CORNER-POINT DETECTION ON CUDA

Jan Ryba
Bachelor Degree Programme (3), FIT BUT
E-mail: xrybaj02@stud.fit.vutbr.cz

Supervised by: Adam Herout
E-mail: herout@fit.vutbr.cz

ABSTRACT

Corner point detection is one of many functions used in computer vision for tasks such as tracking, detecting objects, comparing images and much more. Many of the algorithms are realy complex and require a lot of CPU time. This is where the CUDA platform comes in. CUDA kernels run parallely on graphic acelerators can rapidly decrease time needed for execution, allowing even these complex calculations to work in real time or even better.

1. ÚVOD

Náplní této práce je prozkoumání algoritmů pro detekci rohů v rastrovém obraze a hlavně možnosti jejich implementaci a na platformě CUDA[1]. CUDA (Compute Unified Device Architecture) je unifikovaná architektura pro využití grafických akcelerátorů společnosti nVidia pro obecné výpočty. Architektura grafických akcelerátorů je založena na vysokém stupni paralelizace a rychlých numerických operacích, což jí dělá ideálním kandidátem pro řešení detekce rohů a samozřejmě i pro velké množství dalších aplikací. CUDA poskytuje API pro jazyk C/C++, práce je díky tomuto značně usnadněna.

Dá se říci, že v této architektuře bůže být budoucnost supercomputingu. Společnost nVidia vyrábí i grafické akcelerátory (značené TESLA) určené výhradně pro tyto účely. Jak bylo proneseno na jedné z přednášek společnosti nVidia: „Vize budoucnosti je taková, že CPU bude využíváno spíše jen pro řízení celého systému a hlavní výpočetní síla bude v grafických akcelerátorech“.

Implementace rohových detektorů na grafických akcelerátorech již existují a podávají výborné výsledky. Například implementace SIFT on GPU [3], nebo projekt OpenVIDIA[4].

2. ROZBOR

Pro detekci rohových bodů existuje celá řada algoritmů různé složitosti a různé účinnosti. Jako základní testovací reference pro architekturu CUDA bude sloužit Moravcův operátor[2]. Je sice nejjednodušším z rohových detektorů, ale na demonstraci a průzkum možností architektury CUDA poslouží dobře. Na CPU byl Moravcův operátor iplementován, bez větších pokusů o optimalizaci, tím se stal pokud možno výpočetně náročným, a tak ideálním pro srovnání jak si s touto výpočetně náročnou výzvou poradí CUDA. Později na

konečné srovnání se použije již optimalizovaná verze Moravcova operátoru a případně Harrisův operátor[2]. Jde nám hlavně o průzkum CUDA.

2.1. MORAVCŮV OPERÁTOR

Míru rohovitosti vypočte z matice 4x4 pixelů kolem daného bodu. Je důležité zmínit, že k prvkům matice se v průběhu výpočtu přistupuje mnohokrát. Na CPU byl algoritmus implementován tak, že se pozice v paměti vždy vypočítává znovu. Toto by samozřejmě šlo výrazně optimalizovat, ale pro srovnání CPU a CUDA se nám pro začátek více hodí větší výpočetní náročnost. A budeme se snažit co nejvíce optimalizovat výpočet na CUDA, jelikož cílem je vytvořit co nejrychlejší operátor právě na této platformě.

Výsledkem Moravcova, ale i jiných, operátorů je tzv. mapa rohovitosti o velikosti původního obrázku, na kterou musíme aplikovat prahování a posléze najít lokální maxima. Teprve až v těchto bodech máme detekovány rohové body.

2.2. ARCHITEKTURA CUDA

Jelikož provádění operace se děje na čipu grafického akcelerátoru nastává problém, jak sdílet paměť s procesorem. Je tedy nutné zajistit nahrání dat z paměti RAM do paměti grafické karty před začátkem operace a výsledná data poté nahrát do RAM na PC, aby nad těmito daty mohl dále pracovat procesor. Funkcím, které provádí výpočet na grafickém akcelerátoru se říká „kernel“ a vždy potřebují zadat počet bloků a počet vláken v bloku. Důležitým elementem jsou různé typy pamětí. Globální paměť představuje RAM grafického akcelátoru, přístup je velmi pomalý. Paměť textur a konstant pracují nad globální pamětí, výhodou je že mají 6kB cache. Sdílená paměť o velikosti 16kb, kterou mohou sdílet vlákna v rámci jednoho bloku, ke které je velmi rychlý přístup při správně zarovnaném přístupu. Nakonec sada registrů příslušná každému multiprocesoru, sloužící převážně pro lokálně definované proměnné.

2.3. IMPLEMENTACE NA CUDA

Základním testem bylo implementovat Moravcův operátor na CUDA stejně jako pro CPU a zjistit rychlostní rozdíl. Jedno vlákno na CUDA počítalo rohovitost právě jednoho pixelu. Výsledek byl dobrý, jelikož i na takto neoptimalizovaném příkladu bylo zrychlení oproti CPU 15-30x v závislosti na velikosti obrázku. Čím větší obrázek, tím vyšší zrychlení oproti CPU. Konkrétně tedy z 226ms na CPU na 10ms na CUDA u obrázku o velikosti 640x480 na AMD althon X2 2,6Ghz a nVidia GF9800GT.

Další snahou bylo toto řešení optimalizovat. A to redukcí výpočtů adres a počtu přístupů do globální paměti, díky uložení základní matice 3x3 pixelů kolem bodu do lokálních registrů. Výsledkem bylo pouze minimální zrychlení. Vysvětlení je na jednoduché. Jelikož při čekání na prvek z paměti je možno provádět výpočty, tak výpočty adres a další výpočty byly vsunuty do těchto míst, kdy by se jen zbytečně čekalo na data. A optimalizace počtu přístupů do paměti nepomohla jelikož architektura vyčítá z globální paměti bloky o velikosti 4B a my počítáme s prvky o velikosti 1B, zbylé bajty se tedy uchovají pro další přístup anebo se distribuují dalším vláknům, která potřebovala data ze stejného 4B bloku.

2.4. OPTIMALIZACE POMOCÍ PAMĚTI TEXTURY

Jako nejjednodušší a slibnou možností optimalizace je použití cachované textutovací paměti. Nastaví se reference textury nad blokem v globální paměti a prvky se vyčítají pomocí speciální funkce, kde se zadává souřadnice prvku. Čas běhu se zmenšil na 5,5ms běhu na

grafickém akcelerátoru u obrázku 640x480. Další možností optimalizace v tomto případě bylo upravení počtu vláken v bloku na násobek 64, konkrétně na 256 z původních 20 vláken v bloku, kdy může překladač a plánovač ještě více optimalizovat některé úkony, hlavně co se týče přístupů do paměti. Díky tomu se běh zrychlil na 3,3ms pro obrázek 640x480.

2.5. Optimalizace pomocí sdílené paměti

Ke sdílené paměti je rychlý přístup. Ale nese sebou různé problémy, hlavně jde zarovnaný přístup a problematičnost návrhu. Okolí vypočítávaných bodů se také musí nahrát do sdílené paměti a díky tomu přepočítávat souřadnice. Před vykonáváním vlastních výpočtů je nutné mít všechna potřebná data načtena ve sdílené paměti. Možnosti využití jsou různé, například jedno vlákno počítá rohovitost jednoho bodu, nebo jedno vlákno počítá rohovitost více bodů. Zatím se zkoušením dosáhlo rychlosti 3,2ms na obrázku 640x480, u obou případů.

2.6. Další možnosti optimalizace a vývoje

Náročným prvkem jsou i přesuny dat mezi RAM a grafickým akcelerátorem. Například u verze s texturovací pamětí zabere přesun dat na a z grafického akcelerátoru 2,4ms z celkových 3,3ms. Kdy se přenáší obrázek na Grafický akcelerátor a celá mapa rohovitosti zpět. Tento přístup je nevýhodný. Lepší přístup je provést prahování a detekci lokálních maxim na grafickém akcelerátoru a nazpět poslat jen souřadnice detekovaných rohů. Jako další průběh může také být implementace Harrisova operátoru[3], ten je podobný Moravcovu operátoru, ale výpočetně náročnější. Implementace Harrisova operátoru na CUDA by mohla být rychlá na základě provedeného zkoumání Moravcova operátoru na CUDA.

3. ZÁVĚR

Platforma CUDA má velkou budoucnost, už jen díky rychlému vývoji grafických akcelerátorů. CUDA zůstává použitelná na jakýchkoliv akcelerátorech různých řad díky velké míře unifikace a je použitelná na systému Windows, Linux a Mac OS.

Pro potřeby různých grafických výpočtů, i obecných, se CUDA zdá být opravdu vhodná. Pro dosažení cíle stačí jen zvolit správnou možnost optimalizace. Použití paměti pro textury je výhodné, kvůli rychlosti a menší náročnosti návrhu programu. Stejně tak výhodné je použití sdílené paměti, kvůli rychlosti a výhledově lepšího využití pro přímou lokalizaci rohových bodů. Poměr cena/výkon je u GPU doopravdy velmi dobrý. Můžeme tedy výpočty na GPU nazvat s trochou nadsázky supercomputingem pro masy.

LITERATURA

[1] nVidia Corporation, nVidia CUDA programing guide v2.1, Santa Clara, California, USA, 2008

[2] Kaněčka Petr, Vyhledávání význačných bodů v obraze, [diplomová práce], FIT BUT

[3] Changchang Wu , SIFT on GPU, University of North Carolina at Chapel Hill, http://www.cs.unc.edu/~ccwu/siftgpu/

[4] Stránky projektu OpenVIDIA : Parallel GPU Computer Vision: http://openvidia.sourceforge.net/