jazyku CUDA C, prevzaté z [9].

Obrázok 10 ukazuje porovnanie volania CUDA kernelu oproti bežnej funkcie jazyka C.

Hierarchia vlákien

Premenná threadIdx je 3-zložkový vektor, ktorý umožňuje identifikovať vlákna pomocou jedno/dvoj/troj rozmerného indexu vlákna. Vlákna zoskupené dokopy tvoria jedno/dvoj/troj rozmerný blok vlákien (thread block). Toto poskytuje prirodzený spôsob ako spustiť výpočet naprieč prvkami tvoriacimi vektor, maticu alebo objem (Obrázok 11).

Maximálny počet vlákien na jeden blok je obmedzený. Je to z dôvodu aby sa všetky vlákna v jednom bloku mohli nachádzať na jednom procesorovom jadre a zdielať jeho obmedzené pamäťové zdroje. Na aktuálnych GPU je počet vlákien na jeden blok obmedzený na 1024. -napriek tomu je možné spustiť kernel na viacerých rovnako veľkých blokoch vlákien. Bloky sú, podobne ako vlákna, organizované do jedno/dvoj/troj rozmernej mriežky blokov vlákien (grid of thread blocks). Počet blokov vlákien v mriežke je väčšinou daný veľkosťou spracovávaných dát alebo počtom procesorov v systéme, ktorý môže značne prekročiť.

Od blokov vlákien sa vyžaduje, aby ich bolo možné vykonávať nezávisle (paralelne alebo sériovo) a v ľubovolnom poradí. Táto požiadavka na nezávislosť dovoľuje blokom vlákien, aby boli plánované v ľubovolnom poradí na ľubovolnom počte jadier. Toto dovoľuje programátorovi písať kód, ktorý škáluje s počtom jadier.

Vlákna v rámci jedného bloku môžu spolupracovať prostredníctvom zdieľania dát cez zdieľanú pamäť (shared memory). Prístup do tejto pamäte je ale nutné synchronizovať. Presnejšie je možné špecifikovať synchronizačné body pomocou bariér.

Bariéra funguje tak, že pred ňou všetky vlákna v bloku čakajú predtým, než je ktorékoľvek vlákno pustené ďalej. Pre efektívnu kooperáciu sa očakáva, že zdielaná pamäť má nízku odozvu a je blízko pri jadre procesora (podobne ako L1 cache). Ďalej sa od bariérovej funkcie očakáva, že bude nenáročná.

Hierarchia pamätí

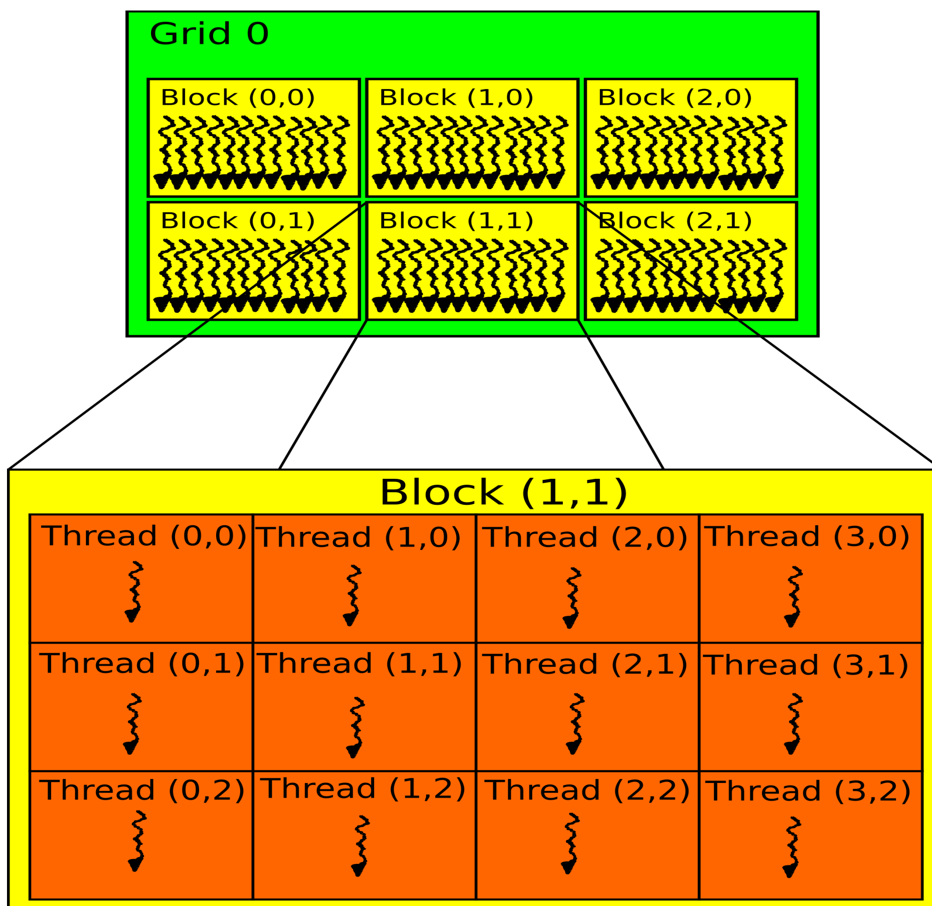
CUDA vlákna dokážu, počas ich vykonávania, pristúpiť na dáta z viacerých pamäťových priestorov. Každé vlákno má svoju privátnu lokálnu pamäť. Každý blok vlákien má spoločnú zdieľanú pamäť viditeľnú všetkými vláknami v rámci daného bloku (Obrázok 12). Táto zdieľaná pamäť má rovnakú životnosť ako samotný blok. Všetky vlákna majú prístup do rovnakej globálnej pamäti. Vlákna si tiež medzi seba delia registre SM. Okrem toho existujú dva pamäťové priestory, slúžiace len na čítanie, dostupné pre všetky vlákna: pamäť konštánt a pamäť textúr. Globálna pamäť, pamäť konštánt a pamäť textúr sú optimalizované na rôzne použitia. Pamäť textúr poskytuje iné módy adresovania ako aj filtrovania dát, pre niektoré špecifické dátové formáty. Globálna pamäť, pamäť konštánt a pamäť textúr zostávajú uchované medzi spusteniami kernelu v rámci jednej aplikácie.

Heterogénne programovanie

Programovací model CUDA predpokladá, že CUDA vlákna sú vykonávané na fyzicky separátnych zariadeniach, ktoré pracujú ako koprocesor ku hostiteľovi, ktorý vykonáva program v jazyku C. Ďalej sa predpokladá, že si obe strany (hostiteľ a zariadenie) udržujú vlastný separátny pamäťový priestor v DRAM. Tieto priestory sa nazývajú pamäť hostiteľa (host) a pamäť zariadenia (device). Program má teda na starosti pamäť globálnu, konštánt a textúr ktoré sú viditeľné kernelom cez volania CUDA runtime. Toto zahŕňa aj alokáciu a dealokáciu a presun dát medzi hostiteľom a zariadením.

Compute Capability

Compute Capability (výpočtové schopnosti) zariadenia sú reprezentované číslom verzie, tiež niekedy nazývané aj SM verzia (SM version). Pomáha identifikovať, ktoré funkcie hardware podporuje. Aplikácie sú schopné túto informáciu zistiť za behu. Toto číslo sa skladá z veľkej (X) a malej (Y) verzie v tvare X.Y. Zariadenia s rovnakým číslom veľkej verzie spadajú pod rovnakú architektúru (Pascal, Maxwell, Kepler …)



Obrázok 11: Rozdelenie výpočtových jadier, prevzaté z [7].

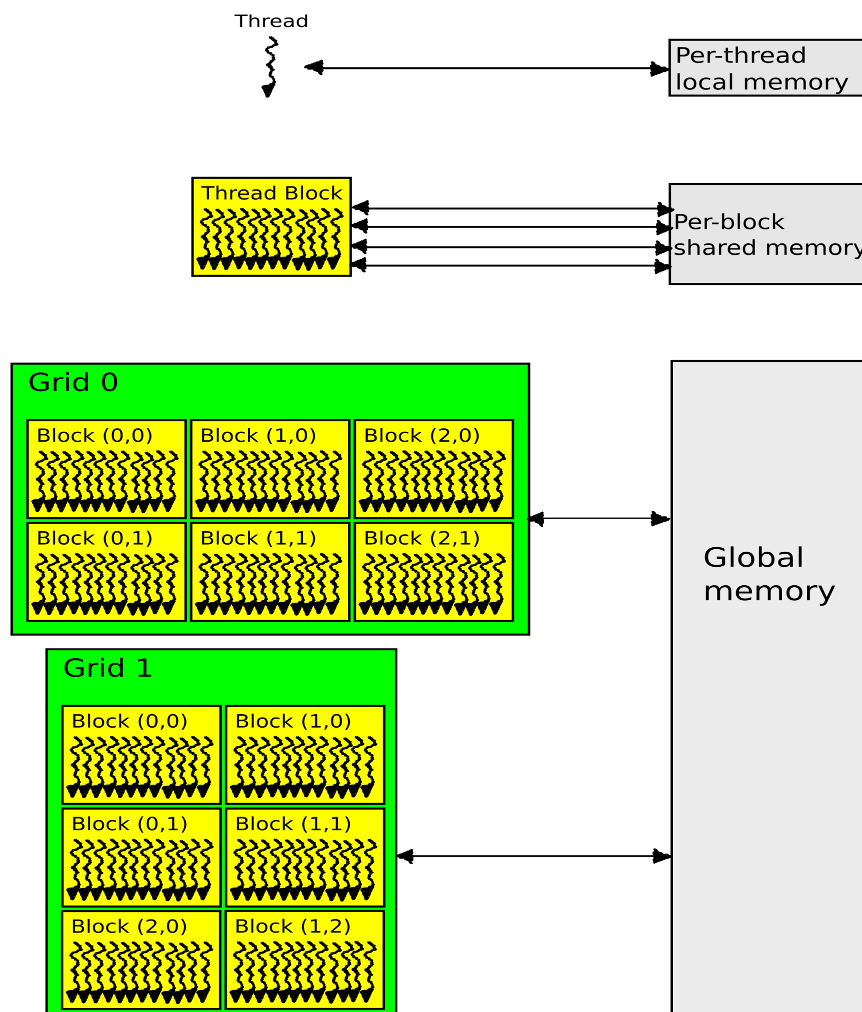
CUDA dôležité súčasti

CUDA platforma poskytuje programátorom dve API, ktoré s ňou umožňujú pracovať: Runtime API a Driver API. Runtime API je postavené nad nízko úrovňovým API v jazyku C, CUDA driver API. Driver API poskytuje programátorovi vyššiu úroveň kontroly nad aplikáciou tým, že sprístupňuje ďalšie koncepty ako: CUDA kontext, CUDA moduly. CUDA kontext je analógia procesov pre zariadenie a CUDA modul je obdobou dynamicky pripojovaných knižníc. Väčšina aplikácií ale nepotrebuje pracovať na nízkej úrovni a teda si vystačí s Runtime API.

Pamäť zariadenia

Programovací model CUDA predpokladá, že systém je zložený z hostiteľa a zariadenia, kde obaja majú svoj vlastný oddelený pamäťový priestor. Kernely operujú z pamäti zariadenia, takže runtime poskytuje funkcie na alokáciu, de-alokáciu a kopírovanie pamäte na zariadení. Okrem toho poskytuje runtime funkcie na presun dát medzi hostiteľskou pamäťou a pamäťou zariadenia.

Lineárna pamäť existuje na zariadení v 40-bitovom adresnom priestore, takže separátne alokované entity môžu navzájom na seba odkazovať pomocou ukazovateľov.



Obrázok 12: Pamaťový model CUDA, prevzaté z [7].

Asynchrónne paralelné vykonávanie

CUDA sprístupňuje nasledujúce operácie ako nezávislé úlohy, ktoré môžu byť vykonávané naraz:

- Výpočet na hostiteľskej strane

- Výpočet na strane zariadenia
- Presun dát z hostiteľskej pamäti do pamäti zariadenia
- Presun dát z pamäti zariadenia do hostiteľskej pamäti
- Presun dát v rámci pamäti daného zariadenia
- Presun dát medzi zariadeniami

Paralelné vykonávanie medzi hostiteľom a zariadením

Paralelné vykonávanie medzi hostiteľom a zariadením je umožnené pomocou asynchrónnych knižničných funkcií, ktoré navrátia kontrolu hostiteľskému vláknu pred tým, než zariadenie dokončí požadovanú úlohu. Pri používaní asynchrónnych volaní môžu byť mnohé operácie na zariadení zaradené do fronty a spracované CUDA ovládačom, až keď sú požadované zdroje dostupné. Tento prístup pomáha odľahčiť záťaž spracovania zariadenia z hostiteľského vlákna, aby toto vlákno mohlo vykonávať iné úlohy. Nasledujúce operácie na zariadení sú asynchrónne z pohľadu hostiteľa:
- spustenie kernelu
- kopírovanie v rámci pamäti jedného zariadenia
- kopírovanie z hostiteľskej pamäti do pamäti zariadenia, kde blok kopírovaných dát je do 64 KB
- kopírovanie pamäti vykonávané funkciou so suffixom Async
- funkcie pre hromadné nadstavovanie hodnôt v pamäti

Programátor môže globálne zakázať asynchrónne spúšťanie kernelov pre všetky CUDA aplikácie bežiace na systéme. Táto funkcia by sa však mala využívať len na účely odstraňovania chýb. Spustenia kernelu sú synchrónne v prípade, že hardwarové počítadlá sú zberané pre profiler (Nsight, Visual Profiler) pokiaľ nie je povolené paralelné profilovanie kernelov. Asynchrónne kopírovanie pamäti sa tiež stáva synchrónnym ak zahŕňa hostiteľskú pamäť, ktorá nie je viazaná na stránku.

Paralelné vykonávanie kernelov

Niektoré zariadenia s compute capability 2.x a vyššou sú schopné spúšťať viac kernelov naraz. Aplikácie sú schopné si túto funkcionalitu overiť cez vnútorný atribút zariadenia. Maximálny počet naraz spustených kernelov závisí taktiež od compute capability zariadenia.

Compute capability	2.x	3.0	3.2	3.5	3.7	5.0	5.2	5.3
Maximálny počet naraz spustených kernelov	16	16	4	32	32	32	32	16

Tabuľka 1: počet spustených kernelov v závislosti na compute capability.

Kernel z jedného CUDA kontextu nemôže byť vykonávaný zároveň s kernelom z iného CUDA kontextu. U kernelov, ktoré využívajú veľa textúr alebo veľké množstvá lokálnej pamäte, je menej pravdepodobné, že budú vykonávané naraz s inými kernelmi. Stále totiž platí, že všetky vlákna, teda aj vlákna iných kernelov, sa delia o zdroje patriace SM.

4.2 OpenCL

OpenCL (Open Compute Language) je framework pre programovanie paralelných výpočtov na rôznych hardvérových platformách. OpenCL nie je na rozdiel od frameworku NVIDIA CUDA závislý na hardware od jednej firmy. Programy, ktoré sú napísané v OpenCL, sú schopné pracovať na univerzálnych procesoroch, grafických kartách, ale napríklad aj na procesoroch architektúry CELL alebo signálnych procesoroch DSP. Za vývojom a štandardizácie OpenCL stojí skupina Khronos Group, ktorá zastrešuje tiež vývoj OpenGL štandardu alebo nového Vulkan štandardu. Skupina Khronos pozostáva z firiem ako AMD, NVIDIA, Apple, IBM, Intel, Microsoft, Valve … OpenCL je otvorený štandard, ktorý bol predstavený v roku 2008. Aktuálne sa nachádza vo verzií 2.2.

Programovanie pomocou OpenCL zahrnuje 2 časti. Naprogramovanie takzvanej hostitelskej časti, ktorá je vykonávaná na klasickom procesore a časti kernelu, ktorý sa vykonáva na zariadení podporujúcom OpenCL (CPU, GPU, iné) [4].

Pre základný popis programovej architektúry OpenCL si ho môžeme rozdeliť do hierarchicky usporiadaných modelov:

- Platformový model
- Pamäťový model
- Vykonávací model
- Programovací model

4.2.1 Platformový model

Platformový model obsahuje hostiteľský systém, na ktorý je pripojené jedno alebo viac zariadení podporujúcich OpenCL. OpenCL zariadenie pozostáva s jednej alebo viacerých výpočtových jednotiek (CU – compute unit, podobne ako CUDA jadrá). Tieto výpočtové jednotky ďalej obsahujú niekoľko výpočtových procesorov (PE – processing elements).

4.2.2 Vykonávací model

Vykonávací model OpenCL sa skladá z dvoch hlavných častí – Kernel a hostiteľský program. Kernel je funkcia ktorý vykonáva zariadenie OpenCL. Hostiteľský program definuje použité OpenCL zariadenie, stará sa o zápis a čítanie dát do/z pamäti OpenCL zariadenia a o spúšťanie jednotlivých kernelov.

Vykonávací model tiež definuje, akým spôsobom budú kernely vykonané. Keď má byť spustený výpočet nad dátami, je pre daný kernel vytvorený globálny indexový priestor. Na každom výpočtovom procesore je vykonávaná jedna inštancia kernelu. Túto inštanciu nazývame pracovná jednotka (work item) a je definovaná jednoznačne svojim indexom v rámci globálneho indexového priestoru. Pracovné jednotky môžu byť zhlukované do pracovných skupín (work group). Pracovné skupiny sú vykonávané na výpočtových jednotkách (CU) a pracovné jednotky sú vykonávané na výpočtových procesoroch (PE). Každá pracovná skupina má svoj jednoznačný index a každá pracovná jednotka má v rámci svojej pracovnej skupiny unikátny lokálny index.

To nám určuje, že každá pracovná jednotka môže byť identifikovaná dvoma spôsobmi:

1. pomocou svojho global ID

2. pomocou work group ID a local ID a údaje, aké veľké sú pracovné skupiny (Veľkosť pracovnej skupiny je obmedzená a závisí na použitom zariadení)

Indexový priestor, ktorý využíva OpenCL sa nazýva NDRange. NDRange je N-Dimenzionálny indexový priestor, kde N môže nadobúdať hodnoty 1, 2, 3. To nám dáva možnosť rozdeliť inštancie kernelov do dimenzií, aké potrebujeme využiť pre spracovávané dáta (vektory, matice, 3D matice). V hostiteľskom programe sa musí definovať kontext vykonávania kernelov. Kontext zaisťuje vykonávací model a musí obsahovať tieto zdroje:

1. Zariadenie OpenCL ktoré hostiteľský program využíva.

2. Kernely OpenCL (funkcie) ktoré budú na zariadení OpenCL spustené

3. Programové objekty: zdrojové a spustiteľné kódy programov ktoré implementujú dané kernely.

4. Pamäťové objekty ktoré sú viditeľné pre hostiteľský systém, a aj pre OpenCL zariadenie, ku ktorým môžu pristupovať inštancie kernelov spustené na pracovných jednotkách.

Ďalej je nutné vytvoriť fronty príkazov (command queue) k jednotlivým OpenCL zariadeniam. Tieto fronty slúžia na vkladanie jednotlivých príkazov, ktoré sa majú vykonávať na zariadeniach.

Medzi tieto príkazy patrí vykonanie kernelu, manipulácia s pamäťovými objektami a synchronizačné príkazy. Pre príkazové fronty je možno nastaviť, či sa majú príkazy vykonávať v poradí v akom boli zadávané (in order) alebo sa poradie nemusí dodržiavať (out of order).

4.2.3 Pamäťový model

Pracovné jednotky v OpenCL majú prístup do 4 rôznych pamäťových priestorov:

Globálna pamäť – Táto pamäť je prístupná všetkým pracovným jednotkám pre čítanie aj zápis vo všetkých pracovných skupinách. Nie je cachovaná.

Konštantná pamäť – Časť globálnej pamäte, ktorá je prístupná iba na čítanie a je inicializovaná hostiteľským programom. Na niektorých zariadeniach môže byť cachovaná čo zvyšuje jej rýchlosť.

Lokálna pamäť – Je to pamäťová oblasť, ktorá je prístupná v rámci pracovnej skupiny (work group). Táto pamäť sa nachádza priamo na grafickom čipe a je veľmi rýchla. Využíva sa na alokáciu premenných, ktoré môžu byť zdielané medzi pracovnými jednotkami.

Privátna pamäť – Táto pamäť je vyhradená iba pre konkrétnu pracovnú jednotku. Do premenných ktoré sú definované v tejto pamäti nemôžu zapisovať iné pracovné jednotky.

4.2.4 Programovací model

OpenCL vykonávací model podporuje dátovo paralelné a úlohovo paralelné programovacie modely, ale aj ich kombinácie. Primárny model, pre ktorý je OpenCL navrhnutý je dátovo paralelný.

Dátovo paralelný programovací model definuje výpočet ako vykonávanie rovnakých sekvencií inštrukcií nad viacerými jednotkami dátových objektov. Vykonávací model jednoznačne definuje počty pracovných jednotiek, ale tiež to, ako do týchto jednotiek budú mapované dátové elementy. V najstriktnejšom dátovo paralelnom modeli pripadá jedna dátová jednotka na jednu pracovnú jednotku. V OpenCL je využitý dátovo paralelný model, kde ale striktné mapovanie jedna k jednej nie je nutne vyžadované. Výhoda dátovo paralelného modelu je, že pracovné jednotky v rámci pracovnej skupiny môžu spolu komunikovať.

Úlohovo paralelný model je model, kde jedna inštancia kernelu je vykonávaná nezávisle na indexovom priestore. Tím pádom je možné spúšťať súbežne niekoľko inštancií rôznych kernelov. V tomto programovacom modeli ale nemôžu pracovné jednotky medzi sebou komunikovať.

V prípade spracovávania obrazu sa bude využívať dátovo paralelný programovací model, pretože potrebujeme vykonávať rovnaké operácie nad množstvom dát.

4.2.5 Programovací jazyk OpenCL C

Pre písanie aplikácií využívajúcich OpenCL sa používa programovací jazyk OpenCL C, ktorý je založený na jazyku C podla normy ISO/IEC 9899:1999 – C Language specification (v skratke C99). OpenCL C sa ale od C99 líši niekoľkými rozšíreniami:

Vektorové dátové typy

Kľúčové slová pre určenie adresového priestoru

Kľúčovými slovami na určenie, odkiaľ je možné kernel volať, a kde bude vykonávaný

Kernelové funkcie

Ale má aj niektoré obmedzenia oproti C99:

Nemožno používať ukazovatele na funkciu

Rekurzia je zakázaná

Kernelové funkcie musia byť typu void (bez návratovej hodnoty)

Kernelové funkcie a ostatný programový kód ktorý má byť vykonávaný na OpenCL zariadení sa zapisuje do súboru s príponou .cl. Hostiteľský program sa zapisuje v C/C++ súboroch a kernelové funkcie sú z nich spúšťané. Pre každú kernelovú funkciu je nutné vytvoriť tzv. Kernelový objekt. Súbory .cl sa nekompilujú spoločne s kompletným programom, existujú dve možnosti ich kompilácie.

Offline kompilácia

V tomto prípade sa súbory .cl prekompilujú vopred do binárneho súboru pomocou externého kompilátoru. Riadiaci program si tento binárny súbor pri behu načíta a z už prekompilovaných kernelov vytvorí kernelové objekty pre dané zariadenie, na ktorom sa bude spúšťať výpočet.

Online kompilácia

Pri online kompilácií je situácia odlišná. Riadiaci program si pri svojom vykonávaní načíta súbor .cl ktorý je pre vybrané OpenCL zariadenie prekompilovaný až pri behu aplikácie a následne sa z tohoto binárneho prekompilovaného kódu vytvoria kernelové objekty ako v predchádzajúcom prípade [5].

5 Návrh riešenia filtrácie 3D skenov s pomocou GPU

Pre jednoduchosť som sa na začiatku rozhodol implementovať 2D grafické filtre. Pre tieto filtre nebude pravdepodobne vidieť taký prínos pri spracovaní na GPU v porovnaní s CPU z dôvodu pomerne veľkej náročnosti z pohľadu kompilácie programu a prípravy na spracovanie dát. Taktiež je potrebné dáta kopírovať do grafickej karty a načítať ich potom nazad do RAM pre ďalšie spracovanie a vykreslenie. Výsledky sa tiež budú líšiť podľa toho aký zložitý filter je implementovaný, rozhodol som sa implementovať dva filtre, jeden Sobelov (hranový) a druhý matematicky náročnejší Gaussov filter (rozmazávanie). Pri druhom filtri by mohol byť väčší rozdiel medzi spracovaním na CPU voči GPU hlavne pri využití väčšieho konvolučného jadra ako v prípade prvom. Tieto dva filtre budú ďalej implementované aj v 3D verzii. Rozhodol som sa pre implementáciu cez OpenCL, keďže dovoľuje aj spúšťanie na iných grafických kartách resp. iných zariadeniach. Spracovanie obrázku sa rozloží do pracovných skupín, po riadkoch alebo po blokoch. Chcem sa najprv zamerať na implementáciu cez globálnu pamäť (naivná implementácia), ktorú potom budem optimalizovať, napríklad použitím lokálnej pamäte alebo vektorových dátových typov.

Pre zrýchlenie spracovania som využil prenos údajov do lokálnej pamäte. Pred presunom je potrebné zvážiť, ktoré údaje sa budú ukladať do lokálnej pamäte. V 2D variantách filtrov sú údaju uložené po riadkoch za sebou vo vektore. V prípade horizontálneho filtru sa do lokálnej pamäte presunie jeden riadok, a vypočítajú sa body na tomto riadku. V prípade vertikálneho filtru sa do lokálnej pamäte presunie viacej riadkov, alebo časť viacerých riadkov a vypočítajú sa možné body obrázku. V 3D variantách filtrov funguje prenos podobne, a v smere cez viacero snímkov sa načítajú dáta na rovnakej súradnici ale cez viacero snímkov a spočítajú sa možné body.

6 Implementácia

Rozhodol som sa rozšíriť framework VPL o podporu akcelerácie cez GPU. Framowork VPL je framework na spracovávanie medicínskych dát, v 2D a v 3D. Obrázky sú ukladané v otvorenom dicom formáte. Framework umožňuje 2D a 3D filtrovanie medicínskych CT dát.

VPL (voxel processing library) je kolekcia 2D a 3D nástrojov na spracovávanie obrazu s otvoreným zdrojovým kódom, zameraná na medicínske obrázky. Obsahuje rutiny pre spracovanie volumetrických dát, ako napríklad 3D filtrovanie, detekcia hrán, segmentácia…

VPL je nástupca MDSTK (Medical data segmentation toolkit). VPL je multiplatformný toolkit, pracuje na platformách Linux, Windows a Mac OS X. Je vysoko modulárny a profituje z rýchleho šablonového a škálovatelného C++ kódu.

Rozličné moduly VPL frameworku môžu komunikovať medzi sebou cez súbory, pomenované/nepomenované rúry a zdielanú pamäť [6].

6.1 2D grafické filtre

Pre zjednodušenie som sa najprv zameral na filtráciu 2D obrázkov. Na implementáciu akcelerovaných filtrov používam knižnicu OpenCL. Na rozdiel od Nvidia CUDA funguje aj na grafických kartách iných výrobcov, resp. iných zariadeniach.

6.1.1 Implementácia hranového filtru Sobel

Ako prvý bol implementovaný hranový filter sobel. Výpočet je postupne vykonávaný pre každý jeden pixel ako už bolo vyššie zmienené.

Ukážka zdrojového kódu ktorý implementuje rátanie filtru s konvolučným jadrom na CPU je nasledovná (Text 1):

```
tResult Value = Denom * (
        2 * tResult(SrcImage(x + 1, y))
        + tResult(SrcImage(x + 1, y - 1))
        + tResult(SrcImage(x + 1, y + 1))
        - 2 * tResult(SrcImage(x - 1, y))
        - tResult(SrcImage(x - 1, y - 1))
        - tResult(SrcImage(x - 1, y + 1))
        );
```

Text 1: 2D sobelov filter pre CPU.

Premenná Value obsahuje výslednú hodnotu pre daný filter. Premenná SrcImage obsahuje hodnoty pixlov pôvodného obrázka. Premenné x a y reprezentujú súradnice práve rátaného bodu. Hodnota Denom obsahuje hodnotu ktorou je potrebné celý obrázok vydeliť, v danom filtri ide o hodnotu 0.25. Z algoritmu je tiež možné vidieť že nie je možné rátať výsledné hodnoty okrajových bodov pretože pre výpočet je potrebné body mimo pôvodný obraz. Tento „rám“ ktorý nie je možné zrátať sa líši v závislosti od použitého filtra. Implementácia na grafickej karte je nasledovná (Text 2):

```
__kernel void gpu_sobelX(__global const float* restrict inputImage,
        __global float* restrict outputImage, const int width)
 {
    int gid = get_global_id(0);
      outputImage[gid]   =   (2 * inputImage[gid + 1]
                 + inputImage[gid - width + 1]
                 + inputImage[gid + width + 1 ]
                 - 2 * inputImage[gid - 1]
                 - inputImage[gid - width - 1]
                 - inputImage[gid + width - 1] )/4;
 };
```

Text 2: 2D sobelov filter v OpenCL.

Výpočet je trochu iný, pretože pre výpočet bol využitý jednorozmerný vektor do ktorého boli hodnoty z obrázku uložené, jednotlivé body v x súradnici ležia indexami za sebou samotný bod ktorý je rátaný je označený indexom gid ktorý je pre každý beh cyklu unikátny. Program je zavolaný taký počet krát aby sa zrátali všetky body. Šírka obrázku je označená premennou „width“. Týmto spôsobom je možné bod ktorý je na CPU indexovaný ako *(x + 1, y - 1)* indexovať s využitím nasledovného vzťahu *gid - width + 1.* vzťah *gid + 1* odpovedá *x+1* vzťah *-width* zodpovedá vzťahu *y-1.*

6.1.2 Implementácia Gaussovho vyhladzovacieho filtru

Tento filter je možné jednoducho rátať pomocou dvoch filtrov, každý v samostatnej osi, preto sa ďalej budem zaoberať výpočtom filtru v jednej osi. Prvým predpokladom pre výpočet celého filtru v oboch osiach je výpočet konvolučného jadra. V tomto prípade ide o vektor. Pomocou uplatnenia tohto vektoru v dvoch osiach získame celkový filter pre 2D prípadne pre 3D priestor. Pokiaľ by sme chceli implementovať filter ktorý bude mať pre X a pre Y rôzne vzdialenosti, napríklad z dôvodu vyššej citlivosti v jednej osi s čoho by vyplývalo aj viacej bodov je možné vektor upraviť a použiť dva vektory o rôznych dĺžkach každý pre príslušnú os.

Výpočet vektoru nie je potrebné a ani vhodné implementovať do grafickej karty pretože tento vektor sa ráta iba jeden krát a réžia pre jeho výpočet na grafickej karte by bola extrémne vysoká v porovnaní s klasickým výpočtom na CPU. Tento vektor bude do grafickej karty odovzdaný prostredníctvom vstupného parametru, podobne ako obrázok nad ktorým bude filtrácia vykonávaná.

Na prepísanie do grafickej karty nie je potrebné robiť veľké zásahy, opäť som zvolil odovzdanie do grafickej karty pomocou jednorozmerného vektoru ako to bolo v predchádzajúcom prípade. Potom je tento algoritmus nasledovný (Text 3):

```
__kernel void gpu_gaussX(__global const float* inputImage,
            __global const float* core, __global float* outputImage)
{
   int gid = get_global_id(0);
   int i=0;
   float Sum =0;
   for (i=0; i<Size; i++)
   Sum+=core[i]*inputImage[gid+i-Half];
      outputImage[gid]    = Sum;
};
```

Text 3: 2D Gaussov filter v OpenCL.

Ide o najjednoduchšiu možnú variantu prepisu do grafickej karty, problémom je že vstupné dáta sú reprezentované pomocou jednorozmerného vektoru a pri výpočte kedy sa global id mení postupne od najnižších indexov až po najvyššie sa prekrývajú požiadavky na prístup do globálnej pamäte. V tomto prípade sa jednotlivé jadrá navzájom blokujú a hodnota načítaná jedným jadrom je po výpočte zahodená a je znovu načítaná druhým jadrom. V tomto prípade sa čas výpočtu pre obrázok s rozmermi 1024*1024 vyšplhal na závratných 33ms. V prípade implamentácie pre Y os kedy sa jednotlivé požiadavky na prístup k pamäti neprekrývajú je čas spracovania daného obrázku približne 2ms. Podrobnejšie výsledky je možné vidieť v kapitole 7.2, pre maximálnu elimináciu práce s globálnou pamäťou je vhodné načítať si do lokálnej pamäte čo najväčší segment dát z ktorého je

možné vyrátať čo najviac bodov výsledného obrazu. Prácu s pamäťou možno pokladať za najzdĺhavejšiu časť výpočtu.

Pre implementáciu s využitím lokálnej pamäte je zdrojový kód nasledovný (Text 4):

```
__kernel void gpu_gaussX(__global float* inputImage,
                        __global const float* core,
                        __global float* outputImage,
                        __local float* LocalMemory)
{
    const int g0 =  get_global_id(0);
    const int s0 =  get_global_size(0);
    const int g1 =  get_global_id(1);
    const int gid = g0+s0*g1;
    const int lid = get_local_id(0);
    event_t event = async_work_group_copy(LocalMemory, (__global float*)(inputImage+gid-lid-
Half), (get_local_size(0)+Size), 0);
    const int inv_gid = g1+s0*g0;
    wait_group_events(1,&event);
    int i=0;
    float Sum =0;
    for (i=0; i<Size; i++)
    {
        Sum+=core[i]*LocalMemory[lid+i];
    }
    outputImage[gid]    = Sum;
}
```

Text 4: 2D Gaussov filter v OpenCL s lokálnou pamäťou.

Pri lokálnej variante, som využil asynchrónne kopírovanie dát do lokálnej pamäte s čakaním na jej vykonanie. Až v tomto okamihu sú dáta pripravené na spracovanie a pracovná skupina vypočíta priradené výpočty.

6.2 3D grafické filtre

Tieto filtre sa rátajú podobne ako v prípade 2D filtrov, ako je zmienené v kapitole 2.3 je možné 2D grafický filter rozložiť na filtrovanie v jednotlivých osiach, takže ide o separabilné filtre. Pre hranové

je potrebné rozšíriť maticu z 2D priestoru do 3D priestoru kedy dostávame kocku s číslami ktorá nám reprezentuje samotné konvolučné jadro. Pre výpočet je ešte vhodnejšie použiť lokálnu pamäť pretože pre výpočet jedného bodu je potrebné načítať väčšie množstvo čísel z pamäte. Celkové dáta v trojrozmernom priestore taktiež predstavujú výrazne väčší objem pre spracovanie, v porovnaní s 2D obrazom.

Pri spracovávaní 2D obrazových dát je pravdepodobné že celkový čas potrebný pre výpočet na CPU bude nižší ako v prípade celkového času potrebného pre spracovanie na grafickej karte. Najdlhší čas pri výpočte na grafickej karte je samotná kompilácia a prípravu programu pre grafickú kartu ktorú je potrebné vykonávať pri každom spustení programu. V prípade že je použitý filter viacnásobne s rôznymi vstupnými dátami je možné fixný čas potrebný pre kompiláciu programu zanedbať a uvažovať iba čas potrebný pre filtrovanie jedného z obrazov. Napríklad by bolo možné jednotlivé grafické filtre dopredu kompilovať pri spustení finálneho programu, takže spustenie by bolo dlhšie ale práca s programom by bola následne rýchlejšia ako v prípade že by bol pre výpočet použitý procesor.

Pre spracovanie väčšieho objemu dát, respektíve pri spracovaní jedného súboru ktorý môže obsadiť v pamäti viac než štvrtinu celkovej kapacity je potrebné si uvedomiť ako sa táto pamäť bude správať v prípade požiadaviek na načítanie dát, a tiež akým spôsobom je v pamäti uložený samotný obrázok. V prípade že spracovávame dvojrozmerný obrázok je v pamäti uložený ako jednorozmerný vektor. Pokiaľ sa pozeráme na prvky v rámci jedného stĺpca alebo v jednom riadku môžeme povedať že tieto prvky sa v pamäti nachádzajú jeden za druhým, v závislosti na tom či sú uložené vo formáte s prioritou stĺpcov alebo riadkov.

Ako už bolo spomenuté nie sú použité štruktúry pre 2D alebo pre 3D vstupné dáta, ale sú uložené jednoducho ako jednorozmerný vektor (Tabuľka 2). Preto môžeme vravieť že dáta v jednotlivých riadkoch ležia za sebou, dáta v jednotlivých stĺpcoch sú od seba potom vzdialené o počet prvkov v jednom riadku, a dáta na rovnakej súradnici vrámci jednotlivých rezov sú od seba vzdialené počtom bodov v jednom reze.

0	1	2	3	4	5	6	7	8	9
10	11	12	13	14	15	16	17	18	19
20	21	22	23	24	25	26	27	28	29
30	31	32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47	48	49
50	51	52	53	54	55	56	57	58	59
60	61	62	63	64	65	66	67	68	69
70	71	72	73	74	75	76	77	78	79
80	81	82	83	84	85	86	87	88	89
90	91	92	93	94	95	96	97	98	99

Tabuľka 2: Uloženie dát pre 2D obrázok.

Prvým problémom je načítavanie súčasne tých istých bodov rôznymi jadrami grafickej karty, druhým problémom je načítavanie dát ktoré ležia ďaleko od seba. V takomto prípade je načítanie jedného bodu časovo náročnejšie.

6.3 Použitie lokálnej pamäte

Pre zrýchlenie spracovania som využil prenos údajov do lokálnej pamäte. Pred presunom je potrebné zvážiť, ktoré údaje sa budú ukladať do lokálnej pamäte. V 2D variantách filtrov sú údaju uložené po riadkoch za sebou vo vektore. V prípade horizontálneho filtru sa do lokálnej pamäte presunie jeden riadok, a vypočítajú sa body na tomto riadku. V prípade vertikálneho filtru sa do lokálnej pamäte presunie viacej riadkov, alebo časť viacerých riadkov a vypočítajú sa možné body obrázku. V 3D variantách filtrov funguje prenos podobne, a v smere cez viacero snímkov sa načítajú dáta na rovnakej súradnici ale cez viacero snímkov a spočítajú sa možné body.

7 Praktické testy grafických filtrov a dosiahnuté výsledky

Upravil som program pre filtrovanie a skúšal som ho spúšťať s rôznymi parametrami.

7.1 Testovanie a meranie

Meranie časov výpočtu pre grafické filtre je náročnejšie ako pri meraní na CPU. Pokiaľ ide o meranie času na CPU stačí zmerať čas výpočtu samotného algoritmu, vstupné dáta a výstupné dáta sú už uložené v pamäti RAM a nie je potrebné ich po počítaní nikam presúvať, taktiež je jasné že program pre CPU je už dopredu pripravený a nie je potrebné ho nanovo kompilovať. Pri meraní času výpočtu na GPU je potrebné uvažovať aj čas potrebný pre kompiláciu programu do grafickej karty priamo na mieru, kopírovania dát do grafickej karty a taktiež načítanie spracovaného výstupu naspäť do RAM.

Výsledky je pomerne ťažké porovnať s výsledkami ktoré sa uvádzajú na internete pretože sú často porovnávané len samotné časy výpočtu. Pre grafickú kartu tiež existuje niekoľko možných prístupov pre zrýchlenie spracovania dát. Napríklad je možné presunúť balík dát do lokálnej pamäte. Pri využití lokálnej pamäte sú výpočty opäť rýchlejšie ale je potrebný zarátať čas potrebný na presun, tento je v rade testov zanedbávaný, preto sa pokúsim uviesť výsledky pre jednotlivé princípy a navzájom ich porovnať.

Pre meranie časov výpočtu som použil knižnicu chrono. Do pôvodného VPL filtračného modulu, a do nového GPU filtračného modulu som pridal nasledujúce makrá (Text 5):

```
#define INIT_TIMER auto start =
      std::chrono::high_resolution_clock::now();
#define START_TIMER  start =
      std::chrono::high_resolution_clock::now();
#define STOP_TIMER(name)
      std::cerr << "RUNTIME of " << name << ": " << \
      std::chrono::duratioin_cast<std::chrono::nanoseconds>( \
      std::chrono::high_resolution_clock::now()-start \
      ).count()/1000000.0 << " ms " << std::endl;
```

Text 5: Makrá pre meranie času.

Pomocou ktorých meriam dobu na vykonanie požadovaného úseku v milisekundách. Výsledné časy sú zapísané na štandardný chybový výstup.

Ďalej som implementoval pomocné shell skripty ktoré spúšťajú požadovaný filter s požadovanými parametrami a veľkosťou vstupu na pôvodnej implementácii pre CPU, a novej implementácii pre GPU a odkladajú štandardný chybový výstup do súboru. Tento shell skript je spúšťaný zo skriptu v jazyku Python3, ktorý spúšťa jednotlivé testy viac krát (bežne 100 krát) a vytvára tabuľku dát do .csv súboru. Z nameraných dát sa potom vytvorili grafy ktoré možno vidieť v tejto práci.

7.2 Dosiahnuté výsledky

Pre testovanie bol použitý počítač s nasledovnými parametrami.

CPU: Intel i5-4200U CPU @ 1.60GHz

GPU: Nvidia GeForce 840M

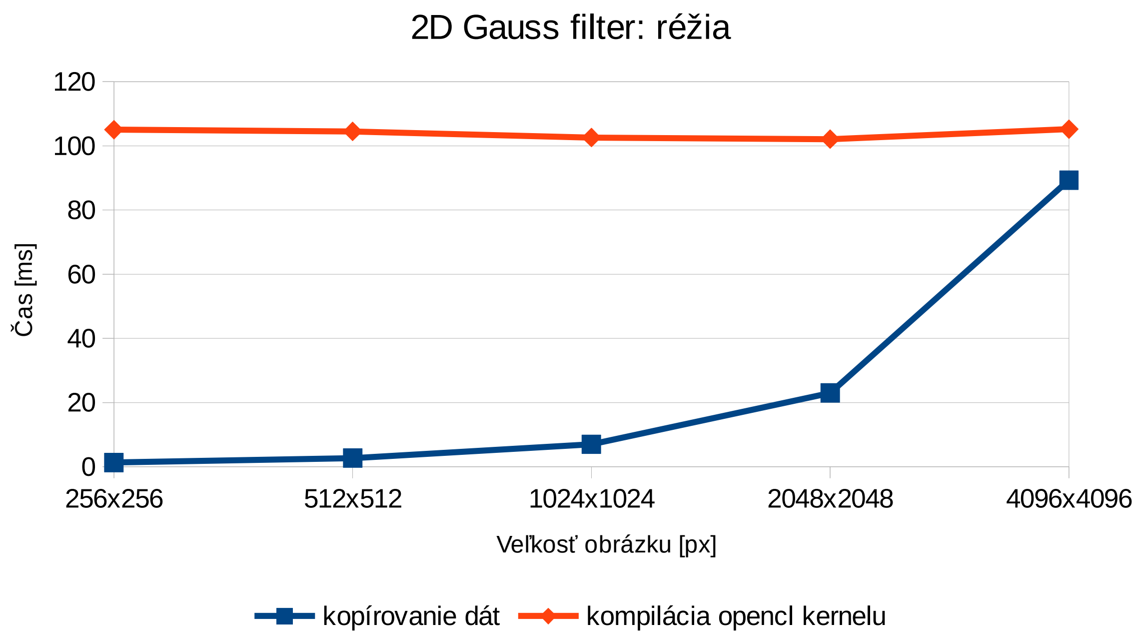
Z informácií ktoré je možné nájsť na internete s prvého pohľadu vyplýva že spracovanie na grafickej karte je vo všetkých prípadoch výhodné a rýchlosť spracovania je niekoľko násobne vyššia ako v prípade spracovania rovnakej operácie na CPU. Avšak často ide len o uvažovanie samotného času potrebného pre vykonanie daného úkonu. V praxi je okrem výpočtu potrebné vykonať i ďalšie operácie.

Ako bolo už skôr spomenuté jednou z výhodou je že program je vytvorený presne pre dané GPU ktorá je v počítači osadená, avšak kompilácia programu zaberá istý čas ktorý je nutné taktiež uvažovať. Ide o fixnú hodnotu a veľkosť vstupného obrázku prípadne počet obrázkov tento čas neovplyvňuje. Takže je možné povedať že pokiaľ by program pracoval stále s rovnakým filtrom bude tento čas zanedbateľný v porovnaní s ostatnými časmi pri viacnásobnom použití filtru.

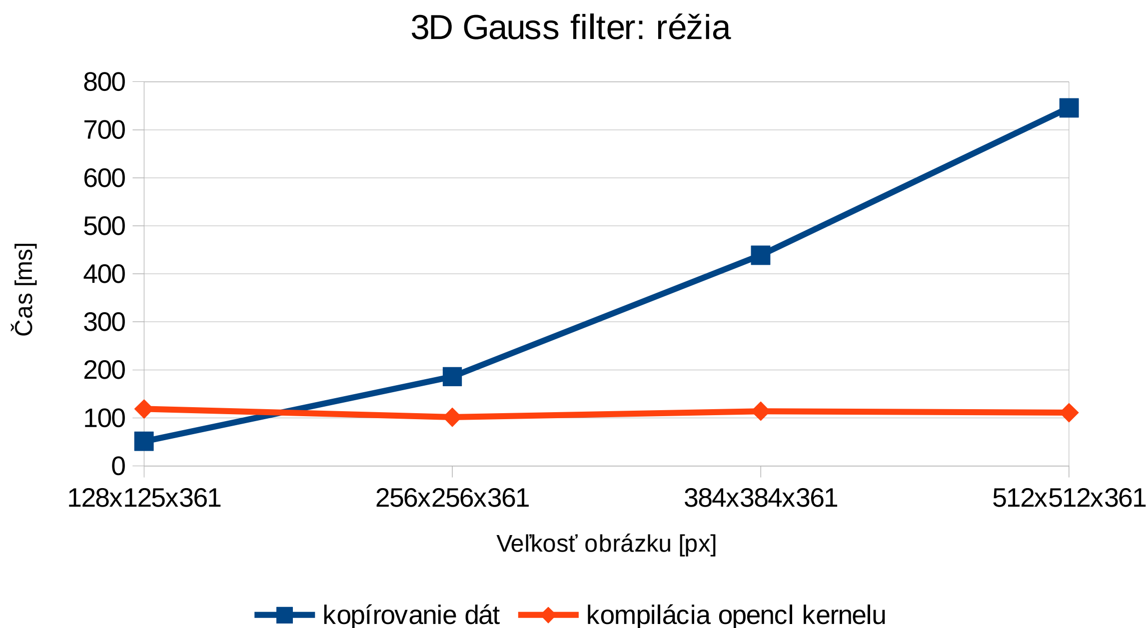
Ďalším nezanedbateľným časom je čas potrebný pre prípravu dát. Vstupné dáta nad ktorými je vykonávaná filtrácia je potrebné presunúť do pamäte GPU aby mohla byť filtrácia vôbec spustená. Po úspešnom dokončení výpočtu je znovu potrebné vykonať presúvanie, tento krát presunutie výsledných dát naspäť do pamäte RAM pre ďalšie spracovanie alebo uloženie . Tento čas s veľkosťou dát samozrejme rastie. Ideálne je pokiaľ by sa nad jednými dátami vykonávalo viacero rôznych filtrov v takomto prípade stačí aby boli vstupné dáta do GPU presunuté iba raz a výpočet filtru by sa vykonal viacero krát, samozrejme výsledky je potrebné opäť presunúť s GPU do RAM. Teoreticky je možné na dáta použiť viacero filtrov za sebou, v tomto prípade by zase stačilo aby bol presunutý celkový výsledok po vykonaní všetkých filtrov. Čas potrebný pre kompilovanie programu a kopírovanie dát pre 2D gaussov filter je možné vidieť na obrázku 13.

V prípade 3D gaussovho filtra (obrázok 14) je čas kopírovania veľkého počtu dát výrazne vyšší ako čas kompilácie a preto je ho možné v niektorých prípadoch zanedbať.

Hodnoty grafu sú vynesené ako priemer zo 100 opakovaní s rovnakými vstupnými dátami. Hodnoty medzi jednotlivými behmi programu sa čiastočne líšili čo je spôsobené predovšetkým operačným systémom ktorý počas behu programu obsluhuje aj iné programy a procesy ktoré súčasne bežia na PC.



Obrázok 13: Réžia 2D Gauss filtra.



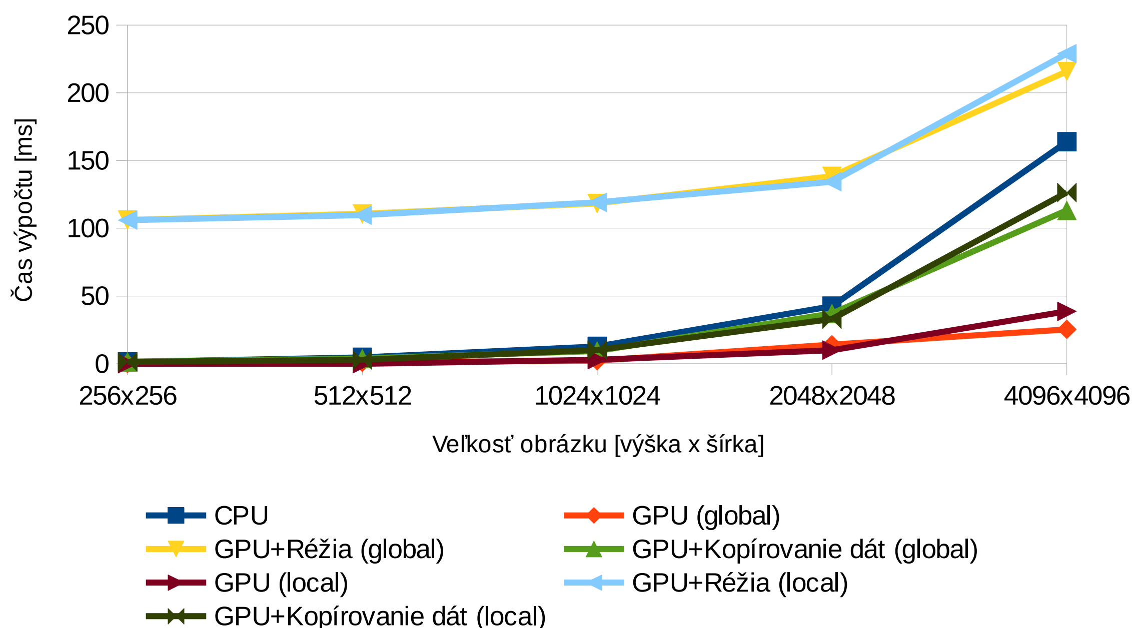
Obrázok 14: Réžia 3D Gauss filtra.

7.3 Testovanie 2D Gaussovho filtra

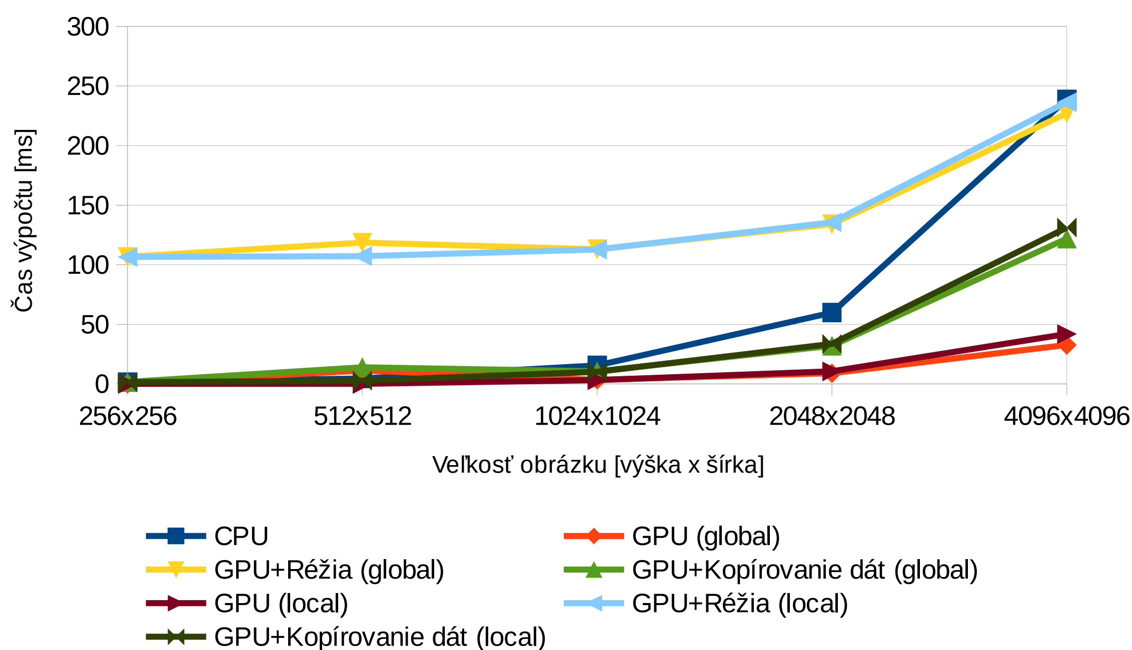
Pre testovanie tohto filtru bol pustený s rôznym okolím pre spracovanie (s rôznym parametrom sigma) pre malé okolie je filtrovaný bod rátaný len s využitím blízkych susedov, s rastúcou hodnotou sa zväčšuje aj počet susedných bodov potrebných pre výpočet a s tým rastie aj celková doba výpočtu filtru. V prípade 2D filtra je vykonaná filtrácia v dvoch osiach a filter je tým pádom jednoduchší v porovnaní s 3D verziou.

V grafoch je testovaných viacero typov algoritmov, najjednoduchšia verzia je s prístupom do globálnej pamäte na GPU, ďalším riešením je možnosť kopírovať dáta do lokálnej pamäte ktorá je menšia a rýchlejšia a následne pracovať s lokálnymi premennými. Pri ladení algoritmu v 2D verzii sa ukázalo že je tiež závislé v ktorej osi je filter vykonávaný, tento fakt je závislý na usporiadaní bodov a ich miesta v pamäti. Ukázalo sa že vhodným riešením je pridať k spracovaniu obrázku transponovanie, spôsobom že je výsledný bod do pamäte uložený tak aby bol nachystaný na rýchle sekvenčné načítanie pri opakovaní filtru. Transponovanie tiež zjednodušilo program pretože dvojnásobné spustenie jedného filtru zabezpečí filtrovanie v oboch osiach a tým pádom je aj kompilácia kernelu rýchlejšia.

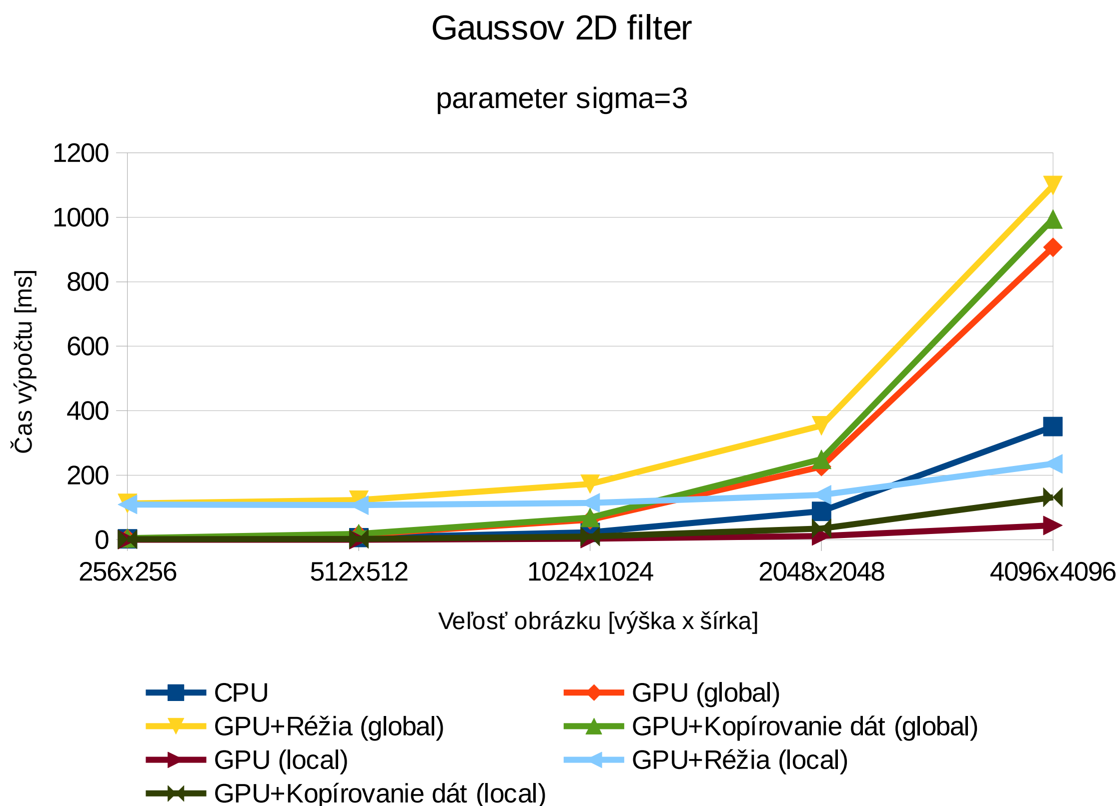
Avšak aj pri použití týchto metód je výpočet zdĺhavejší než na CPU hlavne z dôvodu času potrebného pre kompilovanie programu, Taktiež je možné vidieť, že v grafe pre sigma rovná 3 je spracovanie na GPU s využitím globálnej pamäte zdĺhavejšie než spracovanie na CPU. Toto je spôsobené predovšetkým veľkým odstupom dát v pamäti potrebných pre výpočet jedného pixlu. Zlepšenie nastáva pokiaľ je vybraný balík dát presunutý do lokálnej pamäte a následne sa vyrátajú všetky body ktoré je možné s daného balíka vyrátať.



Obrázok 15: Meranie - 2D gaussov filter, sigma 1.



Obrázok 16: Meranie - 2D gaussov filter, sigma 2.



Obrázok 17: Meranie - 2D gaussov filter, sigma 3.

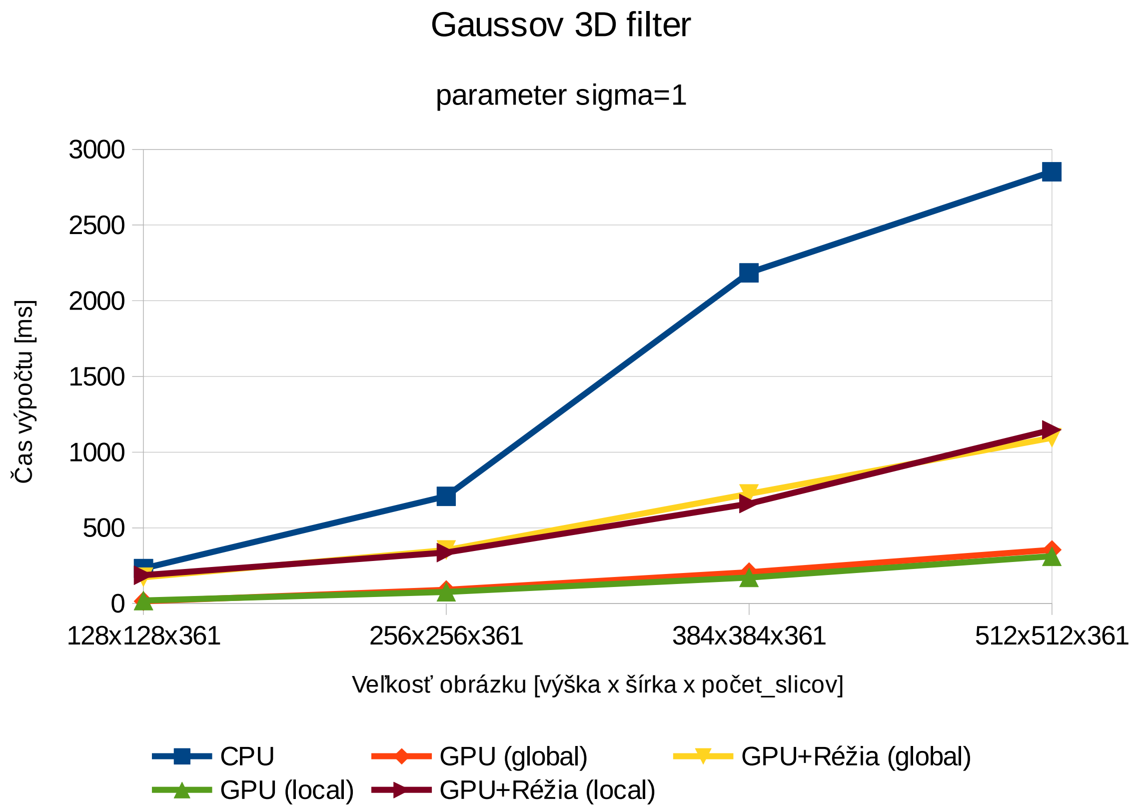
7.4 Testovanie 3D Gaussovho filtru

Podobne ako v predchádzajúcom prípade bol tento filter spúšťaný viacnásobne pre rôzne okolie sigma. So vzrastajúcim okolím sa doba výpočtu na GPU a CPU líšila výraznejšie. Celková veľkosť vstupných dát je limitovaná na 2GB z dôvodu testovania 32 bitovej aplikácie. V prípade väčších obrazových dát je potrebné dáta rozdeliť na úseky a rátať po častiach.

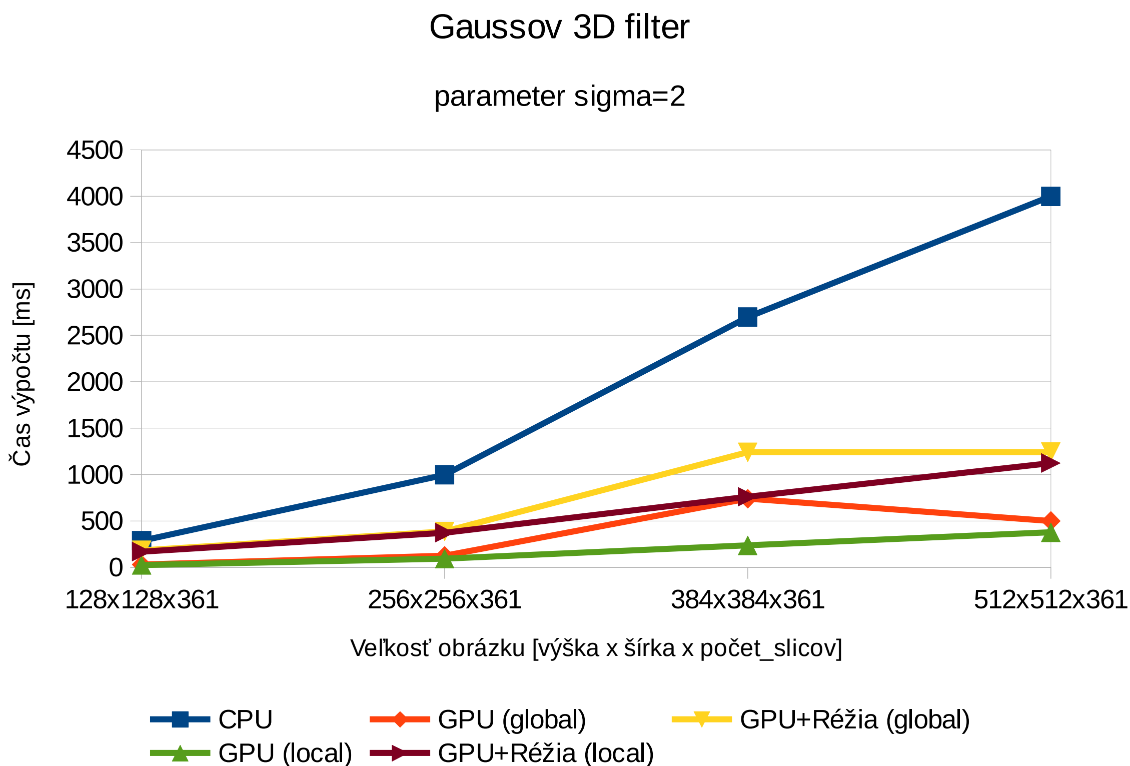
V tomto prípade je tiež vidieť že optimalizácia na grafickej karte s využitím lokálnej pamäte dosahuje taktiež lepšie výsledky podobne ako v predošlom prípade.

Tiež je možné vidieť že čím je výpočet filtra zložitejší tým viac sa prejavuje rozdiel v čase pri využití lokálnej pamäte na GPU v porovnaní s priamym prístupom do globálnej pamäte GPU. Ako problematické sa taktiež ukázalo kopírovanie dát do lokálnej pamäte. V prípade filtru v X osi je potrebné načítať balík dát ležiaci v pamäti za sebou. V prípade filtrácie v Y ose sa načítava balík dát kde jednotlivé body ležia vo vzdialenosti ktorá zodpovedá počtu bodov v X osi.

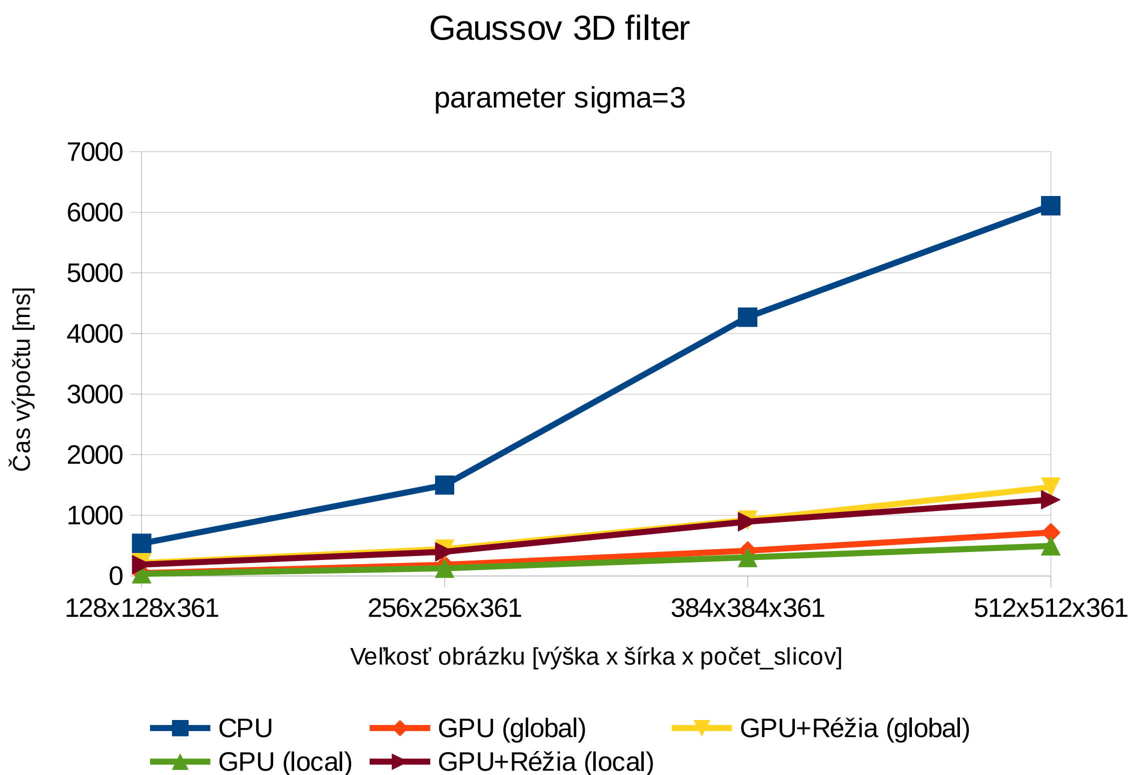
V prípade Z filtru sa opäť načítavajú dáta ktoré sú od seba uložené ďaleko, tentokrát sú vzdialené o počet bodov v rámci jedného rezu (slice). Z tohto dôvodu sú časy pre filtrovanie v rôznych osiach rôzne. Výpočet v X osi je najrýchlejší a v Z osi najpomalší.



Obrázok 18: Meranie - 3D gaussov filter, sigma 1.



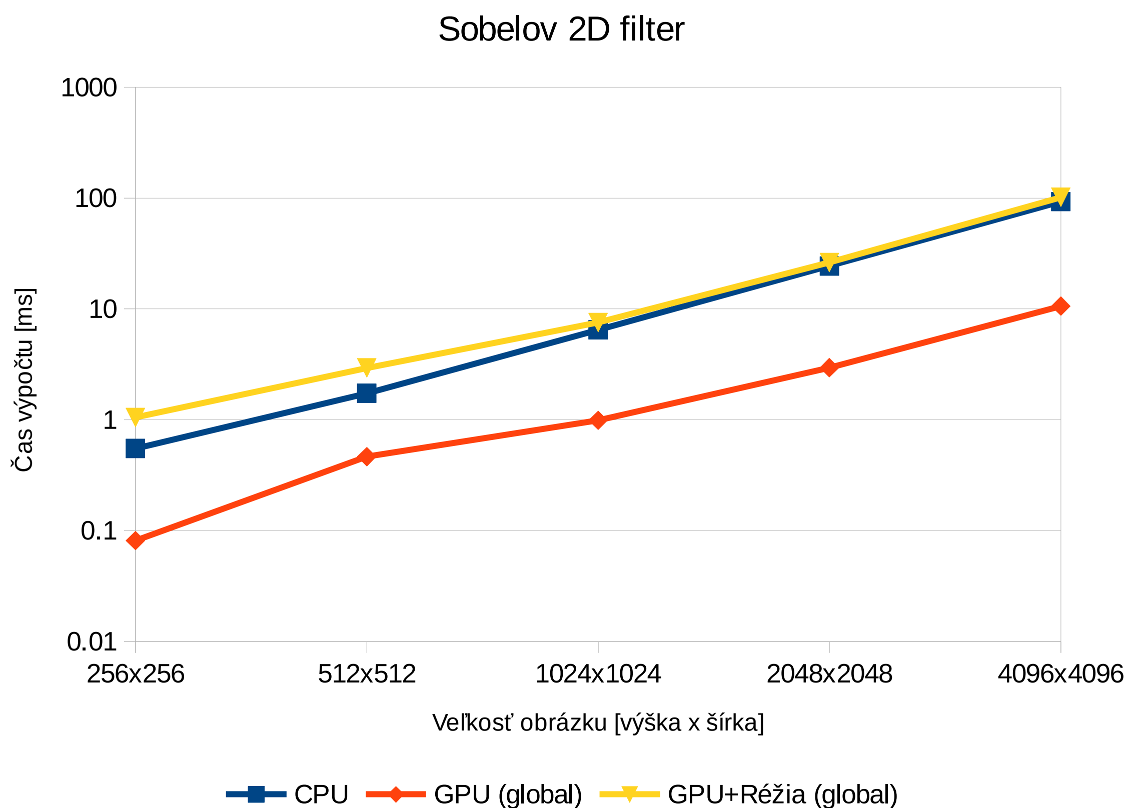
Obrázok 19: Meranie - 3D gaussov filter, sigma 2.



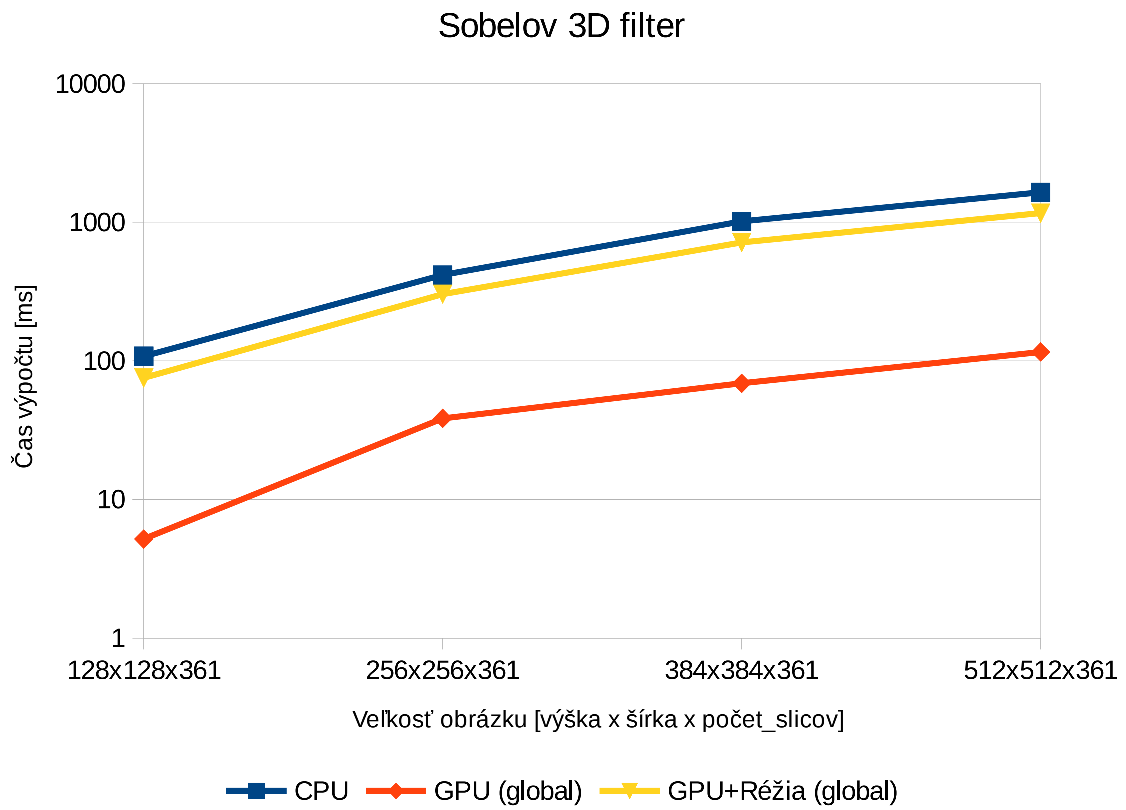
Obrázok 20: Meranie - 3D gaussov filter, sigma 3.

7.5 Testovanie hranového filtru Sobel

V porovnaní s Gaussovím filtrom je výpočet tohto hranového filtru jednoduchší. Pre výpočet sa vždy využívajú body ktoré sú vo vzdialenosti 1 od rátaného bodu. Ako grafy naznačujú výpočet na grafickej karte je zhruba 10x rýchlejší ako na CPU avšak je potrebné zarátať si aj čas potrebný na kopírovanie dát a kompilovanie programu. Vo všeobecnosti je možné povedať že čím je výpočet náročnejší tým výhodnejšie je výpočet paralelizovať.



Obrázok 21: Meranie - 2D sobelov filter.



Obrázok 22: Meranie - 3D sobelov filter.

8 Záver

Cieľom práce bolo overiť možnosti akcelerácie spracovania 2D a 3D obrazových dát s použitím GPGPU. Počas riešenia bolo overených viacero prístupov ktoré sú štandardne používané. Ukázalo sa že doba výpočtu je veľmi závislá na spôsobe ktorým je program napísaný. Využitie lokálnej pamäte dokázalo výpočet v porovnaní s globálnou pamäťou v niektorých prípadoch výrazne zrýchliť, avšak pokiaľ je pamäťový blok pre presunutie zle zvolený môže spôsobiť aj spomalenie celého výpočtu.

Taktiež sa ukázalo že niektoré metódy ako napríklad rozvinuté cykli neprispeli k zrýchleniu algoritmu, taktiež využitie dátového typu float 4 neprispelo k zrýchleniu, ale práve naopak ku miernemu spomaleniu celkového výpočtu. Keďže bol algoritmus testovaný na GPU od spoločnosti NVIDIA ktorá nemá priamu podporu pre prácu s vektormi je tento výsledok očakávaný , zaujímavé by bolo vidieť rozdiel na grafickej karte od spoločnosti AMD ktoré majú podporu pre prácu s vektormi.

Počas riešenia sa tiež ukázalo že mnohé informácie ktoré sú udávané nerátajú napríklad s časmi potrebnými pre presúvanie, predovšetkým príklady s globálnou pamäťou ukazovali výrazné zrýchlenie, ktoré by bolo možné iba v prípade zanedbania času potrebného na presunutie dát s globálnej pamäte GPU do lokálnej pamäte. Tieto presuny sa ukázali ako najpomalšie s celého procesu filtrácie. Preto je výpočet pomocou GPU vhodný predovšetkým na veľmi zložité operácie nevyžadujúce neustále presúvanie dát. Napríklad použitie pri výpočte a učení neurónových sietí by mohlo byť ešte výraznejšie zrýchlené v porovnaní so zrýchlením pri grafických filtroch.

V budúcnosti by bolo zaujímavé zamerať sa na ďalšiu možnosťou prístupu k dátam v GPU. Je možné využitie štruktúr pre 2D alebo 3D obrázky, tento spôsob využíva na GPU špeciálne pamäťové štruktúry ktoré sú určené pre spracovávanie textúr. Výhodou je že podľa typu grafického filtra dokáže GPU prichystať dáta ktoré budú počas filtrácie využívané. Nevýhodou je výrazne vyššia zložitosť pri programovaní a tiež fakt že tieto štruktúry je možné používať iba v globálnej pamäti.

Literatúra

[1] PIKORA, Jan. Implementace grafických filtrů pro zpracování rastrového obrazu, Brno, 2008. Bakalářská práce. Masarykova Univerzita, Fakulta informatiky.

[2] ČERVEŇANSKÝ, Michal RNDR. NON-GRAPHICS COMPUTATIONS ON GPU IN THE OPENGL ENVIRONMENT, Bratislava, 2010. Dizertačná práca. Univerzita Komenského v Bratislave, Fakulta matematiky, fyziky, a informatiky. Katedra aplikovanej informatiky

[3] REDA, Khairi. *A study of OpenCL image convolution optimization* [online]. [vid. 2017-04-25]. Dostupné z: https://www.evl.uic.edu/kreda/gpu/image-convolution/

[4] KHRONOS OPENCL WORKING GROUP. *The OpenCL Specification* [online]. 2011 [vid. 2017-04-03]. Dostupné z: https://www.khronos.org/registry/OpenCL/specs/opencl-1.1.pdf

[5] CACEK Pavel. Akcelerace algoritmů komprese dat s využitím GPU, Brno, 2013. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií.

[6] *3dimlab / vpl — Bitbucket* [online]. [vid. 2017-03-22]. Dostupné z: https://bitbucket.org/3dimlab/vpl

[7] CUDA C Programming Guide [online]. nedatováno [vid. 2017-03-22]. Dostupné z: http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#axzz4PbO2SXOC

[8] BLÁZSOVITS, Gábor. *Interaktívna učebnica spracovania obrazu* [online]. 2006 [vid. 2017-05-02]. Dostupné z: http://dip.sccg.sk/

[9] *CUDA Parallel Computing | What is CUDA? | NVIDIA UK* [online]. [vid. 2017-03-22]. Dostupné z: http://www.nvidia.co.uk/object/cuda-parallel-computing-uk.html

Zoznam príloh

Príloha 1. DVD

Príloha 1. DVD

Obsah DVD je rozdelený do adresárovej štruktúry:

- BP obsahuje samotný text technickej správy v odf a pdf formáte
- PLAGAT obsahuje návrh prezentačného plagátu
- TESTY obsahuje meracie BASH a PYTHON skripty a namerané hodnoty
- VPL pôvodná VPL implementácia, commit: 13cb0a8
- VPL_xkozov01 zmeny potrebné pre openCL implementáciu a samotná implementácia

Ďalšie informácie je možné nájsť v súbore README.txt