VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

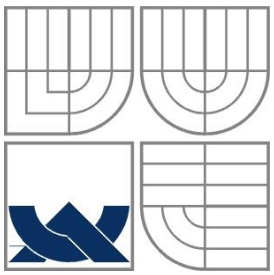
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

KNIHOVNA PRO ZPRACOVÁNÍ OBRAZU V GPU

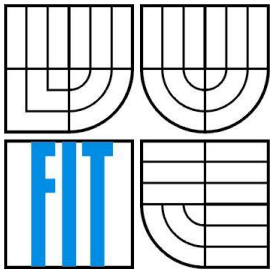
BAKALÁŘSKÁ PRÁCE
BACHELOR´S THESIS

AUTOR PRÁCE MICHAL ČERMÁK
AUTHOR

BRNO 2008



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

KNIHOVNA PRO ZPRACOVÁNÍ OBRAZU V GPU

GPU IMAGE PROCESSING LIBRARY

BAKALÁŘSKÁ PRÁCE

BACHELOR´S THESIS

AUTOR PRÁCE — MICHAL ČERMÁK
AUTHOR

VEDOUCÍ PRÁCE — Doc. RNDr. Ph.D. PAVEL SMRŽ
SUPERVISOR

BRNO 2008

Zadání bakalářské práce/7060/2007/xcerma18

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav počítačové grafiky a multimédií Akademický rok 2007/2008

Zadání bakalářské práce

Řešitel: **Čermák Michal**
Obor: Informační technologie
Téma: **Knihovna pro zpracování obrazu v GPU**
Kategorie: Zpracování obrazu

Pokyny:

1. Prostudujte dostupnou literaturu zabývající se technologií CUDA.
2. Prostudujte literaturu na téma algoritmy pro zpracování obrazu.
3. Navrhněte knihovnu pro zpracování obrazů v GPU pomocí technologie CUDA.
4. Implementujte navrženou knihovnu a sadu algoritmů pro zpracování obrazu (nutno konzultovat).
5. Vyhodnoťte dosažené výsledky.

Literatura:

- Žára, J. a kol.: Počítačová grafika principy a algoritmy, Grada, Praha 1992
- Žára, J. Beneš B. Felker P.: Moderní počítačová grafika, Computer Press, 1998
- dále dle pokynů vedoucího

Při obhajobě semestrální části projektu je požadováno:

- Prostudujte dostupnou literaturu zabývající se technologií CUDA.
- Prostudujte literaturu na téma algoritmy pro zpracování obrazu.
- Navrhněte knihovnu pro zpracování obrazů v GPU pomocí technologie CUDA.

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: **Smrž Pavel, doc. RNDr., Ph.D.**, UPGM FIT VUT
Datum zadání: 1. listopadu 2007
Datum odevzdání: 14. května 2008

L.S.

doc. Dr. Ing. Jan Černocký
vedoucí ústavu

LICENČNÍ SMLOUVA
POSKYTOVANÁ K VÝKONU PRÁVA UŽÍT ŠKOLNÍ DÍLO

uzavřená mezi smluvními stranami

1. **Pan**

Jméno a příjmení: **Michal Čermák**
Id studenta: 79067
Bytem: Brtnička 9, 675 27 Předín
Narozen: 07. 05. 1986, Třebíč
(dále jen "autor")

a

2. **Vysoké učení technické v Brně**
Fakulta informačních technologií
se sídlem Božetěchova 2/1, 612 66 Brno, IČO 00216305
jejímž jménem jedná na základě písemného pověření děkanem fakulty:

.....
(dále jen "nabyvatel")

Článek 1
Specifikace školního díla

1. Předmětem této smlouvy je vysokoškolská kvalifikační práce (VŠKP):
bakalářská práce

Název VŠKP: Knihovna pro zpracování obrazu v GPU
Vedoucí/školitel VŠKP: Smrž Pavel, doc. RNDr., Ph.D.
Ústav: Ústav počítačové grafiky a multimédií
Datum obhajoby VŠKP: .....

VŠKP odevzdal autor nabyvateli v:
tištěné formě počet exemplářů: 1
elektronické formě počet exemplářů: 2 (1 ve skladu dokumentů, 1 na CD)

2. Autor prohlašuje, že vytvořil samostatnou vlastní tvůrčí činností dílo shora popsané a specifikované. Autor dále prohlašuje, že při zpracovávání díla se sám nedostal do rozporu s autorským zákonem a předpisy souvisejícími a že je dílo dílem původním.
3. Dílo je chráněno jako dílo dle autorského zákona v platném znění.
4. Autor potvrzuje, že listinná a elektronická verze díla je identická.

Článek 2
Udělení licenčního oprávnění

1. Autor touto smlouvou poskytuje nabyvateli oprávnění (licenci) k výkonu práva uvedené dílo nevýdělečně užít, archivovat a zpřístupnit ke studijním, výukovým a výzkumným účelům včetně pořizování výpisů, opisů a rozmnoženin.
2. Licence je poskytována celosvětově, pro celou dobu trvání autorských a majetkových práv k dílu.
3. Autor souhlasí se zveřejněním díla v databázi přístupné v mezinárodní síti:
   - ☐ ihned po uzavření této smlouvy
   - ☐ 1 rok po uzavření této smlouvy
   - ☐ 3 roky po uzavření této smlouvy
   - ☐ 5 let po uzavření této smlouvy
   - ☐ 10 let po uzavření této smlouvy

   (z důvodu utajení v něm obsažených informací)
4. Nevýdělečné zveřejňování díla nabyvatelem v souladu s ustanovením § 47b zákona č. 111/ 1998 Sb., v platném znění, nevyžaduje licenci a nabyvatel je k němu povinen a oprávněn ze zákona.

Článek 3
Závěrečná ustanovení

1. Smlouva je sepsána ve třech vyhotoveních s platností originálu, přičemž po jednom vyhotovení obdrží autor a nabyvatel, další vyhotovení je vloženo do VŠKP.
2. Vztahy mezi smluvními stranami vzniklé a neupravené touto smlouvou se řídí autorským zákonem, občanským zákoníkem, vysokoškolským zákonem, zákonem o archivnictví, v platném znění a popř. dalšími právními předpisy.
3. Licenční smlouva byla uzavřena na základě svobodné a pravé vůle smluvních stran, s plným porozuměním jejímu textu i důsledkům, nikoliv v tísni a za nápadně nevýhodných podmínek.
4. Licenční smlouva nabývá platnosti a účinnosti dnem jejího podpisu oběma smluvními stranami.

V Brně dne: .....

....

Nabyvatel Autor

Abstrakt

Tato práce se zabývá architekturou grafických karet Nvidia a s ní související programátorské rozhranní CUDA, které je využito při tvorbě knihovny akcelerující algoritmy zpracování obrazu. Velký důraz je kladen na testování výkonnostního zisku oproti optimalizované a používané knihovně OpenCV.

Klíčová slova

GPU, GPGPU, grafická karta, akcelerace, SIMD, paralelní zpracování, zpracování obrazu, FFT, konvoluční filtr, konvoluce, konverze barevného modelu.

Abstract

This work is concerned with architecture of recent Nvidia graphics cards and application programming interface CUDA. That is used to create accelerated image processing library. It place emphasis on testing performance gain compassion with high optimized and used OpenCv library.

Keywords

GPU, GPGPU, graphics card, acceleration, SIMD, parallel computing, image processing, FFT, convolution filter, convolution, color space conversion.

Citace

Čermák Michal: Knihovna pro zpracování obrazu v GPU, Brno, 2008, bakalářská práce, FIT VUT v Brně.

Knihovna pro zpracování obrazu v GPU

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením docenta, Pavla Smrže. Další informace mi poskytl doktor Stanislav Sumec. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……

Michal Čermák

11. května 2008

Poděkování

Chtěl bych využít této příležitosti, abych poděkoval svému vedoucímu, panu docentu Smržovi, za vedení a odbornou pomoc.

Obsah

Seznamy

Grafy

Obrázky

Tabulky

1 Úvod

Současné grafické akcelerátory se staly, díky snaze o vytváření více a více reálných 3D scén, univerzálním výpočetním nástrojem s vysokým výkonem. Díky tomu překročily hranice svého původního využití. Umožnil tak vývoj nové disciplíny GPGPU – General Purpose using of Graphics Processing Unit (využití grafického akcelerátoru k obecným výpočtům). Ta usnadňuje využívat vysoký výkon grafických karet při výpočtů obecných paralelních algoritmů. Je to poměrně mladé odvětví, zaměřené na snadnou akceleraci paralelních výpočtu.

I když je výkon současných více-jádrových procesorů značný, v některých aplikacích není dostatečný. Příkladem může být i zpracování multimediálních dat (v reálném čase), počítačové vidění a jiné. V těchto odvětvích by se hodilo zvýšit výkon často používaných funkcí. To bude úkolem knihovny zpracování obrazu, která je cílem této práce. Důležité je, že její rutiny budou akcelerovány pomocí grafického procesoru.

Grafický čip je stavěn na zpracování minimálně stovek až tisíců nezávislých vláken, kdy každé z vláken zpracovává relativně malé množství dat. Oblast zpracování obrazu je tedy ideálním kandidátem na to, být tímto způsobem urychlována. Výpočty nad jednotlivými pixely jsou spojeny s vlákny, které je zpracovávají.

Upřesňuji, že i když se jedná o knihovnu zpracování obrazu, což je vlastně původní účel grafické karty, je implementována pomocí technik GPGPU a pomocí obecného GPGPU rozhranní Nvidia CUDA. To je rozdíl oproti původním způsobům GPU akcelerace, kdy byly výpočty mapovány na textury. Pro více informací o tomto způsobu doporučuju nastudovat bakalářskou prácí Lukáše Poloka (1) a srovnat tyto dva způsoby.

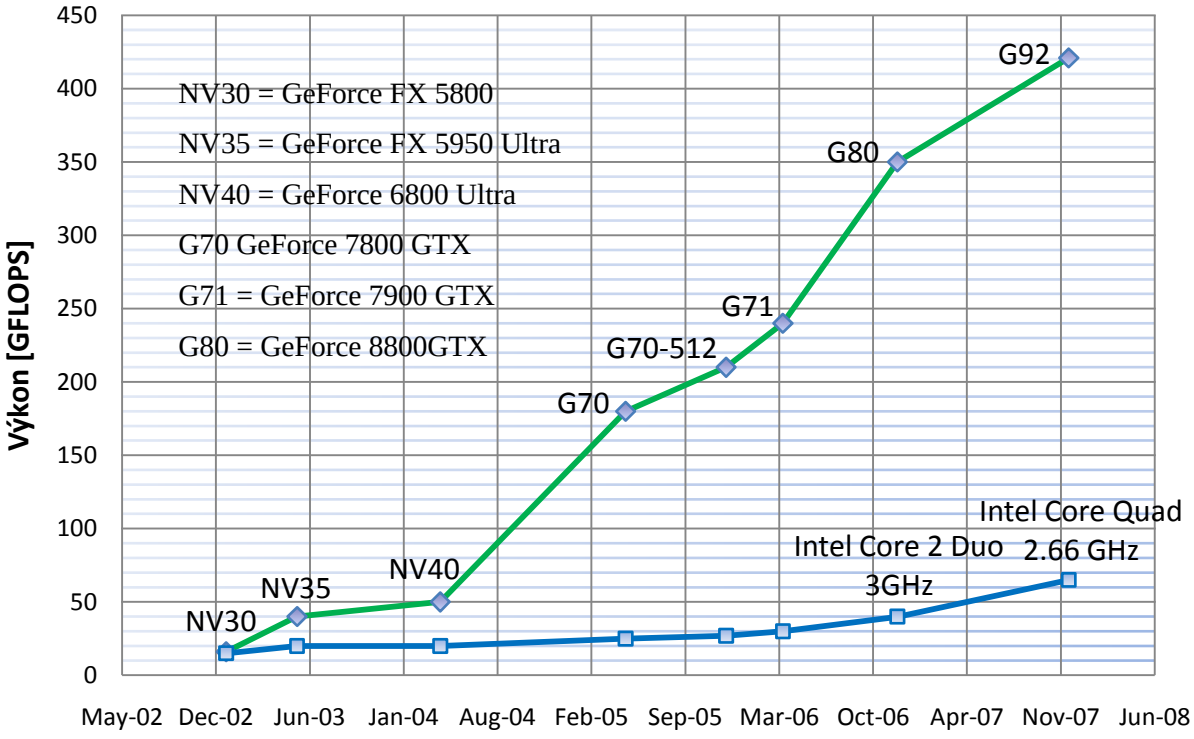
V kapitole bezprostředně následující (2) jsou porovnávány architektury procesoru a grafického akcelerátoru. Diskutovány jsou i rozdíly ve výkonu. Následuje chronologický přehled vývoje grafických čipů a dává je do souvislosti s vývojem GPGPU. Důležitým parametrem, který s výkonem GPGPU aplikací nepřímo souvisí je i rychlost a propustnost komunikačního rozhranní grafické karty. Proto jsou popsány i jeho vlastnosti. Hlavní část této kapitoly je věnována popisu hardwarové implementace grafického čipu a paměťového subsystému. Kapitola 3 popisuje programátorské rozhranní a definuje způsoby efektivního programování grafické karty, které je nutné důsledně dodržovat. Specifikuje způsoby využití sběrnice, paměťového subsystému, management vyrovnávacích paměti a další. Následuje kapitola 4, která se věnuje efektivní implementaci několika funkcí pro zpracování obrazu. Vybrány jsou funkce konverze barevného modelu obrazu, konvoluce a FFT. Poslední 5. kapitola se věnuje hlavně zhodnocení výkonů implementace knihovny a ukazuje možnosti dalšího vývoje. V přílohách (6) čtenář najde tabulky současně používaných GPGPU grafických karet, popis rozhranní knihovny, v textu použitých zkratek, parametry testovací grafické karty a další.

2 Architektura grafické karty

Psaní GPGPU programu předpokládá poměrně hluboké znalosti v oblasti hardware a principů funkce grafické karty. Následující kapitola tyto znalosti čtenáři předkládá. Před tím si, v první podkapitole, dovolím odbočku a srovnám architektury a výkonnost GPU s procesorem.

2.1 Motivace, aneb proč používat GPU

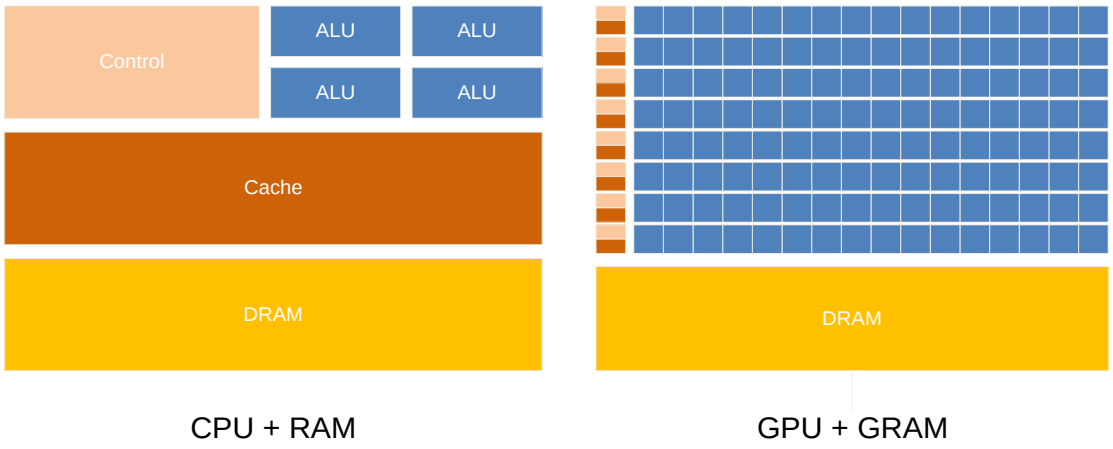
Hlavním důvodem je jednoznačně výkon dnešních grafických karet. V grafu níže lze vidět postupné zvyšování rozdílu výkonnosti GPU a CPU od konce roku 2002. Do budoucna se tento deficit procesoru bude dále zvyšovat. Plány výrobců grafických čipů (AMD a Nvidia) jsou impozantní a není důvod jim nevěřit. Např. ve třetím čtvrtletí roku 2008 nastoupí nová řada karet založených na G200, výkonem přibližujícím se 1TFlops. Naopak výrobci procesorů (Intel, AMD) narazili na frekvenční meze (3GHz) dnešních technologií, a hrubý výkon se zvedá jen zvyšováním počtu procesorových jader. Dokud tedy nepřijde na trh jiná architektura, nůžky mezi výkonem grafických karet a procesorů se budou dále rozevírat. Navíc vývojový cyklus GPU je díky relativně nízkým výsledným taktům krátký. Je možné jej, na rozdíl od procesoru, navrhovat automaticky pomocí počítače.



Graf 1 - Porovnání rychlosti GPU a CPU[1]

[1] Obrázek převzat a aktualizován z (3)

Je zajímavé, že současné čipy s přibližně stejným počtem tranzistorů (700 miliónů) podávají tak rozdílný výkon. Odlišnost je v architekturách, které jsou velmi zjednodušeně zobrazeny na následujícím obrázku.



Obrázek 1 - Porovnání GPU a CPU architektury[2]

Grafický procesor je v podstatě jen velké množství SIMD výpočetních jednotek (jednotlivé řádky v pravé části obrázku). Velikosti cache (tmavě oranžová) jsou omezené. To je dáno tím, že GPU je stavěno pro výpočty s velkou intenzitou aritmetických operací a tím je schopno odstínit zpoždění hlavní paměti. Velká cache tedy ve skutečnosti není potřebná. Stejně tak jsou malé možnosti řízení toku programu. Naopak procesor nabízí sofistikované metody řízení toku vykonávání instrukcí, jako například predikce skoků, dopředné načítání argumentů do cache a jiné. Velikosti cache L2 se pohybují v řádech jednotek MB a tvoří dokonce více než polovinu plochy čipu (u nejnovějších 4jádrových procesorů 12MB).

Díky těmto rozdílům je naprogramování efektivního algoritmu pro GPU o mnoho složitější než CPU. Programátor musí být s konkrétní architekturou velmi dobře obeznámen, aby se vyhnul chybám. Už to, že se jedná o několik SIMD čipu, každá neefektivita v programu způsobí zpomalení běhu velkého množství vláken. Další problémy s efektivitou jsou při přístupu k hlavní a sdílené (paměť na čipu) paměti atd. Více viz další kapitoly.

[2] Obrázek převzat z (3)

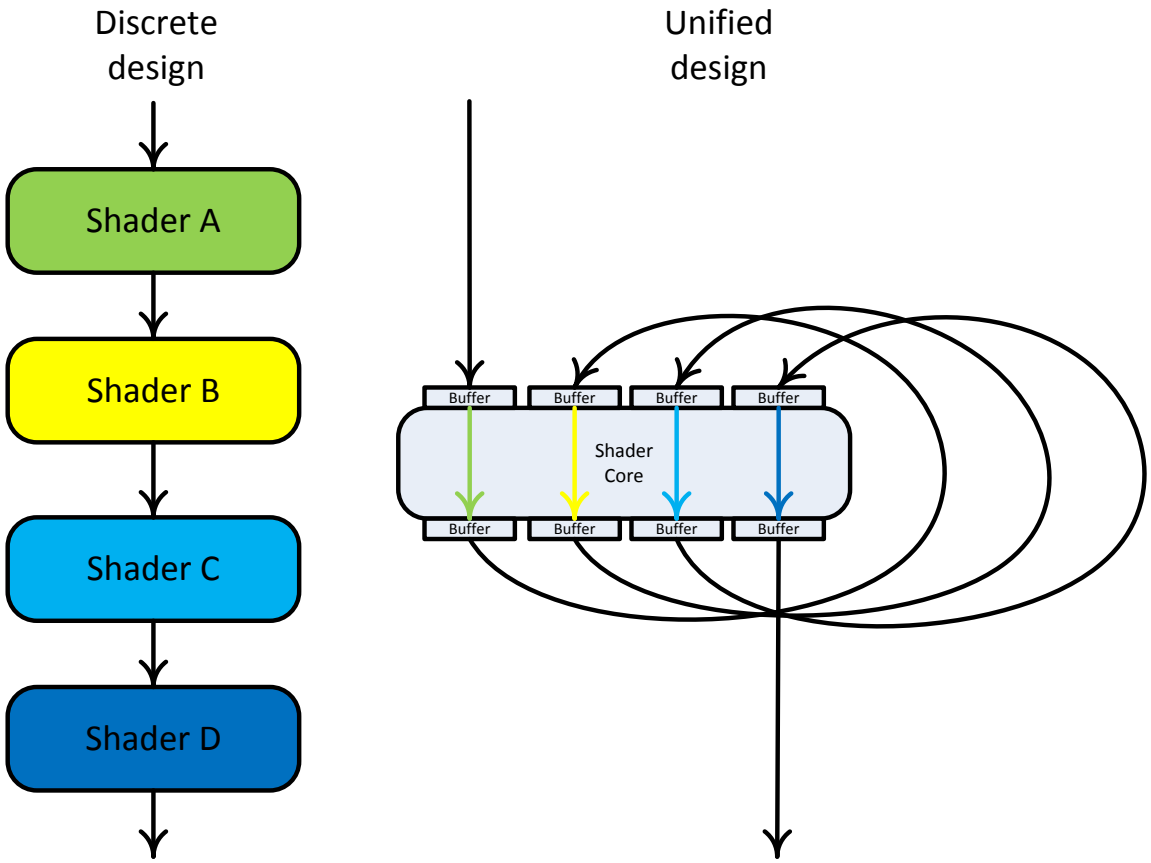
2.2 Stručný pohled na vývoj GPGPU hardwaru

Do roku 2000 dominovaly grafickému trhu karty podporující rozhraní DirectX7, ve kterých vertexové a pixelové výpočty probíhaly hardwarově v TnL (Transform and Lighting) jednotkách, která právě DirectX7 vyžadovala.

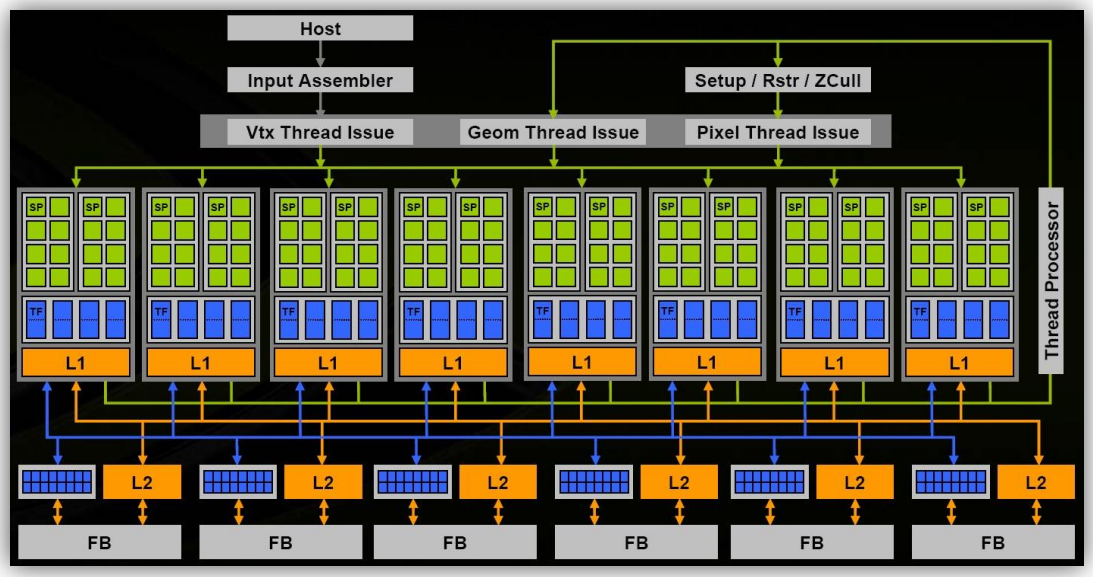
O rok později nastoupily DirectX8 čipy, které jako první podporovaly pixel a vertex shadery. Shader je program, který je aplikován na každý element počítané scény (vrchol, pixel). Díky tomu bylo možné generovat mnohem realističtější scény (jejich tvorba mohla být ovlivňována softwarově). V této chvíli se zrodilo GPGPU, i když programování bylo velice složité. Výpočty musely být definovány pomocí grafických API. Nejčastěji se prováděli v pixel shaderech a byly mapovány na výpočty nad texturami.

DirectX9 karty přinesly další vzestup výkonu GPU, výpočty však musely být neustále mapovány přes grafická API. Přicházejí první implementace a využití. (folding@home – výpočty proteinových bází na kartách ATI x19xx). Právě prostředky dostupné v tomto období využil ve své bakalářské práci také Lukáš Polok. Více informací naleznete v jeho práci (1).

S příchodem shader model 4.0 a DirectX10 přibývají geometry shadery. Tři typy výpočetních jednotek pro troje shadery by znamenalo další zvýšení plochy čipu, které by ale nepřineslo výkon, jen podporu technologií potřebných pro kompatibilitu s DirectX10. Jediný způsob, jakým se lze z této situace dostat, je navržení unifikované výpočetní jednotky, která by mohla být využita pro všechny typy shaderů. Tento princip ilustruje obrázek 2. Z hlediska GPGPU je důležité, že došlo k velkému vylepšení možností shaderů, a díky unifikaci i zvětšení počtu jednotek, které lze využít (u DirectX9 a níže bylo možné použít pouze pixel shadery). Přišly první programovací API speciálně vytvoření pro GPGPU – AMD CtM (Close to metal) a Nvidia CUDA (Compute Unified Device Architecture). Následuje obrázek unifikovaného designu čipu G80 – obrázek 3, který koresponduje s obrázkem 2.



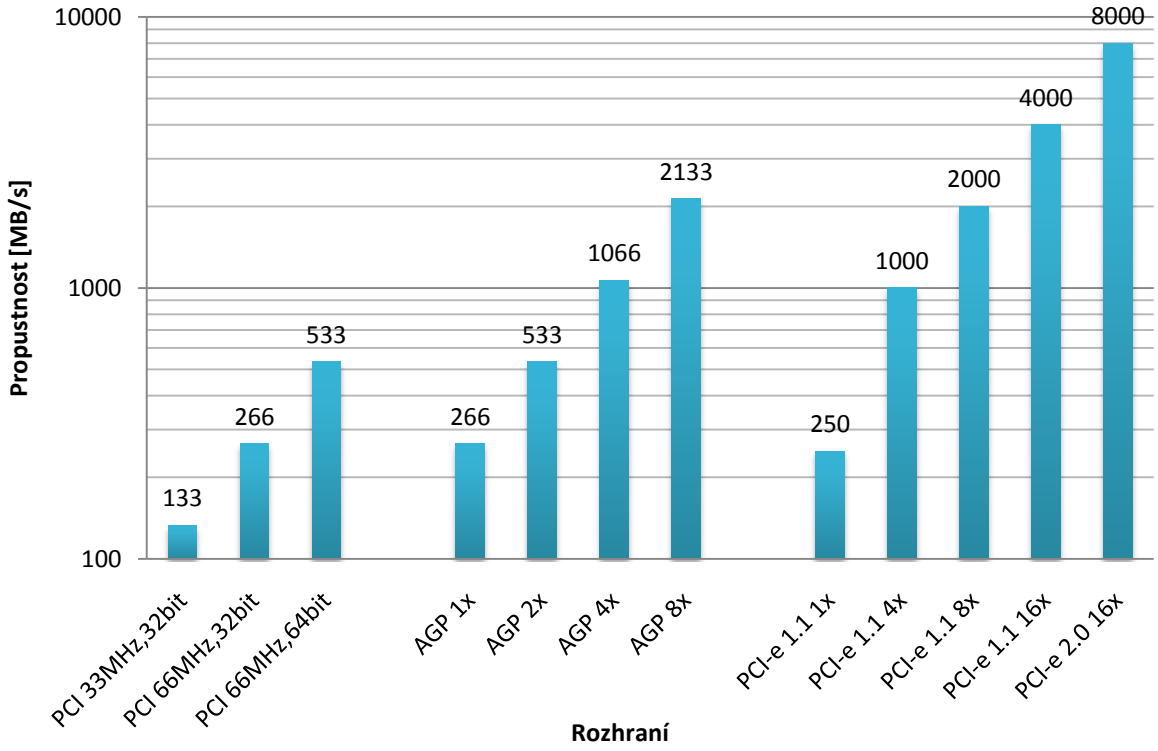
Obrázek 2 - Porovnání diskrétního a unifikovaného designu shaderů GPU



Obrázek 3 - Blokové schéma čipu Nvidia G80 a G92 – ukázka unifikovaného designu

2.2.1 Rozhraní grafických karet

Jedním z faktorů, který umožnil rozvoj GPGPU, je zvyšování rychlosti rozhraní pro připojení grafických karet. V době vlády klasické PCI a první verze AGP byla procesorová sběrnice (FSB) mnohokrát rychlejší než sběrnice grafické karty. Procesor tedy mohl, se započtením zpoždění přenosu dat z paměti, požadovanou úlohu vykonat rychleji, i když vlastní výpočet trval déle. To se ale s příchodem dedikované AGP a následně i PCI-expres změnilo. PCI-expres má ve své nejnovější revizi 2.0 propustnost 500MB/s na jeden kanál, a v případě použití obvyklých 16 kanálu propustnost 8GB/s oběma směry. V této chvíli je limitující spíše propustnost hlavní paměti, která má v době psaní této práce obvyklou propustnost 3-10GB/s.



Graf 2 - Porovnání rozhraní pro připojení grafických karet

2.2.1.1 Alokace paměti

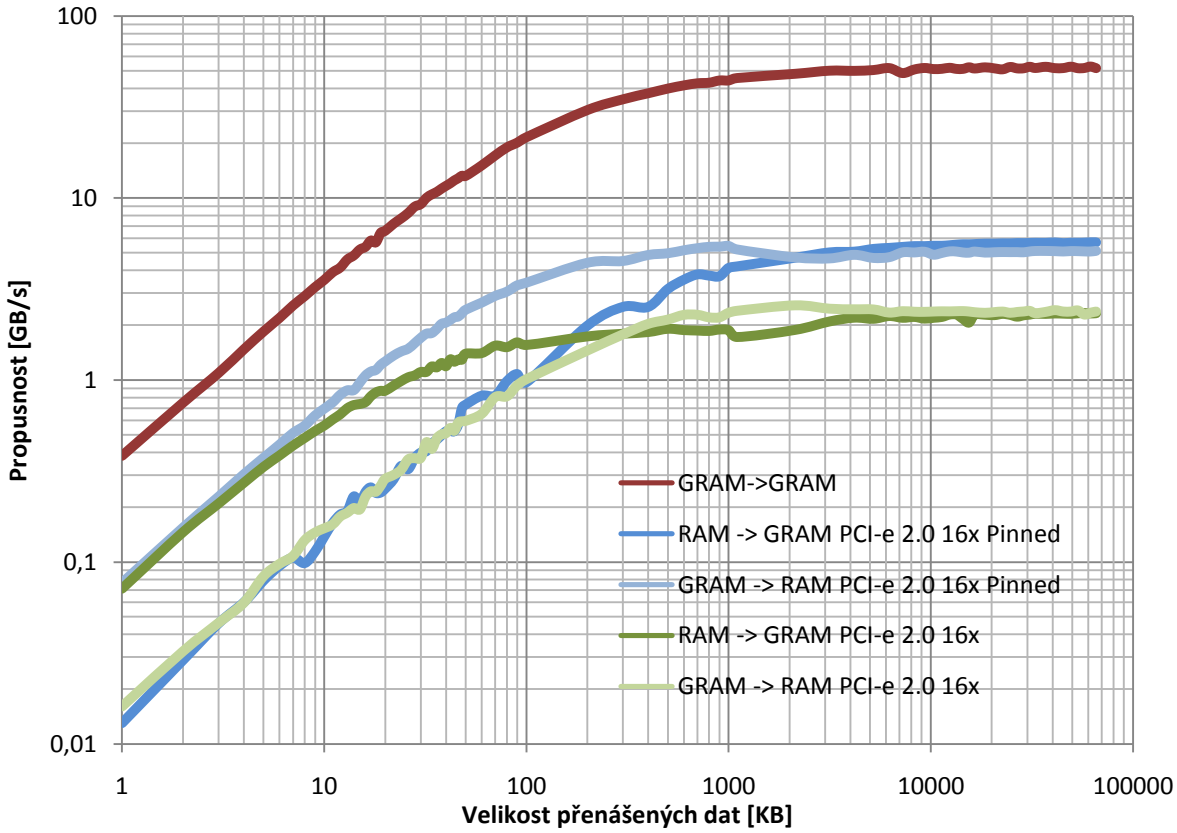
V CUDA API je možné alokovat bloky paměti v hlavní RAM dvěma způsoby. Alokace klasickým způsobem pomocí rodiny funkcí malloc, které vytváří stránkovanou paměť, a tzv. stránkově-uzamčenou (pinned memory). Tato možnost může značně urychlit DMA přenosy obecně. Pro více informací o této alokaci viz (2). To se potvrdilo i měřením a ukazují ji i následující graf, ve kterých jsou porovnány přenosy těmito dvěma způsoby. Pro měření je použita testovací grafická karta a počítačová sestava.

Modré odstíny funkčních závislostí propustnosti náleží přenosu stránkově zamknuté paměti z RAM do GRAM a zpět. Saturovaná hodnota je v obou případech kolem 5.7GB/s, což nejspíše způsobuje omezení rychlosti hlavní paměti RAM. Ve srovnání s tímto, přenos stránkovaných dat dosahuje saturace při 1.5GB/s (téměř čtyřikrát méně!).

Základní desky a grafické karty, které mají podporu pouze PCI-express 1.1 při přenosu stránkovaných dat nepředstavují omezení. Při přenosu nestránkovaných však jejich přenosové pásmo

nestačí a jak čtení, tak zápis saturuje na hodnotě 2.5GB/s. Zbytek do maximální hodnoty 4GB/s tvoří pravděpodobně přenos řídících signálů.

Pro srovnání je do grafu zanesena i propustnost čtení a zápisu z grafické paměti (GRAM) do GPU. Maximální hodnota je přibližně 55GB/s, z čehož plyne závěr, že je maximálně vhodné omezit přenosy přes PCI-e rozhraní. Na trhu jsou i karty s propustností 80-120GB/s, což ještě více umocňuje předchozí závěr. Více viz příloha: Parametry současných grafických karet. Nejvhodnější je tedy přenos stránkově uzamčených dat (pinned) z RAM do GRAM, následně do GPU, v něm provést co nejvíce výpočtů (případné mezivýsledky ukládat samozřejmě do GRAM) a výsledky buď přímo zobrazit, nebo přenést zpět do RAM.

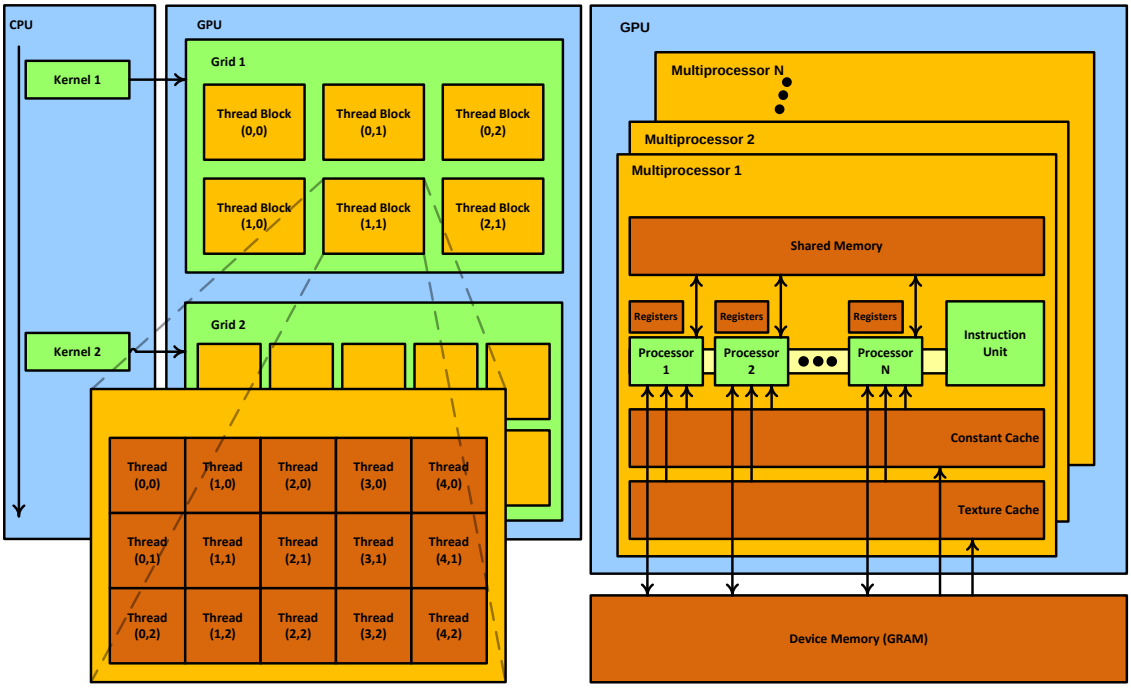


Graf 3 - Závislost propustnosti na velikosti přenášených dat

2.3 Architektura G80, G92 a programový model

Tato podkapitola nastíní architekturu grafické karty z pohledu GPGPU. Více informací naleznete v CUDA manuálu (3).

Na pravé části obrázku 4 je vidět, že GPU je několik (16)[3] SIMD (Single Instruction – Multiple Data) vektorových multiprocesorů. Každý multiprocesor tedy obsahuje několik (8) procesorů, který zpracovává vlákna v jednom thread-blocku (blok vláken). Viz levý obrázek. Vláken v bloku, jak je vidět z obrázku, může být více, než je procesorů, maximálně však 512. Mohou být organizovány do jedno, až tří dimenzionální matice. Používá se princip časového multiplexu. Ten umožňuje, aby se, ve chvíli, kdy vlákna čekají na přenos dat z hlavní paměti, mohli provádět jiná vlákna, která mají data připravena (v registru nebo sdílené paměti). Důsledkem je to, že při dostatečném počtu vláken v bloku a jejich dostatečné výpočetní náročnosti jsou procesory neustále vytížené a velikost potřebné cache je poměrně malá.



Obrázek 4 - Dělení do vláken, HW implementace SIMD multiprocesorů, paměťový model

Vlákna v jednom bloku spolu mohou vzájemně komunikovat pomocí sdílené paměti a jejich běh může být synchronizován. Toho lze využít v mnoha algoritmech.

Protože je počet vláken v bloku omezen (počtem dostupných registrů a sdílené paměti) a je nutné využít všechny multiprocesory je GPGPU program (kernel) vhodné rozdělit na více bloků vláken. Tzv. gridu bloků. Jednotlivé bloky spolu principiálně nemohou komunikovat a není zde možnost synchronizace a předávání informací mezi nimi. Bloky jsou postupně vykonávány na multiprocesorech v nedefinovaném pořadí a jejich počet je prakticky neomezený. Za tím stojí i jednoduchá škálovatelnost napříč jednotlivými řadami (a snad i budoucími generacemi) čipů Nvidia. Výkonné řady (8800, 9800) jednoduše obsahují více multiprocesorů (16 multiprocesorů) než, main-streamové a low-endové (2-8 multiprocesorů), a umožňují tedy běh více bloku paralelně a tím i rychlejší vykonání celého výpočtu.

[3] Hodnoty v závorkách platí pro testovací kartu

2.4 Paměťový model GPU

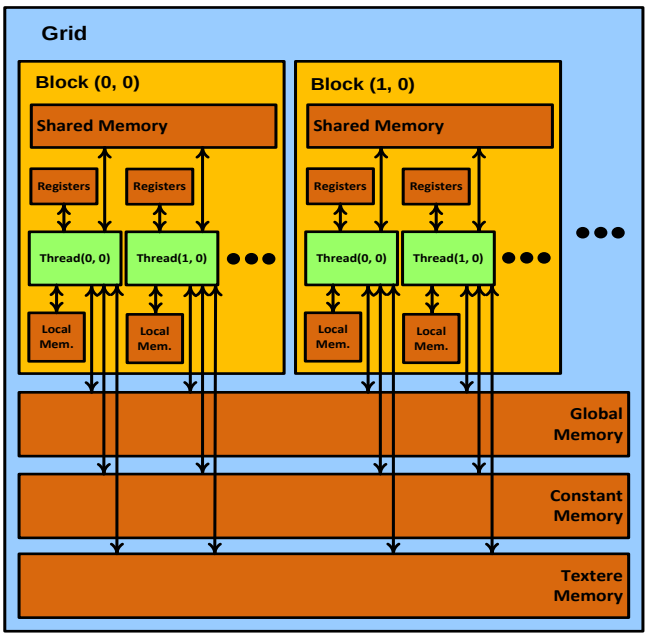
Pro programování grafické karty je nutné znát programový model, a umístění jednotlivých druhů paměti. Opět je tato kapitola pouze informativní a pro více informací je možné se obrátit na (3).

V pravé části obrázku 4 je zobrazena hardwarová implementace paměťové hierarchie a na obrázku 5 jeho logický, programový obraz.

Globální, konstantní a texturová paměť logického modelu leží v hlavní paměti grafické karty (Device Memory - GRAM). Paměť konstant a textur je cachována, globální ne. Texturová paměť a její cache se v GPGPU nejčastěji nepoužívá, globální a konstantní paměť naopak velmi často.

Lokální paměť každého vlákna leží též v hlavní paměti, a ukládají se do ní data, které jsou příliš velká na to, aby se vešla do registrů. Vzhledem k velkému rozdílu rychlostí vybavení dat z registrů (jeden cyklus) a z lokální paměti ležící v hlavní paměti (stovky cyklů) je výhodné lokální paměť příliš nevyužívat.

Sdílená paměť a registry jsou vestavěny do čipu a mají rychlou vybavovací dobu (jeden cyklus u obou v případě, že nedochází ke konfliktu mezi bankami sdílené paměti). Jak již bylo naznačeno, velikost těchto vestavěných pamětí je díky architektuře malá (pro čipy G92 - 8192 registrů a 16KB sdílené paměti na multiprocesor) a je vhodné optimalizovat algoritmy tak aby této paměti používaly co nejméně.



Obrázek 5 - Paměťový model GPU[4]

Průběh typického GPU programu je tento:

- Data jsou přenesena z hlavní paměti přes PCI-express pomocí DMA do hlavní paměti grafické karty.
- Každé vlákno v bloku načte potřebná data do sdílené paměti multiprocesoru.
- Běh vláken je synchronizován. Tím se zajistí, aby všechna vlákna měla načtena potřebná data.
- Je vykonán výpočet.
- Běh vláken je synchronizován.
- Data se uloží zpět do hlavní paměti grafické karty.
- Výsledek je zobrazen, nebo přenesen zpět do hlavní paměti.

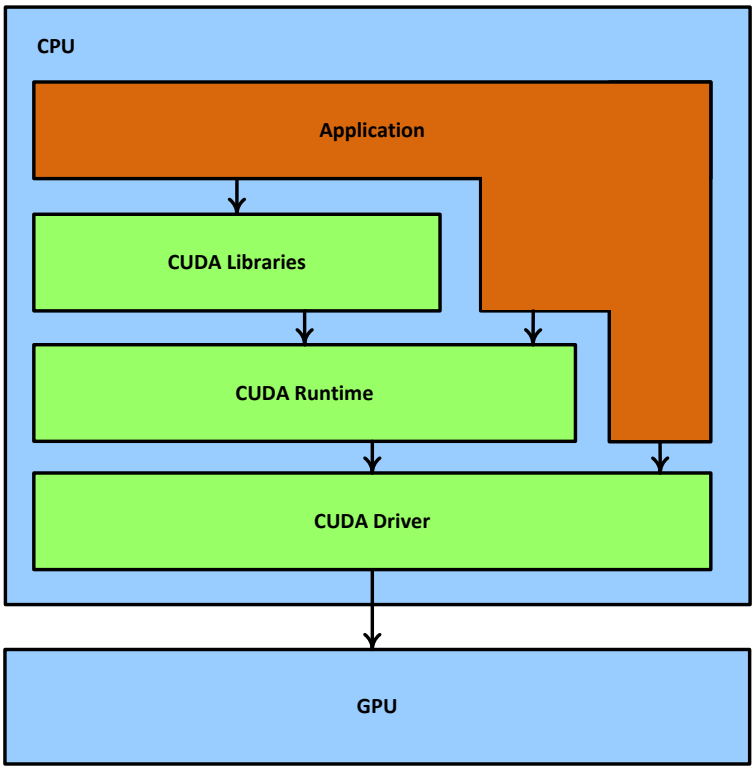
[4] Obrázek převzat a aktualizován z (3)

3 Programování a optimalizace

Tato kapitola letmo představuje nové programové API společnosti Nvidia, které se používá k programování Nvidia GPU od řady 8xxx. Následuje popis zásad, který definuje jakým způsobem toto API využívat. Navazuje na předchozí a snaží se vysvětlit spojení mezi hardwarem a efektivním programováním GPU.

3.1 Architektura Nvidia CUDA

CUDA (Compute Unified Device Architecture) je HW a SW architektura pro akceleraci výpočtů pomocí grafické karty (GPU), bez nutnosti mapovat výpočty na grafické API (DirectX). Je použitelná na nových kartách společnosti Nvidia (G80, G84, G86, Quadro, Tesla[5]). Vrstvový model je zobrazen níže.



Obrázek 6 - Architektura knihovny CUDA[6]

Je vidět, že programátor má přístup ke všem možnostem GPU, a jestliže ví, co dělá, může psát výkonnější aplikace. Ještě bych zmínil, že v API jsou k dispozici efektivní multi-vláknové knihovny BLAS (CuBLAS) a FFT (CuFFT).
Programovací jazyk je velmi podobný C a po kompilaci do objektového kódu je možné je společně linkovat (i s C++ apod.) a vzájemně volat své funkce. Rozdíly jsou popsány zde:

- Přidány kvalifikátory umístění funkcí. To jestli má být funkce vykonávána v CPU nebo GPU a odkud může být volána (__device__, __global__, __host__).

[5] velmi zajímavé řešení – výkon blížící se 1,6 TFlops při použití 4 grafických karet G80. Možnost umístění do 19“ racku
[6] Obrázek převzat z (3)

- Obdobné kvalifikátory pro definici s jakou pamětí chceme pracovat. (__device__, __constant__, __shared__). Viz paměťový model.
- Direktivy a typy pro definování rozdělení aplikace do gridu a bloku vláken.

Každá z vrstev definuje velké množství funkcí. Pro jejich pochopení doporučuji nastudovat dokumentaci k CUDA API (3)

3.2 Optimalizace a pravidla pro dosažení vysokého výkonu

V této podkapitole budou vysvětleny základní principy efektivního používání grafické karty. Ty jsou velice důležité. Vysoký výkon grafického čipu by totiž mohl být snadno promrhán neefektivním využíváním některého ze subsystému grafické karty.

3.2.1 Využití PCI-express přenosů

Na grafu 4 je zobrazena závislost relativní výkonnosti GPU aplikace oproti CPU verzi v závislosti na velikosti dat v aritmeticky velice jednoduchém algoritmu (konverze obrazu mezi barevnými modely). Tato konkrétní úloha má velmi malý poměr aritmetické složitosti k velikosti dat. Je vidět, že v saturovaných oblastech je výkon GPU se započítání PCI-e přenosů velmi malý (2-3x větší než CPU). Naproti tomu bez započítání přenosů se dá dosahovat 30 až 50 násobku rychlosti klasické implementace. Procento doby výpočtu z celkové doby je tedy pouze přibližně 40 procent. Celkově se tedy akcelerace na GPU vyplatí i u těchto výpočtů (2x-3x urychleno), ale mnohem výhodnější je počítat aritmeticky složitější algoritmy (v poměru k velikosti dat), kde doba výpočtu nebude tak extrémně malá oproti celkové době. Další možností je na jedněch datech vykonat více aritmeticky jednoduchých operací.

Následuje odvození relativní výkonnosti grafické karty oproti procesoru. Všechny použité proměnné jsou vysvětleny v příloze.

$$P = \frac{t_{GPUTotal}}{t_{CPUTotal}} = \frac{t_{TransferRAMtoGRAM} + t_{GPU} + t_{TransferGRAMtoRAM}}{t_{CPUTotal}}$$

$$C = \frac{k_i}{k_{Idata}} \rightarrow k_i = C \cdot k_{Idata}$$

$$t_{TransferRAMtoGRAM} = \frac{k_{Idata}}{v_{RAMtoGRAM}\,(k_{Idata})}$$

$$t_{TransferGRAMtoRAM} = \frac{k_{Odata}}{v_{GRAMtoRAM}\,(k_{Odata})}$$

Zavedeme zjednodušující předpoklad, že aritmetická složitost výpočtu a velikost zpracovávaných dat je dostatečná na to, aby bylo odstíněno zpoždění přenosů z grafické paměti do GPU. Pro n výpočtu v grafickém čipu tedy platí:

$$t_{GPU} = \sum_{i=1}^{i=n} \frac{k_i}{v_{GPUProcess}} = \sum_{i=1}^{i=n} \frac{C_i \cdot k_{Idata}}{v_{GPUProcess}}$$

$$P = \frac{t_{GPUTotal}}{t_{CPUTotal}} = \frac{\frac{k_{Idata}}{v_{RAMtoGR\,AM}(k_{Idata})} + \sum_{i=0}^{i=n} \frac{C_i \cdot k_{Idat\,a_i}}{v_{GPUProcess}} + \frac{k_{Odata}}{v_{GRAMtoRAM}\,(k_{Odata})}}{t_{CPUTotal}}$$

Nechť jsou rychlosti přenosu, rychlost zpracování instrukcí, velikost vstupních dat a doba zpracování na procesoru pro daný soubor dat neovlivnitelná. Jediná hodnota, kterou můžeme měnit, je relativní aritmetická složitost, kterou ovlivníme výběrem urychlovaných algoritmů nebo jejich počtem.

3.2.2 Optimalizace použití sdílené paměti.

Vzhledem k tomu že počet registrů na vlákno je omezený (je závislý na počtu vláken v bloku) a některá data potřebuje více vláken, bylo by neefektivní je uchovávat duplicitně v registrech. Proto je přímo na čipu vestavěna velmi rychlá **sdílená paměť** (Shared memory). Sdílená proto, že k ní mohou přistupovat všechna vlákna v jednom bloku. Hardwarová implementace paměti, která by byla schopna číst nebo zapisovat data N paralelně vykonávaných vláken do jakéhokoli místa ve sdílené paměti není možná, a proto je rozdělen do 16 bank. Jedna banka je v jednom taktu schopná obsloužit pouze jeden požadavek na čtení nebo zápis. Počet bank je volen s ohledem na aritmetickou propustnost multiprocesoru, která je právě jedna aritmetická operace v osmi vláknech (8 procesorů v jednom multiprocesoru) v jednom taktu. Ve dvou taktech je tedy 16 operací, což odpovídá možnostem sdílené paměti dodávat multiprocesoru data, tak že nedochází ke zbytečným prodlevám. Toto je ale možné jen ve chvíli, kdy nedochází k požadavkům ke čtení nebo zápis ze stejné banky. Tento fenomén se nazývá **konflikt při přístupu do sdílené paměti.** Pokud nastane, je nutné operace serializovat a v extrémním případě může být vykonávání kódu až 16 krát pomalejší, ve chvíli, kdy nastane požadavek na čtení všech vláken z jedné banky. Toto je samozřejmě nežádoucí.

Obecně je tedy nutné zajistit to, aby vektor šestnácti po sobě jdoucí vláken[7] pracoval s rozdílnými bankami s tím, že data jsou uložena postupně v bankách po 32 bitech. Mějme například pole 32 čísel float. Prvky 0 – 15 obsadí banky 0 – 15 a prvky 16 – 31 opět cyklicky banky 0 – 15.

Ke zjištění, zda skutečně v kódu nedochází ke konfliktům ve sdílené paměti, slouží pomocná funkce cutCheckBankAccess[8] respektive makro CUT_BANK_CHECKER(array, index), které zjednodušuje volání této funkce. Obě tyto možnosti lze použít pouze v Device-emulation modu a v případě konfliktu vytisknou na konzoli příslušnou zprávu.

3.2.3 Zarovnané operace s hlavní pamětí

Grafická karta nenabízí oproti procesoru jenom mnohem vyšší aritmetický výkon grafického procesoru, ale i vysokou propustnost paměťového systému. To je dáno širokou paměťovou sběrnicí (256bit-512bit) oproti FSB (64 nebo 128 bit) a možnosti použít výše taktované paměti (1.8GHz a více). Celková propustnost pamětí je tedy, s dosazením hodnot pro testovací kartu:

$$propustnost = frekvence \cdot \text{šířka sběrnice}$$
$$propustnost = 1940 \cdot 10^6 \cdot 256/8$$
$$propustnost = 57.8 \cdot 10^9 B/s = 57.8\text{GB/s}$$

Tato hodnota ostatně odpovídá grafu 3.

Takovou propustnost je možné dosáhnout pouze za podmínky, kdy po sobě jdoucích 16 vláken[9] manipulují se sekvenčním blokem hlavní paměti (N-té vlákno čte N-tou položku v tomto bloku). V opačném případě se musí iniciovat více paměťových přenosů a celková propustnost rapidně klesá. Další podmínkou je, že jedno vlákno čte (zapisuje) data o velikosti 4, 8 nebo 16 Bytů. Poslední

[7] Pro 2D a 3D bloky vláken se pořadí daného vlákna spočítá vzorcem: $t_{ID} = x + yD_x$ respektive $t_{ID} = x + yD_x + zD_xD_y$. D_k je k dimenze bloku vláken a x, y, z jsou indexy.

[8] Funkce je součást cutil (CUda UTILs)knihovny a je deklarována v souboru cutil.h. Stejně tak i makro CUT_BANK_CHECKER.

[9] V terminologii CUDA knihovny je tento balík vláken nazývám half-warp.

podmínka říká, že bloky by měly být zarovnány na násobek velikosti zpracovávaného bloku. Ten má tedy velikost 64B, 128B, 256B. Nesplnění byť jen jedné z podmínek vede ke drastickému zpomalení práce s hlavní pamětí.

Nyní uvedu způsob, jak lze zajistit, aby bylo možné efektivně zpracovávat data i o velikosti různé od 4, 8 a 16 Bytů. Pokud je velikost dat, které potřebuje jedno vlákno, možné rozdělit na součet 4, 8 a 16 $(x \cdot 4 + y \cdot 8 + z \cdot 16 + c), kde\ c = 0$, provedeme x čtení 4 bytového bloku, y 8 bytového a z 16 bytového do **sdílené paměti.** Pokud $c \neq 0$ použijeme podmíněný výraz, který zajistí, že data do sdílené paměti uloží pouze část N vláken, kde N je voleno tak aby byla načtena všechna data. Jestliže je počet načítaných bytů celé číslo, a využijeme-li první techniku, může c nabývat pouze hodnot 1, 2 nebo 3. Budeme-li načítat 4 Bytové bloky, bude N pro $c = x \rightarrow N = \frac{x}{4}$. Pokud funkce není omezena velikostí sdílené paměti, je též možné jednoduše alokovat větší oblast sdílené paměti a 4 Bytový blok načíst celý bez nutnosti vyhodnocovat podmínku, což vede k větší efektivitě (pokud je omezení ve výpočetní síle GPU).

3.2.4 Obsazenost multiprocesorů.

Jak již bylo řečeno, jeden multiprocesor vykonává kód vláken jednoho bloku. Protože vláken může být více, než je jednotlivých procesorů, dochází k časovému multiplexu a přepínání jednotlivých warp[10] a tím i k zastiňování čekání na přenosy dat z hlavní paměti.

Ve chvíli, kdy je zpracováván algoritmus s malou relativní aritmetickou složitostí, stává se, že hlavní paměť nezvládne dodávat data dostatečně rychle. Aritmetické jednotky nejsou vytížené na maximální hodnotu. Pokud je ovšem dostatek volných zdrojů, spouští se na multiprocesoru více bloků v jednom okamžiku a dochází k lepšímu zastínění zpoždění hlavní paměti. Spouštění více bloků se děje automaticky. Zdroji[11] se v tomto případě myslí trojice:

- Počet registrů
- Velikost sdílené paměti
- Maximální počet spuštěných vláken všech bloků na jednom multiprocesoru

Každý z těchto zdrojů definuje počet bloků, který je pro daný výpočet schopen obsloužit. Skutečný počet spuštěných bloků je definován zdrojem s nejmenší kapacitou a je shora omezen na maximálně 8 spuštěných bloků.
Pro dosažení větší efektivity je vhodné udržovat obsazenost multiprocesoru bloky na co největší hodnotě. K tomu částečně pomáhá i soubor aplikace Excel: CUDA_Occupancy_calculator.xls, který je součástí CUDA SDK. V něm po zadání hodnot zdrojů výše zjistíme co je omezující faktor algoritmu.
Ke zjištění obsazení registrů a sdílené paměti jednoho bloku se používá speciální kompilační mód nvcc kompilátoru, kde parametr klasické kompilace s parametrem -c zaměníme za –cubin (více informací o kompilačních módech viz (4)).

[10] Dále nedělitelný balík 32 vláken, zpracovávaný ve 4 cyklech.
[11] Hodnoty těchto zdrojů naleznete v příloze, v kapitole parametry testovací grafická karta.

Výsledný soubor má tuto strukturu:

```
code  {
        name = _Z18BGRuch8toGRAYuch24PcS_j      - vnitřní název kompilované funkce
        lmem = 0                                - počet registrů lokální paměti
        smem = 3100                             - počet bytu využitých sdílenou pamětí
        reg = 8                                 - počet využitých 32b registrů
        bar = 0
        bincode  {                              - binární kód funkce
                0x1000000d 0x0403c780 0xd0800e01 0x00400780
                ⋮
                0x861ffe03 0000000000 0xf0000001 0xe0000001
        }
        const  {
                segname = const                 - značí konstantní oblast hlavní paměti
                segnum = 1                      - počet obsazených segmentů
                offset = 0
                bytes = 8                       - celkový počet bytu
                mem  {
                        0x000003ff 0x00000003   - binární vyjádření konstant
                }
        }
```

3.2.5 Šablony

Přestože je CUDA jazyk velice podobný jazyku C, přebírá jisté vlastnosti z C++. Tou hlavní je podpora šablon. Jestliže je algoritmus založený převážně na provádění cyklů (například konvoluce) s předem známým počtem průchodů (velikostí konvoluční matice), můžeme, díky celočíselným šablonám dosáhnou výrazného zvýšení rychlosti vykonávání kódu. Díky nim se nemusí v každé iteraci počítat podmínka cyklu a provádí se pouze sekvence výpočtu, které byly rozgenerovány v době kompilace.

4 Implementace knihovny a testování

Tato kapitola popisuje do hloubky vlastnosti jednotlivých algoritmů. Jsou zde využívány a diskutovány všechny principy efektivního zpracování. V neposlední řadě je porovnáván výkon výsledných GPU funkcí a jejich protějšků v OpenCV[12].

4.1.1 Model knihovny

Knihovna je koncipována do tří úrovní, tak, aby její uživatel byl odstíněn od pro něj nepodstatných informací a mohl s knihovnou komunikovat jednoduchým voláním funkcí. Rozdělení do úrovní je takovéto:

V první, nejvyšší úrovni leží uživatelský program, s knihovnou komunikuje voláním minimalizovaného počtu funkcí. Jejich seznam najdete v příloze.

Druhé úrovni náleží konfigurace volání funkcí třetí úrovně (výpočet počtu bloků a vláken) a také umožňuje sekvenčně vyvolat několik funkcí úrovně 3, tak aby výsledek tvořil složitější úlohu.

Třetí úroveň obsahuje optimalizované GPU rutiny (kernely)

4.1.1.1 Přenosy z grafické paměti

Knihovna OpenCV je využívána k načítání obrázků ze souborů funkce cvLoadImage. U ní bohužel nejde ovlivnit způsob alokace pro načítaný obrázek a nelze tedy využít alokace stránkově zamknuté paměti. Možností by bylo vytvořit kopii alokovaných dat do zamknuté paměti a ty přesunovat do GPU a manipulovat s nimi. Čas potřebný na tuto operaci by ovšem nevyvážil úsporu doby potřebnou k přesunu.

V dalších verzích knihovny bude vytvořena modifikovaná verze funkce cvLoadImage, která bude přímo alokovat stránkově uzamknutá data a k ní odpovídající dealokační protějšek cvReleaseImage.

4.2 Konverze mezi barevnými modely

Protože konverze barevného modelu obrazu jsou velmi často používané, je nutné, aby byly efektivní a výkonné. U ostatních, dále implementovaných a o mnoho aritmeticky složitějších operací (konvoluce) není problém obraz převést na 4 Bytovou položku (RGB->RGBA), data zpracovat a převést zpět. U jednoduchých barevných konverzí by byl čas převodu srovnatelný s dobou převodu. Připomínám, že použití 4 bytových datových typu je výhodně kvůli sdílené paměti, kdy nevznikají konflikty mezi bankami při přístupech typu:

```
uchar3 pixel = data[threadIdx.x];
```

kde "data" je pole ve sdílené paměti. Následuje popis metody, kde je možné se převodu vyhnout.

[12] Open Computer Vision library. Pro více informací o této knihovně viz domovská stránka projektu http://sourceforge.net/projects/opencvlibrary/

4.2.1 Konverze z modelu ABCu8 na model DEFu8

Nejjednodušší konverze barevného modelu pracují s konverzní maticí. Výstupní pixel je dán vzorcem:

$$\begin{pmatrix} A \\ B \\ C \end{pmatrix} = \begin{pmatrix} k_0 & k_1 & k_2 \\ k_3 & k_2 & k_3 \\ k_6 & k_7 & k_8 \end{pmatrix} \begin{pmatrix} D \\ E \\ F \end{pmatrix} \begin{pmatrix} l_1 \\ l_1 \\ l_1 \end{pmatrix}$$

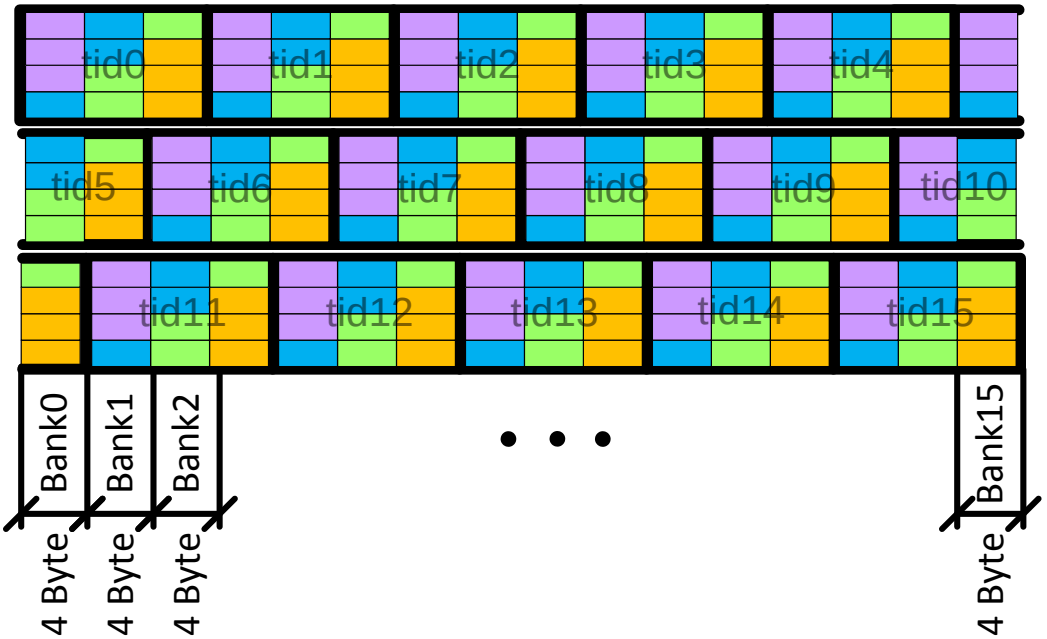
Konverzní matice je výkonné funkce předávána ne pomocí pole v globální paměti, ale pomocí paměti konstant. Ta má, pokud se konstanta nalézá v cache a všechny vlákna bloku čtou stejnou konstantu, zpoždění shodné se zpožděním registru a propustnost jedna 32 bitová položka na takt a blok.

Některé barevné převody navíc vyžadují výsledek součinu vektoru vstupního pixelu a matice K vynásobit vektorem L, který upraví výsledné hodnoty pixelu. Vektor L je uložen podobně jako matice K v paměti konstant.

Počítejme s tím, že zpracováváme 3 bytové pixely. Jestliže by jedno vlákno zpracovávalo jeden pixel, jeden half-warp (16 vláken) by načítal 48 Bytů. Každé vlákno by načetlo 4 Byte, a do sdílené paměti je uložilo pouze prvních 12. Hlavní problém by ale byl v zarovnání. Toho by při tomto způsobu šlo dosáhnout pouze velice složitě. Lepším řešením je načíst celkově 192 Byte, kdy každé vlákno bude načítat 12 Byte pomocí tří 4 Bytových přenosů. V tom případě bude přenos perfektně zarovnaný při všech třech čteních. Stejně tak bude probíhat i uložení zpět do hlavní paměti.

Při tomto uspořádání také nedochází ke konfliktům mezi bankami. Například, pokud načítáme první položku (která může reprezentovat modrou barvu u modelu BRG) prvního pixelu zpracovávaného jedním vláknem, která je na následujícím obrázku zobrazena purpurovou barvou, zjistíme, že tato položka leží pro každé vlákno v jiné bance (žádná neleží vertikálně nad jinou).

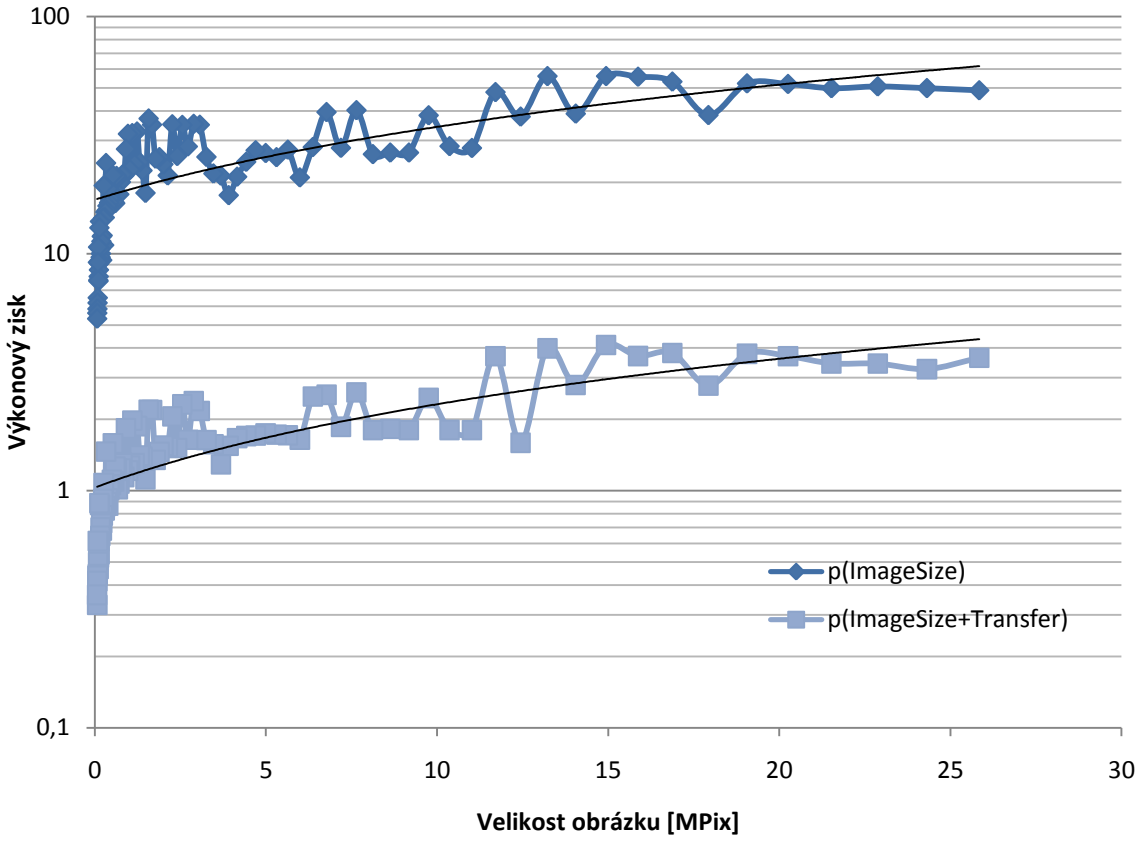
Vlastní konverze jednoduše vynásobí všechny pixely konverzní maticí a uloží zpět do sdílené paměti.



Obrázek 7 - Uložení pixelu ve sdílené paměti a jejich zpracování vlákny

Výkonnost

Následuje graf výkonnostního zisku GPU implementace oproti jejímu CPU protějšku na velikosti obrázku. Bohužel, i když byl čas CPU implementace brán jako průměr sta měření, jsou v něm skoky, které se projevují v grafu. Pravděpodobný důvod jsou optimalizace, náhodné jevy apod. Do grafu je proto vynesena i extrapolace této funkce.



Graf 4 - Výkon převodu z modelu ABC na DEF

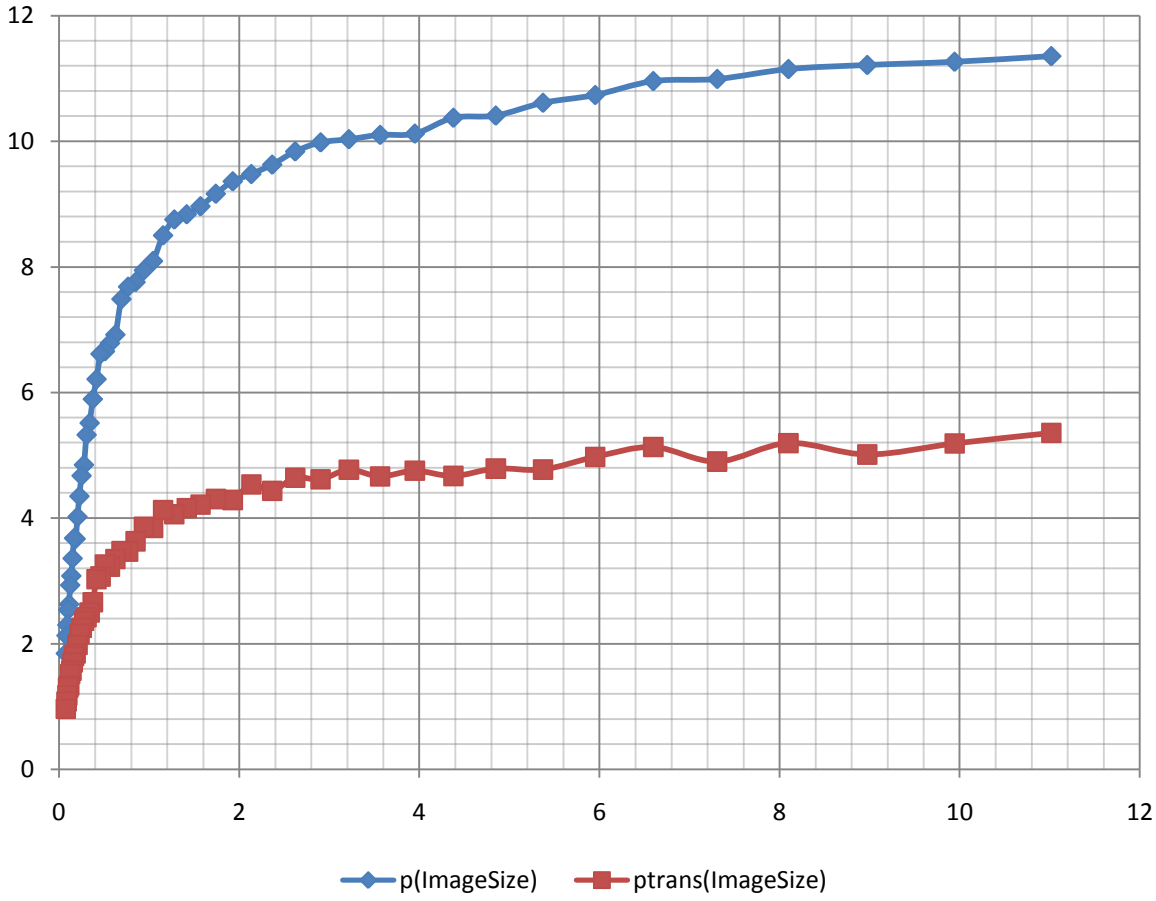
I pro malé velikosti vstupního obrazu je výkonnostní zisk více než dobrý, ovšem díky malé aritmetické složitosti je celkový zisk se započtením přenosu malý.

4.2.2 Konverze z modelu RGB_u8 na model Lab_u8

Tato konverze je ukázkou využití víceúrovňové architektury knihovny, kdy složíme více jednoduchých, elementárních funkcí, tak aby vykonali jednu relativně složitější úlohu.

Obrázek převedeme do RGBA32f modelu
Provedeme konverzi z RGB(A) do formátu XYZ(W) a provedeme korekci výsledku
Provedeme převod do Lab[13] formátu
Převedeme zpět na RGB(A)8u model, kvůli zobrazení nebo dalšímu zpracování.

4.2.2.1 Výkonnost



Graf 5 - Výkon převodu z modelu RGB na Lab

Oproti předchozí úloze, kde je maximální výkonnostní zisk přibližně 40, zde dosahuje pouze 11. Důvod je jednoduchý. GPU neobsahuje hardwarové děličky jako procesor a musí výpočty obsahující dělení počítat **iteračně**. I tak je dosahovaný výkon více než dobrý, a i při započítání přenosu přes sběrnici je asi pětinásobný oproti CPU implementaci. Navíc pokud porovnáme výkony se započítaným přenosem u tohoto a předcházejícího převodu, kdy byl maximální zisk pouze 2, vidíme, že závěr odvozený zde platí.

[13] Pro více informací o tomto formátu a způsob převodu z RGB na Lab viz wikipedia

4.3 Konvoluční filtry

Architektura grafického čipu si vynucuje, aby byla úloha rozdělena na více nezávislých úloh. V tomto případě můžeme obrázek rozdělit na matici pravoúhlých plošek, které reprezentují úlohu zpracovávanou jedním blokem vláken. Jedna elementární ploška je na následujícím obrázku zobrazena zeleně. Protože konvoluce pixelu vyžaduje znalost hodnot okolních pixelů, je nutné při zpracování načítat i oblast zobrazenou na obrázku oranžově. Sjednocení těchto oblastí je načteno do sdílené paměti bloku.

Zarovnaný přístup k hlavní paměti vyžaduje to, aby šířka zelené oblasti byla celočíselným násobkem jednoho zarovnaného přístupu. Tomu odpovídá například níže popsaný a implementovaný algoritmus načítání do sdílené paměti. V něm jsou voleny tyto parametry:

Velikost pixelu na 4 Byty.

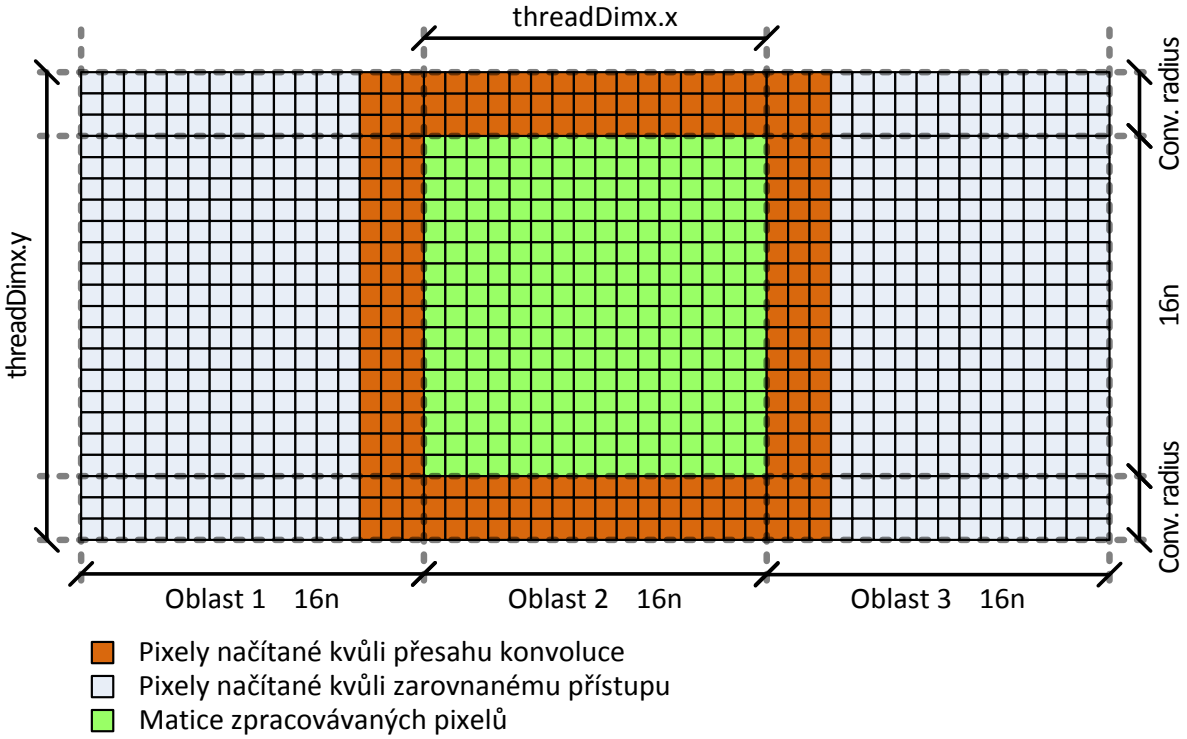
Blok vláken o velikosti threadDimx.x = 16, threadDimx.y = 16 + 2 * radius konvoluční matice[14] (v tomto případě, kdy je r = 3, threadDimx.y = 22).

Vlastní algoritmus načítání:

Načtení oblasti 1, pouze oranžové pixely jsou načteny do sdílené paměti.

Načtení oblasti 2, všechny pixely jsou přesunuty do sdílené paměti.

Načtení oblasti 3, se stejnou podmínkou jako 1).



Obrázek 8 - Zpracovávaná oblast v globální paměti

Čtení modrých pixelů se může zdát jako neefektivní, ale je řádově rychlejší, než kdybychom pixely potřebné z důvodu přesahu konvoluce načítali pomocí nezarovnaného přístupu. Výška zpracovávané oblasti je volena blízká šířce, nejlépe stejná. V tom případě je poměr ploch zelené a oranžové oblasti nejvyšší (dokázat to lze derivováním poměru ploch).

[14] Počítám se čtvercovou konvoluční matici s poloměrem k a velikostí 2k+1.

Vlastní výpočet je implementován jednoduše, jako dva cykly, které skalárně vynásobí konvoluční matici s maticí okolí počítaného pixelu. Výsledek (zelená oblast) je přesunut zpět do hlavní paměti.

4.3.1.1 Efektivita

Výpočet konvoluce na procesoru bude pravděpodobně proveden průchodem po jednotlivých řádcích obrázku. Zaveďme zjednodušující předpoklad, že procesor je schopen uložit do cache celý tento řádek a dále N řádků pod, a N řádků nad, kde N představuje poloměr konvoluční matice. Každý pixel je v tomto případě načten pouze jedenkrát. Naopak GPU musí **některé** pixely načíst dvakrát, nebo dokonce čtyřikrát (rohové, oranžově označené oblasti). Navíc, pokud přidáme faktor zarovnaného čtení z hlavní paměti, je **každý** pixel načten minimálně dvakrát. Poměr velikosti načítaných dat u GPU a CPU je tedy:

Nezarovnaný přístup:

$$k = \frac{tile.x \cdot tile.y + 2 \cdot tile.x \cdot convRadius + 2 \cdot tile.y \cdot convRadius + 4 \cdot convRadius^2}{tile.x \cdot tile.y}$$

$$k = 1 + \frac{2 \cdot convRadius\ (tile.x + tile.y + 2 \cdot convRadius)}{tile.x \cdot tile.y}$$

Zarovnaný přístup:

$$k = \frac{(tile.y + 2 \cdot convRadius) \cdot 3 \cdot tile.x}{tile.x \cdot tile.y}$$

$$k = \frac{3 \cdot tile.x \cdot tile.y + 6 \cdot convRadius \cdot tile.x}{tile.x \cdot tile.y}$$

$$k = 3 + \frac{6 \cdot convRadius}{tile.y}$$

Hodnoty, $tile.x$ a $tile.y$, pro které je provedená implementace:

$$tile.x = 16$$
$$tile.y = 16$$

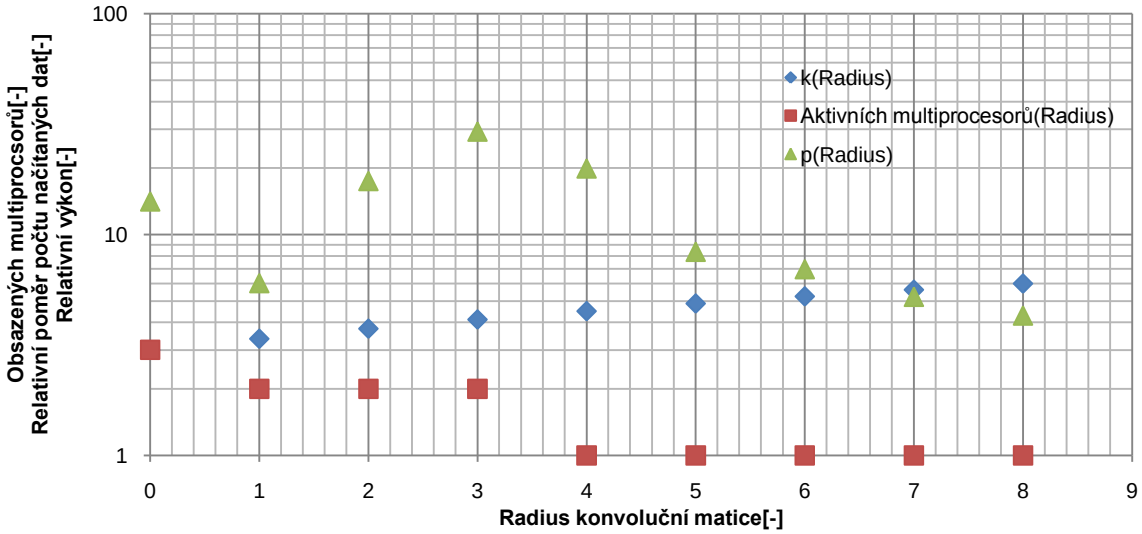
Provedeme dosazení:

$$k = 3 + \frac{6 \cdot convRadius}{16}$$
$$k = 3 + 0.375 \cdot convRadius$$

Tuto diskrétní funkci vyneseme do grafu společně s naměřenými výsledky poměru časů potřebných k výpočtu konvoluce na CPU a GPU.

$$P = \frac{t_{GPU}}{t_{CPU}}$$

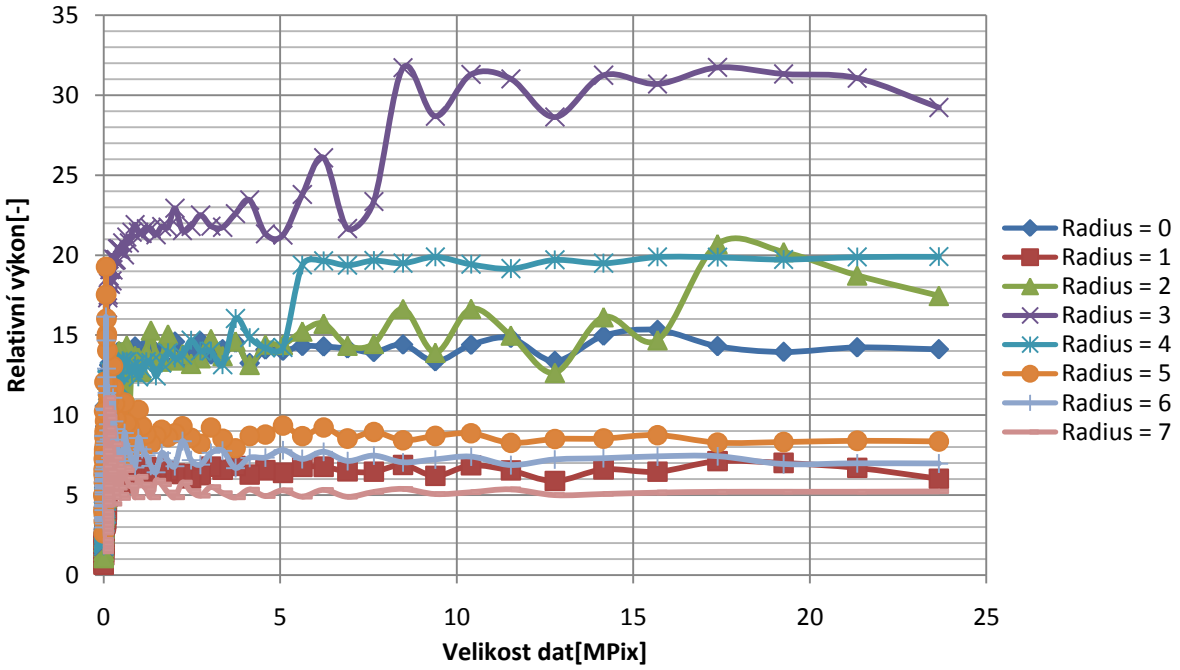
Další funkcí v grafu je počet obsazených multiprocesorů v závislosti na poloměru konvoluční matice. I tento parametr má částečně vliv na výkon. Více o této problematice viz kapitola obsazenost multiprocesorů.



Graf 6 - Závislost výkonu na velikosti konvoluční matice

Jak je vidět, se stoupajícím poloměrem konvoluční matice stoupá jak poměr velikosti načítaných dat u GPU a CPU - k(convRadius), i počet používaných multiprocesorů, který se nepříznivě projevuje klesáním poměru výkonnosti -p(convRadius).

Pro úplnost ještě uvedu graf závislosti p na velikosti vstupního obrázku, parametrizovaný velikostí konvoluční matice.



Graf 7 - Závislost výkonu konvoluce na velikosti obrázku a velikosti konvoluční matice

Podle grafu je vidět, že pro malé vstupní obrázky (<<1MPix) relativní výkonnost klesá. Důvod je v tom, že malý počet spuštěných bloku neobsadí všechny použitelné multiprocesory. Dále je zajímavé, že nejvyšší výkon je dosahován pro poloměr konvoluční matice rovný třemi. To koresponduje s grafem 6. Pro tento poloměr je v ideálním poměru aritmetická složitost a poměr načítaných dat u CPU a GPU. U malých velikostí konvolučního jádra není paměťový subsystém GPU schopen dostatečně rychle dodávat potřebná data (malá aritmetická složitost výpočtu) a naopak u velkých průměrů stupá poměr navíc načítaných dat, který výpočet zpomaluje.

Poslední poznámka je ke způsobu měření. Každý čas byl změřen desetkrát, a k výpočtu poměru byl zvolen medián z těchto časů. I tak se v grafu vyskytují skoky, které mohou být způsobeny jak náhodným vytížením výpočetních zdrojů vnějšími vlivy, tak i jejich hardwarovými omezeními (například od jisté velikosti dat nejsou dostatečná velikost potřebných vyrovnávacích pamětí jak v CPU tak GPU apod.)

4.3.1.2 Zhodnocení

Počet vláken v bloku omezen na 512 a počet vláken algoritmu je parametrizován velikostí zpracovávaného bloku a poloměrem konvoluční matice tímto vzorcem (vyplývá obrázku 8):

$$512 \leq threadDim.x \cdot threadDim.y = tile.x(tile.y \cdot 2 \cdot convRadius)$$

Pro konvoluční matice s větším poloměrem, než je 8 je nutné spustit více vláken než 512 (což není díky omezením hardwaru možné. Více viz specifikace použité grafické karty) a tyto filtry tedy není možné při zachování rozměrů zpracovávané plochy. Řešením je zmenšením této plochy. Její šířka je pěvně daná zarovnaným přístupem k hlavní paměti, a zbývá tedy redukce výšky. To ovšem vede k tomu, že poměr načítaných dat, ke zpracovávaným stoupá a výkon klesá.

Obecným řešením by bylo nastavení $threadDim.x = tile.x, threadDim.y = tile.y$. Data už by nebyla načítána jako tři sloupce, ale jako devět čtvercových bloků rozměrech $tile.x$ a $tile.y$. To by navíc umožnilo spuštění více bloků současně. Například při $tile.x = tile.y = 16$, a $convRadius = 1$, je u implementované metody počet vláken $16 \cdot 18 = 288$. U diskutovaného algoritmu $16 \cdot 16 = 256$. Při maximálním počtu spuštěných vláken na jeden multiprocesor (768), je možné spouštět 2 respektive 3 bloky současně a tím i větší možnosti GPU zakrývat zpoždění hlavní paměti.

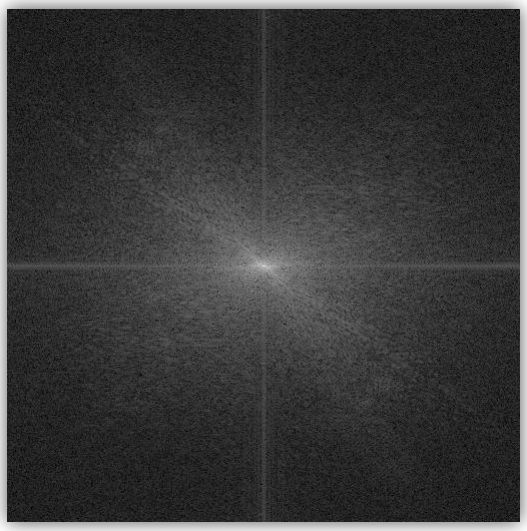
Popsaný algoritmus má i přes tyto chyby pro malé velikosti konvoluční matice relativně dobrý výkon pět až třicetinásobek výkonnosti procesoru (bez započítání přenosů přes PCI-express) v závislosti na poloměru matice a velikosti vstupních dat.

4.4 Filtry využívající Fourierovu transformaci

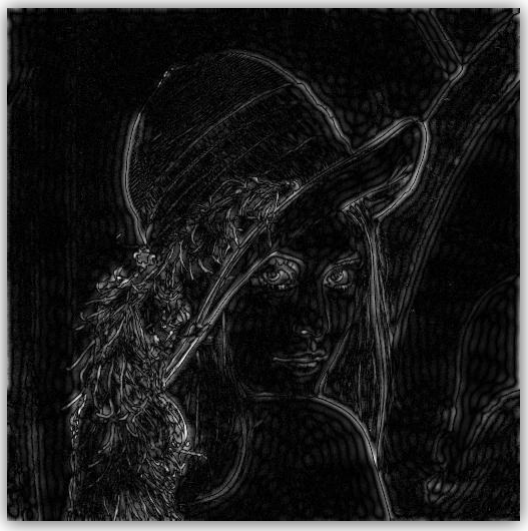
Pokud nelze filtr vytvořit pomocí oddělitelné konvoluce a konvoluční jádro u maticové konvoluce má příliš velké rozměry (nebo obecně větší než dovoluje implementace v mojí knihovně), stává se filtrování v prostorové doméně (spatial domain) neefektivní. Protože je obraz dvou-dimenzionálním signálem, není důvod se k němu tak nechovat a nevyužít možnosti filtrování ve frekvenční doméně. Více o této problematice viz (5).

K tomuto účelu se používá algoritmus rychlé Fourierovy transformace, která zajišťuje efektivní převod mezi těmito dvěma doménami. Ten je navíc snadno paralelizovatelný. V knihovně CUDA je implementován pod názvem CUFFT. S její pomocí se mi podařilo implementovat jednoduché rozhranní pro volání FFT z uživatelských programů. Uživateli stačí předat vstupní a výstupní obrázek a 2D masku filtru, která bude v jednotlivých bodech obsahovat váhu bodu v intervalu $\langle 0, 1)$. Následující obrázky ukazují příklady využití v ukázkové aplikaci.

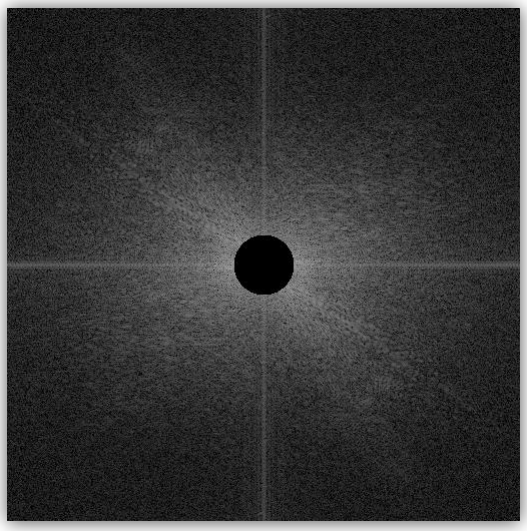
Obrázek 9 - Lenna



Obrázek 12 - Lenna FFT

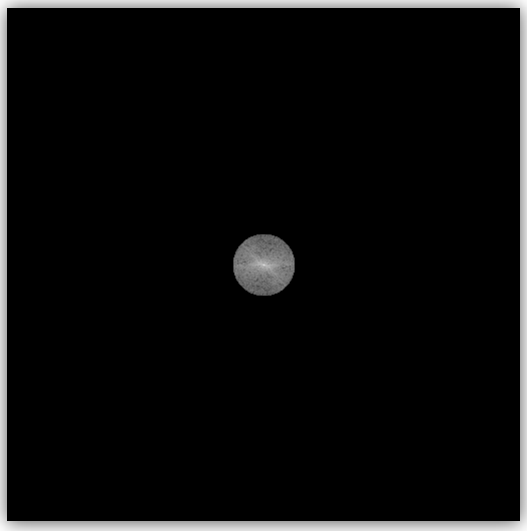


Obrázek 10 - Lenna - ideální horní propust



Obrázek 13 - Lenna - ideální horní propust FFT

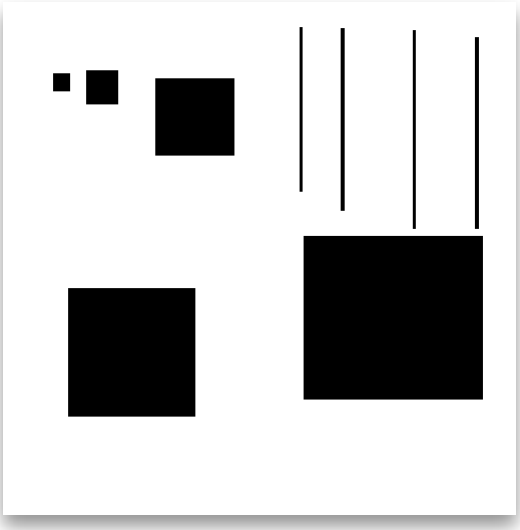
Obrázek 11 - Lenna - ideální dolní propust



Obrázek 14 - Lenna - ideální dolní propust FFT

Obrázek 9 ukazuje klasický testovací obrázek Lenny v černobílém provedení. K získání spektra napravo je tento obrázek nutné vynásobit funkcí $f(x, y) = (-1)^{x+y}$, která zajistí to, že střední hodnota bude ve středu spektra (viz jeden jasný bod ve středu obrázku 12) a logaritmickou normalizaci jednotlivých bodů do intervalu $\langle 0, 1)$. Pokud bychom na toto spektrum aplikovali inverzní Fourierovu transformaci, získali bychom původní obrázek.

Hrany a obecně rychlé změny v obraze jsou asociovány s vysokými frekvencemi. Proto pokud potlačíme nízké frekvence ideální horní propustí (viz spektrum na obrázku 13) získáme po inverzní Fourierově transformaci obrázek 10, který obsahuje pouze vysoké frekvence. Opačného výsledku dostaneme po filtrování dolní propustí. Vysoké frekvence budou potlačeny, a s nimi i hrany, rychlé přechody a šum v prostorové oblasti. Obraz tedy bude rozostřen. Všimněte si, že u obou vyfiltrovaných obrázků (Obrázky 10 a 11) vznikají artefakty. Lépe viditelné jsou na následujících obrázcích:



Obrázek 15 – Testovací obrázek

Obrázek 16 - ideální dolní propust

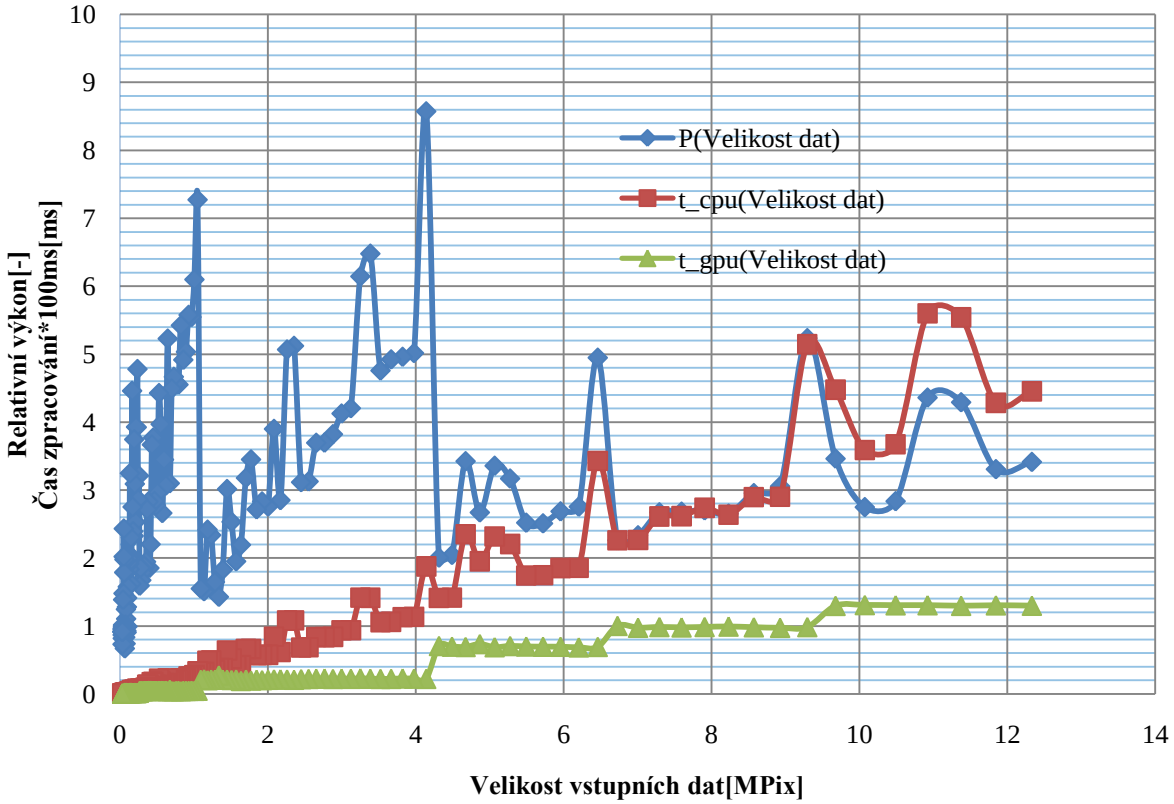
Tento jev je nazýván „kroužkování“ (anglicky ringing). Je způsoben ostrou hranou filtru.

4.4.1.1 Výkonnost

K tomu aby bylo vykonávání FFT algoritmu efektivní, je nutné dodržet požadované velikost vstupních obrazových matic. Nejčastěji je to libovolná mocnina dvou, nebo malých prvočísel (3, 5, 7). Více viz (6) v kapitole accuracy and performance. V obou případech byly vstupem stejné obrázky, které byly předem doplněny nulami vpravo a pod obrázkem, podle výše zmíněného pravidla, tak aby FFT dosahovalo optimálního výkonu.

Podobně jako v předchozích kapitolách následuje graf funkce relativní výkonnosti a časů výpočtů u CPU a GPU v závislosti na velikosti dat. Lze si všimnout, že počet možných optimálních hodnot velikosti vstupního pole je menší než u CPU implementace, což v grafu, u měření výkonnosti GPU implementace, vytváří schody. Ty se projevují i v poměru výkonů.

Celkově je GPU implementace knihovny CUFFT oproti optimalizovanému OpenCV protějšku dva až pět krát výkonnější. Lze předpokládat, že knihovna se bude dále vyvíjet a bude dále optimalizována.



Graf 8 - Závislost výkonu FFT a času převodu na velikosti obrázku

5 Závěr

Výsledkem mojí práce je knihovna funkcí, umožňující zpracování obrazu pomocí grafického akcelerátoru. V této chvíli není vzhledem k malému počtu funkcí v praxi dost dobře použitelná a ani si nedávává za cíl nahradit OpenCV. Cílem bylo spíše demonstrovat technologickou pokročilost současných GPU a ukázat, že v souboji proti optimalizovaným algoritmům v OpenCv je možné dosahovat i třicetinásobného výkonu. Na vývoj skutečně použitelné GPU náhrady OpenCV by bylo potřeba nejen mnohem více času, ale i vývojářů.

Byly vybrány tři rodiny rozdílných a užitečných algoritmů. Jmenovitě to byly *konverze barevného modelu*, *filtrování konvoluční maticí* a *filtrování pomocí FFT*. Všechny byly implementovány s maximálním důrazem na efektivitu. V prvních dvou je dosahovaný výkon špičkově dvaceti až třicetinásobný, v poslední více než pětinásobný, což je vzhledem k použitému testovacímu procesoru více než dobrý výkon.

Částečně tato práce navazuje na práci Lukáše Poloka (1), který stejně jako já, vytvářel akcelerovanou knihovnu pro zpracování obrazu. Na rozdíl od něj, umožňuji ve struktuře definující GPU obrázek uchovat ukazatel do grafické paměti. Lukáš musí při každé operaci obrázek přenášet do paměti, což je značně limitující. Dalším rozdílem bylo, že používal programové rozhranní pro popis grafických operací OpenGL, na rozdíl od mnou použitého obecného GPGPU rozhranní CUDA. Také se v jeho práci netestuje výkonnost, která je z mého hlediska stěžejní.

Z hlediska dalšího vývoje by bylo vhodné doplnit další používané funkce tak, aby vytvořená knihovna pokrývala větší množinu možností OpenCV. V této chvíli je nutné, pokud není GPU obdoba funkce, obrázek přenášet tam a zpět z grafické karty. To zbytečně degraduje výkonové možnosti knihovny. S větším počtem implementovaných GPU funkcí by pravděpodobnost tohoto jevu klesala. Dalším cílem je vytvořit obdobu funkce cvCreateImage tak, aby používala stránkově uzamknutou paměť a urychlil se tak přenos dat po PCI-express sběrnici.

Posledním a hlavním cílem je využít k výpočtu i grafických karet AMD-ATI (viz tabulka současně používaných GPGPU karet) a jejich rozhranní AMD-CTM. Jejich absolutní výkon, který u nadcházející řady HD 4xxx bude přibližně 1TFlops, a vysoká propustnost paměťového subsystému z nich dělá ideálního kandidáta na GPGPU použití. Tím by byla knihovna použitelná na většině jak pracovních, tak i domácích počítačů s externí grafickou kartou. Integrované GPU (např. Intel) bohužel nedosahují takového výkonu a vývojářské podpory, aby se daly použít.

Cením si hlavně zkušenosti, kterou jsem získal s programováním SIMD paralelní architektur, které budou do budoucna velice oblíbené a používané. Důkazem je čip IBM Cell, GPU, plánované projekty Intel Larrabee, AMD Fusion a další. Musím uznat, že přechod z klasického sekvenčního programování na datově paralelní, které využívá GPU, nebyl jednoduchý a jeho programování vyžadují značnou intelektuální námahu.

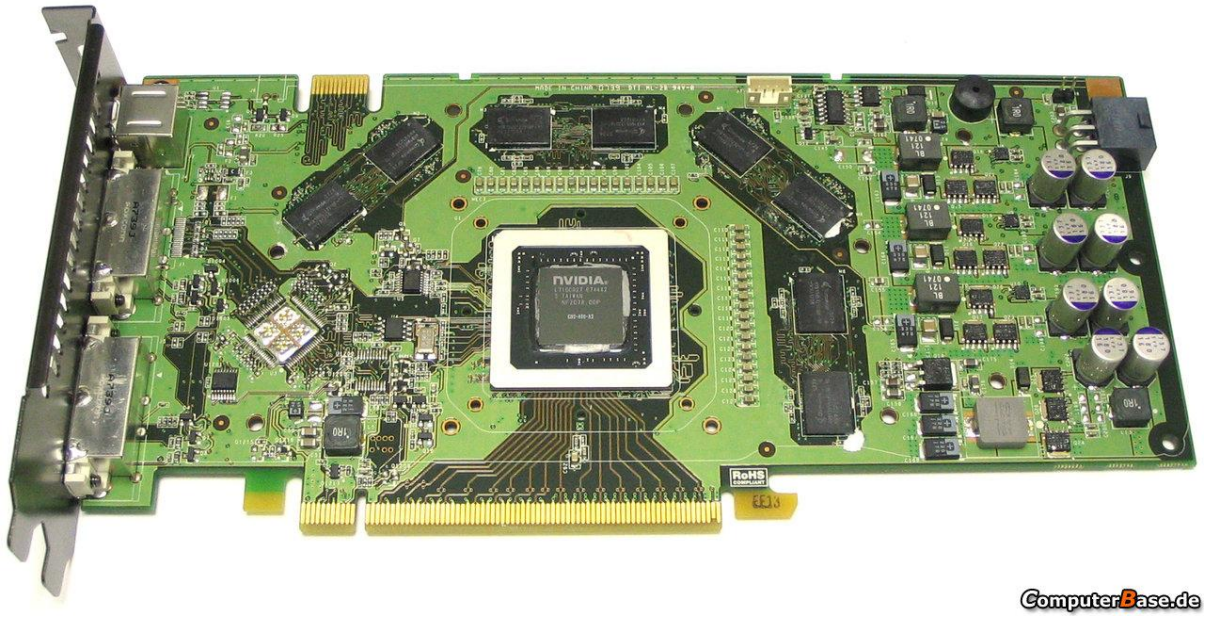
Citovaná literatura

1. **Polok, Lukáš.** *Zpracování obrazu v reálném čase, bakalářská práce.* Brno : FIT VUT v Brně, 2007.
2. **Liedtke, Jochen, a další.** How To Schedule Unlimited Memory Pinning of Untrusted Processes Or Provisional Ideas About Service-Neutrality. *www.ibm.com.* [Online] [Citace: 10. 5 2008.] http://i30www.ira.uka.de/research/documents/l4ka/ideas-about-service-neutrality.pdf.
3. **NVIDIA Corporation.** *NVIDIA CUDA Compute Unified Device Architecture - Programming Guide.* Santa Clara : NVIDIA Corporation, 2007.
4. **NVIDIA Corporation .** *The CUDA - Compiler Driver NVCC.* Santa Clara : NVIDIA Corporation, 2007.
5. **Gonzalez, Rafael C. a Woods, Richard E.** *Digital Imge Processing.* New Jersey : Prentice-Hall, Inc., 2002. ISBN: 0-201-18075-8.
6. **NVIDIA Corporation.** *CUDA-CUFFT Library.* Santa Clara : NVIDIA Corporation, 2007.
7. **Garland, Michael.** General-Purpose Computation Using Graphics Hardware. *http://www.gpgpu.org.* [Online] http://www.gpgpu.org/asplos2008/ASPLOS08-3-CUDA-model-and-language.pdf.
8. **Harris, Mark.** General-Purpose Computation Using Graphics Hardware. *http://www.gpgpu.org.* [Online] 2007. [Citace: 01. 05 2008.] http://www.gpgpu.org/sc2007/SC07_CUDA_5_Optimization_Harris.pdf.

6 Přílohy

6.1 Specifikace použité grafické karty

Typ karty: 8800 GTS 512
Velikost paměti: 512MB
Frekvence paměti: 1940 MHz (GDDR3)
Frekvence jádra: 650 MHz
Frekvence shader core: 1625MHz
Spotřeba při zatížení: 150W
Počet SIMD multiprocesorů: 16
Verze CUDA rozhraní: 1.1
Maximální počet vláken na jeden blok: 512
Maximální velikost x, y a z dimenze bloku vláken: 512, 512, 64
Maximální velikost (x, y, z)matice bloků: 65535
Velikost warp[15]: 32 vláken;
Počet registrů na multiprocesor: 8192;
Velikost sdílené paměti na multiprocesor: 16KB organizovaných do 16 banků
Velikost paměti konstant: 64KB
Konstant cache na multiprocesor: 8KB
Texture cache na multiprocesor: 8KB
Maximální počet aktivních bloků na multiprocesor: 8
Maximální počet aktivních warp na multiprocesor: 24
Maximální počet aktivních vláken na multiprocesor: 768
Maximální počet instrukcí na kernel[16]: $2*10^6$



Obrázek 17 - Grafická karta Nvidia 8800GTS 512MB

[15] Každý SIMD multiprocesor je složen z 8 procesorů, takže je schopen obsloužit nedělitelnou skupinu 32 vláken (warp) ve 4 cyklech.

[16] V tomto smyslu GPU funkce

6.2 Specifikace testovací sestavy

Procesor: Intel Core Quad 6600 2.4GHz
Základní deska: Gigabyte X38-DS5, 2 sloty PCI-express 2.0 16x
Paměť: 4GB DDR2 1066 MHz, časování 5-5-5-15
Operační systém: Windows Xp SP2, Linux – Fedora Core 8
Vývojové prostředí: Visual Studio 2005
Ostatní komponenty mají na testované hodnoty minimální vliv.

6.3 Parametry současných grafických karet využitelných v GPGPU

6.3.1 Nvidia

Model	9800GX2	9800GTX	9600GT	8800Ultra	8800GTX
Series	GeForce 9	GeForce 9	GeForce 9	GeForce 8	GeForce 8
GPU	G92 x 2	G92	G94	G80	G80
Release Date	18.3.2008	31.3.2008	21.2.2008	2.5.2007	8.11.2006
Interface	PCI-E 2.0 x16	PCI-E 2.0 x16	PCI-E 2.0 x16	PCI-E x16	PCI-E x16
Core Clock	600 MHz	675 MHz	650 MHz	612 MHz	575 MHz
Shader Clock	1500 MHz	1688 MHz	1625 MHz	1500 MHz	1350 MHz
Memory Clock	1000 MHz (2000 DDR)	1100 MHz (2200 DDR)	900 MHz (1800 DDR)	1080 MHz (2160 DDR)	900 MHz (1800 DDR)
Memory Bandwidth	128 GB/sec	70.4 GB/sec	57.6 GB/sec	103.68 GB/sec	86.4 GB/sec
FLOPS	768 GFLOPS	432.128 GFLOPS	208 GFLOPS	384 GFLOPS	345.6 GFLOPS
Shaders	2 x 128	128	64	128	128
Model	**8800GTS-512**	**8800GTS**	**8800GT**	**8600GTS**	**8600GT**
Series	GeForce 8	GeForce 8	GeForce 8	GeForce 8	GeForce 8
GPU	G92	G80	G92	G84	G84
Release Date	11.12.2007	8.11.2006	29.10.2007	17.4.2007	17.4.2007
Interface	PCI-E 2.0 x16	PCI-E x16	PCI-E 2.0 x16	PCI-E x16	PCI-E x16
Core Clock	650 MHz	500 MHz	600 MHz	675 MHz	540 MHz
Shader Clock	1625 MHz	1200 MHz	1500 MHz	1450 MHz	1180 MHz
Memory Clock	970 MHz (1940 DDR)	800 MHz (1600 DDR)	900 MHz (1800 DDR)	1000 MHz (2000 DDR)	700 MHz (1400 DDR)
Memory Bandwidth	62.08 GB/sec	64 GB/sec	57.6 GB/sec	32 GB/sec	22.4 GB/sec
FLOPS	416 GFLOPS	230.4 GFLOPS	336 GFLOPS	92.8 GFLOPS	75.52 GFLOPS
Shaders	128	96	112	32	32

Tabulka 1 - Grafické karty Nvidia

6.3.2 AMD-ATI

Model	HD4870	HD4850 DDR3	HD4850	HD 3870 X2	HD 3870
Series	Radeon HD 4k	Radeon HD 4k	Radeon HD 4k	Radeon HD 3k	Radeon HD 3k
GPU	RV770	RV770	RV770	RV670 x 2	RV670
Release Date	31.5.2008	31.5.2008	31.5.2008	28.1.2008	15.11.2007
Interface	PCI-E 2.0 x16	PCI-E 2.0 x16	PCI-E 2.0 x16	PCI-E 2.0 x16	PCI-E 2.0 x16
Core Clock	850 MHz	650 MHz	650 MHz	825 MHz	775 MHz
Shader Clock	1050 MHz	850 MHz	850 MHz	825 MHz	775 MHz
Memory Clock	1935 MHz (3870 DDR)	1143 MHz (2286 DDR)	1728 MHz (3456 DDR)	900 MHz (1800 DDR)	1125 MHz (2250 DDR)
Memory Bandwidth	123.84 GB/sec	73.152 GB/sec	110.592 GB/sec	115.2 GB/sec	72 GB/sec
FLOPS	1008 GFLOPS	816 GFLOPS	816 GFLOPS	1056 GFLOPS	496 GFLOPS
Shaders:	480	480	480	2 x 320	320
Model	**HD 3850**	**HD 3650**	**HD 3470**	**HD 2900 XT**	**HD 2600 XT**
Series	Radeon HD 3k	Radeon HD 3k	Radeon HD 3k	Radeon HD 2k	Radeon HD 2k
GPU	RV670	RV635	RV620	R600	RV630
Release Date	15.11.2007	23.1.2008	23.1.2008	14.5.2007	28.6.2007
Interface	PCI-E 2.0 x16	PCI-E 2.0 x16	PCI-E 2.0 x16	PCI-E x16	PCI-E x16
Core Clock	668 MHz	725 MHz	800 MHz	743 MHz	800 MHz
Shader Clock	668 MHz	725 MHz	800 MHz	743 MHz	800 MHz
Memory Clock	828 MHz (1656 DDR)	800 MHz (1600 DDR)	950 MHz (1900 DDR)	825 MHz (1650 DDR)	1100 MHz (2200 DDR)
Memory Bandwidth	52.992 GB/sec	25.6 GB/sec	15.2 GB/sec	105.6 GB/sec	35.2 GB/sec
FLOPS	427.52 GFLOPS	174 GFLOPS	64 GFLOPS	475 GFLOPS	192 GFLOPS
Shaders	320	120	40	320	120

Tabulka 2 - Grafické karty AMD/ATI

Karty řady Radeon HD 4k nebyly v době psaní bakalářské práce na trhu a jejich konečné parametry zatím nejsou přesně známé a jsou pouze spekulativní. Avšak pokud se budou reálné parametry těmto spekulacím alespoň přibližovat, bude stát za to je zahrnout do výběru při rozhodování jakou grafickou kartu pro GPGPU.

Další výhodou všech GPU AMD-ATI v tabulce je nativní podpora 64bitových FP čísel (double). Záleží na oblasti využití, ale 32bit FP (single) nemusí v některých případech stačit (například numerické derivace apod.). Nvidia uvede podporu práce s těmito čísly až s nadcházející generací GPU.

6.4 Značení proměnných

P – výkonový zisk
$t_{GPUTotal}$ – celková doba GPU výpočtu
$t_{CPUTotal}$ – celková doba CPU výpočtu
$t_{TransferRAMtoGRAM}$ – doba přenosu do GPU
$t_{TransferGRAMtoRAM}$ – doba přenosu z GPU
t_{GPU} – doba GPU výpočtu
C – relativní aritmetická složitost
k – celkový počet instrukcí
k_{Idata} – celková velikost vstupních dat
k_{Odata} – celková velikost výstupních dat
$v_{RAMtoGRAM}$ (k_{Idata}) – rychlost přenosu dat z RAM do GRAM
$v_{GRAMtoRAM}$ (k_{Odata}) – rychlost přenosu dat z GRAM do RAM
$v_{GPUProcess}$ – rychlost zpracování instrukcí na GPU
$v_{CPUProcess}$ – rychlost zpracování instrukcí na CPU
k – Poměr velikosti načítaných dat u GPU a CPU
$tile.x$ – šířka zpracovávané oblasti
$tile.y$ – výška zpracovávané oblasti
$convRadius$ – poloměr konvoluční matice
$threadDim.x$ – šířka matice vláken, která zpracovává jednu oblast
$threadDim.y$ – výška matice vláken, která zpracovává jednu oblast

6.5 Popis rozhranní knihovny

Rozhranní knihovny je definováno v hlavičkovém souboru cvGPUlib.h, které stačí vložit do uživatelského programu.

Funkce mohou pracovat in-place, pokud není definováno jinak. Pokud není operace podporování, funkce vrací `G_NON_SUPPORTED`, naopak pokud proběhne úspěšně `G_OK`.

6.5.1 Konverze barevného modelu

6.5.1.1 Konverze modelu váhovou maticí

Provádí konverzi barevného modelu vstupního obrázku `gIImage`. Výsledek je uložen do výstupního obraz `gOImage`. Konverze je definována kódem `code`.

```
gpuStatus GPUcvCvtColor (gpuIplImage *gIImage, gpuIplImage *gOImage,
int code);
```

Podporované jsou 3 i 4 kanálové obrázky s bitovou hloubkou 8U a 32F.

Podporované hodnoty `code`:

- `CV_BGR2XYZ` – konverze modelu BRG na XYZ
- `CV_RGB2XYZ` – konverze modelu RGB na XYZ
- `CV_XYZ2BGR` – konverze modelu XYZ na BRG
- `CV_XYZ2RGB` – konverze modelu XYZ na BRG
- `CV_RGB2Gray` – konverze modelu BRG na úrovně šedé.
- `CV_BGR2Gray` – konverze modelu RGB na úrovně šedé.

Funkce výše pouze volá funkci GPUcvCvtColorUserDefined, pouze jí předává správně vyplněné konverzní matice. Jestliže uživatel potřebuje definovat vlasttní konverzní matici, může tuto funkci volat přímo s touto definicí:

```
gpuStatus GPUcvCvtColorUserDefined (gpuIplImage *gIImage,
gpuIplImage *gOImage, float* symbolMatrix, float* updateVector);
```

`symbolMatrix` je matice float čísel o velikosti 4x4. `updateVector` je vektor o velikosti 1x4. Pro více informací o jejich významu viz implementace.

6.5.1.2 Konverze do modelu LAB

Knihovna využívá stejně jako OpenCv shodnou hlavičku funkce jako v předchozím případě, kód je ovšem `CV_BGR2Lab` nebo `CV_RGB2Lab`. Podporovaná jsou pouze 4 kanálové 32f obrázky, z jiných formátů musí být na tento převedeny pomocí konverzních funkcí.

6.5.2 Konvoluční filtry

Funkce nepracující in-place.

```
gpuStatus gpuCvFilter2D( gpuIplImage *gIImage, gpuIplImage *gOImage, const
CvMat* kernel, int radius);
```

Vstupem je obrázek `gIImage`, výstupem `gOImage`. `CvMat` je matice 32f čísel definující konvoluční jádro, a `radius` je poloměr konvolučního filtru. Podporovány jsou pouze 4 složkové 8U obrázky.

FFT filtry

```
gpuStatus gpucvFFTFilter(gpuIplImage *gIImage, gpuIplImage *gOImage,
gpuIplImage *gOSpektrum, gpuIplImage *gIMask);
```

Vstupem je obrázek `gIImage`, výstupem `gOImage` a spektrum `gOSpektrum`. `gIMask` je matice 32f čísel definující filtrovací masku. Ta musí mít, stejně jako výstupní obrázek, stejnou velikost jako vstupní obrázek. Podporovvány jsou pouze jednosložkové 32F obrázky.

6.5.3 Ostatní funkce

Tato kapitola popisuje funkce, které nebyly diskutovány v teoretické části, avšak jsou v knihovně obsaženy.

6.5.3.1 Inicializační funkce

```
extern "C" gpuStatus gpuInit ();
```

Před první použitím kterékoli funkce z knihovny musí být GPU inicializována. To zajistí tato funkce.

6.5.3.2 Fukce pro konverzi IplImage na gpuIplImage, kopírování z a do GPU

```
extern "C" gpuStatus gpuCreateImage (CvSize size, int depth, int channels,
gpuIplImage **gImage);
```

Funkce pro vytvoření struktury obrázku. Uvnitř této funkce je alokova struktura IplImage, ke které lze přistupovat.

```
extern "C" gpuStatus gpuCreateTMPImage (CvSize size, int depth, int
channels, gpuIplImage **gImage);
```

Funkce pro vytvoření struktury obrázku. Uvnitř této funkce je alokova struktura IplImage, ale naozdíl od předchozí nejsou alokována data na straně procesoru. Kopírování z a do paměti RAM způsobí chybu.

```
extern "C" gpuStatus gpuReleaseImage (gpuIplImage *gImage);
extern "C" gpuStatus gpuReleaseTMPImage (gpuIplImage *gImage);
```

Kompletní uvolnění obrázku.

```
extern "C" gpuStatus gpuCopyImage (gpuIplImage *gImage, gpuDirection
direction);
```

Kopírování obrázku z RAM (gpuDirection == G_RAM_TO_GPU) do GRAM a naopak (gpuDirection == G_GPU_TO_RAM)

6.5.3.3 Testovací funkce

```
extern "C" gpuStatus gpuMakeStripes (gpuIplImage *gOImage);
```

6.5.3.4 Operace s alfa kanálem

```
extern "C" gpuStatus gpuAddAlpha (gpuIplImage *gIImage, gpuIplImage
*gOImage);
extern "C" gpuStatus gpuRemoveAlpha (gpuIplImage *gIImage, gpuIplImage
*gOImage);
```

6.5.3.5 Změna barevné hloubky

```
extern "C" gpuStatus gpuConvertToUchar (gpuIplImage *gIImage, gpuIplImage
*gOImage);
extern "C" gpuStatus gpuConvertToFloat (gpuIplImage *gIImage, gpuIplImage
*gOImage);
```