Masarykova univerzita
Fakulta informatiky

Paralelní zpracování obrazu ve vysokém rozlišení

Diplomová práce

Bc. Martin Pulec

Brno, 2011

Masarykova univerzita
Fakulta informatiky

ZADÁNÍ DIPLOMOVÉ PRÁCE

Datum: 21. dubna 2011

Student: Bc. Martin Pulec

Program: Aplikovaná informatika

Obor: Aplikovaná informatika

Garant oboru: doc. Ing Jiří Sochor, Csc.

Vedoucí práce: Mgr. Jiří Matela

Název práce: Paralelní zpracování obrazu ve vysokém rozlišení

Zadání: Kolekce patologických snímků Atlases (atlases.muni.cz) je tvořena snímky o rozlišeních v řádu stovek megapixelů. Obrazové úpravy těchto snímků současnými nástroji vips/nip2 představují výpočetně poměrně náročný úkol. Student nastuduje proces zpracování těchto snímků a jednotlivé obrazové úpravy používané při jejich produkci. Pro jednotlivé úpravy bude navržen a implementován paralelní design vhodný pro běh na GPU. Cílem je celkové urychlení zpracování patologických snímků.

Základní literatura:

CUPPIT, John; MARTINEZ, Kirk. Vips: an image processing system for large images [online]. In. SPIE conference on Imaging Science and Technology. SPIE, 1996 [cit. 2011-04-21]. s 19-28. Dostupný z WWW: < http://eprints.ecs.soton.ac.uk/2227/ >.

GONZALES, Rafael C.; WOODS, Richard E. Digital Image processing. 3. vyd. Upper Saddle River, NJ : Pearson Prentice Hall, 2008. ISBN 978-0-13-168728.

Souhlasím se zadáním (podpis, datum)

21.4.2011 — student(ka)

21.4.2011 — vedoucí diplomové práce

garant oboru

* KTP
KPGD
KPSK
KIT

* Hodící se označte!

Prohlášení

Prohlašuji, že tato diplomová práce je mým původním autorským dílem, které jsem vypracoval samostatně. Všechny zdroje, prameny a literaturu, které jsem při vypracování používal nebo z nich čerpal, v práci řádně cituji s uvedením úplného odkazu na příslušný zdroj.

Vedoucí práce: Mgr. Jiří Matela

Poděkování

Rád bych na tomto místě poděkoval Mgr. Jiřímu Matelovi za čas, který věnoval vedení mé práce, za cenné podněty a rady, kterými k jejímu řešení přispěl. Rovněž jsem vděčný za technické připomínky RNDr. Petru Holubovi, Ph.D. Nakonec bych chtěl poděkovat svým rodičům, kteří mě po celou dobu studia podporovali a poskytovali mi zázemí.

Shrnutí

Tato práce si klade za cíl urychlit grafické operace prováděné nad digitálními snímky patologické tkáně v prostředí programu nip2. Tyto úpravy se v současné době počítají na procesoru, což je při počtu a velikosti dlaždic, ze kterých se jednotlivé skeny skládají, časově velmi náročný úkol.

Účelem práce je proto návrh, implementace a nasazení těchto operací na GPU za účelem celkového urychlení zpracování těchto obrázků.

Klíčová slova

Digitální zpracování obrazu, patologické snímky, obrázky ve vysokém rozlišením, dávkové zpracování, CUDA, VIPS, nip2.

Obsah

Seznam obrázků

Seznam tabulek

Seznam zdrojových kódů

Kapitola 1

Úvod

Novodobé informační a komunikační technologie přinášejí řadu způsobů vylepšení práce do různých exaktních i humanitních oborů. Tyto přínosy mají různou kvalitu, což je dané specifickými požadavky těchto oborů, ale nezanedbatelným faktorem je také to, jak dokáží tyto obory, respektive lidé se jim věnující, tyto možnosti absorbovat a využít.

Exaktní obory mají ve využívání IT náskok, což se projevuje možností symbiózy těchto oborů s IT odvětvím a vzájemným přínosem pro všechny zúčastněné.

Právě takovým příkladem je využití počítačů ke zpracování digitálního obrazu, konkrétně pak úpravami snímků patologické tkáně, jejichž akcelerace je náplní této práce.

Oblast digitální fotografie je jedním z nejstarších odvětví, kterými se zabývá odvětví komunikačních technologií. Tato dlouhá tradice trvající od počátku 20. století[1] dovolila etablovat se tomuto oboru jak v teoretické tak praktické rovině se širokou škálou aplikací.

Tyto možné aplikace se zasahují od „amatérských“ úprav fotografií z dovolené Photoshopem až akademické nebo průmyslové uplatnění v nejrůznějších oborech od biologie po detekci defektů integrovaných obvodů. Stejně jako spektrum použití je i rozmanitý výčet nástrojů, které umožňují toto zpracování. Zjevně „jedna velikost nesedí všem“, proto výstupní kontrola kvality elektrotechnické firmy a návrhář vzhledu webové stránky nebudou zřejmě používat tytéž programy.

Zpracování obrázků bylo vždy doménou CPU. Toto tvrzení se nedá dezinterpretovat tak, že by grafické karty vyšly naprázdno. Naopak, v oblasti počítačových her byly grafické operace prováděné dedikovanou kartou významným činitelem, který v konečném důsledku znamenal i značný výkonnostní nárůst těchto karet a postupnou etablaci v obecné výpočetní paralelní prostředí. Problémem grafických karet však bývala jejich nízká univerzálnost, protože i pro špičkovou počítačovou hru stačilo v nedaleké minulosti mít

1. viz kapitola 6

„zadrátováno“ několik málo operací, které procesor dedikoval k provedení grafické kartě. Operace jako vyplnění texturou a obdobné sice mají svoje ekvivalenty i ve Photoshopu, ale nejsou zde natolik klíčové. Kromě toho zmíněné vyplňování oblasti texturou je v konzumních grafických nástrojích o dost primitivnější než ve 3D, protože zpravidla není nutná projekce na nakloněnou rovinu a většinou ani interpolace textury. Proto jsou toto věci, které procesor zvládá při běžných parametrech v rozumném čase. Většinu neherních úprav bychom hledali ve firmware grafické karty marně.

Kromě toho už jsem se dotkl faktu, že procesor zvládá většinu úprav relativně rychle. To je nutné si uvědomit, protože zde nevzniká dostatečný tržní tlak na urychlení výpočtů. Určitě ne takový, který vytváří okruh počítačových hráčů. Tím pádem byli programátoři grafických aplikací donedávna nuceni provádět veškeré výpočty na procesorech.

Neexistuje žádná zřejmá dělící čára mezi tím, které úpravy jsou určeny amatérům a které jsou zaměřené jenom na omezený okruh uživatelů. Jisté rozdíly lze pozorovat až s přihlédnutím k tomu, jak se ta která operace provádí – ať již jde o rozdíl interaktivní versus dávkové zpracování nebo o atributy vlastních úprav, které se mohou lišit třeba použitými barevnými prostory[2].

V popředí zájmu této práce je zpracovávání velkých skenů patologických snímků pořizovaných Ústavem patologie Lékařské fakulty MU sloužící jako učební a diagnostická pomůcka při výuce na zmíněné fakultě. Kolekce těchto obrázků vytváří atlasy těchto patologických snímků. Nejstarší z nich je atlas dermatopatologie. Jednotlivé obrázky mají rozlišení zhruba 1 Gpix, proto je jejich zpracování poměrně specifické jak z hlediska paměťových nároků, protože celý sken se nevejde do paměti RAM, tak z hlediska výpočetní náročnosti při těchto objemech. Při takto velikých obrázcích už výpočetní čas poměrně neúměrně narůstá na to, aby se na procesorech prováděl v rozumném čase.

Obrázky jsou při snímání rozřezány do jednotlivých dlaždic o velikosti zhruba 5 Mpix. Každá z těchto dlaždic je posléze upravovaná grafickými úpravami, jimiž se zabývám v této práci. Tyto úpravy jsou pouze malou částí v celkovém procesu zpracování od pořízení snímků až po publikaci ve webovém atlasu. Další kroky zahrnují například spojování rozřezaných obrázků, komprese obrazu a eventuálně opět rozřezání na menší dlaždice. To se provádí za účelem snížení počtu IO operací, které jsou potřeba na získání malého obdélníkového výřezu celého obrázku tak, jak ho zobrazuje

2. viz sekce 6.2

virtuální mikroskop[3].

Referenčním nástrojem pro tuto práci bude knihovna VIPS a její grafický frontend nip2. Tyto nástroje jsou v současnosti využívány k provádění grafických operací, které nahrazujeme GPU zpracováním. Z toho důvodu je pro účely této práce důležitý přesný postup, kterým tyto nástroje dané úpravy provádějí.

Ve 2. kapitole se zabývám kontextem této práce, jednotlivými atlasy snímků, procesem jejich pořizování a dalšími souvisejícími kroky při jejich zpracování.

Kapitola 3 podává základní informace o výpočetní architektuře CUDA, kterou jsem zvolil pro zpracování, jaké jsou její možnosti, výhody a omezení, a také pojednává o tom, proč právě tato architektura byla vyhodnocená jako nejvýhodnější pro realizaci cílů této práce. Zejména se věnuji konkrétním specifikům platformy CUDA, která jsou významná pro tuto práci.

Následující, 4. kapitola, uvádí knihovnu VIPS a program nip2, které se používají doteď pro zpracování snímků, jejich vlastnosti a důsledky plynoucí z jejich používání ve vztahu k mé práci.

Kapitola 5 je rešerší prací, které se vztahují k obsahu této práce. Dále se zevrubně věnuji grafickým nástrojům, které bych mohl potenciálně využít při implementaci projektu.

Teoreticky zaměřená kapitola 6 podává vysvětlení základních principů obrazových úprav, které se provádí.

Dvě následující kapitoly, 7. a 8, jsou už konkrétním uplatněním této teorie. První z nich má spíše přehledový charakter a vysvětluje jednotlivé grafické úpravy, kterým se věnuji. Druhá se již věnuje vlastnímu nasazení na cílovou architekturu.

V 9. kapitole vyhodnocuji výsledky, kterých se mi podařilo dosáhnout, jaké přesnosti a zrychlení finální implementace dosahuje, její možné nasazení a vhodnost výsledné implementace ve vztahu k vytyčeným cílům. Nejdůležitější poznatky z měření a hodnocení shrnuji v 10. kapitole. V podkapitole 10.1 rekapituluji výsledky ve vztahu k možným navazující pracím ať již by se jednalo o rozšíření finálního produktu nebo o práce vztahující se k problematice grafického zpracování patologických snímků.

Příloha D podává některá doporučení ohledně případného přidávání dalších úprav, aby co nejlépe zapadly do kontextu výsledné aplikace.

3. 2. kapitola

1.1 Cíle

Cíle této práce jsou specifikovány jednoduše – celkové urychlení grafických úprav jakožto části celého procesu. Toto urychlení by mělo být ideálně komplexní, nejlépe kompletní urychlení od načtení až po zápis, nejen vlastní GPU části.

Druhý hlavní požadavek je logicky na co nejlepší kompatibilitu výstupů s nástroji nip2/VIPS, protože v nich se budou tyto úpravy zadávat a očekává se srovnatelný výsledek.

Kapitola 2

Atlas patologických snímků

Atlas, respektive atlasy, patologických snímků vytvářené na Masarykově univerzitě slouží jako studijní a diagnostická pomůcka pro studenty a specialisty Lékařské fakulty MU primárně, případně použitelná pro jiné zainteresované subjekty univerzálně. Dnes již soubor hypertextových atlasů, přístupných z `http://atlases.muni.cz/` (je potřeba registrace a souhlas se zřeknutím se zodpovědnosti provozovatele), zahrnuje atlasy dermatopatologie, fetální a novorozenecké patologie, patologie kostní dřeně a lymfomů.

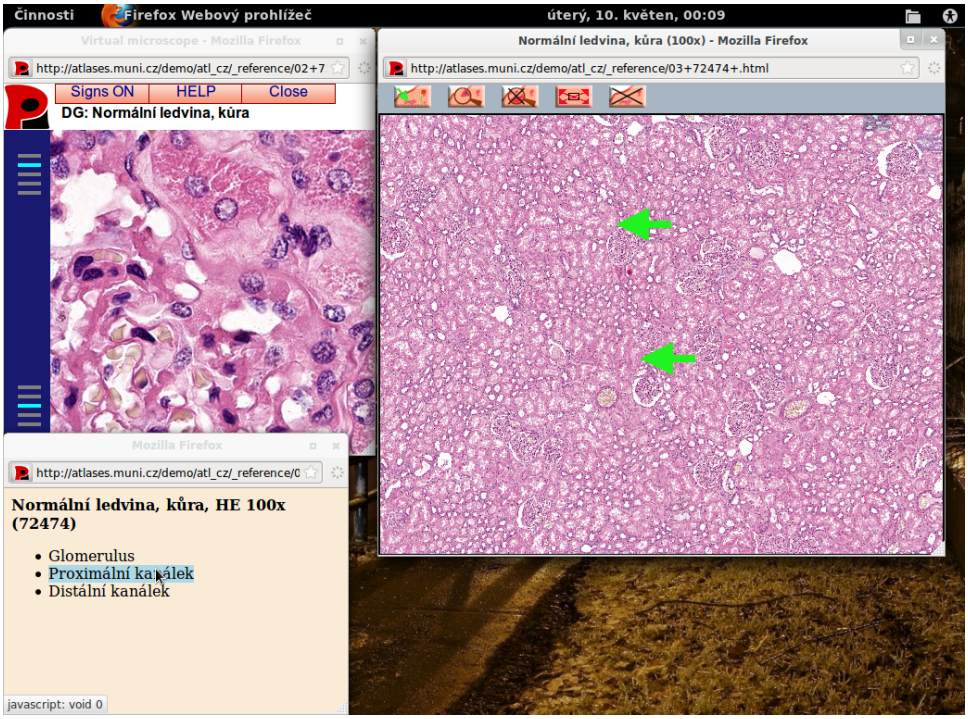
Výsledné obrázky jsou přístupné prostřednictvím webového prohlížeče a umožňují poměrně sofistikované prohlížení obrázků tzv. virtuálním mikroskopem, jehož rozhraní ukazuje obrázek 2.1. Celý proces produkce obrázků od jejich pořízení se skládá z několika kroků, které nyní ve stručnosti nastíním. Věnuji se pouze IT zpracování, a tedy nechávám přípravu preparátů a všechny procesy s ní související stranou.

2.1 Pořizování skenů

Pro pořizování se používá sada mikroskopů. Tyto mikroskopy pořizují snímky v různých rozlišeních, nejčastěji 2–5 Mpixelů na dlaždici, přičemž tyto dlaždice dávají dohromady celý obrázek. Výstup je je ukládán v běžném grafickém formátu (TIFF, JPEG). Snímající kamera je poháněna motorkem, který se pohybuje po snímaném objektu a pořizuje jednotlivé skeny.[1]

Jednotlivé obrázky mají 3x8, respektive 3x12, bitů na obrazový bod, přičemž v druhém případě se k označení takových souborů používá standardní atribut TIFFTAG_MAXSAMPLEVALUE. To bohužel způsobuje nemalé potíže, neboť prakticky žádný rozšířený program (pochopitelně s výjimkou utilit z libtiff) tento atribut nečte a nepracuje s ním. Všechny takové nástroje pak vnímají výsledný obrázek jako 16-bitový, a tedy skutečných 12 bitů vnímají jako nejnižších 12 bitů ze šestnácti. V důsledku toho se všechny úrovně

1. Více informací o snímacích zařízeních je možné najít na adrese `http://atlases.muni.cz/atlases/kuze/atl_cz/main+uvod+sect_howto.html#sect_tech+sect_hardware`.



Obrázek 2.1: Webové rozhraní atlasů

obrázku zobrazí do dolní $\frac{1}{16}$ výsledné intenzity. Takto zobrazený obrázek se pak jeví jako skoro černý. Do nelichotivé množiny těchto nástrojů náleží i referenční knihovna VIPS. Tento nedostatek se pak musí řešit ne úplně čistým způsobem, který popisuji v sekci 7.7.1.

2.2 Obrazové úpravy

Obrazové úpravy slouží k „vylepšení“ vzhledu obrázků případně k vyvážení nedostatků zanesených do jednotlivých snímků při jejich pořizování (třeba posunutá gama obrázku oproti hodnotě zobrazovacího zařízení).

Urychlení těchto úprav je náplní této práce a proto je podrobně popisuji v 7 kapitole.

2.3 Spojování dlaždic

Protože je potřeba z jednotlivých malých dlaždic vytvořit celistvý, bezešvý obrázek, je následujícím krokem spojení jednotlivých dlaždic. V naivně spojeném obrázku by se jistě vyskytovaly artefakty u okrajů způsobené dvěma příčinami – za prvé již při pořizování snímků na sebe nemusí jednotlivé dlaždice navazovat na úrovni jednotlivých pixelů. Druhý zdroj rušivých artefaktů jsou pak obrazové úpravy, které (například unsharp mask) mohou zanést další inkonzistence.

Proto je jako další krok implementovaný algoritmus, který, kromě jednoduchého spojení, tyto artefakty odstraňuje. Výsledný obrázek, který obsahuje celý sken, je pak spojený do jediného JBIG souboru.

2.4 Finální úpravy pro vhodné zobrazení webovým prohlížečem

Posledním krokem v produkci atlasů je konečná úprava pro zobrazení webovým prohlížečem. Vzhledem k tomu, že skeny už jsou na potřebné kvalitativní úrovni, tak je potřeba zejména vyřešit efektivní přístup k obrázkům na serverové straně z hlediska diskových přístupů.

Z hlediska IO a charakteru prohlížení, které probíhá prostřednictvím virtuálního mikroskopu, ve kterém se zobrazuje malý výřez celkového obrazu v jeden okamžik, není jeden velký soubor ideální. Důvod je ten, že zobrazujeme výřez řádově o stovkách pixelů v obou rozměrech. Každý řádek z tohoto výřezu je potřeba číst zvlášť, což ve výsledku znamená v několik stovek IO operací na jediný pohled. Klasické rotační disky (pole složená z rotačních disků nevyjímaje) jsou v případě náhodných přístupů velmi pomalé (10 ms

na operaci a více), což je těžko akceptovatelné, protože by to znamenalo ve výsledku sekundy na vygenerování jediného pohledu.

Jako řešení tohoto problému jsou obrázky znovu „rozsekány“ na malé díly a uloženy do TARu[2]. Pokud jsou jednotlivé nově vzniklé dlaždice stejně velké nebo větší než výřez, kterým prohlížíme, tak to znamená, že je nutné načíst nanejvýš 4 dlaždice na jedno zobrazení. Pokud navíc ukládáme dlaždice po řádcích (myšleno řádek matice dlaždic), tak je možné přečíst oba dva sousední dlaždice ve vertikálním rozměru v jediné IO operaci (využíváme toho, že TAR je jediný soubor). Jako výsledek těchto úprav je potřeba provést pouhé dvě IO operace, což už je z hlediska disků akceptovatelné. Navíc, čtené bloky jsou relativně malé a data, která se načtou navíc se, při relativně vysoké rychlosti sekvenčního čtení, na výsledném čase výrazněji neprojeví.

2. Tar je archivační formát umožňující uložení více souborů do jediného tak, že je jednoduše zřetězí obsah obsažených souborů za sebe.

Kapitola 3

CUDA

Mnohajadernou obecnou výpočetní architekturu CUDA™ vyvinula NVidia jako evoluci dedikovaných grafických operací prováděných na GPU.

Milníkem započínajícím cestu k obecné výpočetní architektuře CUDA bylo zařazení možnosti programování shaderů prostřednictvím vysokoúrovňového jazyka (GLSL, HLSL). Takto vytvořené podprogramy jsou stále úzce svázané s grafikou – jsou volány nadřazeným grafickým API (OpenGL, Direct3D), původní (neunifikované) shadery mají úzce ohraničenou oblast působnosti – pro počítání nad jednotlivými vrcholy a pixely. Každopádně i tak dává shaderový jazyk určité možnosti využití ve specifických oblastech i mimo grafiku.

Zobecnění výpočetních shaderů dalo vzniknout obecnějším výpočetním architekturám. Zatímco ATI/AMD se ubíralo slepou uličkou nízkoúrovňového přístupu k HW (Close-to-Metal, později CAL), NVidia vsadila na použití použití podmnožiny jazyka C99 a částečně i C++. To se ukázalo jako šťastnější volba a i díky tomu NVidia s CUDA dodnes na poli GPGPU víceméně dominuje.

Ačkoliv jazyk, který používá CUDA, vychází z jazyků C a C++, architektura jako taková je uzpůsobena vlastnostem karet od NVidie. To se stalo bariérou tomu, aby ostatní výrobci převzali API této platformy tak, jak je. Jako překlenutí vakua a pravděpodobného chaosu, který by mohl vzniknout, pokud by každý HW výrobce měl svůj jazyk, vznikl pod záštitou Khronos Group otevřený standard pro framework OpenCL. Programovacím jazykem frameworku OpenCL je opět podmnožina jazyka C99, přičemž celá platforma je obecnější a předpokládá nižší společný jmenovatel podporovaných vlastností, než CUDA. Cílem je totiž podpora širšího spektra zařízení, nejen grafických karet. Všechny specifické schopnosti grafických karet jsou brány jako rozšíření.

Nyní je vhodné místo napsat, proč jsem se rozhodl právě pro CUDA a ne například pro OpenCL nebo shadery. V porovnání se shaderovými jazyky je CUDA lepší v tom, že tyto jazyky jsou poměrně málo obecné – úpravy by s nimi zřejmě bylo možné realizovat, nicméně v porovnání s CUDA ne-

mají tyto jazyky vyšší přidanou hodnotu jako sofistikovaná kontrola vláken za pomocí běhové podpory a velmi pravděpodobně i nižší výkon. Jedinou výhodou použití shaderů oproti platformě CUDA je to, že jsou všeobecně použitelné (GLSL na kartách podporujících OpenGL alespoň 2.0).

Mimo shadery je potřeba jako alternativu uvážit také OpenCL. To má tu výhodu, že se jedná o otevřený standard, který je používaný i pro jiné karty než od NVidia. Rozhodnutí, proč použít CUDA namísto OpenCL je do jisté míry pragmatické. První důvod je, že v době psaní této práce není implementace žádného výrobce na té úrovni jako CUDA – NVidia stále implementuje pouze OpenCL verzi 1.0, která například není vláknově bezpečná. Navíc nemá pro OpenCL implementovanou veškerou funkcionalitu v tom rozsahu jako CUDA. AMD je z hlediska OpenCL o trochu dále, ale jejich hardware poskytuje zhruba jen podmnožinu vlastností oproti jejich protějškům od NVidie. (Ačkoliv zde v posledních měsících odvádí AMD dobrou práci a počáteční ztrátu z velké části umazalo, stále je ještě na tom ve srovnání s NVidií hůře. Alespoň z celkového pohledu, protože tyto dvě architektury se v mnoha ohledech liší.) Druhý důvod, proč preferovat platformu CUDA, je ten, že CUDA je poměrně dobře zažitá a přijímána, což se odráží také v tom, že výpočetní uzly, na kterých by mohl být výsledný program nasazen, disponují vesměs kartami tohoto výrobce, karty od AMD jsou už méně časté. Podle určitých měření se dosahuje na týchž kartách od NVidie výkonu pod OpenCL v průměru o 30% nižší než CUDA [9].

Z těchto důvodů se jeví implementace v CUDA v době psaní této práce výhodnější oproti OpenCL, ačkoliv případný převod do OpenCL by neměl být složitý, protože jazyky těchto API jsou do jisté míry obdobné.

3.1 Architektura

Na architekturu CUDA je možné pohlížet několika pohledy. Pro vývojáře je nejdůležitější API a výpočetní model. Kromě toho hraje stejně jako ve většině ostatních HPC aplikacích důležitou roli i hardwarový pohled. Tyto aspekty rozvádím v následujících oddílech. Dále bych se ještě rád krátce vyjádřil k software.

3.1.1 Programovací model

Grafické karty NVidia nabízejí mnohajaderné výpočetní prostředí a výpočetní model. Nejmenší prováděcí jednotkou karty je thread (vlákno), proto se z hlediska programátora jedná o SIMT. Architektonicky jde spíše o hybridní model SIMT/SIMD, eventuálně je u novějších karet možné mluvit i o

MIMD (uvažujeme-li paralelní kernely). Vlákna jsou organizována do warpů fixní velikosti (typicky 16 nebo 32), přičemž thready v jednom warpu jsou prováděny vždy současně. Při znalosti tohoto se často ani neprovádí synchronizace mezi těmito thready, protože je zřejmé, že provádí tutéž instrukci (implicitní synchronizace).

Z uživatelského hlediska se thready sdružují do 1–3 dimenzionálních bloků. Takovýto blok je poté plánován současně, ovšem je nutné se synchronizovat, pakliže velikost bloku přesahuje velikost warpu. Jednotlivé bloky jsou logicky organizovány v gridu, který může mít opět 1–2 dimenze. Mezi různými bloky není žádná vazba a ani záruka pořadí provádění.

Identita threadu v bloku a bloku v gridu není pouze orientační, ale je velmi důležitá pro identifikaci dílu dat, který bude ten který thread zpracovávat. S tím souvisí primární zaměření architektury CUDA na datově paralelní zpracování. Při tomto způsobu se data zpracovávají tak, že se velká množina dat rozdělí na podmnožiny, které se počítají na jednotlivých výpočetních uzlech. Dělit se dá i rekurzivně na různých úrovních (například redukce). Hardwarově nabízí CUDA dvě logické úrovně dekompozice – na bloky a jednotlivé thready. Zpracovávané množiny by měly být ideálně výpočetně disjunktní anebo alespoň s minimem potřebné komunikace k vyřešení vazeb mezi jednotlivými díly dat. Pro architekturu CUDA platí, že je potřeba minimalizovat především vazby mezi jednotlivými bloky. Architektura nedává k dispozici žádná primitiva pro intrablokovou synchronizaci. Oproti tomu uvnitř bloků je možné se synchronizovat velmi levně.

Vedle datového paralelismu je používaný model úlohového paralelismu. Tento výpočetní model je vhodný zejména pro víceprocesorové systémy/clustery, neboť jednotlivé výpočetní uzly jsou z velké části nezávislé. Pro grafické karty tento model není výhodný ze dvou důvodů – jednak mají grafické karty omezený počet jednotek pro dekódování instrukcí (typicky jedna nebo dvě [13, s. 10] na multiprocesor) a s tím souvisí i druhý bod, že CUDA API explicitně nezpřístupňuje možnost provádění více instrukčních toků. Pomineme-li možnost spouštění souběžných kernelů, která je určena spíše k optimálnímu vytížení GPU, tak tento programový model tíhne k řízení provádění výpočtů ve vlastní režii než ke skutečnému multitaskingu.

Pojem kernel označuje výpočetní jednotku, funkci, jež se provádí na GPU, ale je volána pomocí API z vlákna hostitelského systému. Tato funkce musí mít předem připravené výpočetní prostředí, hlavně data (jako parametr lze předávat typicky pouze několik skalárních hodnot) a určené výpočetní zdroje.

CUDA nabízí ve skutečnosti dvě různá rozhraní – Driver API a Runtime API (C for CUDA). Odlišnosti jsou hlavně v syntaxi obslužných rutin

volaných z hostitelského systému, ale GPU kód je u obou stejný. První jmenované rozhraní nevyžaduje žádná rozšíření jazyka pro hostitelský kód, takže je možné jej kompletně přeložit jakýmkoliv překladačem a následně slinkovat s knihovnou Driver API, přeloženými kernely a device kódem[1]. Druhý rozdíl oproti Runtime API je ten, že Driver API pracuje explicitně s kontexty, které se musí ručně vytvářet a předávat jako parametry jednotlivých volání. To má výhodu v tom, že je možné ovládat jedním vláknem více kontextů (a tedy i například více GPU), předávat je mezi vláky apod., ovšem za cenu upovídanějšího kódu.

Runtime API, respektive rozšíření jazyka C, C for CUDA, doplňuje syntaxi jazyka C od CUDA-specifická rozšíření, zejména volání kernelu speciální syntaxí. Pro překlad takového zdrojového kódu se už ovšem už musí použít překladač nvcc:

```
kernel<<<d_grid, d_block, dyn_sh_mem, stream>>>(params...);
```

Tato přidaná syntaxe je výhodná proto, že nemusíme znát přesný název funkce jako s Driver API – tím je myšleno především plně dekorovaný C++ název kernelu, který musíme jako řetězec předat funkci, která spouští kernely. Tento řetězec není v případě C++ shodný s názvem funkce jako v C, ale zahrnuje v sobě i překladačem specificky zakódované typy parametrů, případně i parametry šablony. Pokud nepoužíváme šablony nebo přetížené funkce, tak stačí specifikovat kernel funkci jako **extern "C"**, v opačném případě je však potřeba používat plně dekorované názvy, což je poměrně nešikovné – potřebujeme buď reflexivně přečíst vygenerované symboly, znát přesný mechanismus dekorování nebo spolupracovat s příslušným modulem překladače.

Kontext se v Runtime API vytvoří implicitně při prvním volání některé funkce Runtime API[2] a tento kontext je pak využíván všemi dalšími voláními. Zde došlo ke změně chování od 4. verze CUDA Toolkitu. Až do verze 4 se vytvářel rozdílný kontext pro každé vlákno volající příslušnou funkci Runtime API. Toto chování poněkud komplikovalo vícevláknové a multi-GPU zpracování, protože například paměť (v RAM nebo na kartě) alokovaná v jednom vlákně nebyla použitelná voláními s kontextem druhého vlákna. Proto došlo k revizi tohoto chování a verze 4.0 a vyšší již vytváří právě jeden kontext pro proces a jedno GPU sdílený všemi vlákny.

Runtime API nicméně nezpřístupňuje veškerou funkcionalitu jako Driver API – například konverze sRGB textur při čtení na lineární není s Runtime

1. Poslední dva typy kódů ovšem přeloženy překladačem od NVidie být musí.
2. Ovšem ne všechny funkce potřebují, a tedy vytváření kontext. Typicky se jedná o analogie těch Driver API funkcí, které vyžadují parametr typu CUcontext.

API možné. Není možné ani používat současně v jednom programu volání Driver a Runtime API funkcí.

3.1.2 Hardware

Základní hardwarovou jednotkou je z pohledu architektury CUDA karta, protože CUDA zatím neumožňuje transparentně nahlížet na více karet jako jedno homogenní prostředí. Karta má několik (v případě high-end karet nejčastěji 15) multiprocesorů, ze kterých každý disponuje větším počtem aritmeticko-logických jednotek. Každý multiprocesor disponuje vlastní jednotkou dekódování instrukcí, která je sdílená všemi ALU. Co se týče vazby na bloky, tak platí, že jeden blok provádí právě jeden multiprocesor. Ten může současně provádět i více bloků, pakliže jsou malé. Ve skutečnosti zde může být opačný problém, pokud máme jen několik bloků, jejichž počet není dělitelný počtem multiprocesorů, tak velmi pravděpodobně bude GPU při provádění poslední sady bloků nevytížený. Jinak ale není možné dělat příliš závěry ohledně plánování provádění bloků, poněvadž CUDA přepíná mezi skupinami threadů (warpem) velmi dynamicky v závislosti na tom, jestli má data nebo na ně teprve čeká.

Hierarchie pamětí grafické karty je v podstatě dvouúrovňová – největší, ale i nejpomalejší je globální paměť. Ta má kromě relativně[3] nízké šířky pásma také velmi vysokou latenci, ovšem její obsah je perzistentní po celou dobu alokace, nejen po dobu spuštění kernelu. Podstatně rychlejší je paměť umístěná přímo na multiprocesoru, která je virtuálně rozdělena (o poměru nebo alespoň preferenci může rozhodovat programátor) mezi sdílenou paměť a L1/L2 cache. Obě dvě jsou realizované stejnými obvody, ale rozdíl mezi nimi je v tom, že obsah sdílené paměti si určuje programátor explicitně, kdežto cache funguje v obvyklém smyslu. Pro maximální výkon je nutné dodržovat určité přístupové vzorce k jak globální tak lokální paměti [4]

Data ke zpracování kartě jsou přenášena přes sběrnici PCIe. Pakliže máme hodně dat, na kterých provádíme operace, které probíhají vzhledem k propustnosti sběrnice rychle, tak existuje možnost, že se sběrnice stane úzkým hrdlem. Proto nabízí novější možnost překrývaných přenosů s výpočty[5]. Ty fungují tak, že používáme-li více CUDA streamů, tak je možné aby jeden stream prováděl kernel, zatímco druhý přenášel data z nebo na kartu. Bohu-

3. Relativně k ostatním GPU pamětem. Ve vztahu k šířce pásma hlavní paměti se jedná naopak o velmi vysoké hodnoty.
4. Viz [12]. Další ne zcela vhodné vzorce lze najít v diskuzi na `http://forums.nvidia.com/index.php?showtopic=181432`.
5. overlapped transfers

žel zde existuje omezení platící pro všechny generace GPU až do současnosti, že se musí jednat o prostý blok dat a ne se zarovnáním začátku řádku na určitou hranici – to jsou pole alokovaná například s cudaMallocPitch[6].

Nejnovější karty navíc zvládají přenášet data přes sběrnici zároveň v obou směrech – jeden stream na kartu a jiný do hlavní paměti. Stejně jako u překrývaných přenosů se musí jednat o jednoduché přenosy bez nějaké zvláštní intervence paměťového řadiče karty ($pitch = line\_size$).

Pro oba způsoby překrývaných datových přenosů platí, že se musí jednat o neodswapovatelnou paměť (page-locked) na straně hostitele, navíc alokovanou pomocí CUDA API.

Existuje ještě možnost paralelního běhu více kernelů na jednom GPU. Tato možnost však z hlediska této práce není nijak zvlášť zajímavá, protože její největší přínos je pro nedostatečné vytížení GPU, což při množství, granularitě dat a charakteru operací, které na nich provádíme, není tento případ.

3.1.3 Software

NVidia pro CUDA uzpůsobila překladač nvcc, který je odvozený z překladače Open64. Ten zase vychází z gcc a ponechává z něj C/C++ frontendy a většinu optimalizačních schémat [11]. Nvcc je používaný pro překlad jak CPU tak GPU kódu, pro CPU kód využívá backendů Open64, backend překlad pro grafické karty má nvcc vlastní. NVidia využívá vlastní instrukční sadu PTX. PTX assembler může být následně přeložen přímo do strojového kódu karty (cubin objekty) a slinkován přímo do výsledné aplikace, nebo distribuovaný odděleně a překládaný až v době běhu (just-in-time kompilace).

S překladem volně souvisí i způsob, jakým CUDA nakládá s funkcemi v device kódu. Pre-Fermi karty ve skutečnosti neměly žádnou podporu pro funkce a všechny funkce se inlinovaly přímo do kódu. To pak také znamenalo nemožnost použití rekurzivních volání, ukazatelů na funkce a virtuálních funkcí. Nové karty už mají přímou podporu pro funkce, čímž řada omezení odpadá.

6. CUDA C Programming Guide toto neuvádí úplně přesně, protože tvrdí, že není možné kopírovat bloky alokované touto funkcí. Ve skutečnosti nejde ani tak o alokaci, ale o proces kopírování, a jak jsem ověřil, tak CUDA má právě problém s kopírováním pomocí cudaMemcpy2DAsync a příbuzných funkcí i tehdy, když je paměťový prostor alokovaný pouze voláním cudaMalloc.

Kapitola 4

Knihovna VIPS a nástroj nip2

Projekt VIPS vznikl v roce 1989 jakožto dílčí část projektu VASARI, která měla na starost zpracování obrázků. Tento EU financovaný výzkumný projekt měl za úkol sledovat proces degradace barev na starých malbách. Snímky měly rozlišení 20 pixelů na milimetr, což znamená pro malbu o rozměrech 1x1 metr 1,6 GB.

Žádný z tehdy dostupných nástrojů nebyl schopen pracovat s takto velikými obrázky, proto vznikla knihovna VIPS. VIPS má některé unikátní rysy, které se nenacházejí u žádných jiných nástrojů.

Nejprve se jedná o líné vyhodnocování výrazů – jakákoliv úprava není vyhodnocena do té doby, než je potřeba (jestli vůbec) [8]. Operace se typicky řetězí, což znamená, že spuštění tohoto řetězu operací způsobí až žádost o výsledek. Případně, což je vlastnost často využívaná v nip2 je možné si efektivně vyžádat i jen malý výřez výsledného obrázku a knihovna VIPS vypočítá jen tuto část.

Knihovna VIPS umožňuje použití několika způsobů načítání a zpracování obrázků podle toho, co je efektivnější (IO, paměť). Mimo klasického zpracování celého obrázku najednou také zpracování po částech, což je vhodné pro velké obrázky, které se nevejdou do operační paměti a *in-place* zpracování [8].

Z hlediska specifik, zajímavých pro tuto práci je důležité, že neumí pracovat s 12 bitovým RGB, s výjimkou některých filtrů (gama korekce), které toto umožňují. Tato podpora však není systematická v celém systému.

Libvips umí pracovat s RGB soubory, buď s přiloženým ICC[1] profilem nebo i s jistými omezeními bez něj (viz příští sekce). Většina úprav se však provádí v barevném prostoru L*a*b* (sekce 6.2.1) případně obdobném. Z hlediska převodu mezi prostory je proto nutné to, aby byl zdrojový barevný prostor známý, takže v případě absence ICC profilu se zpravidla jako barevný prostor RGB obrázku uvažuje sRGB. Za zmínku stojí, že pokud převádíme obrázek s implicitně předpokládaným sRGB, VIPS předpokládá

1. ICC profil obsahuje informace o použitém barevném prostoru.

jeho gamu hodnotu 2,2, což není úplně v souladu s definicí, ale tento rozdíl je natolik malý [6, 17], že je to poměrně běžný postup.

4.1 Nip2

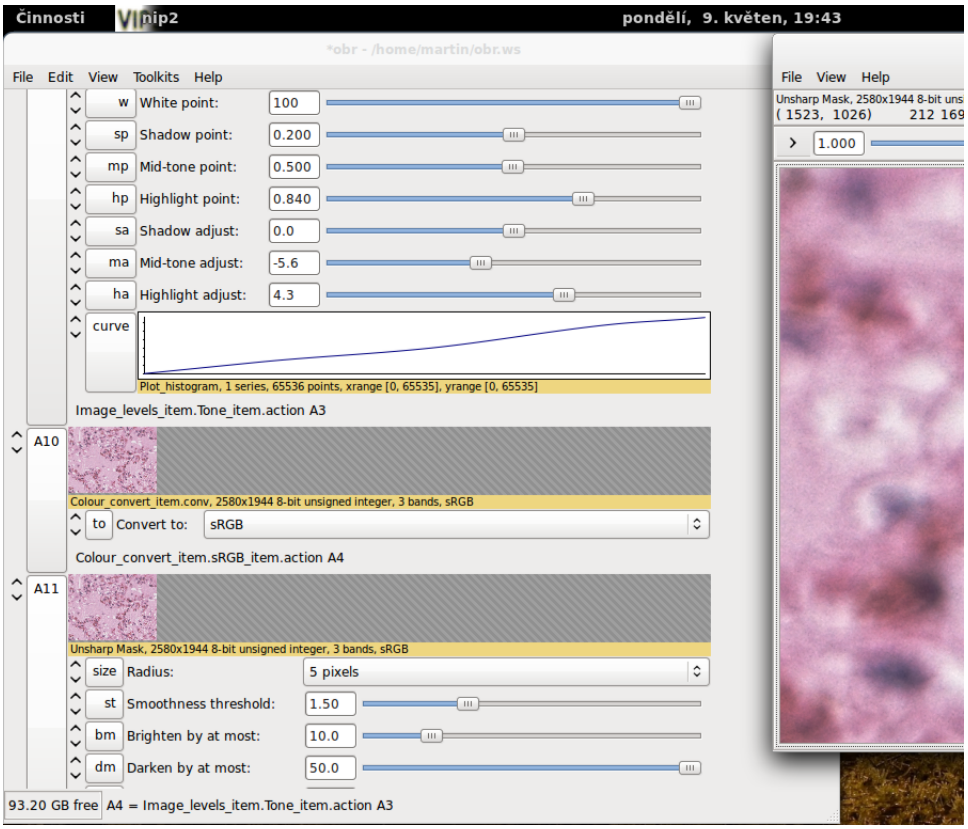
Uživatelské rozhraní ke knihovně VIPS tvoří GUI nip2 napsané v GTK 2.0. Jak tvrdí oficiální webová stránka[2], „jeho GUI má být na půl cesty mezi Excelem a Photoshopem“ – je špatné pro retušování a obdobné úpravy, ale stejně tak mocné pro všechno ostatní. Typickou ukázku tohoto rozhraní představuje obrázek 4.1. V hlavním okně je vidět několik úprav a způsob jejich řetězení. Kromě úprav prostřednictvím dialogu je rovněž možné zadávat příkazy (úpravy) přímo výrazem. Například zadání „$A1 * 1.1$“ způsobí lineární škálování koeficientem 1,1 všech barevných kanálů, nejčastěji gama-komprimovaných RGB hodnot, a tedy zesvětlení obrázku.

Nip2 není grafické rozhraní k libvips v tom smyslu, že by pouze předávalo požadavky knihovně. Nip2 je, stejně jako editory typu Adobe Photoshop nebo GIMP, vybavený interpretem vlastního skriptovacího jazyka. V případě nip2 se jedná o jednoduchý líně interpretovaný, objektově orientovaný funkcionální jazyk. Ovšem na rozdíl od zmíněných grafických editorů, v nip2 jsou i všechny zabudované úpravy implementované pomocí tohoto jazyka, který slouží jako spoj mezi vlastními dialogy operací a operacemi knihovny libvips. Funkce knihovny VIPS jsou zpřístupněna v tomto programovacím jazyku jako grafická primitiva. Tento rys může být pro uživatele velmi příjemný, protože i zabudované úpravy je možné ihned upravovat, případně vytvářet vlastní bez nutnosti jakéhokoliv zásahu do běhu aplikace (rekompilace, znovuspuštění).

Nevýhodou nip2 je to, že neumí pracovat s ICC profily souborů, ani je dále předávat knihovně VIPS, která by je transparentně používala, takže zjednodušeně řečeno, všechny osmibitové RGB soubory jsou brány jako sRGB. Škála úprav, které je možné provádět na 16-bitových souborech je velmi omezena na takové úpravy, které pracují přímo nad RGB soubory, což je třeba gama korekce. V každém případě jsou to úpravy, u nichž není potřebná interpretace barevných kanálů. Z úprav které se provádí nad patologickými snímky[3] dále už žádná úprava nepracuje přímo nad RGB, a tedy není možné je dělat nad 16-bitovými obrázky. Ostatně, sRGB je definovaný jako osmibitový prostor, proto nejde vydávat obrázky s širšími kanály za sRGB.

Pokud potřebujeme pracovat s 16-bitovým RGB, tak nám v nip2 v ur-

2. `http://www.vips.ecs.soton.ac.uk/index.php?title=VIPS`
3. uvedené v kapitole 6.3



Obrázek 4.1: Nip2 7.24.1

čité fázi nezbude, než jej degradovat na 8-bitový RGB (převod přímo do CIE XYZ eventuálně L*a*b* nip2 neumožňuje z důvodů uvedených výše). K tomu používá nip2 takový postup, že každý 16-bitový barevný kanál ořeže na 8-bitů a výsledný soubor pak prohlásí za sRGB.

Kapitola 5

Předchozí a související práce

V podkapitole této kapitoly shrnuji především dostupné nástroje, které by bylo možné využít pro řešení této práce. Co se týče vědeckých prací, tak není možné nezmínit diplomovou práci Jiřího Jelínka [8]. Tato práce se zabývá zpracováním patologických snímků se specifickým akcentem na jejich zobrazování na skládaných SAGE displejích. Mimoto zevrubně rozebírá mechaniku zpracování obrázků pomocí nip2. Z pohledu této práce je důležitý návrh algoritmu pro vyvážení bílé, který je aktuálně používán, takže jeho akcelerace bude zahrnuta v konečné implementaci. Pro účely uvedení architektury CUDA jsem využil NVIDIA CUDA C Programming Guide [12].

Co se týče ostatních odborných prací, existují práce věnující se dílčím otázkám jako třeba rekurzivní počítání Gaussovo rozostření. Pro něj existují dva význačné algoritmy – dobře známý Derichetův algoritmus [3] a také práce od van Vlieta a spol. [18]. Oba dva tyto přístupy mají různou míru přesnosti aproximace pro různé parametry [1] [5].

Další studovaná práce je GPU Gems 2, jejíž 20. kapitola se věnuje provádění bikubické interpolace na grafických kartách za využití speciálních interpolačních jednotek moderních GPU [16].

Mimoto využívám řady dílčích prací věnujících se provádění jednotlivých grafických úprav. Jde většinou o dílčí poznatky, proto tyto práce cituji v příslušných partiích textu.

Vedle odborných prací jsem také provedl rešerši dostupných softwarových nástrojů a knihoven, které by se daly použít pro řešení. Své poznatky shrnuji v následující sekci.

5.1 Nástroje

Akcelerace grafických výpočtů pomocí GPU je velmi dobře zvládnutý obor, který se vyvinul již dávno před existencí unifikované výpočetní architektury

1. Jedná se zejména o hodnotu σ, která ovlivňuje poloměr filtru. σ je zde směrodatná odchylka Gaussova rozdělení.

grafických karet a návazně zkoumaný a využívaný i pro CUDA. Zdálo by se tedy, že není problém navázat a ideálně i využít již existujících prací a knihoven. Bohužel, z rešerše, co jsem provedl, vyplývá, že situace není zdaleka tak ideální, jak by se mohlo na první pohled zdát.

Ponejprve je důležité uvést, že, ačkoliv by podle očekávání mělo být vhodných knihoven nebo nástrojů dostatek, není tomu tak. To je podle mě způsobeno tím, že architektura CUDA je stále relativně nová. Druhým důvodem pak je, že obecně výpočty na malé množině obrázků nejsou při dnešních rychlostech CPU zas až tolik náročné na absolutní čas, což znamená, že vytvářet konzumní nástroj pro zpracování fotek například z dovolené by se nevyplatilo. Navíc, bavíme-li se o například GIMPu, tak ten není vůbec zamýšlen pro dávkové zpracování, takže tam není těch pár sekund na provedení operace limitující. Jednou z čestných výjimek je ImageMagick, který dává možnost provést některé úpravy v OpenCL, pokud je slinkovaný s knihovnou OpenCL. To ovšem není zatím obvyklé, protože ta se zatím běžně distribuuje pouze s kompletním běhovým prostředím OpenCL konkrétního dodavatele, ačkoliv je možné ji stáhnout i ze stránek Khronos Group. V současnosti však ImageMagick umožňuje akcelerovat na GPU pouze konvoluci.

Když přejdeme k úžeji specializovaným nástrojům, tak tady už existují snahy o využití potenciálu GPU. Ovšem problémem těchto nástrojů je právě jejich úzké zaměření – aby byly použitelné pro řešení této práce, bylo by vhodné, aby implementovaly alespoň významný podíl operací, které potřebujeme. Navíc je důležité, aby jejich framework byl dostatečně rozšiřitelný, aby umožnil implementovat vše zbývající. Ideálně by se také měla implementace operací v takové knihovně co nejvíce blížit referenční implementaci, kterou implementuje VIPS[2]. Dále jsou zde určité výkonnostní požadavky, protože různé frameworky nemusí podporovat více GPU nebo překrývané přenosy s výpočtem, což nutně nemusí vadit, ale v případě, že bychom potřebovali co nejvyšší výkon by to mohlo být úzkým hrdlem.

Poměrně problematickým rysem těchto nástrojů také bývá jejich zaměření – poměrně populární je v CUDA obor počítačového vidění[3]. Ačkoliv i zde se dají najít podnětné nástroje nebo alespoň součásti použitelné pro grafiku, těžiště těchto projektů leží poněkud stranou od našeho zájmu. Relativně bližší oblastí, která je zkoumána a kde je CUDA využívána je oblast zpracování biomedicínského obrazu. Nicméně nejedná se často o zpracování obrazu ve smyslu grafických úprav, ale často o poměrně speciální úlohy typu práce s tomografickým obrazem, tedy ve 3D.

2. Úplnou kompatibilitu nelze uvažovat, protože i VIPS se uchyluje k některým kompromisům (viz kapitoly 4, 7 a 8)
3. computer vision

Začnu výčtem od zdroje, nástroji dostupnými od NVidie. NVidia se na poli softwarových grafických nástrojů prezentovala knihovnou OpenVIDIA, zaměřenou na počítačové vidění. Tato knihovna má poměrně široký záběr, nicméně jejímu využití brání to, že je poměrně letitá, a proto napsaná pro OpenGL. Dalším, už zajímavějším projektem je NVidia NPP[4]. NVidia NPP není ani tak knihovnou, jako spíše frameworkem pro CUDA, obsahující různá paralelní primitiva pro rozličné účely, nicméně prominentní z těchto oblastí je právě grafika a video. Tento framework se bude dokonce dodávat společně s CUDA Toolkitem od verze 4[5]. Pro grafiku jsou významná ta primitiva, která umožňují provádět generickou konvoluci, aritmetické operace atp. Relativní nevýhodou tohoto frameworku je pro naše účely příliš malá ohebnost, co se týče toho, jak zpracovávat různé relativně nestandardní barevné prostory jako L*a*b*, protože NPP je zaměřené primárně na zpracovávání RGB a prostorů odvozených z něj (Y'CbCr). Primitiva není například možné použít pro konvoluci jednoho barevného kanálu, pokud tento barevný kanál neoddělíme předem. Také obecnost tohoto frameworku je spíše na škodu, protože pak nemáme k dispozici vysokoúrovňové operace, které bychom potřebovali, ale pouze ty generické. Licence projektu je sice volná, ale ne ve smyslu GPL, takže není možné NVidia NPP distribuovat společně s výsledným projektem (což by bylo před vydáním CUDA Toolkitu 4 vhodné). Z těchto důvodů jsem se rozhodl daný framework nepoužívat, i když byly některé jeho myšlenky podnětné. Rovněž jsou inspirativní příklady z CUDA SDK, které jsou však málokdy použitelné tak, jak jsou. Navíc jsou modifikovatelné a redistribuovatelné pouze pod relativně restriktivní licencí.

Z ostatních prominentních knihoven můžeme jmenovat CUVILib, kvalitní proprietární nástroj, který umožňuje provádět některé grafické operace, ne však takové, které bychom potřebovali. Další velmi kvalitním nástrojem je ImageUtilities z projektu FlowLib [19], která se však také nekryje s našimi potřebami, ovšem obsahuje některé velmi pěkné myšlenky a nápady, které jsem ve své práci využil. Dále existuje další knihovna zaměřená na oblast počítačového vidění – GpuCV. Ta je bohužel z přímého použití dopředu vyřazena, stejně jako OpenVIDIA tím, že počítá v OpenGL shaderech. Většina dalších pak bohužel nedosahuje potřebné produkční kvality a spíše se jedná o ukázkové nebo pahýlovité pokusy (cudaOpenCV).

4. NVidia Performance Primitives
5. CUDA Toolkit 4 není ještě v březnu 2011 oficiálně vydaný. Samozřejmě se však dá NVidia NPP stáhnout samostatně.

Kapitola 6

Digitální zpracování obrazu

První použití digitálního obrazového materiálu sahá do 1. poloviny 20. století a je úzce spjato s vývojem sítí a jejich využitím v oblasti sdělovacích prostředků. To umožnilo tehdejším novinám otiskovat fotografie získané prostřednictvím transatlantického podmořského kabelu, přičemž doba potřebná k získání takových fotografií se zkrátila z více než týdne na pouhé jednotky hodin [4]. I po devadesáti letech zůstává aspekt rychlého šíření jednou z největších výhod, nadále umocněnou masovou dostupností datových sítí.

Digitální obraz ovšem také přinesl některé problémy, které analogová fotografie buď vůbec nepociťuje, nebo nejsou ve spojité doméně řešitelné. Jedním ze zásadních problémů je přenositelná interpretace chromatické informace různými zobrazovacími systémy. K řešení této otázky byla zřízena organizace CIE (viz sekce 6.2.1).

Příjemnou vlastností digitálních obrázků ovšem je, že je možné na ně jednoduše aplikovat různé obrazové filtry, což je i náplní této práce.

6.1 Digitální obraz

Obsah digitálního obrazu je určený prostřednictvím funkce $f(x, y)$, kde x a y jsou rovinné souřadnice a hodnota této funkce udává intenzitu barvy (monochromatické body) nebo jednotlivých barevných složek (chromatické obrázky) v daném bodě [4]. Obecně se dá uvažovat spojitá doména této funkce: $\{i \times j | i, j \in \mathbb{R}^+, i \leq m, j \leq n\}$, často se však spokojíme pouze pro konečnou podmnožinu $\mathbb{Q}^2$ diskrétních hodnot, což i odpovídá uložení s využitím pouze konečného množství hodnot. Pakliže je přece jenom potřeba celý definiční obor funkce f, použije se zpravidla interpolace k získání nepokrytých hodnot.

V praxi je obrázek kvantizovaný pomocí konečného množství hodnot (ne nutně závislého na velikosti obrázku). V nejjednodušším případě se jedná přímo o vzorkování rovinných bodů (nazývané pixely). Jinou možností je ukládat koeficienty vlnových délek jednotlivých konstituujících frekvencí, jako v případě JPEG/JPEG2000.

6.2 Barevné prostory

Ke zaznamenání barevné informace jsou zavedeny barevné prostory. Téměř ve všech případech jde o třírozměrné vyjádření barevné informace. Některé barevné prostory jsou vhodné pro zpracování člověkem (výběr, úprava), jiné, například RGB, pro přímé zobrazení na monitoru [14].

Faktorem, který je potřeba uvažovat, je to, že zobrazení mnohých prostorů závisí na zobrazovacím zařízení, které se použije. Na druhou stranu existují prostory, jejichž interpretace je jednoznačně daná – to je případ prostorů z rodiny CIE a také barevného prostoru sRGB.

6.2.1 Rodina prostorů CIE

Organizace CIE[1] je standardizační skupina, která vznikla za účelem specifikace a standardizace barevných pojmů a prostorů. Barevné prostory definované touto mezinárodní organizací (XYZ /6.2.1/, L*a*b* /6.2.1/ aj.) nejsou zpravidla používány přímo pro uložení obrázků, ale používají se pro provádění úprav v situacích, kdy je žádoucí přesná interpretace barevných hodnot.

CIE XYZ

Na základě experimentálních měření ve 20. letech 20. století byl navržený barevný prostor CIE RGB, jehož barevné složky byly odpozorovány na základě lidského vnímání [15].

Tento prostor neměl zcela optimální vlastnosti (možné negativní koeficienty), proto následovala v roce 1931 specifikace prostoru XYZ, který tyto nedostatky odstranil. Z prostoru XYZ se často odvozuje prostor xyY, který separuje chromicitu barvy (x a y).

Obecně platí, že jakoukoliv barvu libovolného RGB barevného prostoru je možné beze zbytku reprezentovat i v souřadnicích XYZ, což umožňuje použití XYZ jakožto dobře definovaného absolutního prostoru k vzájemným převodům. Navíc, převod z některého z RGB prostorů je možné provést lineárním zobrazením (maticí), takže je taková transformace i výpočetně přijatelná. Vizuálně se vztah mezi xyY a RGB dá zobrazit pomocí gamutu, což je zobrazení těch hodnot xy, které je možné reprezentovat daným RGB prostorem. Gamuty RGB prostorů mají tvar trojúhelníku a z jeho plochy a polohy se dá mimo jiné odvodit spektrum barev, které je možné tímto prostorem reprezentovat.

1. Commission internationale de l'éclairage

CIE L*a*b*

Nevýhodou prostoru XYZ je, že metrika jejich jednotlivých složek není uniformní z hlediska lidského vnímání. Tento nedostatek vedl CIE k vytvoření prostorů L*a*b* a L*u*v*. Oba prostory zlepšují nerovnoměrnosti XYZ z původního poměru 80:1 na přijatelnějších 6:1 [15], ovšem za cenu větší výpočetní náročnosti.

Knihovna VIPS používá z těchto dvou výhradně L*a*b* [2, kapitola 1.2], protože na výpočet převodu do tohoto prostoru je potřeba méně matematických operací než do L*u*v*, což je výhoda i pro implementaci na GPU, proto budu v dalším textu prostor L*u*v* ignorovat.

Složka L^* reprezentuje světlost, která je definované jako (lineární) perceptuální odezva na vyzařované světlo obrazovky [15]. Hodnoty a^* a b^* jsou nenormalizované kartézské souřadnice chromatické informace.

Z této definice vyplývá, že L*a*b* je vhodný zejména na takové úpravy, kde se operuje se hodnotami jasu, ale nikoliv s barevností.

Matematicky se definuje převod z XYZ takto [7]:

$$f(x) = \begin{cases} x^{\frac{1}{3}} & \text{pro } Y > \left(\frac{6}{29}\right)^3 \\ \frac{1}{3}\left(\frac{29}{6}\right)^2 x + \frac{4}{29} & \text{jinak} \end{cases} \tag{6.1}$$

$$L^* = 116 f\left(\frac{Y}{Y_n}\right) - 16 \tag{6.2}$$

$$a^* = 500\left[\frac{X}{X_n} - f\left(\frac{Y}{Y_n}\right)\right] \tag{6.3}$$

$$b^* = 500\left[\frac{Y}{Y_n} - f\left(\frac{Z}{Z_n}\right)\right] \tag{6.4}$$

$$\tag{6.5}$$

$X_nY_nZ_n$ zde udávají XYZ souřadnice referenčního bílého bodu daného barevného prostoru. V případě sRGB jde o iluminant D65, který má souřadnice [0,9505; 1; 1,0890][2].

CIE L*C*h*

L*C*h* přechází od kartézských souřadnic a^*b^* k souřadnicím cylindrickým, které jsou vhodné pro takové úpravy, které potřebují pracovat s chromatickou informací. Výška válce určuje světlost, poloměr saturaci a úhel odstín barvy.

V případě L*C*h* je souřadnice L* shodná s příslušnou složkou prostoru L*a*b*, přechod z kartézských a*b* souřadnic do polárních C*h* je

2. Více informací například v [15]

definován následující transformací:

$$C^* = \sqrt{a^{*2} + b^{*2}} \tag{6.6}$$
$$h^* = atan2(b^*, a^*) \tag{6.7}$$

6.2.2 RGB barevné prostory

Důležitou skupinou barevných prostorů jsou prostory RGB. Tyto barevné prostory ovšem přinášejí určité komplikace jak pro interpretaci a úpravy lidským pozorovatelem, tak i pro přenositelnost mezi zobrazovacími zařízeními. Na druhou stranu, tyto aditivní systémy jsou vhodné pro přímé zobrazení na monitoru počítače, proto jsou většinou používané pro uložení obrázků.

Obvykle není zvykem ukládat přímo lineární intenzitu jednotlivých barevných složek z toho důvodu, že historicky měly CRT monitory tu vlastnost, že vyzařovaná intenzita nebyla lineárně úměrná napětí, se kterým se emitovaly jednotlivé elektrony a bylo žádoucí, aby bylo možné obraz rovnou (bez transformací) zobrazit. Vztah mezi vyzařovanou intenzitou barvy a emitujícím napětí je mocninný a odpovídá rovnici:

$$V_{out} = V_{in}^{\gamma} \tag{6.8}$$

Pokud bych měl být úplně přesný, tak jako RGB bývá někdy zvykem označovat lineární hodnoty primárních složek, pro gama-komprimované hodnoty se pak uvádí R'G'B'. Nicméně tento úzus není úplně striktně dodržován a z kontextu bývá většinou zřejmé, o které hodnoty se jedná.

Kromě hodnoty gama je každý barevný RGB systém určen přesné vlnovými délky primárních barev. Posledním důležitým parametrem je hodnota referenčního bílého bodu. Tato hodnota se zpravidla vyjadřuje v **xy** souřadnicích prostoru xyZ nebo teplotou bílého bodu (v kelvinech).

Výčtem všech těchto parametrů je možné přesně určit souřadnice konkrétní R'G'B' barvy v některém absolutním barevném prostoru, například v CIE XYZ.

Pro RGB existují de facto standardní prostory, které mají napevno stanovené parametry (bílý bod, primární barvy), například Adobe 98 RGB, Apple RGB, barevný prostor používaný v PDF nebo sRGB.

V příloze A uvádím obvyklý postup, který se používá, pokud potřebujeme převádět mezi různými RGB barevnými prostory nebo převést do některého z prostorů CIE.

sRGB

Barevný prostor sRGB je v současnosti nejrozšířenějším barevným prostorem z rodiny RGB a implicitně se předpokládá u všech obrázků, které nejsou označeny explicitně jinak[3].

SRGB předpokládá gama hodnotu monitoru přibližně o hodnotě 2,2. Tato hodnota však není oproti většině ostatních R'G'B' prostorů konstantní na celém definičním oboru, ale její funkční hodnota se pohybuje od hodnoty 1 až po 2,4 [17]. Rozdíl oproti konstantní hodnotě gama 2,2 je však natolik malý [6, kapitola 13], že se často zanedbává a používá se jednoduše hodnota 2,2.

Primární barvy tohoto prostoru specifikované standardem ITU-R BT.709 pro vysílání HDTV a D65 je referenční bílý bod.

6.3 Úpravy obrazu

Úpravy obrazových dat lze obecně rozdělit do dvou kategorií – zlepšování obrazu[4] a obrazová rekonstrukce[5]. Obrazová rekonstrukce zahrnuje takové úpravy, které přibližují obraz původní čisté podobě, která neobsahuje šumy a jiná zkreslení zanesená předchozím procesem. Zlepšování obrazu má za úkol uzpůsobit vzhled daného obrazu účelu, za kterým bude použit, za cenu odchýlení se od původní formy. Mezi takové úpravy patří například zvýšení kontrastu nebo saturace barev.

Většina požadovaných úprav by se dá zařadit do první skupiny. Nicméně u některých barevných úprav, které se aplikují, by toto řazení bylo poměrně špatně odůvodnitelné. Tady je třeba vzít v úvahu jiný aspekt, a to ten, že zdrojové obrázky nemají explicitně specifikovaný barevný prostor a přenosovou funkci (gama koeficient), proto některé úpravy mohou mít za úkol spíše převod a přizpůsobení vlastnostem sRGB. V tom případě se postupuje tak, že se upravuje obrázek do té doby, dokud nevypadá vizuálně dobře a výsledek těchto úprav se prohlásí za, pokud úpravy probíhaly na sRGB displeji [17].

6.3.1 Základy úprav

V předchozí sekci jsem uvedl pojem barevných prostorů a různé možnosti jejich transformací, převodů a podobně. Psal jsem rovněž o možných úpravách nad původními nebo odvozenými barevnými prostory. Nyní by bylo namístě

3. K tomu se používá ICC profil, který nese informace o příslušném barevném prostoru.
4. image enhancement
5. image reconstruction

zavést vlastní pojem úpravy (filtru), zavést kategorizaci úprav a vyjasnit některé související pojmy.

Velmi často používanou třídou úprav jsou úpravy, které se provádí nezávisle nad každým bodem obrázku, což je obzvláště výhodné pro GPU zpracování, protože se dá provádět jednoduše paralelně. Další úpravy jsou takové, které pracují s určitou množinou sousedů daného bodu. Z úprav, co mě zajímají, se jedná o interpolaci a unsharp mask. Protože úpravy tohoto mají společné univerzální jádro, které je v grafice klasické, věnuji tomuto obecnému schématu následující podkapitolu. Interpolace se ovšem z daného schématu vymyká proto, že nezůstává zachovaný rozměr obrázku, jinak se jedná o podobný princip.

6.3.2 Mechanika provádění prostorových filtrů

Velká část úprav, které pracují s jistým okolím zpracovávaného bodu se dá zařadit do dvou základních skupin – konvoluce a korelace. Tyto operace mají obrovskou výhodu v tom, že jsou definované obecně, a tedy použitelné pro různé druhy prostorových filtrů.

Základní předpoklad je, že filtr lze v případě dvourozměrných obrázků definovat pomocí matice koeficientů, která určuje, v jakém poměru se do výsledného pixelu bude započítávat hodnota sousedních pixelů. Aby při určování pozice centrálního bodu nevznikaly dvojznačnosti, mají filtry oba dva rozměry liché, čímž je střed filtru jednoznačně určen.

Matematicky se korelace definuje pro filtr velikosti $m = 2a+1, n = 2b+1$ takto [4, kapitola 3]:

$$g(x,y) = \sum_{s=-a}^{a} \sum_{t=-b}^{b} w(s,t) f(x+s, y+t)$$

Situaci pro jeden rozměr ilustruje schéma na obrázku 6.1. Význam jednotlivých symbolů je tento:

f zdrojový obrázek

g zdrojový obrázek

w filtr, který aplikujeme

x horizontální souřadnice zpracovávaného bodu

y vertikální souřadnice zpracovávaného bodu

$$
\begin{aligned}
f &= (0,1,0,0) \\
f_{padded} &= (\mathbf{0},0,1,0,0,\mathbf{0}) \\
w &= (1,2,3)
\end{aligned}
$$

f	**0**	**0**	**1**	0	0	0
w	**1**	**2**	**3**			
g	_	**3**	?	?	?	_
f	0	**0**	**1**	**0**	0	0
w		**1**	**2**	**3**		
g	_	3	**2**	?	?	_
f	0	0	**1**	**0**	**0**	0
w			**1**	**2**	**3**	
g	_	3	2	**1**	?	_
f	0	0	1	**0**	**0**	**0**
w				**1**	**2**	**3**
g	_	3	2	1	**0**	_

$$g = (3,2,1,0)$$

Obrázek 6.1: Korelace

Střed filtru je element s hodnotou 2. Postup si můžeme představit tak, že posunujeme filtr zleva po jednotlivých hodnotách obrázku a do výsledné hodnoty zapisujeme skalární součin hodnot filtru a hodnot vstupního obrázku na příslušných pozicích. Zapisuje se pozice odpovídající středu právě zpracovávaného výseku zdrojového obrazu. Aby bylo možné korelovat i krajní body, zavádí se z obou stran dodatečná nulová výplň široká $\lfloor \frac{m}{2} \rfloor$.

Situace pro konvoluci je analogická s korelací, přičemž filtr se nejprve rotuje o 180° bez ohledu na počet dimenzí, ve kterých pracujeme. Tím pádem by výsledek příkladu na obrázku 6.1 pro totéž w a f byl $(1,2,3,0)$.

Kapitola 7

Používané grafické úpravy

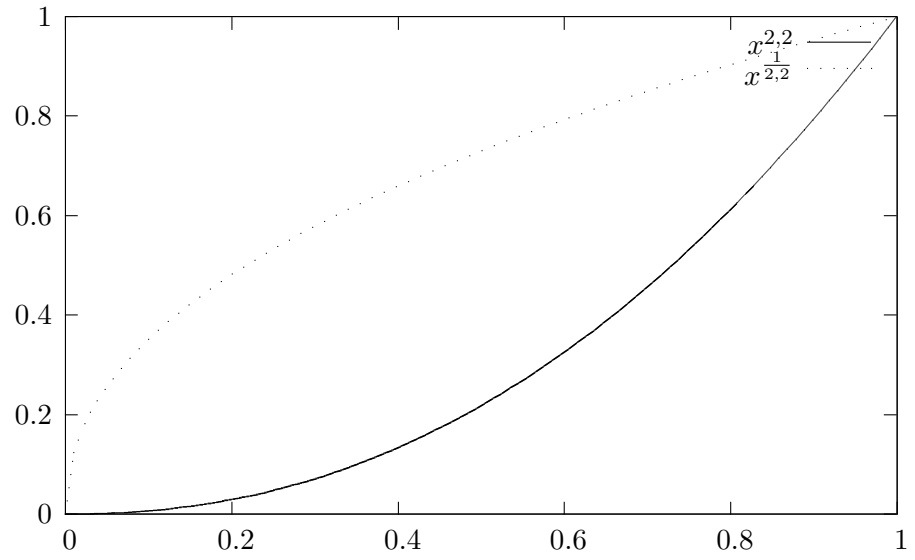
V rámci řešení této diplomové práce je zadána podmnožina úprav, které nabízí nip2 a které se používají k úpravám patologických obrázků. Tyto úpravy mají být akcelerovány grafickými kartami NVidia, což ovšem znamená, že veškeré úpravy musí být kompletně přepsány, a to ideálně tak, aby zachovávaly jak obecnou sémantiku těchto úprav, tak jistá specifika programů nip2 a VIPS.

Nyní zběžně projdu tyto úpravy na obrázcích přesně tak, jak jsou požadovány a zadány v příslušném nip2 souboru. U některých se pokusím nastínit jejich sémantiku a případně jejich souvislosti, přičemž podrobnější informace k implementaci uvádím později.

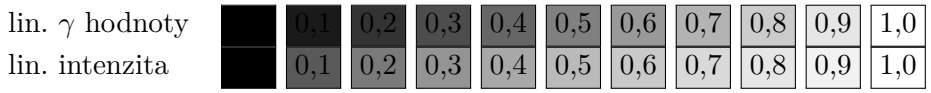
7.1 Gama korekce

Gama korekce je nelineární úprava intenzity barev, která má primárně za úkol kompenzovat nelineární odezvu katodových trubic klasických monitorů. Obvykle se k úpravě využívá mocninné funkce. Obrázek 7.1 ukazuje tvar funkce pro hodnotu gama 2.2, což je obvyklá hodnota na běžném displeji (a zároveň zhruba odpovídá přenosové funkci barevného prostoru sRGB, viz 6.2.2). Hodnota $gama < 1,0$ je kódovací gama používaná pro kompresi obrázků a $gama > 1,0$ se používá pro dekompresi lineárních barevných hodnot.

Obrázek 7.2 ukazuje rozdíl mezi lineární škálou gama-komprimovaných hodnot (gama 2,2) a lineární škálou intenzit. Zjevně vypadá řada lineárních gama-komprimovaných hodnot opticky lineárněji než druhá škála s lineárními intenzitami, což se může zdát poněkud překvapivé. Ve skutečnosti je tomu opravdu tak – rozdíl mezi černou a prvním stupněm šedi je při lineární intenzitě neproporčně větší než ostatní rozdíly, zatímco v případě gama komprimovaných hodnot je to vizuálně zhruba stejný rozdíl jako u ostatních vzdáleností. Vysvětlením je, že lidské oko má zhruba logaritmickou odezvu na úroveň světla, a tedy lineární intenzita se nezdá lineární. Oproti tomu



Obrázek 7.1: Gama korekce – funkce



Obrázek 7.2: Gama korekce – úrovně intenzity

gama korigované hodnoty tuto nelinearitu částečně potlačují[1].

Kromě prosté komprese/dekomprese se tato úprava dá rovněž využít pro převod mezi různými exponenciálními přenosovými funkcemi. To je zřejmě i případ vstupních souborů, které generuje skener, se kterým pracujeme. Požadovaná gama hodnota 0,7 neodpovídá kompresnímu exponentu ($\frac{1}{2,2} \doteq 0,45$) prostoru sRGB, který nip2 implicitně předpokládá, tudíž nejde o lineární RGB hodnoty. Vysvětlením může být, že tato gama komprese slouží jako adaptace zdrojového barevného prostoru do sRGB. Druhou možností je, že takto upravený obrázek jednoduše vypadá lépe.

1. Pro srovnání, L*a*b* používá zhruba inverzní funkci ke gama dekompresi k získání luminozity, viz 6.2.1.

7.2 Úrovně barev

Touto úpravou je možné provést další úpravy úrovní barev, nejen na základě mocninné funkce. Parametry se zadávají prostřednictvím uživatelem definované křivky, která může vypadat například jako jedna z křivek na obrázku 7.1, ale v případě této úpravy se zadává pomocí několika bodů, kterými se křivka proloží.

7.3 Unsharp mask

Unsharp mask je úprava, která má za úkol optické zlepšení ostrosti obrazu. Sestává se ze dvou kroků – nejprve se obstará rozostřená verze obrázku konvolvovaná za pomocí Gaussova filtru [4, kapitola 4] . Tento rozostřovací filtr má hodnoty dané Gaussovým rozdělením se střední hodnotou ve středu filtru. Následně se odečte rozostřená verze od originálu a tento rozdíl se přičte k původnímu obrázku.

7.4 Zmenšení

Zmenšení je konceptuálně poměrně jednoduchá úprava, jejímž jádrem je interpolace původního obrázků na místech odpovídajících novým souřadnicím. Používá se několik interpolačních algoritmů, volba se odvíjí zejména od požadovaného poměru kvalita/rychlost. Jde od tyto (od nejméně kvalitního): nejbližší soused, bilineární a bikubický. Kromě toho je možné také použít algoritmy pro zlepšení hran interpolovaného obrázku.

7.5 Vyvážení bílé

Vyvážení bílé je implementováno tak, jak popisuje Jiří Jelínek [8]. Tato úprava se poněkud odchyluje od referenční implementace, tak jak je implementovaná v nip2, proto lze odkázat případné zájemce přímo tam.

Referenční (nipová) implementace nebyla vhodná, protože špatně škálovala domnělou bílou na šedou a nikoliv bílou. Nyní se používá následující

algoritmus:

$$\begin{aligned} new_r &= r * \frac{white_r}{sel_r} \\ new_g &= g * \frac{white_g}{sel_g} \\ new_b &= b * \frac{white_b}{sel_b} \end{aligned}$$

Vektor sel_3 udává barevné RGB souřadnice vybrané barvy, která má být považována za bílou a $white_3$ referenční souřadnice bílé daného barevného prostoru, na kterým se transformace provádí.

7.6 Saturace barev

Jednou třídou z možných perceptuálně orientovaných reprezentací barev jsou barevné prostory vyjadřující barvu jako trojici: odstín, sytost a jas.

Nejjednodušší reprezentace těchto veličin odvozuje jejich hodnotu přímo z hodnot jednotlivých barevných složek RGB. Tato reprezentace není ovšem dostatečně perceptuálně uniformní, proto se pro přesnější výpočty používá odvození z barevného prostoru CIE L*a*b případně CIE L*u*v. Tyto barevné prostory reprezentují chromicitu v kartézských souřadnicích.

CIE L*C*h* oproti tomu přechází k cylindrické reprezentaci, kde luminance je nezměněna, ale kartézské souřadnice a*b* se transformují na polární souřadnice. Úhel reprezentuje odstín a poloměr saturaci barvy.

Sémantika této úpravy je ve spojení s přecházející poměrně jasná – hlavním účelem je kompenzovat zvýšenou saturaci barev, která vznikne předchozím vyvažováním bílé.

7.7 Ostatní úpravy

Zbylé úpravy jsou technického charakteru a souvisí s konkrétním způsobem, jakým nip2 s obrázky nakládá.

7.7.1 Bitový posun

Tato úprava je zde z důvodu obcházení omezené podpory libvips pro jiné než 16 nebo 8 bitové barevné kanály. V našem případě se jedná o 12 bitovou barevnou hloubku, která se kompenzuje na 16 bitů bitovým posunem o 4 pozice doleva.

V obecném případě by aplikací této úpravy došlo ke změně obrázku, tj. ke zvýšení intenzity jednotlivých barevných kanálů násobkem 2^n, kde n je počet bitů. To není ovšem případ, protože VIPS ignoruje atribut *TIFFTAG_MAXSAMPLEVALUE* a předpokládá maximální hodnotu rovnou 2^n, kde n je počet bitů na vzorek. Tím, že ve svém programu danou hodnotu čtu a zohledňuji, nepotřebuji (a vlastně ani nesmím) tuto kompenzační úpravu provést.

7.7.2 Převod do sRGB

V našem případě se jedná o částečně triviální úpravu, protože nip2 sRGB implicitně předpokládá, a to nepřímo i pro 16-bitové obrázky (viz strana 26), proto není potřeba žádná konkrétní změna vstupního obrázku.

7.7.3 Změna interní reprezentace čísel

Hodnoty jednotlivých barevných kanálů je možné reprezentovat buď celočíselnou konstantou (s nebo bez znaménka) nebo číslem s pohyblivou řádovou čárkou. Pro GPU zpracování je výhodnější pohyblivá čárka, nicméně tradiční CPU zpracování se obvykle provádí v celočíselné aritmetice. V tomto formátu ukládá obvykle i libvips, ačkoliv umožňuje i jiné reprezentace, ale zřejmě není nutné se tohoto příliš držet. I čísla s plovoucí desetinnou čárkou uložená ve 32 bitech (jednoduchá přesnost) jsou schopna pojmout na rozumném intervalu mnohem více hodnot, než 16-bitové celé číslo.

Rovněž ani k této úpravě neimplementuji žádnou akci, protože se jedná o implementační detail, jak jsou obrazová data uložena, a není proto potřeba imitovat chování nip2 v těchto detailech.

Kapitola 8

Návrh a implementace

Architektura, na kterou projekt cílí, je na základě diskuse ve 3. kapitole CUDA. Tato architektura je v mnoha ohledech odlišná od CPU, jak popisuji v téže kapitole, proto se odlišuje i celý návrh jednotlivých úprav a vlastně i způsob nasazení a spouštění jednotlivý výpočetních modulů. Dlužno podotknout, že základní návrh jednotlivých úprav pro CUDA není úplně obtížný, což je dané tím, že většina grafických úprav je přirozeně velmi paralelních a navíc přirozené mapování jednotlivých úprav poměrně přesně odpovídá charakteru architektury. To ovšem neznamená, že by nebyl potřeba pečlivý návrh, jak u složitějších úprav, tak i u jednodušších k dosažení co nejlepšího výkonu a univerzality.

Produkt této práce vzniká z velké částí od počátku, proto je důležité také navrhnout vhodný rámec pro provádění, od parsování vstupních souborů, přes způsob spouštění výpočtů až po plánování IO. To se nakonec ukázalo jako kritické.

8.1 Vstupy

Vstupem pro aplikaci je (uspořadatelná) množina velkého počtu dlaždic v počtu řádově 10-25 tisíc, které jsou uloženy každý individuálně jako samostatný obrázek. Předpokládaným fyzickým úložištěm je diskové pole. Celkový objem dat, které čteme, je řádově ve stovkách GB. Vstupní soubor s úpravami je pracovní soubor programu nip2.

8.1.1 WS soubory

Pracovní soubor nipu je XML soubor obsahující všechny úpravy a jejich parametry. Nepříjemné je, že kromě příslušných úprav obsahuje řadu věcí nesouvisející s grafickými úpravami a vztahující se ke GUI. Kód B.1 (příloha B) ukazuje výřez z takového WS souboru. Obsah tohoto úryvku zahrnuje data uložená nipem vztahující se ke gama korekci. Je z něj vidět další méně příjemný rys, a to ten, že u polí, které se v příslušném dialogu nipu nezmění a

ponechají se na výchozí hodnotě, se tato implicitní hodnota do WS souboru nezapíše. Jinak ovšem obsahuje WS soubor všechna potřebná data.

Nejdůležitější informaci obsahuje značka *iText* s atributem *formula* na 2. vnořené úrovni úpravy. Tento atribut je zhruba příkaz, který se má provést nad obrázkem. Může obsahovat skutečné parametry příkazové řádky. Ostatní atributy, zejména ty, které jsou nastavené pomocí dialogů nipu, jsou uloženy v jiných větvích XML stromu. Ačkoliv má u většiny úprav (zejména zadaných z GUI) příkaz tvar *„název_filtru vstupní_obrázek“*, jindy může jít o značně složitý výraz. V případě úprav, které máme zadané, je jediný složitější výraz „$A1 << 4$“ (k vykompenzování nemožnosti pracovat v nipu s 12-bitovými obrázky). Obecně se však může jednat až o bezkontextovou syntaxi. Vzhledem k tomu, že takovéto výrazy jsou spíše jen okrajové a skoro vždy jdou rozepsat na více menších úprav jsem z tohoto požadavku ustoupil, a předpokládám nanejvýš jednoduché výrazy typu „$Ax + 15$“ s typickými symboly operátorů, což mi dává společně s běžnými příkazy relativně jednoduchou regulární syntaxi.

8.1.2 Vstupní a výstupní soubory

Vstupem jsou typicky buď 8 nebo 12-bitové RGB obrázky, případně není problém s jakoukoliv jinou hloubkou do 16 bitů. Primárně předpokládám uložení v TIFFu. Protože TIFF umožňuje ukládání dat několika možnými způsoby, nabízí libtiff pro čtení vysokoúrovňové API, které ukrývá skutečnou vnitřní strukturu. Nevýhoda tohoto API je, že automaticky ořezává vstup na 8-bitů a není vláknově bezpečné, což jsou důvody, kvůli kterým není pro naše účely vyhovující. Používám tedy nízkoúrovňové funkce této knihovny, u kterých programátor už potřebuje ošetřit čtení z různě strukturovaných souborů sám.

Data mohou být v TIFFu ukládaná buď po pruzích – to znamená nařezání obrazu po řádcích (část, jeden či více) – nebo v dlaždicích, kde dlaždice má obvyklý význam, který používám v textu celé práce. Další možnost volby vnitřní struktury je volba rovinné konfigurace souboru s ohledem na více kanálů – buď lze ukládat všechny vzorky v jedné rovině, v případě RGB tedy: $R_iG_iB_iR_{i+1}G_{i+1}B_{i+1}\ldots$. Druhou možností je pak uložit nejdříve všechny vzorky červené, pak zelené a nakonec modré, což vytvoří tři logické roviny. Obrázky mohou být komprimované bezztrátově (LZW) případně i ztrátově (JPEG komprese). Kromě toho nabízí i plejádu exotičtějších kompresních schémat, které nebudu vyjmenovávat[1]. Výhodou je, že libtiff zvládá kompresi i dekompresi transparentně pro programátora i při použití nízkoúrovňového

1. dají se nalézt v hlavičkovém souboru *tiff.h*

API, takže s nimi není žádná dodatečná práce.

TIFF je velmi bohatý formát na různé možnosti, atributy, konfigurace apod., proto by asi nebylo účelné implementovat celou dekódovací mašinerii pro všechny exotické volby, proto i já předpokládám pouze určitou podmnožinu vlastností, které zpracovávám. Konkrétně RGB 3-kanálové obrázky, šířku kanálu 8 nebo 16 bitů, uložení ve stridech a jedné rovině, ve které jsou uloženy všechny barevné kanály. Na druhou stranu interpretuji (a korektně v celém projektu pracuji s) atribut *TIFFTAG_MAXSAMPLEVALUE*, který identifikuje 12 bitové obrázky hodnotou 4095 (ale logicky může interpretovat i jiné barevné hloubky).

Jednotlivé vstupní dlaždice mají velikost do tří desítek megabajtů, minimálně několik megabajtů, což můžou být předpoklady velmi relevantní pro návrh a nasazení systému.

8.2 Grafické úpravy a jejich nasazení na GPU

Nejhrubší rozdělení úprav, které urychluji, je do dvou skupin – ty které se aplikují nezávisle na každá obrazový bod a ostatní. První skupina je přirozeně vysoce paralelní a dává takřka neomezené možnosti nasazení na GPU, stačí optimalizovat parametry tak, aby byly co nejvýhodnější na GPU. Implementace pak v zásadě může být realizována dvěma způsoby – vyhledávací tabulkou nebo přímým výpočtem na GPU. Obecně je tabulka rychlostně výhodnější, ovšem ne vždy použitelná – můžeme mít tolik možných stavů, že by byla její velikost neúnosně veliká, třeba pokud se zobrazuje z kartézského součinu tří hodnot (barevných kanálů), kde každý má 12 bitů, dává dohromady 2^{36} diskrétních hodnot. Ovšem i v případě, že potřebujeme tabulku pouze pro jediný kanál, v případě 16-bitových hodnot se jedná o 64 Ki diskrétních hodnot, což dává velikost tabulky 128 KiB při zobrazení $a \to b; a \in [0..2^{16}-1], b \in [0..2^{16}-1]$, což je příliš velká hodnota pro uložení celé tabulky do některé z rychlejších pamětí současné generace GPU (L2 cache/sdílená paměť, texturová cache nebo paměť konstant)[2].

U ostatních úprav musí být paralelní návrh poněkud odlišný, proto nyní uvedu, pro který jsem se rozhodl u jednotlivých úprav.

8.2.1 Pomocné funkce

Pomocnými funkcemi rozumím takové úpravy, které nejsou zadané ve WS souboru, ale postupem času se ukázalo, že je lepší některé předzpracování a

2. Kapacita je sice ve stejném řádu, který potřebujeme, ale konkrétní hodnoty jsou 2x nižší nebo méně. Přesné hodnoty je možné najít například v [12].

finalizaci provádět na GPU než na CPU, proto je jejich paralelní návrh také zajímavý.

Jde především o úpravy vstupů a výstupů takové, aby byly pro GPU zpracování co nejvýhodnější. Tady jde primárně o vybalení trojic hodnot (RGB) na více vhodné čtveřice pro GPU zpracování.

První myšlenka byla toto implementovat v CPU s tím, že by to měl procesor zvládat rychle při načítání vstupního souboru, kdy má ještě data v L1 cache, ideálně na rychlosti memcpy. Tento předpoklad se však ukázal mylný a implementace na CPU trvala 50 ms, což je vzhledem k rychlostem, které uvádím v kapitole 9 neúnosné. Mimoto, při současných rychlostech hlavní paměti nemá ani kopírování s memcpy úplně decentní čas – při téže kapacitě 10-15 ms (ovšem bez zátěže procesoru).

Implementace rozbalování a sbalování vektorů jednotlivých bodů je velmi přímočará – každé vlákno přečte svůj vstupní vektor, vypočítá adresu výstupního a na ni zapíše rozbalený/sbalený výstup.

V budoucnu by měly přibýt i pomocné funkce (případně být přidané do stávajících) pro úpravu zarovnání přímo na GPU. Již dříve jsem uvedl, že současný HW NVidie neumí prokládat výpočty s datovými přenosy tehdy, pokud v průběhu přenosu musí zarovnávat řádky na určitou hranici (typicky zarovnání textury). Zatím však toto není aktuální, neboť rychlost, kterou GPU dosahuje je dostatečná, i když se neprokládají přenosy s výpočty.

8.2.2 Unsharp mask

Naivní návrh unsharp mask, který by počítal rozostření zvlášť pro každý bod, je sice možný a funkční, ale podle předběžných měření, které jsem provedl, je natolik špatný (první výsledky dávaly sekundu, později se mi to podařilo srazit na 200 ms), že se ukázalo, že jde o úplně neakceptovatelné řešení. Důvodem pro tak špatný čas zřejmě nebylo ani tak to, že jsme pro každý bod museli přečíst kompletní okolí až po zadaný poloměr, protože to by podle propočtů stále nevysvětlovalo tak vysoký čas. Spíš jde o to, že daný kernel měl složitou řídící logiku. Navíc byl poměrně heterogenní, což výpočetní architektuře CUDA dělá určité problémy. Ukázalo se dokonce, že jakákoliv malá úprava kernelu, která by neměla mít valný vliv (např. vynechání přepínání bufferů v globální paměti) se na celkovém čase projevila až nelogicky mnoho. Každopádně příčina pomalosti se z kódu úplně dobře vysledovat nedala, protože jednotlivé části fungovaly relativně rychle, ale jako celek už ne. To je dle mého soudu trochu nevýhoda architektury CUDA, že je hodně dynamická, takže se může stát, že kód, který není pro danou

architekturu výhodný[3], neběží čas o řád delší, ale rovnou o řády dva nebo tři.

Proto druhá možnost implementace je využití známé vlastnosti konvoluce[4], a to její separovatelnosti na dva nezávislé směry (vertikální a horizontální). V důsledku toho se konvoluce provede ve dvou sousledných krocích, které mají jednodušší řídící logiku.

Jenom technická poznámka – nip2 používá jako parametr *Unsharp mask* poloměr rozostřovacího filtru. Obecně je zvykem používat využívat tento vztah mezi poloměrem filtru a směrodatnou odchylkou Gaussovy funkce:

$$\sigma = \frac{r}{3}$$

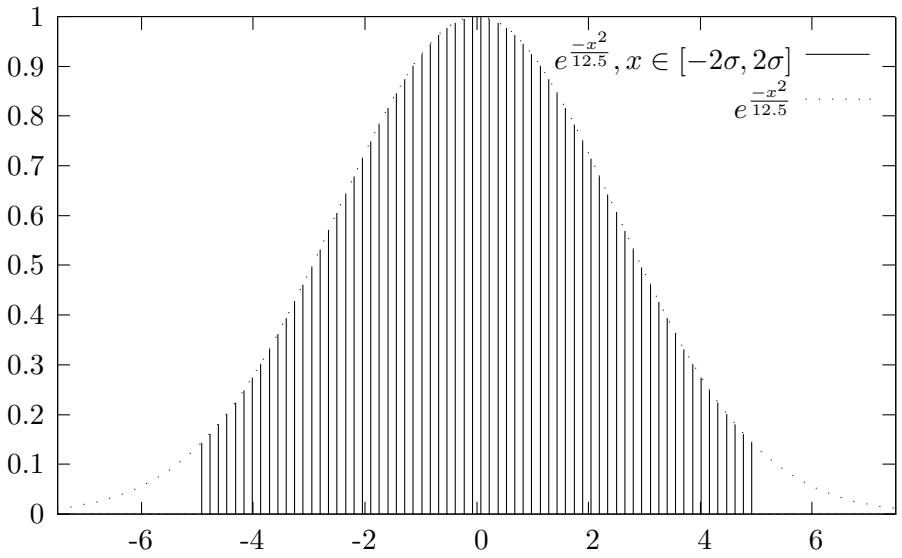
Vztah vychází z toho, že $erf(3) = 0,99998$[5], takže hodnoty v okolí $||N - x||_2 > 3\sigma$ jsou daleko za mezí významnosti. Nip2 oproti tomu využívá vztahu $\sigma = \frac{r}{2}$, tedy bereme okolí pouze 2σ, tedy $erf(2) = 0,99532$, což ukazuje obrázek 8.1. V prostoru se to dá představit také relativně jednoduše, a to tak, že si představíme kružnici, která odpovídá vrstevnicím 3σ a z ní vyřízneme čtverec, jehož všechny vrcholy leží zhruba na této kružnici. To pak znamená, že nejvíce excentričtí sousedé centrálního bodu mají potřebně malou úroveň významnosti, zatímco body ve čtvercovém okolí se stejnými souřadnicemi x nebo y jako zkoumaný bod mají tuto míru ještě relativně vysokou (2σ). Každopádně, toto je spíše detail, který však znamená to, že musíme výsledný bod škálovat hodnotou konstanty $\sum_{n \in neib(X)} w(n)$, protože tato suma dává normalizovanou hodnotu nižší než 1.

Pro grafické karty je nejvýhodnější přístupový vzor do paměti ten, že vlákna se stejnou y-ovou souřadnicí přistupují k sousedním paměťovým hodnotám (ideálně 32 bitovým). Dalším dobrým zvykem (ačkoliv dnes je již tato nutnost z výkonnostního hlediska hodně relaxovaná) je přistupovat n-tým threadem k n-té buňce paměti (u sdílené paměti je to bank). Tyto dvě skutečnosti determinují směr, kterým budeme separovaný filtr provádět, tedy

3. Avšak tím nemyslím bank konflikty a ostatní známé záležitosti, protože tyto věci jsem měl ošetřeny velmi dobře a alespoň podle mých zkušeností nebylo v kódu nic primárně nevhodného pro platformu CUDA.

4. Pro připomenutí, Unsharp masking zostřuje tak, že rozostří obraz pomocí Gaussova rozostření a ke zdrojovému obrázku přičte rozdíl mezi zdrojovým a rozostřeným obrázkem. Toto rozostření je vlastně jádrem celé úpravy a probíhá tak, že se obrázek konvolvuje s filtrem, hodnota jehož koeficientů se snižuje se vzdáleností od středu podle Gaussovy funkce.

5. Funkce $erf(n)$ je jednoparametrická chybová funkce Gaussova rozdělení, která říká, jaké množství dat spadá do intervalu $\mu \pm n\sigma$, kde μ je střední hodnota a σ směrodatná odchylka. Hodnota funkce zjevně nezávisí na odchylce ani střední hodnotě. V tomto kontextu to znamená to, že bereme v úvahu pouze ty sousedy, jejichž váha je větší než $n\sigma$.



Obrázek 8.1: Gaussovo rozostření – pokrytí 2σ

vertikálně.

Sdílená paměť má větší šířku pásma a výrazně nižší latenci než globální paměť. Toho se dá využít k omezení přístupu ke globální paměti na pouhé jedno čtení každého pixelu. Toho se docílí snadno tím, že blok vláken pracuje nad blokem dat, který si přednačte do sdílené paměti. Na výpočet výsledného bloku dat o velikosti $(n * m)$ je potřeba $(n * (m + 2 * r))$ vstupních hodnot, protože prvky u okraje bloku potřebují okolí velikosti r. Data, co jsou při okrajích potřeba navíc oproti $(n*m)$ hodnotám, se buď dají načíst z globální paměti, nebo lépe je předávat mezi jednotlivými iteracemi výpočtu, což je i přístup, který jsem zvolil.

Zbytek implementace úpravy je pak značně triviální – transponujeme obrázek rozostřený v jednom směru, provedeme rozostření i v druhém rozměru, transponujeme nazpět a přičteme ke zdrojovému obrázku vypočtenou masku, která se získá jako rozdíl rozostřeného obrázku a vstupního.

8.2.3 Zmenšení

Algoritmus, jehož implementace pro GPU je diametrálně odlišný od CPU, je algoritmus pro zmenšení, respektive zvětšení obrázku. Vzhledem k tomu, že v obou případech jde o interpolaci, tak budu nadále používat tento termín.

Interpolace je standardní matematický mechanismus získání hodnoty proměnné v bodě spojité domény v případě, že máme pouze diskrétní hodnoty na celém intervalu a pro interpolant obecně nemáme vzorek.

Existují tři základní algoritmy, které se používají pro interpolaci v grafice[6] – nejbližší soused, bilineární a bikubická interpolace.[7] Krátce řečeno, v pořadí, ve které jsem je vyjmenoval, kvalita a výpočetní náročnost stoupá.

Rozdíl GPU zpracování oproti CPU je takový, že grafické karty jsou vybaveny speciálními texturovými jednotkami pro rychlý výpočet interpolantu. Tyto hardwarové interpolace se provádějí bez výjimky bilineárně. (Tedy existuje i volba vyhledání nejbližšího souseda, ale tuto možnost asi málokdo využije.) Současný hardware NVidie zvládá interpolaci ve vysoké kvalitě, nicméně využívá pouze omezeného počtu interpolantů (256^8), což ovlivňuje kvalitu výsledku.

Jak již jsem napsal, přímá HW podpora pro bikubickou interpolaci neexistuje, nicméně to neznamená, že bychom byli odkázáni pouze na přímý výpočet stejně jako na CPU. Pro bikubickou interpolaci se dají rovněž využít schopnost texturovacích jednotek bilineárně interpolovat, a to s využitím algoritmu navrženého v [16].

Nicméně jsem se rozhodl implementovat bilineární interpolaci z toho důvodu, že dosahuje podle mě dostatečné kvality pro 2D obrázky a je rychlejší než bikubická interpolace implementovaná citovaným algoritmem.[9] Interpoluji přímo gama-komprimované hodnoty vstupního obrázku. Další interpolační algoritmy je možné samozřejmě přidat.

8.2.4 Ostatní úpravy

Všechny ostatní úpravy mají jednoduché schéma v tom, že je možné zpracovávat všechny body nezávisle. Optimalizace na výkon vychází ze znalostí architektury a předchozích zkušeností, kde se mi osvědčilo preferovat větší bloky, ale v menším počtu. Tento přístup skrývá drobný zádrhel, že velmi závisí na správném nasazení na jednotlivé multiprocesory GPU, jinak může dojít k hladovění některých multiprocesorů v situaci, kdy už nejsou žádné

6. Jiné obory často používají i jiné postupy, případně jejich modifikace – například ekonomie často využívá monotonní bikubickou interpolaci. Tato interpolace (resp. její obdoba je shodou okolností použita v jiné úpravě, kterou akceleruji, a to pro interpolaci hodnot křivky, která se používá pro tónování barev.
7. Nebudu blíže jednotlivé algoritmy popisovat, protože jsou asi všem se základním matematickým vzděláním jasné, eventuálně se k nim dají potřebné informace lehce dohledat.
8. dle vyjádření zaměstnance NVidie, viz `http://forums.nvidia.com/index.php?showtopic=48546`
9. Opírám se o měření výkonů implementací z NVidia SDK. Výsledky je možné nalézt přímo ve zdrojovém kódu „bicubicTexture“.

nevypočítané bloky, ale malý počet stále blokuje celé GPU potenciálně dlouhou dobu (jednotlivé bloky zpracovávají hodně dat). Tento problém se dá částečně obejít tím, že je počet bloků násobkem počtu multiprocesorů, což je přístup, který jsem použil. Ač se zdá, že není úplně ideální, protože ideální průběh závisí na konkrétním počtu multiprocesorů, není tomu úplně tak. Za prvé proto, že ten počet bloků není zase až tak malý a jedná se o malý násobek počtu multiprocesorů (5-10x). Navíc, vzhledem k tomu, že toto nasazení je pro kartu příhodné, tak je výhodnější i pro podmínky, kdy nasazení se úplně nestrefí do parametrů karty, než abychom spouštěli třeba thread pro každý pixel, což by bylo pomalejší pro všechna možná nasazení. Mimoto, používáme více streamů, takže je šance, že karta spustí souběžně jiný kernel, což vyváží i tuto drobnou neefektivitu.

8.3 Architektura

To, že je čas zpracování jednotlivých úprav dobrý, nemusí nic znamenat, pokud se toto nepodaří promítnout do výsledného produktu a celkový čas zpracování je mnohanásobně horší než suma provádění jednotlivých úprav. Toto riziko je poměrně významné, protože rychlost GPU pro úpravy tohoto typu je výborná, takže není moc prostoru na plýtvání časem.

8.3.1 Parsování vstupu

Nejdříve ovšem krátce nastíním problémy a jejich řešení při zpracování vstupního WS souboru.

Nepříliš výhodné je, že nemám globální přehled o obsahu tohoto XML souboru. To je rozdíl oproti nip2, který jej vytváří, takže dokáže interpretovat všechny položky tohoto souboru. Já potřebuji oproti tomu pouze malý výřez jeho obsahu, nicméně bez těch znalostí je složitější jej získat. Jako řešení jsem použil různé heuristiky, které vycházejí ze způsobu, kterým zhruba nip2 hodnoty ukládá. Například se jedná o to, že každý atribut úpravy má právě jednu hodnotu. To je sice svým způsobem triviální, nicméně nip2 ukládá skoro vždy ke každé vlastnosti úpravy značky několika typů (iText, Slider, Option) a není jasné, která z těchto značek danou informaci nese, protože v různých kontextech se může jednat o různé typy hodnot. Nip2 tuto informaci má, nicméně při parsování, které provádím já, nemám žádné apriorní informace o úpravě a typech jejich atributů. Proto při čtení předpokládám, že u každého atributu úpravy je nejvýše jedna hodnot a obsah prázdných XML značek zahazuji.

Druhým problematickým místem je analýza příkazů, kterou jsem dis-

```
EXPR          ->  IDENT_IMAGE OP ARGS |
                  IDENT_FUNCTION ARGS
IDENT_IMAGE   ->  [a-zA-Z_][a-zA-Z0-9_]*
OP            ->  OPSYM | OPSYM OPSYM |
                  OPSYM OPSYM OPSYM
OPSYM         ->  ! | @ | # | $ | % |^| & | * | _
IDENT_FUNC    ->  [a-zA-Z_][a-zA-Z0-9_]* |
                  IDENT_FUNC ':' IDENT_FUNC
ARGS          ->  IDENT_IMAGE ARGS |
                  IDENT_FUNCTION ARGS |
                  INTEGER ARGS | FLOAT ARGS |
                  STRING ARGS | ϵ
INTEGER       ->  [0-9][0-9]*
FLOAT         ->  INTEGER . INTEGER
STRING        ->  " [a-Z0-9_]* "
```

Obrázek 8.2: Používaná gramatika pro rozpoznávání příkazů

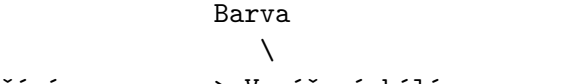
kutoval v sekci 8.1.1. Tím, že neimplementuji kompletní stack nip2, se zužuji pouze na jednoduché příkazy. Jinak by bylo nutné implementovat interpret pro kompletní jazyk nipu, což je vzhledem k komplexnosti úprav, které urychluje dost zbytečné. Pro parsování příkazů používám ekvivalent gramatiky na obr 8.2. Tato gramatika je zjevně typu 0, nicméně sémantika je přehlednější než odpovídající regulární gramatika. Důkaz toho, že existuje ekvivalentní regulární gramatika je, že mám automat, který ji rozpoznává.

8.3.2 Analýza vstupu

Ačkoliv beru WS soubor z větší části bez interpretace jeho obsahu, přece se nedá předběžné analýze vyhnout úplně. Potřebuji nad vstupem provést následující operace:

- doplnění skutečných závislostí na základě jmen, které nip2 přiřazuje jednotlivým mezivýsledkům a na která se odvolávají následující úpravy
- odstranění nepotřebných transformací

První bod je poměrně zjevný – potřebujeme znát skutečné závislosti mezi výsledky, aby bylo možné správně předávat vstupy úpravám a přebírat jejich výstupy. Ačkoliv se jednotlivé úpravy spouští v pořadí, ve kterém jsou zadané, obecně mezi úpravami mohou být složitější závislosti, než že každá



Obrázek 8.3: Závislosti jednotlivých úprav

úprava bere výsledek předchozí úpravy jako vstup. Tím nechci zbytečně řešení komplikovat nějakým případem, který málokdy nastane, ale taková situace skutečně nastává i v případě typického WS souboru, který zpracovávám. Problematickou situaci ukazuji na schématu 8.3.

Ve WS souboru je *Barva* zadaná hned před zadáním *Vyvážení bílé* a vyvážení má tak vlastně dva vstupy – barvu a předchozí obrázek. Ačkoliv nejde o složitý případ, toto zadání už vynucuje obecné řešení.

Implementace detekce závislostí je přímočará – bereme jednotlivé úpravy v pořadí, ve kterém jsou zadané, a postupně doplňujeme do asociativní mapy k jednotlivým proměnným referenci na úpravu, jejímž je výsledkem. Naopak pokud úprava očekává vstup (nebo více) zadaný symbolickým jménem, tak už musíme mít tuto položku definovanou v mapě, takže doplníme skutečnou závislost.

Druhý bod, který jsem napsal nevypadá už tak zjevně, protože teoreticky by nemusely žádné nepotřebné úpravy ve WS souboru existovat. I v opačném případě by nemuselo zpracování úprav navíc rychlostně vadit, pokud by jich nebylo mnoho. Ani jeden z těchto dvou předpokladů neplatí – první proto, že nip2 vyhodnocuje výrazy líně, takže tam nějaké úpravy navíc nevadí. Kromě toho je jejich odstranění z nip2 souboru trochu krkolomné. Druhý důvod je, že úpravy vlastně nemusí být úplně zbytečné – to je případ současné implementace vyvážení bílé[10], která předpokládá ruční zadání hraniční hodnoty barvy, od které je vše považováno za bílou. V nipu tomu ovšem předchází výběr oblasti domnělé bílé a vypočítání barevného průměru této oblasti. Oba kroky mají odraz v pracovním souboru, ale výsledek úprav se předává ručně. To se může zdát divné, ale je to z toho důvodu, že jeden WS soubor náleží k celé množině dlaždic a „bílá“ barva se zjišťuje z jednoho konkrétního. V případě, že by se to tento proces manuálně nepřetrhl, tak by to znamenalo, že v daném regionu je v jiných dlaždicích už úplně něco jiného, než bílá oblast.

Tím, že tyto operace, ač nejsou zbytečné, nejsou zpracovány, je zbytečné je implementovat, respektive krkolomné věnovat energii jejich přeskočení v řetězci úprav.

10. Dá se zjistit buď přímo z WS souboru, nebo při jeho otevření v nip2.

Eliminace nepotřebných pravidel provedu tak, že vycházím od výstupního obrázku (stoku), na který spustím standardní DFS po směru závislostí a nepoužitá pravidla vyřadím.

8.3.3 Rozhraní pro úpravy

Vlastní GPU část frameworku pro spouštění úprav je optimalizovaná takřka výhradně na co nejvyšší propustnost, k čemuž využívá těchto postupů, které nyní uvedu.

K maximalizaci propustnosti je klíčové využití CUDA streamů, a to hned z několika důvodů:

překrývané přenosy s výpočty Překryv je jeden ze základních přínosů streamů, který umožňuje vykrýt minimum z časů přenosu dat na kartu (v součtu obou směrů) a doby výpočtu. Tento přínos není zdaleka zanedbatelný hlavně tehdy, když jsou tyto dva časy zhruba stejné, což je i případ našich výsledků (viz kapitola 9).

souběžné spouštění kernelů Tato vlastnost novějších karet je trochu méně zajímavá, protože spíše cílím na to, aby individuální kernel kartu optimálně využil, což v naprosté většině případů znamená její plné vytížení. Nicméně, je možné, že tato vlastnost přináší i některé další výhody, jako třeba amortizaci latence spouštění kernelů, takže je možné, že i z tohoto se dá těžit. Nicméně, toto je poměrně složité podchytit nějakým měřením, navíc když CUDA profiler jednotlivé streamy serializuje (verze 3.2).

Další důvody ani tak nesouvisí s vlastnostmi CUDA, ale s optimalizacemi workflow, které využití streamů umožňuje dosáhnout.

Důležité je, že volání CUDA rutin spojená se streamy jsou neblokující z hlediska volajícího hostitelského vlákna, což umožňuje uvolnit řídící vlákno pro jinou práci, která by se jinak musela dělat v jiném vlákně. To by bylo značně nepohodlné proto, že by bylo nejen potřeba tato vlákna synchronizovat (což je ovšem stále potřeba pro ostatní vlákna), ale hlavně by bylo řízení rozdělené, což by ztěžovalo zpětnou vazbu, která propaguje rychlost nejpomalejšího prvku v řetězci zpracování (vstup, /dekódování,/ zpracování, výstup).

Rovněž bych předpokládal, že streamové zpracování bude o něco rychlejší proto, že uložíme do fronty požadavků všechna volání najednou a přesné spouštění si pak už řídí API samo. Přinejmenším může být rozdíl tehdy, kdy je CPU hostitelského systému zahlceno a stihne zadat daný požadavek až

tehdy, kdy bude přiděleno volajícímu vláknu další časové kvantum poté, co se ukončí předchozí volání. Navíc je to i s rizikem, že nestihne za toto časové kvantum požadavek odeslat, jestliže provádí mezi CUDA voláními nějaké další výpočty.

Pro jednotlivé grafické operace to pak znamená především to, že je potřeba, aby uměly zařadit kernely do patřičného streamu a pokud možno co nejrychleji vrátit volání nazpět volající funkci.

Správa GPU paměti

Jednou z dalších důležitých záležitostí je správa paměti na grafické kartě.

Ponejprve není vhodné nechávat moduly grafických úprav spravovat paměť ve vlastní režii bez toho, abychom neudělali určitou distribuci informací o přidělené paměti mezi navazujícími úpravami. Navíc je potřeba vzít v úvahu, že v situaci, kdy používáme streamy je dobré mít paměť dopředu alokovanou před spuštěním fronty úprav na dlaždici.

Alokaci paměti pro výstup a případné pomocné buffery si v konečném řešení sice každá úprava řídí sama, ale s určitými omezeními. Jednak k tomu musí využívat funkce poskytované frameworkem a za druhé je alokace oddělena od výpočtů a probíhá předem. Konkrétní úprava musí poskytnout adresu a metadata bufferu, do kterého uloží výsledek. Při určování velikosti alokace a metadat výsledku vychází naopak z výsledků předaných úpravami, na kterých daná operace závisí.

Paměť grafické karty je typicky 1 GiB, což je sice velká kapacita a zdánlivě dostatečná s přihlédnutím k velikosti obrázků a tomu, že zpracovávám jednotlivé obrázky individuálně. Ale toto zdání není opodstatněné – 12 bitové dlaždice o rozměru 5 Mpix zabírají zhruba 40 MB. Při předpokladu 7 bufferů pro výsledky a například 4 pomocných bufferů velikosti obrázku (zhruba vycházeje z typického WS souboru, který je zpracováván), dostaneme 440 MB jenom na zpracování jedné dlaždice. Když navíc máme 2 streamy, tak se skoro blížíme kapacitě paměti karty.

Proto jsem se rozhodl implementovat mechanismus, kterým umožním již nepotřebnou paměť vracet zpět a znovuvyužít ji v dalším zpracování. Využívám toho, že nenechávám jednotlivé úpravy alokovat paměť přímo, ale pomocí funkcí frameworku. Myšlenka je ta, že přiděluji paměť z virtuální haldy, kterou podle potřeby rozšířím, když už není dostatek použitelného volného místa. Když už paměťové umístění není potřeba tak jej uvolním. Protože nemůžu přidělovat ukazatel do reálné paměti do té doby, než je alokovaná, poskytnu volajícím úpravám objekt mající zástupnou adresu s ofsetem od začátku haldy a bázi doplním zpětně.

Pro přidělování paměti používám algoritmu first-fit a probíhá v pořadí, ve kterém probíhají úpravy. Paměť uvolňuji ihned, jakmile není potřeba, přičemž každé uvolnění ihned spustí úplné scelení volného místa.

Další vylepšení je to, že nechám jednotlivé úpravy alokovat dočasné buffery (pro potřebu úpravy) separátní funkcí a tyto uvolním ihned po skončení úpravy. Celkově alokaci ilustruje pseudokód 8.1 (syntaxe je částečně inspirována Pythonem).

```
memoryPool pool

foreach f in Filters:
    foreach prev in Filters
        if prev < f:
            if resultNoLongerNeeded(prev, f):
                pool.dispose(prev)
    pool.open(f)
    f.allocResult(pool)
    pool.close(f)
    pool.openTmp()
    f.allocAditional(pool)
    pool.closeTmp()
    pool.disposeTmp()

pool.allocate()
```

Zdrojový kód 8.1: Pseudokód alokace paměti

Pak už jsou potřeba pouze dvě věci: obstarat si pitch řádku používaný pro voláním cudaMalloc2D, abychom byli schopni imitovat beze zbytku chování nativních funkcí i tehdy, když se paměť pro vypočítanou velikost haldy alokuje až zpětně. Tato hranice zarovnání se zjistí poměrně jednoduše, protože se jedná o hranici zarovnání textur grafické karty. Druhý drobný problém je, že některé zpracovávané úpravy nemají vlastně implementaci. To je příklad ořezání na 8 bitů, které ignoruji a řídící kód této úpravy pouze předá buffer dál, přičemž pomocí jiného volání distribuuje v metadatech sémantiku své úpravy (pro pozdější zpracování při výstupu). Uvolnění bufferu úpravy předchozí úpravy by proto nastalo vlastně předčasně, protože tato pseudoúprava buffer nealokovala, pouze podědila. Řešení je takové, že implementuji dědění referencí pomocí referenčního čítače a k uvolnění dojde až poté, co jeho hodnota klesne na nulu.

Dodatečné optimalizace

Částečnou pozornost si zaslouží i optimalizace mezi jednotlivými úpravami z globálního pohledu. Tento úkol je v obecnosti náročný, ale existují situace, kdy je řešitelný s malými náklady, a to tehdy, když jsou za sebou dvě jednoduché operace využívající vyhledávací tabulky na stejném barevném prostoru. V takovém případě je možné kontrahovat několik takových úprav do jediné, pokud za sebou bezprostředně následují.

8.3.4 Diskový vstup/výstup, dekomprese

Z výsledků, které jsem naměřil (kapitola 9) vyplývá, že vlastní zpracování je hodně rychlé a největší zpomalení nadále způsobují dva faktory – rychlost vstupu/výstupu a doba dekomprese. Vstupní TIFFy jsou komprimované algoritmen LZW, což je komprese obecně nevhodná pro nasazení na GPU, protože se jedná o slovníkovou kompresi, ve které existují extenzivní datové závislosti mezi daty.

Dekompresi je možné na CPU úspěšně zvládnout tehdy, když se použije více vláken pro dekompresi.

Zvládnout pomalé IO je trochu problematičtější. Hlavním problémem není ani tak rychlost sekvenčního čtení z rotačního disku (pole), spíše náhodné přístupy k disku, které čtení relativně malých souborů způsobuje.

Vlastnosti, které se dají částečně využít, je spojování požadavků a výtah. Myšlenka je ta, že zadání mnoha IO požadavků zároveň (od jednotek po desítky) umožní IO subsystému operačního systému respektive řadiči disku požadavky zpracovávat efektivněji s ohledem na celkový čas. Samozřejmě za cenu zpomalení absolutního času individuálních požadavků. Ukazuje se, že toto sice vliv mít může, někdy i docela výrazný, ovšem ne vždy v pozitivním směru. Důležité je totiž to, že výraznou roli hraje plánovač operačního systému, fyzické dispozice disku a v neposlední řadě i souborový systém (zejména nastavení žurnálu, ale i jiné parametry).

Z těchto důvodů je obtížné dopředu stanovovat parametry, za kterých bude IO dostatečně rychlé. Faktem ale je, že v určitých situacích lze pozorovat nárůst výkonu tehdy, když na IO subsystém zatlačíme náporem požadavků (kapitola 9).

Ještě zbývá zmínit, jak dokážeme vytvořit dostatečnou zátěž na IO. Ideálním řešením by bylo využít asynchronní IO. Bohužel, jak se píše v [10], situace ohledně asynchronního IO je v Linuxu taková, že není v současnosti použitelné. Takže jediný způsob, jak obejít blokující vstup/výstupní operace je zadávat každý požadavek z jiného vlákna. Proti předpokladům by toto ne-

mělo způsobit žádný výrazný nápor na procesor systému proto, že taková IO vlákna zvládnou prakticky v prvním časovém kvantu, které jim je přiděleno, odeslat IO požadavek a pak už je vše v režii operačního systému.

Na druhou stranu může nastat efekt naprosto opačný, že bude disk natolik stresovaný náporem IO požadavků, že to povede ke snížení celkové propustnosti.

8.3.5 Nasazení

Nasazení jednotlivých procesních jednotek vypadá následovně – jediné řídící vlákno obsluhuje všechny ostatní jednotky. To jsou jednak vlákna pro čtení/-dekomprimování obrázků, které naplňují jednu sdílenou frontu. Z této fronty pak řídící vlákno předává obrázky střídavě více CUDA streamům. Jakmile je nějaký výstup z grafické karty, hlavní vlákno jej převezme a předá vhodnému výstupnímu vláknu, které právě není zaneprázdněno probíhajícím výstupem.

Kapitola 9

Výsledky

Abych ověřil úspěšnost implementace, provedl jsem sadu měření. Nejprve měřím rychlost zpracování jediného obrázku, a to pouze trvání jednotlivých grafických úprav na CPU respektive GPU. Čas je tedy očištěný od dalších vlivů jako je IO a dekomprese. Poté co tyto výsledky zhodnotím, měřím celkovou dobu provádění pro reprezentativní sadu obrázků. Toto měření probíhá zhruba za stejných podmínek, ve kterých by měl být výsledný produkt nasazený.

Pro všechny následující výpočty byla použita tato konfigurace: grafická karta NVida GTX 480, 6-jádrový procesor Intel® Core™ i7 X980 s frekvencí 3,33 GHz[1]. Vstupem byl dvanáctibitový RGB obrázek o velikosti 2680x1944, tedy nekomprimovaný zhruba o velikosti 31,3 MB.

9.1 GPU zpracování

Dobu zpracování jednotlivých úprav ilustruje tabulka 9.1. Při kompilaci pro GPU nebyly použity žádné speciální přepínače (například '–fast-math'). Výsledky vycházejí z pěti měření na jednom konkrétním obrázku. Úpravy se ne vždy mapují jedna ku jedná na příslušné kernely. Výsledky jednotlivých kernelů je možné nalézt v příloze C.

9.1.1 Analýza

Výsledky jsou na první pohled velmi dobré, protože se rychlost zpracování pomalu blíží k době, jakou potřebujeme data nahrát na kartu, což je horním mez možného zrychlení. U starších generací karet, protože umožňují současně kopírování jen jedním směrem (zároveň s výpočty), se musí uvažovat součet obou přenosů a tomuto času se už výsledky blíží těsně. (Nicméně na starších kartách by provádění bylo zjevně pomalejší, zatímco doba trvání přenosu přes PCIe by se významně měnit neměla.)

1. Procesor dokáže zpracovávat paralelně dvě vlákna na jednom jádře (hyperthreading).

úprava	trvání (μs)
Unsharp mask	4921.8
Saturace	1939,6
Vyvážení bílé	574,3
Gama korekce + křivky	569,1
Zmenšení	378,7
přenos na kartu	5098,2
přenos z karty	1770,1
celkem (bez přenosů)	9491,2

Tabulka 9.1: Délky trvání jednotlivých úprav

Teoreticky existuje prostor ke zlepšování časů, ale to by bylo zřejmě zbytečné s ohledem k rychlosti IO, které, jak ukáži, je v lepším případě o řád pomalejší než doba běhu na GPU.

Navíc dodatečné urychlení by asi moc významné nebylo – na příkladu kernelu využívajícího vyhledávací tabulku bych ukázal, že má propustnost bezmála 145 GB/s[2], což je 82% udávané maximální šířky pásma NVidie GTX 480 (177,4 GB/s).

Pro 8-bitové obrázky trvá celkové zpracování zhruba stejně dlouho jako pro 12-bitové. Důvod je ten, že časově nejnáročnější kernely nejsou závislé ani tak na velikosti obrázku v bajtech, jako na jeho rozměrech, protože zde je limitující výpočetní výkon karty. Mimoto některé kernely unsharp mask pracují pro obojí s floatovými hodnotami odpovídající luminozitě prostoru L*a*b*. U těch kernelů, jejichž čas se odvíjí od rychlosti paměti karty dosahují 8-bitové obrázky lepších časů, ale absolutní čas těchto úprav není v celkové sumě natolik významný.

9.2 Zpracování pomocí nip2 na CPU

Současné zpracování prostřednictvím nip2 je realizováno tak, že se perlovému skriptu zadá WS soubor, vstupní a výstupní adresář. Tento skript se poté postará o vytvoření jednotlivých procesů programu nip2 pro každý individuální obrázek. Množství souběžných procesů je specifikováno v textu skriptu.

Bohužel není možné měřit čistý čas, který nip2 stráví voláními funkcí knihovny VIPS, protože nip2 toto neumožňuje. Je možné změřit pouze hrubý

2. 42MB obrázek (nyní už 4 kanálový) zpracujeme za 574 μs a k paměti přistupujeme 2-krát

velikost obrázku (px)	střední hodnota (s)	směrodatná odchylka
2580x1944	6,367	0,021
1x1	3,796	0,016
rozdíl	2,571	0,025

Tabulka 9.2: Délka trvání úprav na procesoru (nip2)

čas, který spotřebuje nip2 včetně vstupu/výstupu a dekomprese TIFFu a na základě tohoto času a znalosti, jak dlouhou trvá IO a dekomprese dělat kvalifikované závěry. Další vliv, který je potřeba odstínit je ten, že se spouští na každý soubor samostatný proces. Samo spuštění procesu určitě nějakou dobu trvá, navíc se musí při každém spuštění znovu parsovat WS-soubor a připravit celé prostředí na zpracování jediného obrázku.

Nakonec jsem se rozhodl použít pro měření následující postup, který by měl dávat zhruba správné výsledky:

- Vytvořím prázdný obrázek (například 1x1 pixel) a změřím dobu běhu procesu pro tento obrázek a příslušný WS soubor.
- Dekomprimuji vzorový obrázek, jehož dobu zpracování jsem měřil v předchozí sekci. Zajistím a ověřím že je v IO cache operačního systému. Poté naměřím dobu zpracování tohoto souboru.
- Výsledný očištěný čas dostanu tak, že od délky trvání ve 2. bodě odečtu výsledek 1. bodu.

Z rozdílu časů, který uvádím v tabulce 9.2 vyplývá, že čistý čas zpracování na tomto jednom procesoru je lehce nad 2,5 vteřiny, což je více než 250x času, který na dosahuji na GPU. Tento rozdíl ovšem nelze přičítat pouze tomu, že je procesor pomalý. Faktem je to, že architektura VIPS je trochu odlišná od toho, jak jsem jednotlivé operace implementoval já. Ty se ve VIPS sestávají z atomičtějších kroků, k zajištění větší modularity. Na druhou stranu řešení na GPU, které prezentuji, bere jednotlivé úpravy jako celý blok, což dále umožňuje některé další optimalizace a přímočařejší implementaci, ovšem za cenu ztráty obecnosti a znovupoužitelnosti dílčích programových jednotek.

9.3 Celkové výsledky

Nyní se pokusím porovnat výsledky kompletního zpracování celé množiny obrázků od načtení z disku až po zápis výsledků. Referenční množinou je soubor obrázků označený „zoon-vulvitis-5802-11-60x-he“. Celkem jde o 12045

počet vláken	komprese	výstup	střední hodnota (m)	směrodatná odchyka
1	-	disk	40,04	6,42
4	-	-	19,26	0,14
8	-	-	17,13	0,09
4	LZW	-	29,29	0,32
8	LZW	-	17,05	0,16
4	LZW	disk	32,36	0,32
8	LZW	disk	27,97	2,20
12	LZW	disk	26,91	1,19
16	LZW	disk	26,35	0,04

Tabulka 9.3: Zpracování množiny 2 – lokální disk, 1 výstupní vlákno, gpubip

dlaždic o celkové velikosti 325 GB, výstup zabírá 65 GB. Jsou komprimované algoritmem LZW[3], šířka pásma jednotlivých kanálů je 16 bitů, významných (barevná hloubka) je jen 12 nejnižších. Rozměr každé dlaždice je 2580x1944.

Druhou množinou je pak soubor „rhabdomyosa-alv-po-chemo-hab-3022-05-60x-he“ s obrázky o stejném rozlišení jako má předchozí sada, nicméně nyní pouze 8-bitovými. Celkově je jich 24719 o celkové kapacitě 279 GB. Také WS-soubor je jiný (v podstatě podmnožina úprav z prvního).

Pro vstup je použité diskové pole Thor připojené vzdáleně přes NFS 10 GE spojením. Pole zvládá rychlost čtení spojitého souboru zhruba 720 MB/s, v případě náhodných přístupů při souborech velikosti dlaždic, které se zpracovávají, poklesne rychlost zhruba na polovinu. Zápis výstupu jsem testoval na různá zařízení, proto vždy uvedu, kam probíhal. IO scheduler systému je deadline. Používá se běžná CPU/IO priorita procesů bez dalších speciálních nastavení. Kde je to relevantní, je vypnutý odkládací prostor (viz dále). Dále se všechny výstupní soubory vytváří až při zápisu. Přepis má na výkon vliv i tehdy, když výstupní soubory otevíráme s **O_TRUNC**. Někdy jde o pozitivní vliv, jindy negativní, každopádně k určitému ovlivnění dojde.

První měření bylo pro jednovláknový výstup a různé variace počtu dekódovacích vláken a volby, zda-li jsou obrázky komprimované nebo ne. Výstupní zařízení byl buď lokální rotační disk nebo byl výstup vypnutý. Pro toto předběžné měření byla použita druhá sada.[4]

Tabulky 9.3 a 9.4 prezentují výsledky, které jsem naměřil. Z výsledků

3. Jako poznámku bych rád uvedl, že dekomprese obrázku z této množiny trvá zhruba 350 ms.
4. Pro přehlednost budu tuto množinu označovat jako „množina 2“.

počet procesů	komprese	výstup	trvání (m)
6	LZW	disk	442,32
12	LZW	disk	207,68

Tabulka 9.4: Zpracování množiny 2 – lokální disk, nip2

počet procesů	komprese	výstup	trvání (m)
6	LZW	disk	274,65

Tabulka 9.5: Zpracování množiny 1 – pole, nip2

(gpubip) lze vyvodit několik závěrů:

1. Komprese při dostatečném počtu vláken (a samozřejmě také jader procesoru) tolik nevadí. Záleží pouze na rychlosti čtení, jak rychle jsme schopni vyčítat z úložiště. Zde je to u nejrychlejších časů zhruba 350 MB/s. Na dekompresi tohoto toku stačí zhruba 6-8 vláken. Pokud by se podařilo číst rychleji, tak by bylo potřeba ekvivalentně více procesorů/vláken nebo nekomprimovaná data. Nicméně, rychlost čtení byla limitní pouze tehdy, pokud jsem zakomentoval výstup.

2. Jako limitní se ukazuje zápis na úložiště. Zjevně zde není problém s maximální (sekvenční) propustností disku, protože příslušný lokální disk[5] je schopen zvládnout až 110 MB/s při zápisu dostatečně velkého spojitého souboru. Problémem jsou spíše malé diskové operace – výsledné soubory mají něco přes 5 MB, ale zato je potřeba jich v krátkém časovém úseku zapisovat mnoho. Na druhou stranu, diskové pole, ze kterého čteme, je schopné dodávat vyšší počet souborů za časový úsek (triviálně, protože se zapisuje stejný počet souborů, jako se čte), a to o větší velikosti.

V dalších měřeních a úpravách jsem se zaměřil právě na výstup. Pro měření je použita první množina. Tabulku 9.5 s výsledky zpracování nipem dávám spíše jen pro úplnost, protože mě nyní zajímá právě výstup.

V následujících měřeních (gpubip) obměňují různé parametry – čtecí (dekódovací) vlákna, zapisovací vlákna. Jako další měřené úložiště používám další pole, označované jako pole2. Důležité je, že data se čtou vždy z pole1. Mimoto jsem v průběhu dodělal novou frontu dekódovaných vstupů, která předtím závisela na počtu dekódovacích vláken.

5. souborový systém ext4, žurnálovaná pouze metadata (ordered)

počet dekódovacích vláken	počet výstupních vláken	délka fronty	výstup	trvání (m)	směrodatná odchylka
16	1	16	-	13,39	0,12
16	1	16	pole1	37,39	1,44
10	5	10	pole1	25,51	1,45
16	4	16	pole1	20,45	1,63
16	8	16	pole1	20,33	0,86
16	10	16	pole1	20,37	1,75
16	1	16	pole2	24,08	0,74
16	8	16	pole2	17,58	0,39
16	1	16	disk	28,10	3,66
16	3	30	disk	34,34	4,77
16	10	16	disk	34,91	6,93
16	50	50	pole1	19,91	1,05
30	30	100	pole1	19,86	0,22
60	100	80	pole1	19,73	0,87
8	120	60	pole2	18,11	0,15
16	96	48	pole2	15,96	0,52

Tabulka 9.6: Zpracování množiny 1 – gpubip

Na výsledcích je vidět, že jsem zdaleka nezkoušel všechny kombinace, což samozřejmě ani není možné. Nicméně zdánlivě chybí i zajímavé kombinace. Důvod je ten, že jsem na základě předběžných měření vyřadil takové kombinace, které nepřinášely žádný pokrok, abych mohl otestovat jiné možnosti. Zaznamenané výsledky jsou pak ty, které se mi zdály významné – buď s dobrým časem, nebo reprezentují nějakou typickou skupinu.

Dále jsem kromě základních měření pro přiměřený počet čtecích a zapisovacích vláken zkoušel i jiné možnosti. Jednak zvýšit počet zapisovacích vláken a také zvětšením interních bufferů, čehož se docílí přidáním variabilní fronty, která drží přečtené a dekódované dlaždice.

Obě dvě tyto možnosti se zdály podle první sondy nadějné z hlediska celkového času. Smyslem zvýšení počtu výstupních vláken je vlastně zatlačit na IO subsystém operačního systému a disku, aby spojoval požadavky na IO a případně je vyřizoval výtahem. Může být nutné vypnout swap – neznám sice klíč, podle kterého se operační systém rozhodne swapovat a, i když to není nějaké extenzivní swapování, spíše jen sem tam nějaká ta stránka, tak to mírně provádění zbrzdí.

Prvním problémem je, že v systému může docházet k oscilacím tehdy, kdy je zápis pomalejší než zbylé součásti. To zapříčiní, že se nepohybujeme na rychosti nejpomalejšího článku, tedy zápisu, ale ještě na nižší úrovni. Protože je nejpomalejší prvek na konci, systém se v určitém okamžiku zahltí. Informace o zahlcení se propaguje zpět a pozastaví se i ostatní součásti. V situaci, kdy je opět výstupní zařízení schopné zápisu, se provedou operace na datech, která jsou k dispozici. Nicméně pokud je připravených dat málo, tak jsou tato dekomprimovaná data zpracovaná za krátký okamžik. Ovšem dekomprese dalších dat trvá řádově více, takže v tomto okamžiku čekáme zbytečně na dekompresi, která se mohla provést už dříve. Toto je nejjednodušší případ, nicméně k některým oscilacím může docházet i z jiných příčin, i ty by však fronta mohla rovněž mírnit.

Proto jsem přidal variabilní frontu, která umožní mít větší počet rozpracovaných souborů[6]. Ani to samozřejmě neodstraní oscilace zcela, už jen proto, že nemáme dostatek volné paměti – 50 dlaždic zabírá 1 GB RAM, přičemž GPU má tento počet spotřebovaný za 500 ms, což je hodně málo, protože jsem vypozoroval (přesně to naměřit je obtížné), že perioda některých oscilací bývá od jednotek sekund až po desítky sekund eventuálně i trochu víc. Navíc volnou paměť potřebujeme i pro vlákna – při plném zatížení, které je žádoucí, bude držet každé vlákno právě jeden obrázek.

6. Předtím byla v podstatě fronta o délce takové, jaký byl počet dekódovacích vláken. Nicméně přesná implementace byla trošku odlišná.

Výsledky v tabulce 9.6 však tyto hypotézy ne zcela potvrzují. Pro rozumný počet výstupních vláken se dá zrychlení dosáhnout, nicméně pro větší počet už není další zrychlení průkazné, což je v protikladu k tomu, co předběžná sonda naznačovala.

Je vidět, že vícevláknový zápis se u lokálnímu disku projevuje spíše negativně. Oproti tomu u polí jsou časy dosahované vícevláknovými zápisy lepší, ovšem existuje hranice, od které se čas už výrazněji nezlepšuje K určitému zrychlení zřejmě dochází, ale to není podle měření průkazné.

Každopádně bych tyto myšlenky úplně nezavrhoval, protože přinejmenším naměřené časy nejsou horší, dokonce jsou o pár procent lepší než ty s menším počtem vláken. Další skutečností je, že lepší časy z těch, co jsem naměřil, se už blíží rychlosti čtení z pole, takže je dost dobře možné, že už se čas v podstatě dostává na rychlost vstupu s nějakou dodatečnou režií.[7] Je také otázkou, jaké by bylo chování dalších úložišť s jinými parametry.

Ovšem obsahem této práce není primárně optimalizace IO a s přihlédnutím k tomu, že jsou celkové výsledky i přes zpomalení, které vstup a výstup způsobí, poměrně dobré, tak se podle mě jedná o dostatečné zrychlení. Další maximalizace propustnosti IO by podle mého názoru vyžadovala komplexnější přístup, a to ideálně i ve vztahu k dalším procesům, které se na patologických snímcích provádí (viz kapitola 2). Jeden z možných přístupů k dalšímu zlepšení uvádím v 10.1.

7. Na druhou stranu další měření v jinou dobu naměřila hodnoty zhruba o 2 minuty nižší než zatím nejrychlejší čas (16/96/48/pole2). Tento nový čas už je stejně dobrý, jako pouhé čtení z vstupního pole. Zřejmě dochází v různých okamžicích k určitým odchylkám, ty mohou být způsobené například aktuální zátěží pole.

Kapitola 10

Závěr

Potřebné úpravy se podařilo úspěšně implementovat, přičemž časy zpracování úprav na GPU se ukazují jako velmi dobré. Tím pádem je možné zpracovávat i velké soubory dat na prakticky jediném počítači s vyhovující GPU, počtem procesorů pro dekompresi a ideálně i dostatečnou velikostí RAM v relativně slušném čase. To je pokrok oproti původnímu procesorovému zpracování, které na jednom uzlu trvalo nemalý počet hodin a i při nasazení na jednotky uzlů se čas sníží zhruba na hodnotu 5x vyšší než při zpracování na GPU. Při velkém počtu výpočetních uzlů se dá předpokládat, že by se výsledky přiblížily rychlosti GPU blíže, ale nakolik dosavadní postup škáluje pro takové počty procesorů je otázkou.

Na druhou stranu se ukázalo, že už začíná být velmi limitní propustnost vstupu a výstupu pro jednotlivé dlaždice.[1] IO výkon se dá určitými postupy za určitých okolností vylepšit, jak jsem ukázal v předchozí kapitole, ovšem zřejmě zůstane limitní i nadále – grafická karta zvládne zpracovat 100 dlaždic o velikosti 30 MB za sekundu, což dává agregátní šířku pásma 3 GB/s. Ta by byla potřeba i na vstupu, aby se plně využily možnosti grafické karty, což je stěží dostupné se současnými perzistentními úložišti. Na druhou stranu je možné ještě zapracovat na problémech, které přináší náhodný přístup na disk, protože tento přístup neumožňuje dostat se ani rychlost, kterou jsou rotační disky schopny dát k dispozici sekvenčně. To nemusel by být takový problém v případě SSD, proto si myslím, že jejich nasazením by se dalo dosáhnout dalšího významného zrychlení. Pole složená z takových disků by navíc mohla dát daleko větší šířku pásma než pole z rotačních disků.

10.1 Budoucí práce

Pokud bych měl v krátkosti pojednat o případných navazujících nebo souvisejících pracích, které by eventuálně mohly navázat na moji, tak vidím velké

1. Oproti původnímu nipovému zpracování, u kterého byla s přehledem nejvíce omezující rychost CPU (při rozumném počtu procesorů).

množství možností.

Práce, které by blíže souvisely s mojí by mohly zahrnovat přidávání dalších úprav, respektive rozšiřování možností existujících – například přidání implementace dalších, sofistikovanějších interpolačních algoritmů ke současné bilineární interpolaci.

Dále by se dalo hlouběji prozkoumat možnosti urychlení IO – buď čistě v teoretické rovině. Vycházeje z výsledků které jsem naměřil, existuje spousta proměnných, které dosahovaný výkon ovlivňují, a to zejména pro menší zápisy. Druhá možnost by pak byla záležitost, kterou jsme diskutovali s Jirkou Matelou, a tou by bylo eventuální zapisování určitého množství dlaždic do TAR archivu, což by mělo eliminovat náhodné přístupy. Implementace jako taková je poměrně jednoduchá, nicméně je potřeba spíše vyřešit návaznost na další krok procesu zpracování, čímž je spojování dlaždic dohromady. Asi by nemělo moc velký smysl, kdyby se sice ušetřilo pár minut zapisováním do TARu, ale poté by se tento TAR stejně rozbalil a předal ke zpracování dále, což by v součtu s vlastním GPU zpracováním zabralo více času. Proto by bylo potřeba vyřešit toto pokud možno na obou stranách – jak ukládání do TARu, tak i čtení poté rovnou z TARu, aby z této úpravy mohly profitovat obě strany, respektive celý proces.

Při svém řešení jsem implementoval hrubozrnně jednotlivé grafické operace. Nicméně, existuje ještě jiná možnost, jak k řešení přistupovat, která vyplynula z diskuze s Johnem Cupittem, a to by bylo naopak implementace atomických operací ekvivalentních těm, co provádí libvips. Z těchto jednotlivých primitivních operací by se poté složily ty grafické operace, které implementuji nyní.

Těchto atomických operací je větší počet, než se provádí nyní, což by ovšem nemělo z rychlostního hlediska příliš vadit, protože časy, které dosahuji ve své implementaci jsou natolik nízké, že nějaké zpomalení se dá bez problému absorbovat. Nevýhodou je, že počet operací, které odpovídají našim grafickým úpravám, je zhruba 50, a všechny tyto by se musely implementovat. Na druhou stranu by to znamenalo potenciálně daleko vyšší univerzalitu, protože při implementaci významného počtu primitivních operací libvips by bylo možné spouštět velkou část nipových úprav bez toho, aby se tyto úpravy dopředu nějak zohledňovaly. Navíc by se eventuálně otevírala i cesta pro integraci s nip2 na určité úrovni, což by znamenalo velký přínos pro poměrně širokou skupinu potenciálních uživatelů. Nicméně mám určité pochybnosti, jestli by toto téma nebylo zejména implementačně příliš rozsáhlé pro práci typu diplomové nebo bakalářské.

Literatura

[1] John Cupitt a Kirk Martinez. „VIPS: an image processing system for large images“. In: *Proc. SPIE conference on Imaging Science and Technology*. Roč. 1663. SPIE, 1996, s. 19–28. URL: http://eprints.ecs.soton.ac.uk/2227/ (cit. 2011-04-21).

[2] John Cupitt a Kirk Martinez. *VIPS Manual*. 2010. URL: http://www.vips.ecs.soton.ac.uk/supported/current/doc/html/vipsmanual/vipsmanual.html (cit. 2011-03-09).

[3] Rachid Deriche. *Recursively implementating the Gaussian and its derivatives*. Výzk.zpr. RR-1893. INRIA, 1993. 24 s. URL: http://hal.inria.fr/inria-00074778/PDF/RR-1893.pdf (cit. 2011-04-19).

[4] Rafael C. Gonzales a Richard E. Woods. *Digital Image Processing*. 3. vyd. Upper Saddle River, NJ: Pearson Prentice Hall, 2008. 954 s. ISBN: 978-0-13-168728-8.

[5] David Hale. *Recursive Gaussian filters*. Tech. zpr. CWP Report 546. Center for Wave Phenomena, Colorado School of Mines, 2006. URL: http://inside.mines.edu/~dhale/papers/Hale06RecursiveGaussianFilters.pdf (cit. 2011-04-19).

[6] Gerno.t Hoffmann. *CIE Color Spaces*. 2010. URL: http://www.fho-emden.de/~hoffmann/ciexyz29082000.pdf (cit. 2011-03-08).

[7] Gernot Hoffmann. *CIELab Color Space*. 2009. URL: http://www.fho-emden.de/~hoffmann/cielab302203.pdf (cit. 2011-03-09).

[8] Jiří Jelínek. „SAGE-Based Tiled Displays for Pathology Image Rendering“. Dipl práce. Brno, 2009. 54 s. URL: https://is.muni.cz/auth/th/98936/fi_m/ (cit. 2011-02-28).

[9] Kamran Karimi, Neil G. Dickson a Firas Hamze. „A Performance Comparison of CUDA and OpenCL“. In: *CoRR* abs/1005.2581 (2010). URL: http://arxiv.org/pdf/1005.2581 (cit. 2011-05-23).

[10] Dave McCall. *Asynchronous I/O in Linux. Welcome to hell*. URL: http://davmac.org/davpage/linux/async-io.html (cit. 2011-05-14).

[11] Mike Murphy. „NVIDIA's Experience with Open64". In: *Open64 Workshop at CGO*. 2008. URL: www.affinic.com/documents/open64workshop/2008/Papers/101.doc (cit. 2011-05-24).

[12] NVIDIA Corporation. *NVIDIA CUDA^{TM}. NVIDIA CUDA C Programming Guide*. 2010. URL: http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/CUDA_C_Programming_Guide.pdf (cit. 2011-03-09).

[13] NVIDIA Corporation. *NVIDIA's Next Generation CUDA^{TM} Compute Architekture: Fermi*. Tech. zpr. NVIDIA Corporation, 2009. 21 s. URL: http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIAFermiComputeArchitectureWhitepaper.pdf (cit. 2011-05-25).

[14] Danny Pascale. *A Review of RGB Color Spaces. ... from xyZ to R'G'B'*. 2003. URL: http://www.babelcolor.com/download/A%20review%20of%20RGB%20color%20spaces.pdf (cit. 2011-03-06).

[15] Charles Poynton. *Frequently Asked Questions about Color*. 2009. URL: http://www.poynton.com/PDFs/ColorFAQ.pdf (cit. 2011-03-01).

[16] Christian Sigg a Marcus Hardwiger. *GPU Gems 2*. 2. vyd. Addison-Wesley, 2005. Kap. 20. Fast Third-Order Texture Filtering. ISBN: 0-321-33559-7. URL: http://http.developer.nvidia.com/GPUGems2/gpugems2_chapter20.html (cit. 2011-04-19).

[17] Michael Stokes aj. *A Standard Default Color Space for the Internet – sRGB*. 1996. URL: http://www.w3.org/Graphics/Color/sRGB (cit. 2011-02-27).

[18] L. J. van Vliet, I. T. Young a P. W. Verbeek. „Recursive Gaussian derivative filters". In: *Pattern Recognition, 1998. Proceedings. Fourteenth International Conference on*. Roč. 1. Srp. 1998, 509–514 vol.1. DOI: 10.1109/ICPR.1998.711192.

[19] Manuel Werlberger, Thomas Pock a Horst Bischof. „Motion Estimation with Non-Local Total Variation Regularization". In: *IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR)*. San Francisco, CA, USA, červ. 2010.

Příloha A

Převody mezi barevnými prostory

V sekci 6.2 jsem uvedl základní vlastnosti různých barevných prostorů a vztahy mezi nimi. Nyní bych rád krátce shrnul typickou posloupnost kroků pro převody mezi R'G'B' barevnými prostory nebo pro převod z R'G'B' do některého prostoru vhodnějšího pro úpravy (L*a*b*, L*C*h* apod.) a nazpět.

Obecně se postupuje v tomto pořadí:

1. Aplikace inverzní přenosové funkce, tj. gama dekomprese (s $\gamma < 1$).
2. Zobrazení lineárních RGB hodnot do prostoru CIE XYZ. Typicky se jedná o maticové násobení, kde transformační matice je odvozena na základě hodnoty primárních barev a referenčního bílého bodu.[1]
3. Převod do L*a*b*, případně L*u*v*. Dále může následovat převod do dalšího odvozeného prostoru jako L*C*h*.
4. Provedení úprav v cílovém prostoru.
5. Inverzní transformace z prostoru úprav zpět do CIE XYZ.
6. Maticové násobení inverzí transformační matice RGB->XYZ.
7. Aplikování přenosové funkce (gama komprese).

Výše uvedený postup je použitelný pro převod z R'G'B' do cílového CIE prostoru a převedení zpět do zdrojového R'G'B' prostoru. V případě, že chceme ukládat do odlišného R'G'B' prostoru, než je zdrojový, pak body 6 a 7 pracují s příslušnými maticemi cílového prostoru. Pakliže chceme pouze převádět mezi různými R'G'B' prostory, tak vynecháme navíc body 3–5.

1. Pro informace o způsobu odvození převodních matic viz [6].

Příloha B

Ukázka WS souboru

```
<Row popup="false" name="A3">
  <Rhs vislevel="3" flags="7">
    <iImage image_left="0" image_top="0" image_mag="0"
        show_status="false" show_paintbox="false"
        show_convert="false" show_rulers="false" scale="0"
        offset="0" falsecolour="false" type="true"/>
    <Subcolumn vislevel="1">
      <Row name="x">
        <Rhs vislevel="0" flags="4">
          <iText/>
        </Rhs>
      </Row>
      <Row name="super">
        <Rhs vislevel="0" flags="4">
          <iImage image_left="0" image_top="0" image_mag="0"
              show_status="false" show_paintbox="false"
              show_convert="false" show_rulers="false"
              scale="0" offset="0" falsecolour="false"
              type="true"/>
          <Subcolumn vislevel="0"/>
          <iText/>
        </Rhs>
      </Row>
      <Row name="gamma">
        <Rhs vislevel="1" flags="1">
          <Slider caption="Gamma" from="0.001" to="4"
              value="0.74657627118644054"/>
          <Subcolumn vislevel="0"/>
          <iText/>
        </Rhs>
      </Row>
      <Row name="image_maximum_hint">
        <Rhs vislevel="1" flags="4">
          <iText/>
        </Rhs>
```

```
        </Row>
        <Row name="im_mx">
          <Rhs vislevel="1" flags="1">
            <Expression caption="Image␣maximum"/>
            <Subcolumn vislevel="0"/>
            <iText/>
          </Rhs>
        </Row>
      </Subcolumn>
      <iText
          formula="Image_levels_item.Gamma_item.action␣A2"/>
    </Rhs>
</Row>
```

Zdrojový kód B.1: Výřez WS souboru

Příloha C

Trvání jednotlivých kernelů

Tabulka C.1 uvádí naměřené časy jednotlivých kernelů.

název kernelu	počet spuštění	střední hodnota (μs)	směrodatná odchylka
saturate	1	1939,642	2,149
unsharp	1	1829,134	2,730
blur_vert	2	1661,678	2,123
transpose	2	826,259	9,626
unpack_kern	1	784,758	0,376
detachL	1	604,745	0,541
balance	1	574,300	3,949
simpleLoo...[1]	1	569,069	1,516
resize	1	378,669	0,260
pack_kern	1	275,321	0,953
memcpyHtoDasync	1	5098,188	3,482
memcpyDtoHasync	1	1770,094	0,132
celkem (bez přenosů)[2]	1	9491,2	36,8

Tabulka C.1: Délky trvání provádění kernelů

Operace interpolace, saturace barev a vyvážení bílé se mapují 1 ku 1 na příslušné kernely. SimpleLookupFilterImplKernel je kernel, který spojuje úpravy gamy a tónování podle křivky. Naopak unsharp mask se rozpadá do několika kernelů – detachL, blur_vert, transpose a unsharp. U kernelů, které se spouští dvakrát v průběhu měření je uvedený součet časů obou běhů. Zbývající dva kernely – pack_kern a unpack_kern – slouží k vstupnímu předzpracování, respektive výstupnímu postprocessingu, hlavně mají za úkol převod 3 kanálů na 4 a nazpět.

1. simpleLookupiFilterImplKernel
2. Nejde o prostý součet, ale o skutečný spotřebovaný čas GPU včetně latencí spuštění kernelů. Počítáno jako rozdíl časové značky začátku kopírování z karty a časové značky spuštění prvního z kernelů.

Příloha D

Přidávání dalších grafických úprav

V tomto dodatku bych rád shrnul kontrakt pro přidávání nových úprav. Modální slovesa mají obvyklý význam a jejich interpretace odpovídá významu zažitých anglických protějšků.

Každá úprava:

- Musí implementovat rozhraní Filter včetně všech omezení a předpokladů uvedených v Javadocové dokumentaci jednotlivých virtuálních funkcí.
- Hlavičkový soubor musí být obsažen ve filter.cc a příslušné volání kontruktoru operace ve Filter::factory.
- Objekt nemusí implementovat žádné další specifické konstruktory než výchozí (přesunovací, kopírovací).
- Všechna volání musí být thread-safe (ačkoliv současná implementace toto nevyžaduje, je možné například přidání multi-GPU zpracování, které už by toto s Runtime API vyžadovalo)
- Úprava musí korektně zpracovávat 8 a 16-bitové obrázky. Navíc musí správně interpretovat hodnotu udávající maximální úroveň barevného kanálu. Obrázky jsou R'G'B, přičemž vlastnosti těchto barev odpovídají sRGB (se samozřejmou výjimkou bitové hloubky a maximální úrovně, protože sRGB předpokládá 8-bitů a maximální hodnotu 255). Rutiny nemusí kontrolovat u vstupu, jestli není překročena hodnota max_level, ale na druhou stranu musí zajisti, že nebude překročena u výstupních dat.
- Data pro GPU zpracování se ukládají jako pole ushort4 nebo uchar4, přičemž první tři hodnoty vektoru odpovídají RGB hodnotám, hodnota 4. položky je nedefinovaná. Řádek má z důvodu zarovnání délku pitch (v bajtech), která se dá zjistit voláním metody getPitch() objektu typu Image<GPUAllocator>.

- Pro alokaci dat pro výsledek musí metoda initOutputBuffer používat konstruktor Image<GPUAllocator> s parametry typu ImageDesc a MemPool, druhý je předaný volajícím. Pro mezivýsledky na GPU by tento konstruktor měly používat také, přičemž pokud se jedná o dočasné výsledky, měl by konstruktoru předávat parametr memPool, pokud o musí být persistentní mezi více voláními (vyhledávací tabulka), nesmí tento parametr předávat (a předá buď 0 nebo tento parametr vynechá), jinak dojde pravděpodobně k přepisu těchto dat. Volání metod je vždy v tomto pořadí: 1. initOutputBuffer, 2. initAdditionalBuffers.

- Všechna alokovaná data musí objekt uvolnit v destruktoru a velmi pravděpodobně i při volání initOutputBuffer, pokud toto volání není první. To platí i pro data alokovaná z poolu – v současnost sice destruktor neprovádí žádnou akci, ale toto chování se může změnit.

- ComputeAsync musí zařadit výpočet do patřičného streamu, který je předaný jako parametr. Navíc, toto volání musí být co nejrychlejší ideálně jen zavolání kernelu, eventuálně volba patřičné instance šablony kernelu na základě dat. V žádném případě nesmí tato metoda čekat na dokončení operace (pomocí eventu).

- Metoda computeAsync může volat podle návrhu i více kernelů, ale pořád platí omezení v předchozím bodě.

- Žádná metoda by neměla volat CUDA funkce, které ovlivní zpracování i jiných úprav – tedy je povoleno nastavovat např. preferenci cache pro vlastní kernel, ale ne globálně.

- Jazykem pro CPU kód metod je Runtime API.

- Třídy tohoto typu musí být možné korektně instanciovat alespoň v MAX_STREAMS[1] objektech. Toto tvrzení pro většinu úprav není relevantní. Je důležité tehdy, když potřebuje úprava některou z proměnných, které musí být statické na úrovni souboru (např. texturové reference), protože to musí být pro různé streamy různé proměnné. Pro více informací viz úprava Resize.

- Doporučuje se rozdělení úpravy na CPU kód (implementace rozhraní Filter) a GPU kód do samostatných souborů – 1. s příponou .cc a

1. main.cc

druhý .cu. Toto není úplně nutné, ale pro konzistenci doporučuji se podívat, jak je to řešené u dalších úprav.

- Při dotazování na hodnoty atributů (přes Command) by se měly metody dotazovat nejdříve na to jestli je vůbec daná hodnota nastavená. Pokud ne, tak požít výchozí hodnotu, kterou používá nip2. Toto je silně doporučeno pro všechny úpravy, pokud si není autor naprosto jistý, že nip2 tuto hodnotu vždy nastaví (není mi znám žádný takový příklad).

- Úpravy by měly být co nejvíce kompatibilní s nip2 protějšky. Tento požadavek není úplně striktní, ale alespoň sémantiku a velmi podobný výstup by měly implementovat. Na druhou stranu nip2 některé úpravy umožňuje pouze na 8-bitech a/nebo při menší přesnosti, takže snaha o kompatibilitu v tomto by byla samozřejmě kontraproduktivní. Nemluvě o jeho ignoranci maximální nastavené úrovně barevného kanálu obrázku. Také přesnost bývá někdy nižší, než je možné dosáhnout na GPU.

- Pokud kernel nebo device kód potřebuje pomocné proměnné s pohyblivou čárkou (což je na druhou stranu silně doporučováno, ať již kvůli charakteru GPU nebo vyšší přesnosti), kód by měl využívat typ float a typy s typovým konstruktorem obsahující float. Navíc i literály plovoucí čárky by měly být floatové, jinak by Fermi karty a vyšší mezivýpočty prováděly v dvojnásobné přesnosti. Příliš se nedoporučuje halffloat – jednak je to často velmi zbytečné, navíc u tohoto typu už hrozí ztráta přesnosti. Doporučuji reprezentaci barevných hodnot na intervalu [0..1] kdekoliv, kde je to výhodné, a kontrolu pod/přetečení pomocí saturatef().

- Úpravy by neměly svévolně rozšiřovat nebo zužovat přesnost (a to ani modifikací hodnoty max_level).

- Vzhledem k relativně dobré přesnosti (ULP) většiny floatových operací se doporučuje volat rychlejší verze funkcí pracující s desetinou čárkou.

- Jazykem by měl být buď C++ (zjevně alespoň pro implementaci rozhraní) standard 98 s GNU rozšířeními nebo C99. C++ kód by měl být ve jmenném prostoru „gpubip“. V případě céčkového kódu doporučuji v maximální míře statické funkce a neexportované datové

struktury (samozřejmě s výjimkou rozhraní pro ostatní kód). Pro zamezení případných budoucích problému s interoperabilitou doporučuji jako rozhraní pro kód třídy a GPU kód (kompilovaný s nvcc) céčkové rozhraní[2], kvůli případným problémům s jiným dekorovacím schématem C++ implementací. (Nvcc a gcc používají v současnosti totéž schéma.)

- Volání funkcí Runtime API a také kernelů by měly být obalené wrapperem CudaCall (funktor).

2. definované s extern "C"