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

VYUŽITÍ GPU PRO AKCELEROVANÉ ZPRACOVÁNÍ OBRAZU

BAKALÁŘSKÁ PRÁCE
BACHELOR´S THESIS

AUTOR PRÁCE Lukáš Horák
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

VYUŽITÍ GPU PRO AKCELEROVANÉ ZPRACOVÁNÍ OBRAZU

IMAGE PROCESSING ON GPU

BAKALÁŘSKÁ PRÁCE
BACHELOR´S THESIS

AUTOR PRÁCE Lukáš Horák
AUTHOR

VEDOUCÍ PRÁCE Doc. RNDr. Pavel Smrž, Ph.D.
SUPERVISOR

BRNO 2007

Zadání bakalářské práce/6316/2007/xhorak48

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav počítačové grafiky a multimédií Akademický rok 2007/2008

Zadání bakalářské práce

Řešitel: **Horák Lukáš**
Obor: Informační technologie
Téma: **Využití GPU pro akcelerované zpracování obrazu**
Kategorie: Počítačová grafika

Pokyny:

1. Prostudujte základy zpracování obrazu.
2. Analyzujte současné možnosti využití programovatelných GPU pro zpracování obrazu.
3. Vyberte vhodné algoritmy (např. filtrace obrazu, detekce hran, DFT, atd.) a navrhněte metodu jejich akcelerovaného výpočtu pomocí GPU.
4. Experimentujte s vaší implementací a případně navrhněte vlastní modifikace.
5. Porovnejte dosažené výsledky a diskutujte možnosti budoucího vývoje. Zvažte další pokračování v rámci diplomové práce.
6. Vytvořte stručný plakát prezentující vaši bakalářskou práci, její cíle a výsledky.

Literatura:

- Dle pokynů vedoucího.
- http://www.gpgpu.org/
- http://developer.nvidia.com/object/cuda.html

Při obhajobě semestrální části projektu je požadováno:

- Splnění prvních tří bodů zadaní.

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
Konzultant: Španěl Michal, Ing., UPGM FIT VUT
Datum zadání: 1. listopadu 2007
Datum odevzdání: 14. května 2008

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

L.S.

doc. Dr. Ing. Pavel Zemčík
vedoucí ústavu

LICENČNÍ SMLOUVA
POSKYTOVANÁ K VÝKONU PRÁVA UŽÍT ŠKOLNÍ DÍLO

uzavřená mezi smluvními stranami

1. Pan

Jméno a příjmení: **Lukáš Horák**
Id studenta: 78992
Bytem: Boleradice 133, 691 12 Boleradice
Narozen: 14. 03. 1986, Hustopeče u Brna
(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: Využití GPU pro akcelerované zpracování obrazu
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

Akceleraci zpracování obrazu lze provést více způsoby. Tato práce se zaměří na novou technologii využívající grafické procesory s architekturou, která umožňuje univerzální využití. Architektura nese název NVIDIA CUDA™ (Compute unified device architecture). Práce poučí o této technologii a ukáže základní terminologii nutnou pro pochopení následujících kapitol. Pro otestování této technologie je zvoleno zpracování obrazu pomocí konvoluce. Ukážou se některé počáteční a jiné chyby návrhu algoritmu a směr dalšího možného urychlení.

Klíčová slova

Akcelerace zpracování, obraz, CUDA, konvoluce.

Abstract

Acceleration of image processing can be done by more than one way. This work is oriented to new technology using graphics processing unit with architecture which allows universal usage. The architecture has called as NVIDIA CUDA™ (Cumpute unified device architecture). This work instructing about this technology and show basic terminology needed to understand other chapters. Convolution image processing is selected to testing this technology. It describes some first mistakes in algorithm design and show other steps of acceleration.

Keywords

Acceleration of processing, image, Compute unified device architecture, convolution filter.

Citace

Lukáš Horák: Využití GPU pro akcelerované zpracování obrazu, bakalářská práce, Brno, FIT VUT v Brně, 2008

Využití GPU pro akcelerované zpracování obrazu

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením Doc. RNDr. Pavla Smrže, Ph.D. Další informace mi poskytl Ing. Michal Španěl. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……………………
Jméno Příjmení
Datum

Poděkování

Rád bych poděkoval za připomínky a odborné rady vedoucímu mé diplomové práce Doc. RNDr. Pavlu Smržovi, Ph.D. a také Ing. Michalu Španělovi. Dále děkuji Pavlu Hrabcovi za poskytnutí grafického adaptéru, bez kterého bych tuto práci nedokončil.

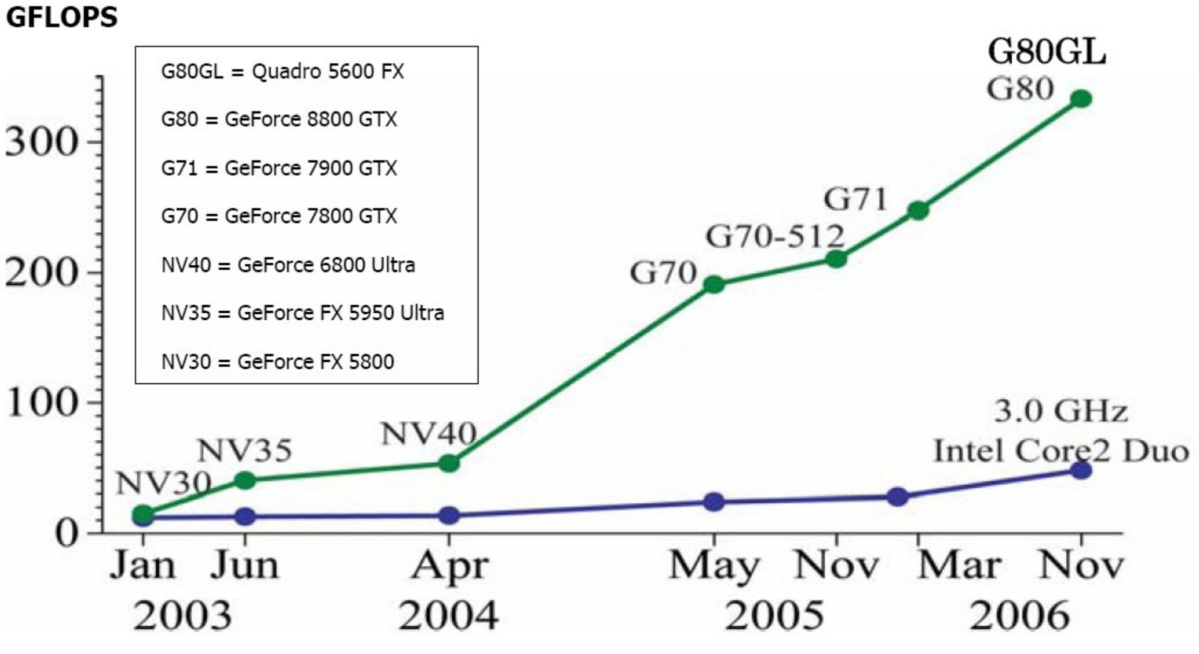
Obsah

1 Úvod

Existují algoritmy pro zpracování obrazu, které stále neběží v okamžitém čase na dnešních procesorech. Taktovací frekvence procesorů, podle které se odvíjí rychlost zpracování informací, je na hranici možností používané technologie. Vývoj procesorů se však nezastavil. Provádí se výzkumy nových technologií, které se možná někdy budou využívat pro stavbu procesorů.

Druhou cestou vývoje je paralelizace. Již procesory Intel® Pentium® měly dvě pipeline a za předpokladu dobře optimalizovaného kódu na úrovni assembleru se skutečně paralelně prováděly dvě instrukce. Ovšem jen instrukce, nikoliv procesy. Dnes jsou na trhu procesory, které běžně disponují dvěmi i čtyřmi jádry. Souběžně může běžet více procesů nebo vláken, což je značný pokrok a časová úleva, pokud se algoritmus může rozdělit.

Některé výpočetní úlohy trvají poměrně dlouho i při využití vícejádrových procesorů. Moderní grafické procesory disponují mnohem větším výkonem, jak zobrazuje Ilustrace 1, díky vysokému počtu jader. Nezávislost jader mezi sebou je určitým způsobem omezena a možnosti využití také.



Ilustrace 1: Počet operací v plovoucí řadové čárce za sekundu pro CPU a GPU (Ilustrace převzata z knihy [1])

Zobrazení virtuální reality v dnešních počítačových hrách se děje hlavně pomocí tzv. Vertex a Pixel (v OpenGL terminologii známé pod názvem Fragment) shaderů. Jsou to krátké programy, které běží na grafických kartách. Tyto programy proudově a velice rychle zpracovávají geometrii a textury k zobrazení. Jsou určitými pravidly omezeny, ale dokážu si představit verze jednodušších algoritmů zpracování obrazu, které by mohly využít tuto technologii.

Zajímavou novinkou jsou grafické procesory od firmy NVIDIA s kódovým označením GeForce řady 8 a 9, Tesla a Quadro. Tyto procesory patří do množiny CUDA-Enabled GPU (Compute unified device architecture), což znamená, že grafické procesory mají sjednocenou architekturu a lze je využít pro obecné programy (možnosti využití nejsou tak omezené jako dříve). S trochou nadsázky lze říct, že výše uvedené procesory se mohou využít téměř na cokoliv.

Záměrem této bakalářské práce je zkoumat možnosti těchto nových grafických procesorů a jejich využití pro akcelerované zpracování obrazu.

Úvodní druhá kapitola popisuje architekturu CUDA-Enabled grafických procesorů, proč se možná stane fenoménem při stavbě superpočítačů, a toolkit, který je potřebnou součástí práce s programy využívající výpočetní sílu těchto procesorů.

Třetí kapitola nás seznámí s rozšířením jazyka C/C++. Rozšíření se týká definice a deklarace funkcí a proměnných. Zajímavým prvkem rozšíření je syntaxe spuštění funkce, která má být spuštěna na grafickém procesoru. Kapitola tedy tvoří potřebný teoretický základ pro orientaci v rozboru a návrhu algoritmů.

Ve čtvrté kapitole popisuji vybraný algoritmus pro zpracování obrazu, navrhnu více verzí, které také implementuji a neustále vylepšuji pro získání uspokojivých výsledků.

Rychlost implementovaných verzí algoritmů je tématem páté kapitoly. Ukážu výsledky rychlostí mezi algoritmem napsaným pro CPU a algoritmy napsanými pro grafické procesory.

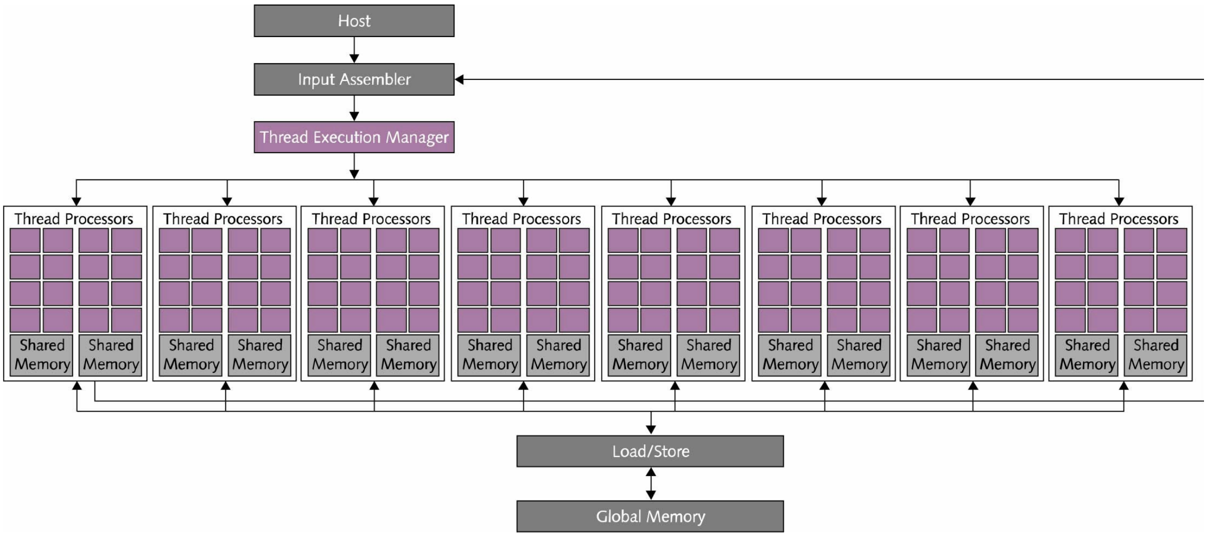
2 NVIDIA CUDA^{TM} technologie

NVIDIA CUDA^{TM} Technologie využívá jen jazyk C pro zpřístupnění výpočetní síly NVIDIA CUDA - Enabled grafických procesorů. Umožňuje programátorům využít tyto grafické procesory pro téměř jakékoliv výpočty a simulace. Důmyslně navržená architektura grafických procesorů umožňuje přenositelnost mezi jakýmkoliv CUDA-Enabled grafickým procesorem. Procesor může být jak výkonově nižší v notebooku, tak i více procesorů vysoce výkonné sestavy.

Zajímavou řadou výrobků firmy NVIDIA je Tesla, což jsou počítačové sestavy „nacpané“ neobvykle větším počtem grafických procesorů než je zvykem u high-end počítačových sestav. Důvodem je jejich využití pro obecné výpočty, nikoliv pro hraní her. Jedna verze sestavy je dokonce i v RACK provedení, lze ji využít pro stavbu superpočítačů.

2.1 Mnohem více výpočetních jader

Grafické procesory obsahují mnohem více výpočetních jader, které mohou být využity k masivnímu paralelnímu chodu úlohy. Návrh a programování paralelních úloh se musí řídit určitými zákony, vycházející z technologických limitů této architektury.

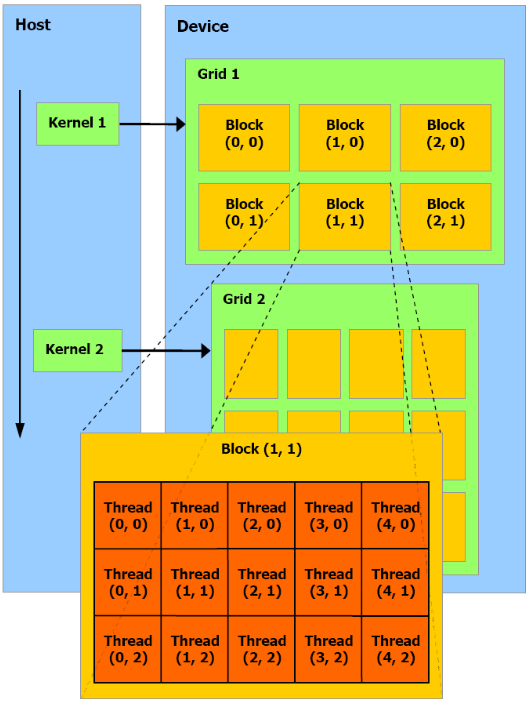


Ilustrace 2: Architektura z pohledu programátora (Ilustrace převzata z knihy [5])

Rozhraní umožňuje základní přístup k jeho globální paměti právě tak, jak CPU přistupuje do operační paměti. Umožňuje se tak větší flexibilita programu. Grafický procesor využívá sdílené paměti s velice nízkou dobou přístupu. Využití této paměti minimalizuje přístup ke globální paměti. Jedná se tedy o obdobu L1 a L2 cache u CPU.

Operační systém (a ovladač zařízení) je zodpovědný za provádění multitaskingu mezi aplikacemi využívající grafickou kartu.

Části programu, které se využívají nejčastěji a mohou běžet paralelně, se izolují do funkcí zvaných *__global__*, ty podrobněji popíši v následující kapitole. Uvedená funkce může běžet na grafické kartě ve více vláknech. Vlákna jsou organizována do bloků. Každá *__global__* funkce je spuštěna jako dávka vláken organizovaných do bloků jednoho gridu.



Ilustrace 3: Organizace běžících vláken

(Ilustrace převzata z knihy [1])

Vláknový blok je soubor vláken, které plně spolupracují a sdílejí data. Vlákna jednoho bloku jsou synchronizována.

Každé vlákno je identifikováno jeho ID. Identifikace nebo-li adresace je založena na parametru, který nabývá až tří rozměrů. Účel tohoto způsobu adresace je dokonalý. Máme například za úkol dávkově zpracovat data jednorozměrného pole o velikosti 30 prvků. Využijeme adresaci pomocí jednorozměrného parametru. Spustíme 30 vláken a každé vlákno zpracovává jeden prvek. Vlákno s adresací pomocí dvourozměrného parametru *(x, y)* využijeme nejspíše pro zpracování dvourozměrné pole a třírozměrný parametr pro třírozměrné pole *(x, y, z)*.

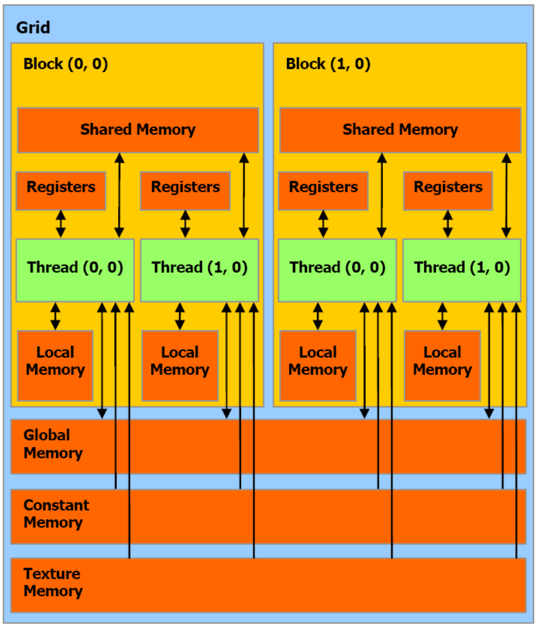
Účelem bloku je oddělit vlákna tak, aby mezi sebou nebyla synchronizována. Dvě vlákna z různých bloků se nesynchronizují a nekomunikují spolu, čímž se dosáhne vyššího výkonu. Čím více nesynchronizovaných vláken, tím vyšší výkon získáme. Ovšem počet souběžně spuštěných bloků je omezený a využít jen jedno vlákno v bloku je plýtvání výkonem. Každý grafický procesor této architektury má k dispozici multiprocesory, ty realizují bloky, a k nim přiřazený počet stream procesorů, které realizují synchronizovaná vlákna. Počet se pohybuje kolem 14 multiprocesorů a 112 stream procesorů - jednomu multiprocesoru je přiřazeno 8 stream procesorů.

Každý blok je identifikován jeho ID. Blok je adresován pomocí až třírozměrného parametru a jsou organizovány do gridu. Grid je celkové zastřešení všech využitých bloků.

2.2 Paměťový model

Každé vlákno běžící na grafické kartě má přístup k následujícím typům paměti:

- Registry vlákna
- Lokální paměť vlákna
- Sdílená paměť bloku
- Globální paměť gridu
- Paměť konstant gridu
- Paměť textur gridu



Ilustrace 4: Paměťový model (Ilustrace převzata z knihy [1])

Globální paměťový prostor, paměťový prostor konstant a paměťový prostor textur lze využívat aplikací přímo. Jsou také sdíleny všemi funkcemi dané aplikace. Navíc jsou optimalizovány pro různé používání. Paměťový prostor textur nabízí i různé módy adresace využitelné pro filtrování dat (pro různé formáty obrazových dat).

Paměť textur je navržena pro rychlé čtení a sdílení s více vlákny. Má k dispozici vyrovnávací paměť. Přístup do této paměti je rychlejší než přístup do globální paměti. Velikost alokované paměti musí být zarovnaná.

Jistě zajímavou možností je spárování alokované textury s některým z nejvyužívanějších 3D API. Grafická karta provede určitou změnu textury pomocí CUDA technologie a výsledek se zobrazí například v 3D scéně pomocí OpenGL.

2.3 Více grafických karet v počítači

Používání dvou či více grafických karet lze jen při použití stejného typu. Při zapnutém SLI režimu lze využívat jen jednu grafickou kartu, je tedy nutné SLI režim vypnout v ovládacím panelu ovladače.

2.4 Toolkit

CUDA^{TM} Toolkit je prostředí jazyka C, které obsahuje speciálně upravený kompilátor jazyka C, potřebné hlavičkové soubory a knihovny, knihovny CUFFT a CUBLAS, Profiler, ovladač zařízení a manuál. Prostředí je dostupné pro operační systémy Mac OS X, Linux (32-bit/64-bit) a MS Windows XP (32-bit/64-bit).

2.4.1 Kompilátor

NVCC je kompilátor pro CUDA zdrojové kódy. Jeho práce spočívá v rozdělení kódu na dvě části. Jedna část kódu poběží na CPU a druhá na GPU. Tyto dvě části se zkompilují pomocí stávajícího kompilátoru (MS Visual C++ kompilátor, gcc). Přesnější informace funkce a využití tohoto kompilátoru jsou v jeho manuálu, knize [4]. Kompilaci zdrojových kódů se lze naučit z ukázek v SDK pro Windows i pro Linux.

Při používání prostředí MS Visual Studia jsou chybová hlášení NVCC kompilátoru překopírována do okna Error list jako položky. Toto prostředí neparsuje hlášení z uvedeného kompilátoru dobře (například položka v okně Error list „Error: Error“ nenapoví, jakou chybu hledat). Kompletní informace jsou vypsány v okně Output.

Kompilátor NVCC dodávaný se současnou vydanou verzí toolkitu 1.1 má problém využívat v linuxu kompilátor gcc/g++ verze 4.2. Nekompatibilitu řeší veřejná verze toolkitu 2.0 beta.

2.4.2 Knihovny CUFFT a CUBLAS

Toolkit obsahuje knihovnu pro práci s rychlou fourierovou transformací (Fast fourier transformation) pro komplexní i reálná čísla a knihovnu obsahující základní funkce lineární algebry (Basic Linear Algebra Subroutines), čímž se myslí vektorové a maticové operace. Obě knihovny zpřístupňují funkce již implementové na GPU. Zajímavostí je, že tyto dvě knihovny nepotřebují přístup přes ovladače grafické karty. Více informací k nalezení je v knihách [2] a [3].

Knihovna CUBLAS je jistě přínosem pro aplikace využívající matice s velkým počtem prvků.

Firma NVIDIA jde vývojářům vstříc a na požádání již některým jedincům zpřístupnila zdrojové kódy těchto knihoven, takže je lze doplňovat i o další funkce.

3 Rozšíření jazyka C/C++

Programování aplikací pro CUDA-Enabled grafické karty je založeno na programování v jazyce C. Pro tyto účely se jazyk C rozšířil o klíčová slova, novou syntaxi a zabudované typy a konstanty. Aby nedošlo k nedorozumění, název kapitoly obsahuje jazyk C++, protože kompilátor dokáže pracovat s jednoduchými šablonami funkcí. Podpora tříd je téměř nulová, z mých zkušeností nevyužitelná.

3.1.1 Deklarace funkcí

Mezi důležitá klíčová slova patří slovo __*global*__, které deklaruje funkci jako podprogram běžící na GPU. Je viditelná jen pro funkce běžící na CPU a nesmí nic vracet pomocí klíčového slova *return*.

Funkce deklarovaná pomocí klíčového slova __*device*__ je také podprogramem běžícím na GPU, ale je viditelná jen pro funkce deklarované klíčovými slovy __*device*__ a __*global*__. Tato funkce se zpravidla zkompiluje jako *inline* funkce.

Klíčové slovo __*host*__ deklaruje funkci běžící na CPU, je viditelná jen pro funkce neběžící na GPU. Využívá se nejčastěji jako kombinace s klíčovým slovem __*device*__. Takto deklarovaná funkce se zkompiluje ve dvojím provedení - GPU i CPU. Toto řešení je v některých případech výhodné.

Omezení funkcí deklarovaných pomocí výše uvedených klíčových slov je uvedeno v podkapitole 4.2.1.4 knihy [1]. Jedno z největších omezení je, že funkce deklarované pomocí __*global*__ a __*device*__ nesmí provádět rekurzivní volání a nesmí mít libovolný počet parametrů.

Funkce kompilované pomocí NVCC kompilátoru nejsou viditelné pro linker. Aby byly viditelné ve zdrojovém kódu kompilovaného pomocí stávajícího překadače, musí mít před vlastní deklarací a definicí následující řetězec: *extern "C"*.

3.1.2 Deklarace proměnných a konstant

Klíčové slovo __*device*__ lze využít i při deklaraci proměnné. Takto deklarovaná proměnná je umístěna do globální paměti grafické karty po celou dobu běhu aplikace. Je přístupná pro všechna vlákna.

Podobně klíčové slovo __*constant*__ deklaruje konstantu, umístěnou do paměti konstant. Je přístupná pro všechna vlákna, dokonce i pro CPU.

Klíčové slovo __*shared*__ deklaruje proměnnou, umístěnou do sdílené paměti bloku vláken. Po celou dobu využívání daného bloku je přístupná všem vláknům v bloku.

Omezení výše uvedených deklarací je popsáno v podkapitole 4.2.2.4 knihy [1].

3.1.3 Syntaxe konfigurace spuštění

Jakékoliv volání funkce deklarované s klíčovým slovem *__global__* musí mít určenou konfiguraci spuštění. Konfigurace určuje rozměr gridu a bloků, které se využijí při rozložení běhu této funkce na vlákna a bloky. Syntaxe je rozšířena o formuli <<< *Dg, Db, Ns, S* >>> umístěnou mezi název funkce a argumenty funkce:

Foo<<<Dg, Db, Ns, S>>>(params);

Dg je parametr zabudovaného typu *dim3* a specifikuje velikost gridu. Parametr Db je typu *dim3* a určuje rozměry bloku. Parametr Ns je typu *size_t* a určuje velikost paměti v bytech, které jsou dynamicky alokované pro každý blok. Tato dynamicky alokovaná paměť je použita pro *__shared__* dynamickou proměnnou (více v podkapitole 4.2.2.3 knihy [1]). *S* je parametr zabudovaného typu *cudaStream_t* a specifikuje využívaný stream. Je to nepovinný parametr s výchozí hodnotou 0.

Funkcím deklarovaným s klíčovým slovem *__global__* jsou zpřístupněny následující proměnné jen pro čtení: *blockIdx* typu *dim3* určující index bloku v gridu a *threadIdx* také typu *dim3* určující index vlákna v bloku. Další proměnná *gridDim* je typu *uint3* a obsahuje velikost gridu spuštěné úlohy. Proměnná *blockDim* je také typu *uint3* a obsahuje velikost bloku spuštěné úlohy. Tyto proměnné se využívají pro identifikaci vlákna a určení jeho podílu práce.

Spouštěcí manažer, který je součástí ovladače, rozvrhne požadavek na vytvoření vláken do multiprocesorů grafické karty (prozatím obvyklý počet je 12, 14 a 16). Když je požadavek nastaven na více vláken než je k dispozici stream procesorů a multiprocesorů, manažer využije tyto výpočetní prostředky vícekrát. Tato vlastnost je výhodná pro přechod na jinou architekturu grafického procesoru (i budoucí) s jiným počtem výpočetních prostředků.

Po zavolání funkce deklarované s klíčovým slovem *__global__* je dobré zabezpečit běh programu odchycením výjimky pomocí makra *CUT_CHECK_ERROR("message").* Následujícím zavoláním *CUDA_SAFE_CALL(cudaThreadSynchronize())* program počká, až všechna vlákna *__global__* funkce doběhnou. Bez tohoto opatření se může stát, že bude program pokračovat paralelně s dobíhající ulohou na GPU.

4 Metody zpracování obrazu

Jako obraz si představujeme nějaký (nástěnný) obrázek, obraz televize, monitoru. Pro zpracování můžeme obraz pochopit jako soustavu obrazových signálů, které jsou multidimenzionální. Při formálním vyjadřování je nejčastěji hodnotou obrazové funkce jas, což je obrazová veličina odpovídající optickému vnímání člověkem.

4.1 Konvoluční filtr

Kapitola nás seznámí s teorií o konvolučních filtrech. Jelikož budeme pracovat s diskrétním obrazem, budeme celou dobu mluvit o diskrétních konvolučních filtrech. Co to je konvoluční filtr? V podstatě je to matice, jako následující:

$$\begin{pmatrix} okolí & okolí & okolí \\ okolí & PIXEL & okolí \\ okolí & okolí & okolí \end{pmatrix} / faktor + offset$$

Pixel, který zpracováváme pomocí výše popsané matice má osmi-okolí. Prvek matice určuje váhu dané složky. Výsledná hodnota je hodnotou jasu, která se získá součtem součinů vstupního obrazu a matice. Součet je dělen faktorem a nakonec se přičte kompenzační hodnota. Matematický zápis výpočtu může mít tvar:

$$O(x, y) = \left(\sum_{ix=-w}^{w} \sum_{iy=-h}^{h} I(x+ix, y+iy)\, m(ix, iy) \right) / faktor + offset$$

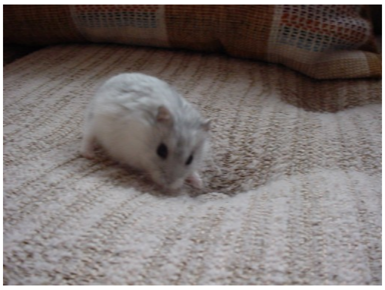
V tomto případě je *I(x, y)* vstupní obraz, *m(ix, iy)* prvky matice, *O(x, y)* výstupní obraz, *w* a *h* poloviční rozměry matice.

Konvoluční matice může mít i různé velikosti než 3x3 a může mít i více hran. Čím větší je matice, tím méně můžeme zpracovat pixelů z celého obrazu. Pixely, které nemají okolí potřebné k výpočtu, se většinou nezpracovávají. Existují i speciální verze zpracování pomocí konvolučního filtru, který zpracuje celý obraz. U této varianty se vynechávají nepoužitelné prvky matice a dynamicky se mění faktor.

4.1.1 Seznam nejznámějších matic a jejich výsledný efekt

- Rozostření (Smoothing) – Okolí má stejnou váhu, výsledná hodnota je průměrem. Potlačují se vyšší frekvence obrazové funkce.

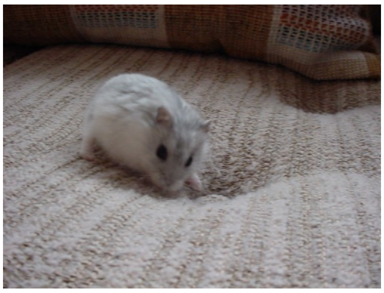
$$\begin{pmatrix}1 & 1 & 1\\1 & 1 & 1\\1 & 1 & 1\end{pmatrix}/9+0$$



Ilustrace 5: Rozostření

- Gaussovo vyhlazení (Gaussian Blur) – Okolí má váhu rozprostřenou do kruhu, výsledný obraz vypadá jako se špatným nastavením ohniska fotoaparátu.

$$\begin{pmatrix}1 & 2 & 1\\2 & 4 & 2\\1 & 2 & 1\end{pmatrix}/16+0$$



Ilustrace 6: Gaussovo vyhlazení

- Zaostření (Sharpen) – S porovnáním gaussova vyhlazení tato matice vypadá téměř stejně, ale okolí má záporné hodnoty. Stupeň zaostření se zvýší spolu se zvýšením váhy hlavního pixelu. Zaostření zdůrazňuje vyšší frekvence, které zvýrazní hrany v obraze, ale zvýrazní i šum. Pro lepší demonstraci zkusím zaostřit neostrý obraz.

$$\begin{pmatrix}0 & -2 & 0\\-2 & 11 & -2\\0 & -2 & 0\end{pmatrix}/3+0$$

Ilustrace 7: Zaostření

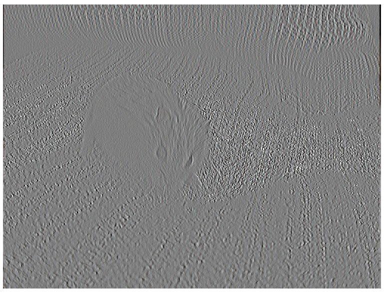
- Zaostření (Mean removal) – Tento filtr je podobný předchozímu zaostření, avšak pracuje i s diagonálními hodnotami okolí. Opět pro lepší demonstraci použiji neostrý obraz.

$$\begin{pmatrix} -1 & -1 & -1 \\ -1 & 9 & -1 \\ -1 & -1 & -1 \end{pmatrix} / 1 + 0$$

Ilustrace 8: Zaostření (Mean removal)

- Reliéf (Embossing) – Reliéf pracuje se zápornými hodnotami okolí, ale i s kompenzační hodnotou, kterou jsme prozatím nechávali nulovou. Kompenzační hodnota nám posune výslednou hodnotu z černé do odstínu šedé pro snadné zobrazení (záporné výsledné hodnoty se totiž ořezávají na nulu). Konvoluční matice má více podob. Mezi nejrozšířenější patří horizontalní detekce, vertikální detekce, horizontálně vertikální detekce, detekce ve všech směrech...

$$\begin{pmatrix} -1 & 0 & -1 \\ 0 & 4 & 0 \\ -1 & 0 & -1 \end{pmatrix} / 1 + 127$$

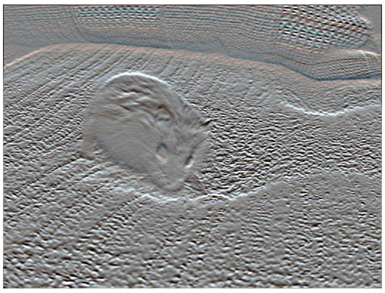


Ilustrace 9: Reliéf

- Detekce hran (Edge detection) – Horizontální a vertikální detekce. Známější verze detekce hran je bez kompenzační hodnoty, kdy je výsledný obraz laděn do černé.

$$\begin{pmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{pmatrix} / 1 + 127$$

$$\begin{pmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{pmatrix} / 1 + 127$$



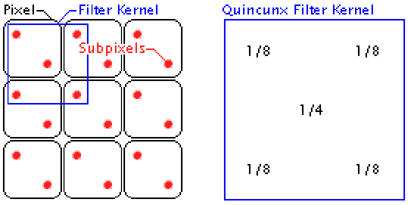
Ilustrace 10: Detekce hran

4.2 Filtr anti-aliasingu - supersampling

Anti-aliasing je technika pro zjemnění ostrých hran, které vznikají, když je vzorkovací frekvence nižší než dvojnásobek frekvence vzorkovaného obrazu.

Jako anti-aliasing obrazu se často používá tzv. supersampling. V této technice se pixel obrazu skládá z jemnějších prvků, subpixelů, které mohou být v pixelu rozmístěny do různých geometrických útvarů – jejich poloha nemusí vytvářet jen čtverec, ale i kosodelník.

Výsledná barva (intenzita barev) se získá vhodnou konvoluční maticí. Konvoluce se nemusí provádět jen na jednom pixelu, ale může získávat hodnoty i ze subpixelů sousedních pixelů.



Ilustrace 11: Ukázka subpixelů a konvoluce (Ilustrace převzata z článku [6])

5 Implementace

Již výše popsaný anti-aliasing používá konvoluci a potřeboval by speciální testovací data na odzkoušení. Proto implementuji jen konvoluci, které stačí obraz v běžné digitální podobě – pole jasu obrazových bodů s jednotlivými složkami barev. Anti-aliasing se z implementované konvoluce může odvodit.

5.1 Implementace testovacího rozhraní

Jak již téma práce napovídá, půjde o práci s obrazem. Pro testovací účely nám postačí statické obrazy načítané ze souborů.

Pro načítání obrazů využiji služeb knihovny FreeImage (implementace není tématem této bakalářské práce). Knihovna podporuje platformu MS Windows 32-bit, Linux a Mac OS X. Umí pracovat s třiceti formáty grafických souborů (další lze pomocí pluginů přidat), z čehož využijeme nejčastěji asi jen tři.

Protože budu pracovat s větším počtem verzí algoritmů pro zpracování obrazu, je dobré si vytvořit jednotné a jednoduché rozhraní. Algoritmy mohou být zpracovány nejen jako dynamické knihovny (implementováno jen pro MS Windows), ale i jako součást testovací aplikace. O všechny algoritmy se stará třída *Manager*. Algoritmy zpracování obrazu se používají prostřednictvím třídy *Algorithm*. Algoritmus může být součástí dynamické knihovny, která je zastřešena třídou *Library.*

Aplikace je napsána v jazyce C++, zdrojové kódy algoritmů zpracování obrazů v jazyce C. Rozhraní aplikace je konzolové, není nutné vytvářet složitá okna.

Aplikace je testovacím rozhraním, výsledek se ukládá do výstupního souboru. Pro spuštění je nutné specifikovat několik parametrů aplikace. Důležitými parametry jsou použitý algoritmus ke zpracování obrazu, vstupní obrázek a výstupní obrázek. Aplikace umožňuje i specifikovat parametry určené jen pro použitý algoritmus (např. soubor s maticí konvoluce). Pro testování umožňuje aplikace vygenerovat testovací obraz o rozměrech zadaných pomocí parametrů.

Každý algoritmus musí mít následující funkce implementované. Jedná se o jednotné rozhraní, kompatibilní s jazykem C:

- *LibraryEnum(AlgorithmSettings *settings)* – Nastaví vlastnosti algoritmu.
- *SetParameterString(const char *name, const char *value, Exception *ex)* – Nastaví parametr algoritmu.
- *SetImages(Image *input, Image *output)* – Použije tyto vstupní a výstupní struktury obrazu.
- *StartAlgorithm(Exception *ex)* – Alokace výstupního obrazu, start vlastního algoritmu zpracování obrazu a uložení výsledku.

- *CleanupAlgorithm()* - Uvolnění prostředků, které algoritmus využívá. Je zodpovědný za uvolnění výstupního obrazu.

5.2 Implementace konvolučního filtru

Konvoluční filtr je implementován ve dvou verzích – první verze poběží pouze na CPU, druhá verze bude využívat rozhraní CUDA, poběží tedy na GPU.

Konvuluční filtr použije uživatelsky zadanou matici, uloženou v souboru. Tento soubor se předává pomocí parametru *-p convMatrixFile file.conv*. Aplikace parametr předá algoritmu pomocí funkce *SetParameterString("convMatrixFile", "file.conv", ex).*

5.2.1 Konvoluční filtr běžící na CPU

Implementace konvolučního filtru pro CPU je provedena do jisté míry podle článku [7]. Rozdílů je více. Nejdůležitější však je, že lze použít libovolně rozměrné matice konvoluce (podmínkou jsou liché rozměry matice, se kterými se lépe manipuluje).

Funkce *StartAlgorithm()* ověří parametry, alokuje výstupní obraz a zavolá funkci *Filter()*, která je vlastní implementací filtru.

Algoritmus se snadněji odladí na CPU než na GPU. Po odladění se stane snadným základem konvolučního filtru pro GPU.

5.2.2 Konvoluční filtr běžící na GPU

Jelikož CUDA aplikace využívají jazyka C a implementace konvolučního filtru pro CPU je napsána v jazyce C, využiji tuto implementaci i pro GPU s drobnými úpravami.

Důležité je rozložení běhu filtru do více vláken. Nejvhodnější přístup je model, kde vlákno zpracovává jeden nebo více řádků.

Využiji funkci *Filter()* z implementace pro CPU a úpravou převedu na funkci běžící na GPU. Úprava spočívá v rozložení úlohy do více vláken. Každé vlákno zpracuje určitý počet řádků z celého obrazu. Proměnná *threadIdx* nařizuje vláknu, kterou část má zpracovat.

Druhá varianta konvolučního filtru pro GPU rozloží práci tolika vláknům, kolik je kanálů barev a počtu prvků matice konvoluce. Jde tedy o „třírozměrné" volání *__global__* funkce – šířka matice, výška matice a počet barevných kanálů. Po zvážení častého a elementárního přístupu této metody do paměti, je tato metoda pomalejší kvůli většímu počtu konfliktních přístupů do globální paměti.

5.2.3 Odlaďování a optimalizace implementace konvoluce na GPU

Výsledek prvního testování konvoluce na GPU mne zarazilo. Konvoluce na GPU je téměř pětkrát pomalejší než konvoluce na CPU. Z mých experimentů s kódem jsem zjistil, že ohromný pokles výkonu způsobuje následující kód, kód samotné konvoluce (ostatní části, jako čtení a zápis byly dostatečně rychlé):

```
uint4 pixel;
for (my = 0; my < convMatrix->height; my++) {
    for (mx = 0; mx < cmw; mx++) {
        const int value = convMatrix->data[mx + my * convMatrix->width];
        const uchar4 inBank = in[mx];

        pixel.x += inBank.x * value;
        pixel.y += inBank.y * value;
        pixel.z += inBank.z * value;
    }
}
pixel.x = pixel.x / convMatrix->factor – convMatrix->offset;
pixel.y = pixel.y / convMatrix->factor – convMatrix->offset;
pixel.z = pixel.z / convMatrix->factor – convMatrix->offset;
```

Proměnná *inBank* obsahuje zpracovávanou část vstupního obrazu. Matice má při testu rozměry 3x3. I při nahrazení rozměrů konstantami se neobjevil patřičný zrychlující efekt. Při odstranění tohoto kódu se dostaví trojnásobné zrychlení oproti konvoluci na CPU, tudíž zde musí být nějaký problém. S dalšími experimenty s kódem vyšlo najevo, že aritmetické operace v daném kódu velice zpomalují.

Po přečtení podkapitoly 5.1.1.1 knihy [1] pojednávající o zvyšování rychlosti kódu je pravděpodobnou příčinou poklesu výkonu využití celočíselných aritmetických operací. Dle prvních dvou bodů v odstavci zabere násobení *32-bitových integerů* 16 strojových cyklů, kdežto sčítání a násobení *float* čísel jen 4 strojové cykly!

Změna vybranných celočíselných proměnných na typ *float* čas běhu zkrátila. Konvoluce na GPU byla již jen třikrát pomalejší než na CPU. Stále to však nebyl uspokojivý výsledek.

Další možnost jak zrychlit je změna vstupního a výstupního pole z typu *unsigned char* na zabudovaný typ *uchar4*. Tato změna minimalizuje přístup do paměti s nezarovnaným ukazatelem a zjednodušuje kód tak, že nedochází k tak četnému konfliktu při přístupu do paměti.

Dělení výsledné sumy faktorem se objevuje třikrát. Dělení je časově náročnou operací. Lze využít zjednodušenou verzi *__fdividef(x,y)*, která nabízí rozumný výsledek za kratší dobu. Po otestování rychlosti se toto zjednodušené dělení na rychlosti konvoluce nijak zvlášť neprojevilo.

Rozepsání cyklu na posloupnost příkazů zrychlilo konvoluci na GPU a to na dvojnásobnou dobu zpracování konvoluce na CPU. Rozepsání cyklu je nežádoucí z hlediska využití libovolně rozměrných matic. Pomalý běh má jistě jinou příčinu.

Přitom chyba byla zcela banální. Při studii využití CUDA rozhraní jsem si prohlížel víceméně jednodušší zdrojové kódy, které využívaly jednodušší variantu volání funkce deklarované pomocí klíčového slova __*global*__.

Při volání této funkce se předávají hlavní dva parametry, které specifikují, jak se funkce rozloží na vlákna. První, *gridDim*, určuje počet bloků (multiprocesorů). Druhý parametr určuje počet jader (stream procesorů) přidělených danému bloku, která jsou mezi sebou synchronizována. Proměnná *gridDim* byla nastavena na hodnotu 1, což byla ona fatální chyba. Vlákna běžela jen v jediném bloku a byla synchronizována. Synchronizace vláken snižovala výkon.

Po experimentování s nastavením proměnných *gridDim* a *threadDim* dosahovala konvoluce na GPU nejlepších výsledků při počtu 32 bloků a 32 vláken na jeden blok. Doba potřebná pro výpočet konvoluce na GPU předstihla konvoluci na CPU přibližně o jednu třetinu.

5.2.4 Konvoluční filtr běžící na GPU (druhá varianta)

Druhý experimentální kód konvoluce na GPU využívá mnohem většího paralelizmu. Pro získání jasu výsledného pixelu spouští tři vlákna (tři barevné kanály). Obrázek je dvourozměrně rozložen do několika bloků. Zpracování každé barvy pixelu zvlášť je časově náročné, proto jsem kód upravil tak, aby zpracovával všechny barevné kanály pixelu v jednom vlákně. Při testech se tato verze konvoluce prokázala jako mírně rychlejší než výše popsaná konvoluce na GPU.

Tento kód ale neprovádí konvoluci celého obrazu. To může být příčinou kratšího běhu oproti první variantě. Rozměr zpracovávané plochy je zarovnán podle nastaveného počtu bloků (proměnná gridDim). Zpracování jen určité části obrazu může být v některých aplikacích výhodné.

6 Výsledky testů

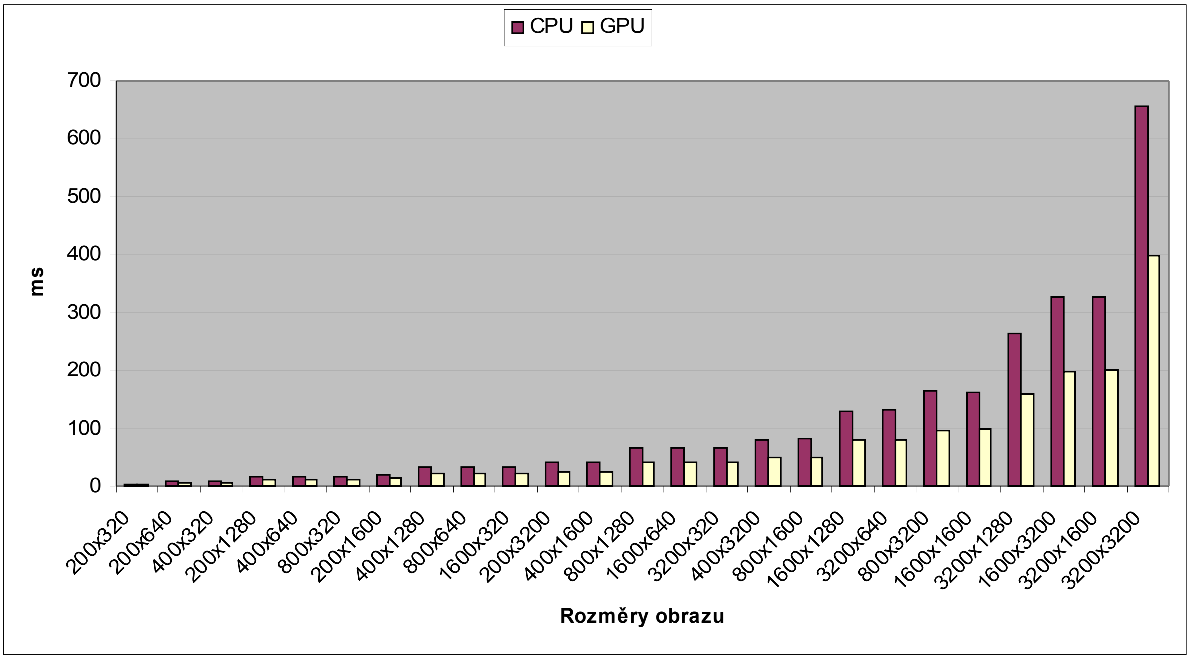
Test zjišťuje poměr doby běhu konvoluce pracující na CPU a na GPU. Pro získání výsledků použiji první verzi konvoluce na GPU, protože druhá verze nezpracovává stejný počet pixelů jako konvoluce pro CPU. Test se provádí několikrát na různě velkých obrazech. Aby byl výsledek přesnější, každý test se opakuje dvacetkrát. Výsledné hodnoty se zprůměrují.

Test je proveden na počítači s procesorem Intel(R) Core(TM)2 Quad CPU 2.40GHz a grafickou kartou NVIDIA GeForce 8800 GT, která má 14 multiprocesorů taktovaných na 600MHz a 112 stream procesorů taktovaných na 1500 MHz. Domnívám se, že tak výkonný centrální procesor výsledky zkreslil.

Ještě je nutno podotknout, že do času naměřeného u konvoluce pro GPU jsou započteny i operace alokace a uvolnění globální paměti na grafické kartě spolu s kopírováním dat v obou směrech (z paměti grafické karty do operační paměti a zpět).

6.1 Graf dle rozměrů obrazu

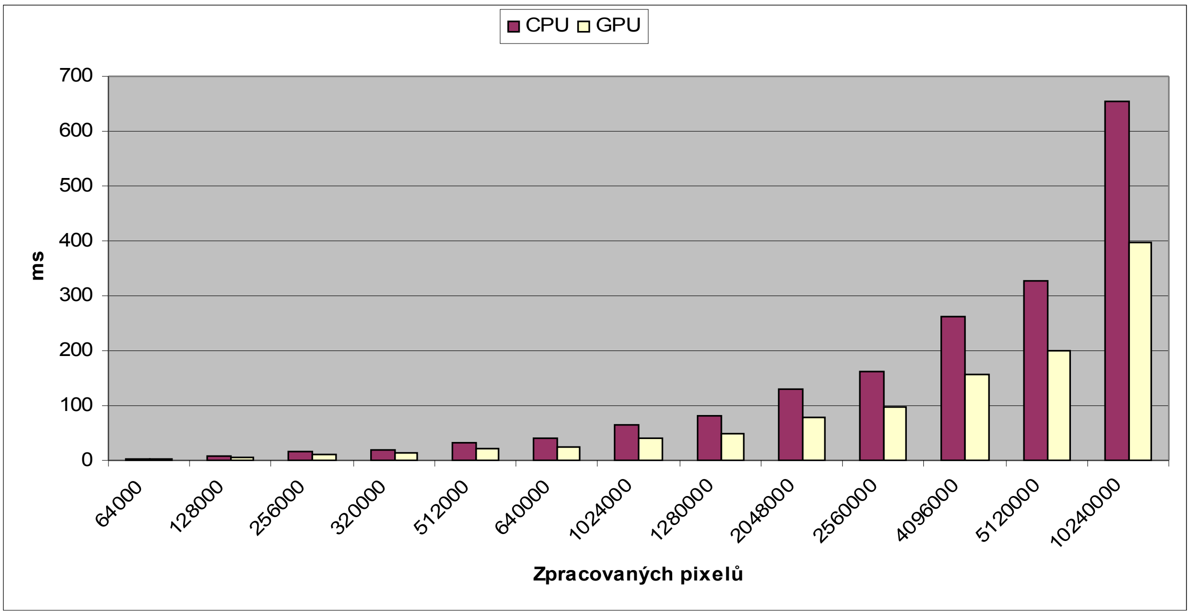
Výsledky jsem pro větší přehled převedl do grafu. Jako podklad jsem použil rozlišení zpracovávaných obrazů. Můžeme vidět, že nárůst výkonu je patrný, i když ne tak výrazně, jak jsem na začátku psaní této práce předpokládal.



Ilustrace 12: Graf výsledku testu dle rozlišení obrazu

6.2 Graf dle počtu zpracovaných pixelů

Níže položený graf je upraven pro porovnání s jinými verzemi konvoluce, především konkurenčními.



Ilustrace 13: Graf výsledku testu dle zpracovaných pixelů

Moji verzi konvoluce pro GPU srovnám s jinou implementací pro GPU, kterou jsem nalezl na diskusním fóru firmy NVIDIA. Konkurenční konvoluce zpracovává jen jeden kanál typu *int*. Dle počtu zpracovaných pixelů je moje verze konvoluce rychlejší, i přes to, že zpracovává 3 barevné kanály. Konkurenční konvoluce se dále vyvíjela a využila místo globální paměti paměť textur. Tato změna přinesla mnohonásobné zrychlení.

6.3 Další možný vývoj

Z hlediska zrychlení je možným vývojem využití paměti textur. To neplatí jen pro tuto bakalářskou práci, ale i pro všechny algoritmy, které zpracovávají obraz.

Tento přístup lze aplikovat na mnohem více druhů zpracování obrazu než jen na konvoluci. Příkladem může být i natrénovaný rozpoznávač obrazů AdaBoost, který by v případě využití texturové paměti mohl běžet úplně realtime. Existují i další možnosti využití.

7 Závěr

Když jsem si vybíral toto téma bakalářské práce, byl jsem informován, že je svým způsobem výzkumná a má za úkol zmapovat a zkoumat možnosti grafických procesorů, o kterých celý čas hovořím. Vybral jsem ji i přes to, že mi bylo zdůrazněno, že mi pravděpodobně nikdo detailně neporadí ohledně návrhu a programování aplikací pro tuto technologii. Věřím, že mé poznatky alespoň trochu pomohou všem, kdo mají zájem o zrychlení svých časově náročných algoritmů. Rozhodně doporučuji pro FFT využít knihovnu CUFFT a pro operace s maticemi knihovnu CUBLAS.

Zjistil jsem, v čem spočívá síla této technologie. Dá se jistě využít i ve více oblastech než je zpracování obrazu. I když při využívání texturové paměti to tak nevypadá. Na domovské stránce této technologie je uveřejněn seznam aktivních projektů z různých oblastí, především simulací a výzkumu. Možnosti této technologie jsou velice rozsáhlé.

Cílem této práce bylo analyzovat současné možnosti využití programovatelných grafických procesorů, vybrat vhodný algoritmus a využít grafický procesor pro akceleraci zpracování. Bylo nutné překonat několik překážek, abych dosáhl vytyčeného cíle. Rychlost implementovaného algoritmu ale rozhodně nedosáhla tak vysokého nárůstu, jak se uvádí u některých aplikací využívající tuto technologii. Jistě je potřeba si obohatit vědomosti a zkušenosti o této technologii, které jsou nezbytné pro návrh a implementaci.

V rámci diplomové práce navrhuji využít znalosti pro akceleraci rozsáhlejšího problému.

Literatura

[1] NVIDIA, Santa Clara, CA 95050, USA. *CUDA Programming Guide 1.1*, 29.11.2007. Dokument dostupný na URL: http://developer.download.nvidia.com/compute/cuda/1_1/NVIDIA_CUDA_Programming_Guide_1.1.pdf (duben 2008).

[2] NVIDIA, Santa Clara, CA 95050, USA. *CUDA CUBLAS Library 1.1*, Září 2007. Dokument dostupný na URL: http://developer.download.nvidia.com/compute/cuda/1_1/CUBLAS_Library_1.1.pdf (duben 2008).

[3] NVIDIA, Santa Clara, CA 95050, USA. *CUDA CUFFT Library 1.1*, Říjen 2007. Dokument dostupný na URL: http://developer.download.nvidia.com/compute/cuda/1_1/CUFFT_Library_1.1.pdf (duben 2008).

[4] NVIDIA, Santa Clara, CA 95050, USA. *CUDA Compiler Driver NVCC*, 5.11.2007. Dokument dostupný na URL: http://www.nvidia.com/object/io_1195170069217.html (duben 2008).

[5] Halfhill Tom: *Paralell processing with CUDA*, 28.1.2008. Dokument dostupný na URL http://www.nvidia.com/docs/IO/47906/220401_Reprint.pdf (duben 2008).

[6] Kabát, Zdeněk: *3D technologie: Anti-aliasing*, 12.11.2003. Dokument dostupný na URL: http://www.svethardware.cz/art_doc-694B0E75F09B32BBC1256DD70074B6BA.html (duben 2008).

[7] Christian Graus: *Image processing for dummies with C# and GDI+*, 23.3.2002. Dokument dostupný na URL http://www.codeproject.com/KB/GDI-plus/csharpfilters.aspx (duben 2008).

[8] Černocký, Jan: *Zpracování obrazů*. Dokument dostupný na URL http://www.fit.vutbr.cz/~cernocky/sig/pred/2d/2d.pdf (duben 2008).

Seznam příloh

Příloha 1. Uživatelský manuál testovacího programu, využití texturové paměti.

Příloha 2. CD se zdrojovými kódy, obrazem plakátu a technickou zprávou.

Příloha

1 Uživatelský manuál testovacího programu

Program *Cudabak* je aplikace pro operační systémy MS Windows a Linux. Obsahuje jednotné rozhraní pro algoritmy, které jsou implementované nebo které se mohou dynamicky načíst. Implementovanými algoritmy jsou konvoluce pro CPU *(convolutionCPU)*, konvoluce pro GPU *(convolutionGPU)* a experimentální konvoluce pro GPU *(convolutionGPU2)*. Tyto algoritmy jsou popsány v kapitole Implementace.

Aby mohl být program správně spuštěn, vyžaduje ke své činnosti alespoň jeden grafický adaptér s podporou CUDA technologie, správně nainstalované CUDA ovladače grafického adaptéru, knihovnu *cutils* dostupnou v SDK a knihovnu *FreeImage*.

1.1.1 Testovací data

Program pomocí jednoho příkazu dokáže vygenerovat testovací obraz o zadaných rozměrech. Pro tuto funkci se využívá spuštění aplikace s parametrem *-g*. Následují parametry jméno výstupního souboru, šířka obrazu, výška obrazu a barevná hloubka obrazu v bitech.

1.1.2 Seznam grafických adaptérů a algoritmů

Pomocí parametru *-l* program vyhledá dynamicky spustitelné algoritmy, které se nachází v dynamických knihovnách, a vypíše je společně se zabudovanými algoritmy. Seznam grafických adaptérů lze vyvolat spuštěním s parametrem *-d*.

1.1.3 Provedení testu

K provedení testu musí být specifikovány následující parametry: název algoritmu, vstupní obraz, výstupní obraz. Implementovaný algoritmus si může vyžádat další parametry. Například pro konvoluci je to soubor s maticí konvoluce. Pro demonstraci lze uvést příklad spuštění testu:

```
cudabak -a convolutionCPU -i input_image.png -o output_image.png -p convMatrixFile sharpen.conv
```

V tomto případě parametr *-p* uvozuje parametr specifický pro danou implementaci algoritmu. Například u algoritmu jiného typu to může být hodnota zanoření, vyhlazení...

1.1.4 Statistika

Aby bylo možné provádět testy rychlosti, má program k dispozici měřič času z knihovny *cutils*, který je velice přesný. Tímto měřičem se určuje doba běhu celého algoritmu (doba trvání hlavní funkce algoritmu *StartAlgorithm()*). Ta je zapsána do souboru, který se určuje parametrem *-s*. Pro statistiku je ale důležité provést více měření. Implementace algoritmu se pomocí parametru *-r* spustí vícekrát, tím se provede více měření. Všechny naměřené časy jsou zprůměrovány a průměr je uložen do statistického souboru.

2 Využití texturové paměti

Texturová paměť nabízí rychlejší přístup než paměť globální. Textura je zastřešení jednodimenzionálního nebo dvoudimenzionálního pole hodnot. Hodnoty nejčastěji reprezentují jas barevných složek bodu, ale může se jednat i o třírozměrný vektor.

Přístup k hodnotám tohoto pole se realizuje pomocí funkcí *tex1D* nebo *tex2D*. Parametry této funkce jsou textura a koordinanty. Tyto koordinanty jsou typu *float* a adresují prvky pole. Umožňují přistupovat k hodnotě, která může být „mezi“ dvěma nebo čtyřmi prvky. Výsledná hodnota takového přístupu je interpolována, avšak záleží na nastavení typu filtrování u textury. Tato vlastnost vychází z již dlouhodobě používaného mechanizmu práce s texturami.

Při přístupu k prvku běžného pole využíváme celočíselnou hodnotu, která adresuje prvek v poli. Například poslední prvek takového pole o velikosti 36 má index 35 (při použití nuly). Ale poslední prvek pole u textury může mít index 1, při využití parametru *normalized* (záleží také na nastaveném adresovém módu).

Datový typ textura využívá C++ generické šablonování pro specifické nastavení. V deklaraci textury se uvádí datový typ hodnot, počet dimenzí a příznak přístupu.

```
texture<float, 1, cudaReadModeElementType> tex;
```

Pole hodnot se přiřadí textuře pomocí funkce *cudaBindTextureToArray*. Před zavoláním musí mít textura nastavené adresové módy, typ filtrování a hodnotu příznaku *normalized*. Pole se vytváří a naplňuje pomocí funkcí *cudaMallocArray* a *cudaMemcpyToArray*.

Alokace pole vyžaduje parametr popisující kanál. Popis kanálu vytváří funkce *cudaCreateChannelDesc*, její parametry jsou rozměr pole (x, y, z, w) a formát dat.

Texturu resp. alokované pole můžeme svázat funkcí *cudaGLMapBufferObject* s OpenGL texturou a používat ji při vykreslování obrazu v OpenGL.