UNIVERZA V LJUBLJANI
FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Klavdij Lapajne

Rezanje šivov na arhitekturah CUDA

DIPLOMSKO DELO
NA UNIVERZITETNEM ŠTUDIJU

Mentor: doc. dr. Patricio Bulić

Ljubljana, 2010

Univerza
v Ljubljani

Fakulteta za računalništvo
in informatiko

Tržaška 25
1000 Ljubljana, Slovenija
telefon: 01 476 84 11
faks: 01 426 46 47
www.fri.uni-lj.si
e-mail: dekanat@fri.uni-lj.si

Št. naloge: 01603/2009
Datum: 15.10.2009

Univerza v Ljubljani, Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Kandidat: **KLAVDIJ LAPAJNE**

Naslov: **REZANJE ŠIVOV NA ARHITEKTURAH CUDA**
SEAM CARVING ON CUDA ARCHITECTURES

Vrsta naloge: Diplomsko delo univerzitetnega študija

Tematika naloge:

Implementirajte metodo rezanja šivov na večjedrnih arhitekturah NVIDIA CUDA. Metodo ustrezno paralelizirajte ter ugotovite morebitne dele algoritme, ki niso primerni za paralelno implementacijo. Časovno primerjajte sekvenčno in paralelno implementacijo rezanja šivov ter poskusite ugotovite morebitna ozka grla, ki otežujejo paralelno izvajanje na arhitekturi CUDA. Za obe izvedbi izmerite obremenitev centralne procesne enote v sistemu.

Mentor:
doc. dr. Patricio Bulić

Fakulteta za računalništvo in informatiko
2
UNIVERZA V LJUBLJANI

Dekan:
prof. dr. Franc Solina

I Z J A V A O A V T O R S T V U

diplomskega dela

Spodaj podpisani Klavdij Lapajne,

z vpisno številko 63020095,

sem avtor diplomskega dela z naslovom:

Rezanje šivov na arhitekturah CUDA

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom

  doc. dr. Patricia Bulića

- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela

- soglašam z javno objavo elektronske oblike diplomskega dela v zbirki »Dela FRI«.

V Ljubljani, dne 14. april 2010 Podpis avtorja: _____

ZAHVALA

Zahvaljujem se mentorju doc. dr. Patriciu Bulić za korektno mentorstvo in vztrajnost tudi v zadnjih trenutkih nastajanja tega dela. Zahvaljujem se tudi svojim staršem in bratu za vso podporo in pomoč, ki so mi jo nudili tekom študija. Zahvala pa gre tudi Tini Kalan in Maši Vukadinovič, za potrpežljivost in vse odkrite napake znotraj tega dela.

Kazalo vsebine

SEZNAM UPORABLJANIH KRATIC

ASCII – American Standard Code for Information Interchange
CPE – centralna procesna enota (ang. GPU)
CUDA – Compute Unified Device Architecture
GDDR – Graphics Double Data Rate
GPE – grafično procesna enota (ang. CPU)
PGM – Portable Gray Map

SPLOŠNI DOGOVORI IN DEFINICIJE:

Dogovor: Pri omembah dimenzije slik velja, da je X širina, Y pa višina obravnavane slike. Enota širine in dolžine je vedno slikovna pika oz. piksel. Dimenzije slike s širino 500 pikslov ter višino 400 pikslov bomo torej podali kot "slika dimenzij X=500, Y=400".

Dogovor: Prva vrstica je vedno zgornja (i=0), zadnja pa spodnja (i=Y-1). Prvi stolpec je vedno levi (j=0), zadnji pa skrajno desni (j=X-1). Piko znotraj slike nam podaja par (i,j).

Dogovor*:* Kot sosednje bomo smatrali elemente tabel in slik, za katere velja, da so z obravnavanim elementom povezani vsaj z enim vogalom (8-sosednost). Pot do sosednjega elementa je torej mogoča, brez da bi pri tem morali prečkati kak drug element.

POVZETEK

Cilj tega dela je optimizacija metode rezanja šivov za izvajanje na arhitkturi CUDA (Compute Unified Device Architecture) [5]. Ugotovimo želimo, če se da splošno metodo učinkovito prenesti v paralelno metodo ter ali je taka izvedba smiselna (hitrejša).

Metoda rezanja šivov je metoda za spreminjanje dimenzije slik. Pri tem se upošteva tudi vsebina slike. Spreminjanje velikosti poteka z željo, da bi na sliki ohranili čim več informacije. To dosegamo z rezanjem optimalnih šivov, to je šivov, ki vsebujejo čim manj informacije. Ugotovimo, da je primernost metode v veliki večini odvisna od vsebine slike in predlagamo optimalno uporabo.

CUDA je visoko paralelna arhitektura, ki teče na grafičnih karticah podjetja NVIDIA. Izkorišča sposobnost grafičnih kartic, da lahko naenkrat nad določenim naborom ukazov in podatkov izvajamo tudi po več tisoč niti. Seveda ima svoje omejitve in slabosti ter velike prednosti pri izvajanju algoritmov, ki so visoko paralelne narave in imajo veliko gostoto aritmetičnih operacij.

Ugotovimo, da je izvedba metode na CUDA primerna, saj pri obdelavi velikih slik daje vidno boljše rezultate. Poleg tega pri izvajanju algoritma na zunanji enoti razbremenimo CPE (centralno procesno enoto). Izpostavimo tudi omejitve in slabosti, ki jih prinaša uvedba CUDA arhitekture.

Ključne besede: rezanje šivov, prilagajanje velikosti slik glede na vsebino, CUDA, paralelno računanje, grafično procesna enota

ABSTRACT

The objective of this thesis is to optimize the Seam Carving method in CUDA (Compute Unified Device Architecture) [5]. We want to determine, if a general method can be effectively transferred into a parallel method, and whether such a step means a better (faster) performance.

The method of Seam Carving is a content aware image scaling method. Changing the size is paired with the desire to keep as much relative information in the picture as possible. This is achieved by carving out the optimal seams, which contain the least information. We find that the suitability of the method is dependent on the content of the image, and suggest optimal use.

CUDA is a highly parallel architecture that runs on NVIDIA graphic cards. It exploits the ability of graphic cards that can simultaneously execute a large amount of threads over a certain set of instructions and data. We present its limitations and weaknesses but also major strengths in implementing the algorithms that are highly parallel in nature and have a high density of arithmetic operations.

We determine that the CUDA implementation gives much better results when used on larger images. In addition, the implementation of the algorithm relieves the load on the host CPU (central processing unit).

Key words: seam carving, content aware image resizing, CUDA, parallel computing, graphics processing unit

1. UVOD

Računalniškega trga že dolgo ne sestavljajo več le 'navadni' računalniki. Poleg njih lahko zasledimo še prenosne računalnike različnih dimenzij, dlančnike in nenazadnje pametne in navadne mobilne telefone, ki so vsi sposobni izvajati multimedijske aplikacije. Med seboj se razlikujejo tako po velikosti kot po zmogljivosti. Skupaj z različnimi dimenzijami naprav pa dobimo tudi različne dimenzije prikaznih površin, tako v sami velikosti kot tudi v razmerju med stranicama površine.

Ko na takih napravah prikazujemo slike, so lahko rezultati zaradi različnega razmerja stranic neprimerni, rezanje robov slike pa tudi ni najboljša rešitev, saj nam lahko pomeni izgubo pomembnih informacij. V ta namen se uporablja metoda imenovana Rezanje šivov (Seam Carving), ki sliko zmanjša glede na njeno vsebino. Pri tem je seveda vsebino slike potrebno analizirati, kar pa je relativno zahteven postopek.

Omenili smo že, da se različne naprave po zmogljivosti lahko krepko razlikujejo. Tako je metoda rezanja šivov za šibkejše naprave neprimerna, saj spreminjanje dimenzije po tej metodi traja predolgo. Pojavila se je ideja, da bi sliki dimenzije spremenili pred pošiljanjem in tako odstranili določeno omejitev, hkrati pa tudi zmanjšali promet znotraj omrežja, saj bi se prenašale manjše slike. Rešitev je seveda primerna le za sisteme z omejenim številom naprav, ki bi koristile to storitev.

Ker je obdelava slik praviloma visoko paralelno opravilo, se je pojavila ideja, da bi lahko tudi metoda rezanja šivov pridobila z uvedbo paralelnih funkcij. Pričujoče delo tako poskuša algoritem paralelizirati. V ta namen se uporablja visoko paralelna arhitektura CUDA, ki teče na večjedrnih grafičnih karticah podjetja NVIDIA. S tem se, poleg izboljšanja časovnih performans metode, razbremeni glavni procesor računalnika, ki bi bil zadolžen za spreminjanje velikosti slik. Z uporabo sodobnih sistemov, ki lahko vsebujejo po več grafičnih kartic, ki lahko na eni kartici združujejo več GPE (grafično procesna enota), bi lahko metodo poganjali vzporedno za večje število slik.

2. PREDSTAVITEV OSNOVNIH PRINCIPOV

Za razumevanje opisanega algoritma si moramo najprej pogledati osnovne ideje, ki jih želimo realizirati. Cilj poglavja je opisati osnovno idejo metode rezanja šivov ter arhitekturo CUDA, ki jo bomo uporabljali pri paralelizaciji te metode. Če ni ob sliki navedeno drugače, velja, da so vse slike, ki prikazujejo delovanje metode rezanja šivov, zmanjšane z (v poglavju 3) opisano paralelno implementacijo algoritma. Prav tako se bomo osredotočili na manjšanje slik. Podobne ugotovitve sicer veljajo tudi pri večanju.

2.1 METODA REZANJA ŠIVOV - SEAM CARVING

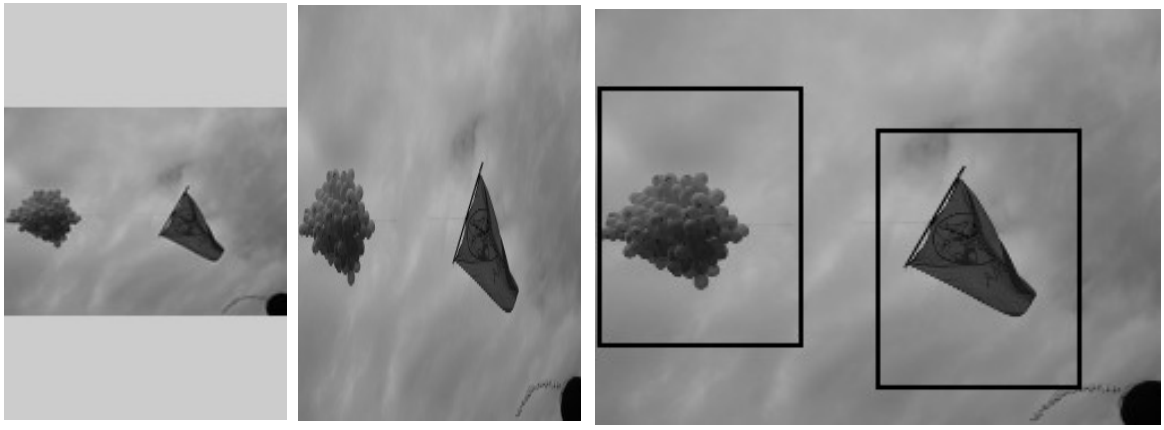
Pri spreminjanju dimenzij slike smo bili do nedavnega omejeni na klasično spreminjanje dimenzij in rezanje slike.

Kot klasične bomo obravnavali metode, ki se za spreminjanje dimenzije slik trenutno uporabljajo v večini grafičnih programov (navadno imenovane resize, resample [8] ali scale image [9]). Ob tem omenimo, da obstajajo različne metode spreminjanja dimenzije slike po tem principu. Razlikujejo se predvsem po tem, kako po odvzemanju vrstic in stolpcev prilagodijo vsebino preostale slike (metoda najbližjega soseda, bi-linearna in bi-kubična interpolacija …). Vendar vsi delujejo po istem principu enakomernega odstranjevanja ali dodajanja stolpcev in vrstic in jih zato lahko v naše namene obravnavamo v isti skupini.

Kot namiguje že ime, rezanje slike izreže del slike, ki se ujema z našimi zahtevami. Metoda je dobra, če imamo le en glavni objekt in veliko nepomembnega ozadja. Dilema se pojavi, če je takih objektov na sliki več. Prav tako se pojavi problem, če je osrednji objekt večji od želenih novih dimenzij slike. Metoda je navadno uspešnejša če jo kombiniramo z eno izmed ostalih tehnik za spreminjanje dimenzij slike.

Obe metodi imata svoje slabosti. Prva se ne ozira na vsebino slike, druga pa se osredotoči le na del slike in zato lahko izpusti pomembne dele slike. Problem prve metode se torej pojavi, ko sliki ob spreminjanju dimenzij spremenimo tudi razmerje med stranicama.

V tem primeru smo omejeni na dve možnosti. Objekte na sliki lahko deformiramo na novo razmerje dimenzij stranic, lahko pa del slike pustimo prazen in s tem ohranimo razmerja stranic. Ob ohranjanju dimenzij postopek sliko enakomerno zmanjša, ob spreminjanju razmerja med stranicami pa sliko deformira. Prav tako manjšanje slike z veliko količino ozadja lahko pomeni majhne in nerazločne glavne objekte. Slabosti te metode si lahko ogledamo v levem delu slike 2.1.



(a) (b) (c)

Slika 2.1: (a) in (b) Primera slabosti klasičnega spreminjanja dimenzije slike. (a) Slika ohrani pravilno razmerje med stranicama, vendar se zato pojavi neizkoriščen prostor. Slika je precej manjša, kot bi želeli. (b) Slika ne ohrani razmerja stranic. Opazimo deformacije glavnih objektov. (c) Originalna slika, na kateri je prikazana dilema, ki se pojavi pri rezanju.

Metoda rezanja slike (ang. crop) se sicer ozira na vsebino slike, vendar je to opravilo prepuščeno uporabniku. Preprostejši avtomatizirani postopki se navadno osredotočijo na rezanje določenega dela slike (sredina, zgornji levi kot ipd.) in se ne ozirajo na vsebino. Pomanjkljivosti rezanja si lahko ogledamo v desnem delu slike 2.1.

Avtomatizirani postopki rezanja (Photoshop [9] – trim, GIMP [10] – auto in zealus crop) sicer upoštevajo vsebino slike, vendar so omejeni na rezanje glede na enobarvno ozadje in kot taki neprimerni za večino slik in fotografij, ki imajo večbarvno ozadje.

Obstajajo tudi naprednejše metode za analizo slike, vendar se tudi te za končno spreminjanje zanašajo na kombinacijo rezanja in klasičnega spreminjanja velikosti slike. Za primernejšo se je izkazala in uveljavila raba šivov, na podlagi katere se je razvilo več metod [2].

Jasno je, da mora metoda, ki hoče prestopiti pomanjkljivosti obeh klasičnih pristopov, upoštevati vsebino slike. V nadaljevanju si bomo ogledali metodo rezanja šivov, ki sliko predimenzionira glede na njeno vsebino. Kot bomo videli, to odpravi pomanjkljivosti zgoraj omenjenih pristopov, toda tudi ta metoda ni brez svojih pomanjkljivosti.

Naša želja je, da bi sliki zmanjšali dimenzije, pri tem pa ohranili vse njene glavne objekte. Prav tako želimo, da se ob spreminjanju razmerja med stranicama glavni objekti v sliki ne deformirajo. Želimo torej, da slika ohrani čim več (za nas pomembne) informacije.

Za razliko od klasičnega spreminjanja dimenzij slike, ki dodaja ali odstranjuje stolpce in vrstice, bomo sliko najprej analizirali, nato pa iz nje rezali šive, ki se prilagajajo vsebini slike (več o šivih si lahko preberemo v 2.1.2). Pri tem ohranjamo glavne elemente slike, ne pa tudi razdalij med njimi. Slika 2.2 nam prikazuje zmanjšanje slike z metodo rezanja šivov. Več primerov si lahko ogledamo konec poglavja 3.

slika 2.2: primer s slike 2.1 zmanjšamo s pomočjo metode rezanja šivov.
Desno original, levo z metodo rezanja šivov zmanjšana slika.

Za določanje vsebine slike, jo moramo najprej analizirati. V ta namen opravimo ti. energetsko analizo slike. Glavni namen te analize je, da vsakemu pikslu znotraj slike določimo, kako vpliva na svojo okolico. Kriterije vplivanja lahko določimo glede na naše zahteve. Pri splošnem spreminjanju dimenzij na fotografijah se za primerno analizo izkaže metoda iskanja robov [4]. Iz slike nato odstranjujemo zaporedja pik, ki jim energetska analiza določi minimalne vrednosti. Slike primerov energetske analize (iskanje robov) so na voljo v poglavju 3.1.2.

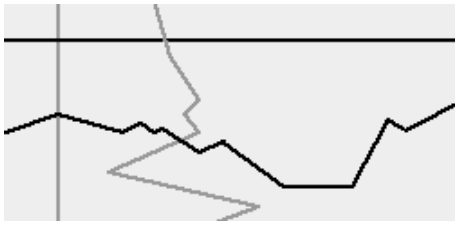
Z malo razmisleka ugotovimo, da direktno brisanje pik z minimalnimi energetskimi vrednostmi ni najbolj logična rešitev, saj lahko pričakujemo velika popačenja slike. Ne ločujemo namreč med pikami znotraj in izven objektov, prav tako pa se ne oziramo na število brisanih pik znotraj posamezne vrstice. Potrebujemo dodatna pravila, ki nam bodo ohranjala širine vrstic in višine stolpcev ter osnovno podobo slike. V ta namen uvedemo šive. Več o šivih si bomo ogledali v nadaljevanju.

2.1.1 Šiv

Spoznali smo že, da želimo s slike odstraniti tiste pike, ki imajo najmanjše energetske vrednosti ter nam tako s slike odstranijo čim manj nam pomembnih informacij. Prav tako smo spoznali, da direktno rezanje teh pik ne prinese želenih rezultatov in da zato potrebujemo dodatna pravila.

Po zgledu klasične metode spreminjanja dimenzij, ki slikam odstranjuje stolpce in vrstice, bomo definirali šiv. Za razliko od stolpcev in vrstic, ki so ravni, se lahko šivi prilagajajo vsebini in so zato različnih oblik. Klub temu pa ostajajo povezane poti z začetnega na nasprotni rob slike.

Da ohranjamo širine in višine vrstic ločimo vodoravne in navpične šive. Pogoj za vsakega je, da se lahko po določeni dimenziji premika le desno oz. navzdol. Slika 2.3 nam prikazuje osnovno razliko med vrstico, stolpcem in šivom.



Slika 2.3: osnovna razlika med vrstico, stolpcem in šivom

Šiv bi torej lahko definirali kot povezano pot med vodoravnima ali navpičnima robovoma, pri čemer velja, da lahko vodoravni šiv prečka določen stolpec, navpični pa določeno vrstico le enkrat. Vodoravni šiv na n elementih označimo $s_x(n)$, navpičnega pa $s_y(n)$. Za sliko širine X in višine Y določimo:

$$s_x(n) = \{(i_k,j_k) \mid i_k \in 0...Y\text{-}1,\ k=0...X\text{-}1;\ j_k=k,\ \mid i_k - i_{(k-1)} \mid \leq 1\} \quad (1)$$

$$s_y(n) = \{(i_k,j_k) \mid j_k \in 0...X\text{-}1,\ k=0...Y\text{-}1;\ x_k=k,\ \mid j_k - j_{(k-1)} \mid \leq 1\} \quad (2)$$

Vrednost šivu določimo tako, da seštejemo energetske vrednosti vseh njegovih pik. Šiv z najmanjšo vrednostjo imenujemo optimalni šiv in predstavlja našo izbiro za rezanje. Pri zmanjševanju širine režemo navpične, pri zmanjševanju višine slike pa vodoravne šive. Če obstaja več šivov z enako vrednostjo, lahko izberemo kateregakoli izmed njih.

Slika 2.4: prikaz optimalnih šivov, izris šivov s programom GIMP [10]. Levo: prikaz dveh šivov, desno: prikaz 100 šivov (število šivov rezanih na sliki 2.3)

2.1.2 Prednosti in slabosti metode

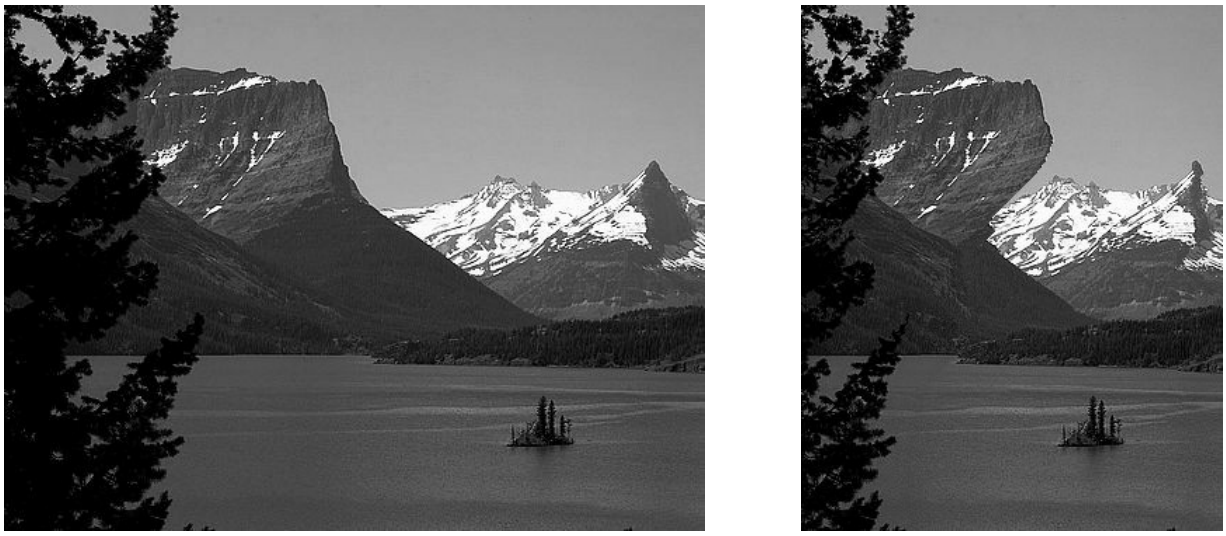
Metoda rezanja šivov nam torej omogoča spreminjanje dimenzij slike, pri čemer ohranja glavno vsebino. Prednosti take metode so očitne. Sliko lahko zmanjšamo, tudi če so njeni glavni objekti daleč narazen, spremeni se le razdalja med temi objekti. Prav tako se izognemo deformacij glavnih objektov, če spreminjamo razmerje med stranicama slike. Ob pomanjšanju slike tudi zagotovimo, da glavni objekti ostanejo čim večji. Poleg tega nam algoritem omogoča tudi druge funkcije, ki si jih bomo ogledali v poglavju 2.1.3.

Kot primere praktične uporabe bi lahko omenili prilagajanje slik na zaslone majhnih naprav

(mobilni telefoni, dlančniki ipd.), ki imajo zaslone z različnimi razmerji stranic ter manjšanje slik na ustrezno dimenzijo za prikaz na spletu. Sliko tako lahko prikažemo, brez da bi izgubili glavno informacijo, ki smo jo z njo uporabniku želeli podati. Prednosti metode nam prikazuje slika 2.2.

Seveda pa tudi ta metoda ni brez svojih napak. V določenih primerih se na zmanjšanem ozadju pojavijo deformacije in artefakti (tipičen pojav je krivljenje ravnih črt v ozadju), ki so posledica dejstva, da šivi niso enakomerno porazdeljeni glede na ozadje. Prav tako se v primeru močnega ozadja (kamni, množica ljudi itd.) ter pretežno enobarvnih glavnih objektov pripeti najslabši mogoč scenarij, ko algoritem zazna manj informacije v glavnih objektih kot v ozadju.

Algoritem pokaže svojo slabšo stran tudi v primeru, ko je potrebno sliko zmanjšati do te mere, da odvzamemo vse ozadje in smo primorani manjšati glavni objekt. Tu se slabosti pojavijo pri kompleksnejših glavnih objektih (zelo pisani objekti, vzorci znotraj objekta, obrazi) ter v primeru, ko moramo objekt zmanjšati za velik faktor. Algoritem namreč iz objekta odstranjuje šive z manj energije kar pripelje do neželenih deformacij. Slabosti metode nam prikazuje slika 2.5.



slika 2.5 [12]: prikaz napačnega delovanja. Levo originalna slika, desno sika zmanjšana z rezanjem šivov. Vidna je neželjena transformacija gorovja v ozadju.

Izboljšave osnovnega algoritma

Deformaciji pomembnejših objektov se lahko ognemo z deklaracijo interesnih con (point /area of interest), ki jih pri računanju energetske funkcije dodatno obtežimo. To zahteva sodelovanje uporabnika, zato postopek ni več avtomatiziran.

Očitno je, da z odstranitvijo šiva s slike spremenimo tudi njeno energetsko sliko ter posledično vrednosti šivov, ki tečejo čez sliko. Pri vnosu artefaktov (npr. prelom ravne črte) pride do povečanja teh vrednosti, saj se srečajo deli slike različnih intenzitet.

Brisanje šiva sliki torej doda novo energijo. Če vnaprej izračunamo, kakšna bo ta sprememba lahko odstranimo tisti šiv, ki v sliko vnese najmanj nove energije (predikcija). S tem se lahko ognemo nekaterim artefaktom, ki bi se sicer pojavili v sliki. To izboljšanje je del izboljšane metode rezanja šivov (improved seam carving) [1].

Metoda je torej odlična, vendar le pod določenimi pogoji? Kako jo najbolje uporabiti?

Pri manjšanju dimenzij slike za prikaz na manjših napravah smo navadno primorani sliko zmanjšati za večji faktor. To pa vključuje tudi večje pomanjšanje osnovnega objekta. Priporočamo, da sliko najprej zmanjšamo s klasičnimi postopki, pri čemer ohranjamo originalno razmerje med stranicama. Za tem uporabimo metodo rezanja šivov in sliko zmanjšamo na njeno končno velikost.

S tem se izognemo tako deformacijam, ki bi nastali pri prevelikem zmanjšanju glavnih objektov z metodo rezanja šivov ter deformacijam, ki nastanejo pri klasičnih metodah ob spreminjanju razmerja med stranicami. Rezultat prikazuje slika 2.6.

(a) (b) (c)

slika 2.6: zmanjšanje enostavnega objekta. Objekt smo najprej iz 2304x3072 zmanjšali na 461x614 (a). Prikazujemo dve zmanjšanji širine z metodo rezanja šivov, prvo za 100 (b), drugo pa za 200 pik (c).

2.2 CUDA

Že dolgo velja, da so računalniške igre ene izmed najzahtevnejših aplikacij, ki jih poganja povprečen uporabnik osebnega računalnika. Eden izmed najbolj izpostavljenih elementov strojne opreme pri tem pa so grafične kartice. Te so tako v zadnjem desetletju doživele velik razvoj. Ker je obdelava grafike visoko paralelno opravilo, so se grafične kartice razvile v visoko paralelne naprave z veliko računsko močjo in zalogo lastnega hitrega pomnilnika (GDDR - Graphics Double Data Rate) [8], ki lahko dosega nivo pomnilnika v napravi sami (512MB, 1GB in več). Na vsaki GPE tako prebiva več multiprocesorjev, izmed katerih ima vsak po osem procesorjev imenovanih jedra (core) [6]. Najnovejši izdelki pa na eni kartici združujejo do 4 GPE.

Posebnost teh grafičnih kartic je prav v močni računski moči na paralelnih strukturah, saj je več tranzistorjev namenjenih računskim kot pa podatkovnih operacijam. Idealne so torej za paralelne probleme z veliko količino aritmetičnih operacij, kjer lahko isto kodo istočasno izvršimo nad večjim številom podatkov. Ker se nad vsemi elementi izvrši ista koda, imamo manjšo potrebo po naprednejšem nadzoru toka izvajanja. Med latenco pri pomnilniških dostopih tečejo druge niti, ki izvajajo aritmetične ukaze. Zadostna gostota aritmetičnih operacij tako lahko povsem zakrije zamike v izvajanje, ki sicer nastanejo pri počasnih dostopih do pomnilnika. Razliko v strukturi CPE in GPE prikazuje slika 2.7.

slika 2.7 [6]: razlika v zasnovi CPE (1) in GPE (2). Vidimo lahko, da ima GPE na razpolago precej večje število ALE (aritmetično logična enota – ang. ALU) ter manj predpomnilnika (ang. cache). Organizacijsko je razdeljena na več multiprocesorjev, izmed katerih ima vsak svoj predpomnilnik, ukazno enoto in večje število ALE. Količina globalnega pomnilnika je sicer velika, vendar so dostopni časi precej daljši. To ustreza omenjenim priporočilom o veliki gostoti aritmetičnih operacij.

Ne preseneča torej, da so se grafične kartice pričele uporabljati tudi kot pomožne enote pri izvajanju programskih problemov izven področja obdelave računalniške grafike. Novembra 2006 je NVIDIA predstavila svojo arhitekturo CUDA, ki omogoča programerjem, da lahko s pomočjo njenih grafičnih kartic izvajajo računsko zahtevne programe.

Priporočljiva je uporaba pri problemih, ki omogočajo visoko stopnjo paralelizacije. Prav tako se priporoča, da imajo obravnavani problemi veliko intenziteto aritmetičnih operacij. Področje paralelnega procesiranja ter arhitektura CUDA sta obsežni področji, podrobna analiza katerih presega namene tega dela. V nadaljevanju si bomo tako ogledali le tiste elemente, ki so nujni za razumevanje osnovnih principov delovanja opisanega algoritma.

2.2.1 Programirni model CUDA

Razvoj grafičnih kartic ter njihovih sposobnosti paralelnega računanja se seveda še vedno veča, tako je bil eden glavnih ciljev pri razvoju CUDA podpora velikemu rangu grafičnih kartic, tako v podpori delovanja arhitekture same, kot tudi v podpori delovanja v CUDA razvitih programov. Želimo torej, da bi lahko naš program poganjali na različnih napravah z velikimi razlikami v zmogljivosti (pomnilnik, število jeder itd.), brez da bi bili pri tem primorani spreminjati kodo (slika 2.8)

V osnovi nam to omogočajo trije ključni elementi. Urejenost niti v neodvisne skupine, ureditev pomnilnika (skupni pomnilnik) ter možnost sinhronizacije niti znotraj bloka [6]. GPE obravnavamo kot pomožno procesno enoto (koprocesor), ki se navadno imenuje naprava (device), medtem ko se glavni procesor naslavlja kot gostitelja (host). Programsko kodo izvajajo niti, ki jih porazdelimo v bloke. Pišemo jo v posebnih funkcijah imenovanih jedra.

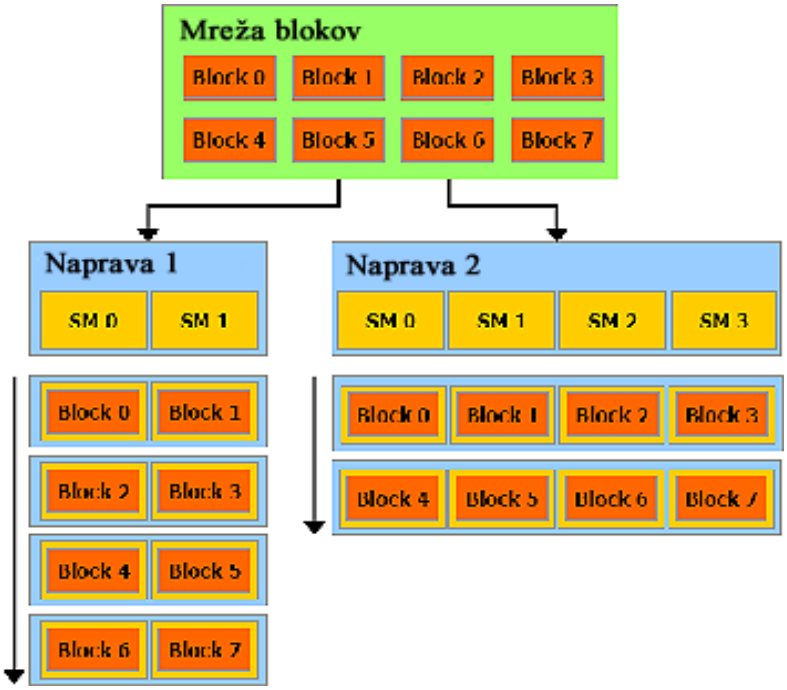
Značilno je, da imamo lahko precej večje število blokov kot procesorskih jeder, ki jih izvršujejo. Na jedro imamo tako lahko aktivnih več blokov, ki se pri izvajanju izmenjujejo, ko določen blok čaka na podatke iz pomnilnika ali sinhronizacijo niti. Neaktivni bloki čakajo v vrsti in se pričnejo izvrševati (postanejo aktivni), ko se do konca izvede eden izmed trenutno aktivnih blokov.

V času pisanja na trgu obstaja velik izbor splošnih in profesionalnih grafičnih kartic, ki podpirajo CUDA. Razlikujejo se tako po moči kot tudi po računskih zmogljivostih in omejitvah [6].

2.2.2 C za CUDA

Funkcije, ki se izvajajo na CUDA karticah, se lahko piše v zbirnem jeziku imenovanem PTX [6]. Vendar pa se kot bolj praktična izkaže uporaba razširitve jezika C, ki so jo poimenovali C za CUDA (C for CUDA). Sestavlja jo nabor funkcij, ki razširjajo delovanje jezika C. Datoteke s kodo lahko vsebujejo kodo namenjeno izvajanju na napravi ter kodo namenjeno gostitelju. Posebne ukaze, ki jih vsebuje C za CUDA bomo spoznali v sledečih poglavjih.

Za prevajanje funkcije nam je na voljo poseben prevajalnik poimenovan nvcc [6]. Ta kodo, namenjeno izvajanju na napravi, prevede v PTX ali binarno kodo. Koda namenjena gostitelju se prevede z gostiteljevim prevajalnikom v zadnjih stadijih prevajanja.



Slika 2.8 [6]: prikaz razporeditve mreže blokov na dve različni napravi. Naprava 1 ima na voljo le 2, naprava 2 pa 4 multiprocesorje. V kodi ustvarimo mrežo blokov. CUDA nato pri izvajanju sama poskrbi za to, da se ti bloki pravilno razporedijo po multiprocesorjih naprave. To nam omogoča veliko prilagodljivost za izvajanje na različnih napravah brez posegov v kodo. Lastnost imenujemo transparentna skalabilnost.

2.2.3 Jedra

Kodo, ki se izvaja na napravi, pišemo v posebnih funkcijah imenovanih jedra (kernel). Zaradi enakega prevoda moramo paziti, da ne pomešamo jedra (kernel) kot funkcije in jedra (core) kot enote na multiprocesorju.

Jedra so torej posebne funkcije, ki jih izvajamo na napravi. Ko kličemo jedro, se to izvrši n-krat s strani n vzporednih niti. Spodnja koda prikazuje definicijo in klic jedra.

```
1.  // definicija jedra
2.  __global__ void VecAdd(float* A, float* B, float* C)
3.  {
4.    ...
5.  }
6.  // koda za gostitelja vsebuje klic jedra
7.  int main()
8.  {
9.    ...
10.  // klic jedra
11.  // dimGrid - dimenzija mreže (št. blokov)
12.  // dimBlock - dimenzija bloka (št. niti na blok)
13.  VecAdd<<<dimGrid, dimBlock>>>(A, B, C);
14.  }
```

Število niti definiramo ob klicu jedra z dvema parametroma: številom blokov v mreži in številom niti na blok (dimGrid in dimBlock - vrstica 11).

V splošnem lahko v isti datoteki pišemo kodo za napravo (jedra) in kodo za gostitelja. Za dobro in pregledno programiranje svetujemo, da se jedra piše v ločeni datoteki. Pri pisanju kode moramo paziti na to, da se nam vsi elementi znotraj polj obravnavajo istočasno, vsak s svojo nitjo. To zahteva direkten izračun indeksa za določeno polje. Navodila za optimalno programiranje nam priporočajo, da imamo v jedrih veliko gostoto aritmetičnih operacij.

Obstaja več vrst jeder. Med seboj se ločijo glede na to, kdo jih lahko kliče. Oznaka __global__ definira jedro, ki ga lahko kliče gostitelj, medtem ko oznaka __device__ pomeni jedro, ki ga lahko kličemo le znotraj naprave, torej iz drugega jedra. Definirana je tudi oznaka __host__, ki pomeni funkcijo, ki se izvrši na gostitelju, vendar isti učinek dosežemo, če ne uporabimo nobene oznake. Oznaka pride do izraza, če jo uporabimo skupaj z oznako __device__. V tem primeru se funkcija prevede tako za napravo kot tudi za gostitelja. [6]

2.2.4 Niti in bloki

Pri obravnavi jeder smo omenili, da se jedro izvede nad določenim številom niti, ki ga definiramo ob klicu jedra. Niti ob tem razdelimo na bloke, ki jih porazdelimo znotraj mreže. Pri tem lahko definiramo eno- dvo- ali trodimenzionalne bloke. Bloke razdelimo v eno- ali dvodimenzionalno mrežo enakih blokov. Dvodimenzionalno mrežo blokov lahko vidimo na sliki 2.8.

CUDA zahteva, da se mora vsak blok niti izvršiti neodvisno od vseh ostalih blokov. To nam omogoča poljubno razvrščanje blokov po multiprocesorjih. Podobno velja za niti. Vsaki niti moramo zagotoviti, da se lahko pravilno izvrši neodvisno od ostalih. Razlika je v tem, da imamo znotraj bloka možnost sinhronizacije niti na določenem koraku. Niti v različnih blokih ne moremo medsebojno sinhronizirati.

Ta omejitev nam omogoča, da lahko bloke izvajamo popolnoma neodvisno. Tako jih lahko razporedimo na poljubno število multiprocesorjev ter izvajamo v poljubnem vrstnem redu. To nam omogoča veliko prilagodljivost za izvajanje na različnih napravah z različnim številom jeder (cores), brez posegov v kodo. Ta zmožnost se imenuje transparentna skalabilnost (transparent scalability) in je prikazana na sliki 2.8.

Ob klicu jedra se vsaki niti v bloku dodeli edinstven indeks, ki jo razloči od ostalih niti znotraj bloka. Niti v različnih blokih imajo enake indekse. Za enolično določitev niti v splošnem torej potrebujemo tudi indeks bloka v katerem se nahaja. Vsakemu bloku in niti se dodeli lasten indeks. S pomočjo teh indeksov lahko znotraj iste kode jedra (kernel) različne niti dostopajo do različnih podatkov.

Omenili smo že, da so lahko bloki in niti večdimenzionalni. To nam omogoča preprosto prilagajanje indeksov niti glede na podatkovne strukture, kot so večdimenzijska polja, matrike ipd. Indeksi niti in blokov se kličejo po standardnih oznakah dimenzij: x, y in z.

Primer izračuna indeksov niti za dvodimenzionalno tabelo prikazuje spodnji klic:

```
1. int i = blockIdx.x * blockDim.x + threadIdx.x;
2. int j = blockIdx.y * blockDim.y + threadIdx.y;
```

Smiselno je torej, da jedro kličemo s toliko nitmi, kolikor je elementov v največjem izmed polj, nad katerimi želimo izvajati določeno operacijo. Tako lahko vsaka nit prek svojega indeksa dostopa do drugega elementa znotraj polja na enak način, kot bi do tega elementa dostopali v programskem jeziku C. Spodnja koda v tabelo A zapiše seštevke indeksov posameznega polja:

```
1. // 2D polje A
2. A[i][j]=i+j;
3. // oziroma s kazalcem
4. *A(i*X + j) =i+j;
```

Znotraj jeder lahko torej razvijamo običajno C kodo, le da se ob tem zavedamo, da se bodo naši ukazi istočasno izvršili na večjem naboru podatkov. Pri tem smo sicer omejeni, saj ne moremo klicati funkcij, ki bi se pri izvajanju programa izvajale izven grafične kartice (npr. printf). Prav tako jedra ne podpirajo uporabe rekurzivnih funkcij [6].

Število niti in blokov na jedro je torej lahko precej večje kot število procesnih enot naprave. Pri številu niti na blok smo omejeni na 512 niti. Priporočljivo je določanje števila niti, ki je večkratnik števila 32 (poglavje 2.2.5). Pri tem smo na posamezno dimenzijo omejeni na 512 (širina), 512 (višina) in 64 (globina) niti. Pri razporejanju blokov v mrežo pa smo omejeni na 65535 blokov na dimenzijo.

Kljub tem zares velikim številkam pa velja opozoriti, da se vse niti ne izvajajo istočasno. Na multiprocesor je tako lahko aktivnih največ 768 niti oziroma 8 blokov. Maksimalno število istočasno aktivnih niti za GeForce GTX 260, ki vsebuje 24 multiprocesorjev [7] je torej kar 18432.

Niti znotraj istega bloka si lahko med seboj izmenjujejo podatke prek omejene količine skupnega pomnilnika. Niti različnih blokov pa lahko komunicirajo le preko počasnejšega globalnega pomnilnika. Pri tem velja opozoriti, da je komunikacija med nitmi priporočljiva le, če jih prej sinhroniziramo, sicer lahko določena nit bere podatke, ki jih druga nit še ni obdelala.

S tem se nam pojavi nova omejitev števila blokov na multiprocesor. Velja, da je količina skupnega pomnilnika, ki je na multiprocesorju na razpolago nitim, omejena na 16 KB in količina registrov na 8192. Tako se število aktivnih blokov določa tudi s tem, koliko skupnega pomnilnika in registrov zahteva posamezen blok ter je odvisno od programa.

Pri porazdelitvi niti in blokov velja omeniti tudi navodilo za optimalno izvajanje kode, ki priporoča, da sta naenkrat na multiprocesorju aktivna vsaj dva bloka. To omogoča menjavanje

med blokoma, ko eden čaka na sinhronizacijo niti ali na podatke iz pomnilnika naprave (če v bloku ni dovolj niti, da bi latenco pokrili z menjavanjem niti znotraj bloka).

2.2.5 Pomnilnik

CUDA programabilni model predvideva, da tako naprava kot gostitelj razpolagata z lastnim pomnilnikom. Vse niti se izvršijo na napravi in torej izvajajo operacije nad pomnilnikom naprave.

To zahteva, da prostor na napravi najprej alociramo, nato pa vanj prenesemo vsebino polj, nad katerimi bomo izvajali operacije. Navadne številske vrednosti (int, float) lahko podamo kot parametre jedra (kernel) in nam jih ni potrebno ločeno kopirati na napravo. Polja v parametrih jedra podajamo s kazalci.

Po končani izvedbi programa polja prenesemo nazaj v gostiteljev pomnilnik ter sprostimo pomnilnik na napravi. Kopiranje podatkov v napravo je relativno zamuden proces, ki ga je zato znotraj programa potrebno minimizirati.

Spodnji primer prikazuje zgoraj opisano upravljanje s pomnilnikom:

```
1.  // koda gostitelja vsebuje pripravo pomnilnika na napravi
2.  int main()
3.  {
4.  int stevilo = 5;
5.  size_t size = N * sizeof(float);
6.   ...
7.  float* h_A = malloc(size);
8.   ...
9.  // alokacija prostora na napravi
10. cudaMalloc((void**)&d_A, size);
11.  ...
12. // kopiranje podatkov na napravo
13. cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
14. ...
15. // klic jedra
16. Jedro<<<dimGrid, dimBlock>>>(A, stevilo);
17.  ...
18. // kopiranje podatkov iz naprave
19. cudaMemcpy(h_A, d_A, size, cudaMemcpyDeviceToHost);
20.  ...
21. // sproščanje pomnilnika
22. cudaFree(d_A);
23. }
```

Pri tem smo uporabili dogovor, da se strukture na gostitelju označuje z h_imeStrukture, strukture na napravi pa d_imeStrukture. V zgornji kodi lahko vidimo primere osnovnih ukazov, ki jih za upravljanje s pomnilnikom na napravi priskrbi C za CUDA.

Pomnilnik na napravi delimo na globalni in lokalni pomnilnik (pomnilnik naprave) ter konstantni, teksturni in skupni pomnilnik ter registre (pomnilnik multiprocesorja). V

podrobnosti se ne bomo poglabljali, omenimo le, da so dostopi do lokalnega in globalnega pomnilnika dolgotrajni. Skupni pomnilnik in registri se nahajajo na čipu in imajo hitre dostopne čase. Prav tako lahko hitro dostopamo do konstantnega (pomnilnik rezerviran za konstante) ter teksturnega pomnilnika [6].

Na zgoraj prikazani način kopiranja se podatki prenesejo v globalni pomnilnik naprave. Spremenljivke in parametri jedra se prenesejo v registre, če pa zanje ni mogoče določiti točne vrednosti pa v lokalni pomnilnik. Tja se podatki (običajno polja) kopirajo tudi, če prevajalnik oceni, da porabijo preveč registrskega prostora.

Vsaka nit razpolaga s 16 KB lokalnega pomnilnika, vse niti znotraj bloka pa še z dodatnim skupnim pomnilnikom. Pri upravljanju s skupnim pomnilnikom (oznaka __shared__) moramo paziti, saj je na voljo le 16 KB skupnega pomnilnika na multiprocesor, na enem multiprocesorju pa je optimalno izvajati vsaj dva bloka. Število blokov na multiprocesor omejuje tudi poraba registrov na blok. Multiprocesor ima na razpolago 8192 32-bitnih registrov. Strukturo pomnilnika prikazuje slika 2.9.

2.2.6 Multiprocesor - SIMT in snopi

CUDA je zasnovana okrog večjega števila večnitnih multiprocesorjev. Ob klicu jedra se bloki niti razporedijo na te multiprocesorje. Ko se bloki na posameznem multiprocesorju izvedejo, se na njem pričnejo izvajati novi bloki, ki so do tedaj čakali v vrsti (slika 2.8).

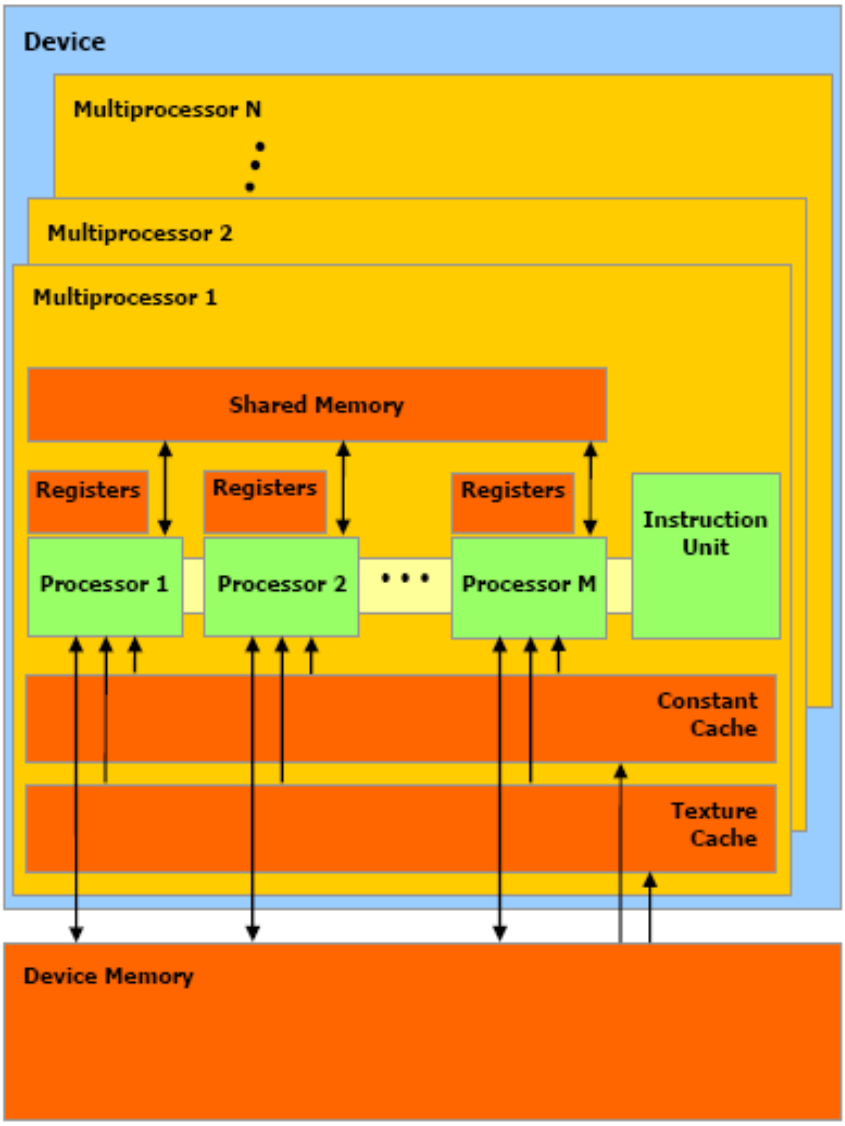
Multiprocesor na dosedanjih implementacijah sestavlja 8 skalarnih procesorjev (jeder – cores), dve posebni enoti za transcedentna števila, ukazna enota ter lastni pomnilnik (slika 2.9).

Za upravljanje s stotinami niti multiprocesor uporablja novo arhitekturo, ki so jo poimenovali SIMT (single-instruction multiple-thread – več niti en ukaz). Multiprocesor vsaki niti določi lasten skalarni procesor, na katerem se vsaka nit izvrši neodvisno od ostalih. [6]. Multiprocesorjeva ukazna enota upravlja z nitmi v skupinah po 32, ki jih imenujemo snopi (warps). Priporočljivo je, da blok vsebuje število niti, ki je večkratnik števila 32.

Ko multiprocesor dobi v izvajanje enega ali več blokov, njihove niti enota SIMT razvrsti v snope. Delitev se vedno izvede na enak način. Vsak blok vsebuje niti, ki imajo lastne indekse razporejene po velikosti. Snopi tako dobijo po 32 zaporednih niti, pri čemer prvi snop prične z nitjo, ki ima indeks 0.

Vsakemu snopu se dodeljujejo ukazi s strani SIMT. Ko snop prejme ukaz, ta velja za vse aktivne niti v snopu. Snop izvaja po en ukaz naenkrat, zato se največja učinkovitost doseže, ko imamo aktivnih vseh 32 niti ter se vse niti strinjajo z izvajanjem predlaganega ukaza. Če naletimo na vejitev ukazne poti zaradi pogojnega stavka, se vsaka izmed mogočih poti izvede zaporedno na nitih, ki izpolnjujejo določen pogoj. Ostale niti morajo ta čas čakati. Zato je priporočljivo, da se takim vejitvam, če je le mogoče, izogibamo.

SIMT ukazi določajo obnašanje posamezne niti. Tako lahko brez večje razlike znotraj jeder (kernel) pišemo kodo za eno nit, ki izvaja med seboj odvisne ukaze ali pa za veliko število niti, ki se neodvisno izvajajo na paralelni strukturi in se pri tem razhajajo znotraj pogojnih stavkov.



Slika 2.9 [6]: struktura GPE. En GPE (najnovejše grafične kartice jih imajo tudi po več) vsebuje več multiprocesorjev. Vsak multiprocesor ima večje število (v času pisanja je to število 8) procesnih enot (procesorjev oz. jeder), ukazno enoto ter lasten pomnilnik. Tega delimo na pomnilnik za teksture, pomnilnik za konstante, skupni pomnilnik in registre. Vsi multiprocesorji na isti GPE si med sabo delijo glavni pomnilnik.

Ker se vsaka nit lahko izvaja po ločeni poti znotraj jedra in deluje na lastnem naboru podatkov, ima vsaka nit lasten pomnilnik, nabor registrov, programski števec (PC) in sklad [5].

CUDA v celoti sama upravlja z izvajanjem in razporejanjem niti. Programerju je to opravilo skoraj popolnoma nevidno in lahko razvija skoraj običajno kodo. Na voljo nam je le preprosta sinhronizacija niti znotraj istega bloka. To močno poenostavi programiranje jeder. NVIDIA hkrati zagotavlja, da njihov sistem upravljanja z nitmi popolnoma eliminira smrtne objeme niti (threadlocks), ne glede na število hkrati aktivnih niti [5].

3. IMPLEMENTACIJA

Koda je bila razvita v programskem jeziku C++, za paralelizacijo pa je bil izbran jezik C za CUDA (poglavje 2.2). Razvojno okolje projekta je MS Visual Studio 2008. Realizirali smo osnovno metodo rezanja šivov.

Za realizacijo algoritma uporabljamo preprost format slik PGM (Portable Gray Map) v ASCII zapisu. Gre za sivinski format slik. Format poda višino in širino slike, maksimalno vrednost skale za sivino ter tabelo celih števil. Vsak piksel znotraj slike se predstavi z eno celoštevilsko vrednostjo, ki predstavlja njegovo sivino glede na skalo.

Ker je cilj ugotavljanje primernosti uporabe CUDA pri metodi rezanja šivov, je realizirano le spreminjanje širine slike (rezanje navpičnih šivov). Spreminjanje višine poteka analogno in na ugotovitve ne vpliva.

3.1 ALGORITEM

Najprej si bomo ogledali osnovni algoritem za rezanje šivov, nato pa še paralelizacijo tega algoritma. Osnovni potek delovanja sestoji iz treh glavnih korakov, ki bodo opisani v nadaljevanju:

- energetska analiza slika
- računanje šivov
- rezanje minimalnega šiva

Kot prvo pa si poglejmo vhodne pogoje, ki jih zahteva osnovna funkcija algoritma.

3.1.1 Vhodni pogoji

Glavna funkcija za vhodne podatke pričakuje 6 parametrov. Prvi štirje so za delovanje algoritma obvezni, zadnja dva pa omogočata nadaljnjo razširitev funkcionalnosti.

- kazalec na polje int (polje sivinskih vrednosti slike)
- širina slike (int)
- višina slike (int)
- želeno zmanjšanje slike (int)
- uporabljana funkcija energetske analize (int)

3.1.2 Energetska analiza slike

Rezanje šivov poteka glede na spremembo energije znotraj slike. Osnovna ideja algoritma je rezanje šivov, katerih odstranitev bo najmanj opazna glede na vsebino slike. Zato nam energija piksla predstavlja pomen tega piksla za sliko.

Za ta namen se pri fotografijah kot najbolj primeren izkaže algoritem iskanja robov [4], ki pikslom določi vrednost glede na to, kako močno se razlikujejo od pikslov okrog sebe. Glede na naše potrebe se seveda lahko odločimo za drugačno analizo (magnituda gradienta, entropija...) [2]. Obstaja pa tudi več različnih algoritmov za iskanje robov (slika 3.1). Zato implementacija omogoča vgradnjo novih funkcij za energetsko analizo, med katerimi lahko preklapljamo z vhodnim parametrom.

Trenutno algoritem uporablja preprosto metodo iskanja robov, ki pikam določi vrednost kot vsoto obteženih absolutnih razlik med vrednostmi sosednjih pik. Na sliki 3.1 smo zaradi jasnejšega prikaza vse vrednosti rezultata energetske analize povečali za faktor 2. Na samo analizo in rezanje šivov taka transformacija ne vpliva, saj se razmere med vrednostmi ohranijo.

Za analizo zadostuje, da primerjamo vrednost pike z vrednostjo točke desno, točke spodaj ter diagonalne točke med njima. Razliko pike glede na ostale sosede upoštevamo pri računanju njihove energije [3].

Vsaka razlika je obtežena z razdaljo med pikama, skupna vsota pa se deli s številom razlik, saj to ni vedno enako (robovi) – enačba (3). Rezultat uporabljene energetske funkcije si lahko ogledamo na sliki 3.1.

$$E(x,y)=\frac{|((x,y)-(x+1,y))|+|((x,y)-(x,y+1))|+\frac{|((x,y)-(x+1,y+1))|}{\sqrt{(2)}}}{3} \qquad (3)$$

Energetsko analizo izvajamo samo enkrat na začetku izvajanja algoritma. S tem preprečimo spajanje objektov na sliki ob odstranjevanju več šivov ter seveda pridobimo na hitrosti. Za vhodne parametre potrebuje višino in širino slike ter kazalec na sliko, ki je predstavljena z enodimenzijskim poljem celoštevilskih vrednosti – sivinsko sliko (tak zapis mora zagotoviti funkcija branja slike).

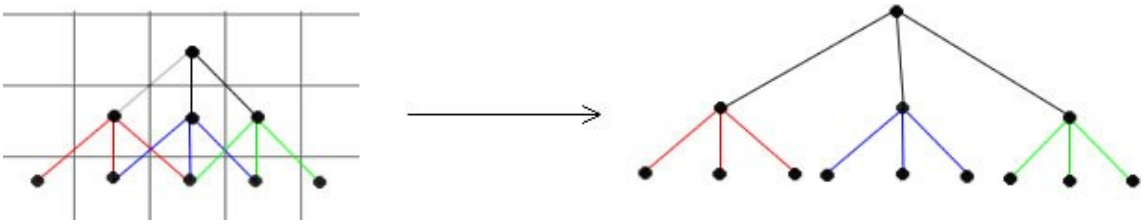
(a) (b) (c)

slika 3.1 [12]: primerjava različnih metod rezanja robov. (a): Sobelova metoda, (b): neon filter, (c): tukaj opisana metoda

3.1.3 Računanje šivov in podšivov

Iz opisa šiva (poglavje 2.2.2) lahko ugotovimo, da je navpični šiv praktično gledano minimalna utežena povezana pot od vrha do dna slike. Pri tem upoštevamo še omejitev, da se lahko premikamo le navzdol po sliki, uteži pa nam predstavljajo vrednosti pike, dobljene z energetsko analizo.

Zastavi se vprašanje, kako lahko pridemo do optimalnega šiva. Vse šive lahko zajamemo v gozd dreves, katerih koreni so piksli prve vrstice slike. Drevo teoretično zgradimo tako, da vsako piko vrstice i povežemo s sosednjimi pikami iz vrstice i-1 (slika 3.2).



slika 3.2: izgradnja drevesa. Vsako vozlišče se poveže s svojimi spodnjimi sosedi.

Imamo torej način, prek katerega se lahko sprehajamo po različnih šivih. Kako pa določimo optimalni šiv, ki ga bomo rezali iz slike? Iz slike 3.2 lahko vidimo, da se izračun vrednosti vseh poti za vsako drevo ne izplača, saj se nam poddrevesa kmalu začnejo ponavljati in iste poti računamo večkrat.

Za najučinkovitejšega se izkaže pristop dinamičnega programiranja, s pomočjo katerega določimo optimalno pot skozi vsako drevo. Po izvedbi algoritma koreni vseh (pod)dreves vsebujejo vrednost minimalne poti skozi (pod)drevo.

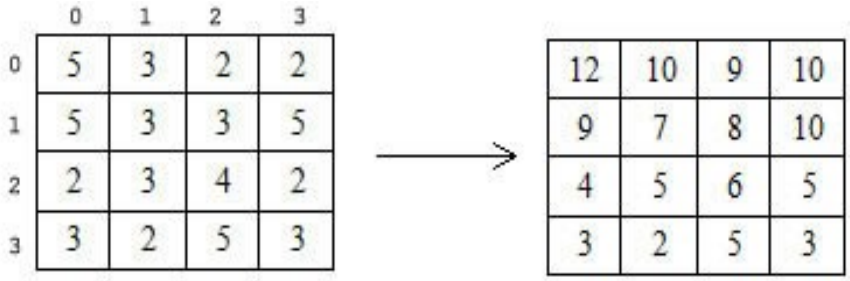
Algoritem izvajamo na tabeli, ki smo jo dobili kot rezultat energetske analize. Drevesa gradimo od listov proti korenu. Vozlišča zadnje vrstice nimajo poddreves in zato že vsebujejo minimalno pot do sebe. Vsakemu nadaljnjemu vozlišču prištejemo vrednost korena minimalnega poddrevesa (formula 4).

$$(x,y)=(x,y)+min[(x-1,y+1),(x,y+1),(x+1,y)] \quad (4)$$

$$min(n)=n+min(n-1); \quad (5)$$

Tako smo za vsako (pod)drevo izračunali vrednost minimalne poti od lista do korena (formula 5). V točkah prve vrstice slike je ta pot enaka navpičnemu šivu, dobljene vrednosti pa so vrednosti minimalnega šiva za posamezno točko.

Postopek moramo ponoviti po vsakem rezanju šiva. Izvajamo ga na primerno zmanjšani tabeli (tabelo z analizo slike zmanjšujemo skupaj s tabelo slike, ko režemo šiv), ki vsebuje energetsko analizo slike. Slika 3.3 nam prikazuje primer izračuna za preprosto tabelo.



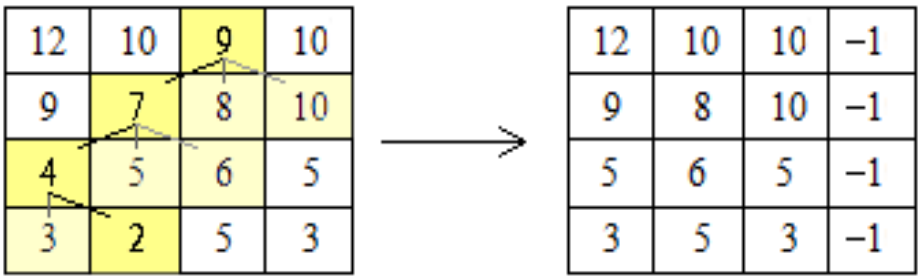
slika 3.3: primer izračuna tabele. Tabelo gradimo od spodaj navzgor tako, da na vsakem koraku elementu tabele prištejemo min. spodnjega soseda (formula 4). Ker je vsaka vrstica odvisna od izračuna prejšnje, je tu mogoče vzporedno računati le elemente znotraj iste vrstice.

Primer izračuna za polje (1,1) na sliki 3.3: (1,1) = 3 + min(4,5,6) = 3 + 4 = 7

3.1.4 Brisanje optimalnega šiva

Z dinamičnim programiranjem smo pridobili tabelo vrednosti optimalnih (pod)šivov za vsako piko prve vrstice. Zanima pa nas optimalen šiv glede na celotno sliko. Najprej poiščemo najmanjšo vrednost prve vrstice, njeno lokacijo zapišemo v ločeno tabelo ter se z nje premaknemo na poddrevo z najmanjšim korenom. Postopek ponavljamo, dokler ne pridemo do dna slike. Po formuli (5) smo tako določili šiv z minimalno vrednostjo. Če je takih šivov več (kadarkoli imamo na poti na izbiro več elementov z minimalno vrednostjo), lahko režemo poljubnega med njimi. Lokacije, po katerih smo se premikali, smo na vsakem koraku zapisali v ločeno tabelo, ki nam bo služila za brisanje šiva.

Dobljeni šiv brišemo tako, da elemente slike za šivom zamaknemo eno polje levo ter s tem šiv prepišemo. Postopek nam na desnem robu tabele pusti prazen stolpec, v katerega vpišemo negativne vrednosti (npr. -1) in s tem simuliramo brisanje stolpca. Določanje in brisanje optimalnega šiva nam prikazuje slika 3.5.



slika 3.4 določanje in brisanje optimalnega šiva na tabeli s slike 3.3. Svetleje so obarvani elementi, med katerimi na določenem koraku izbiramo, temneje pa element na katerega se premaknemo. Pri določanju šiva paralelizacija ni mogoča. Pri brisanju šiva pa so vse operacije med sabo neodvisne in se lahko vsi prepisi izvedejo vzporedno.

Algoritem 'zmanjša' tabelo slike in energetske analize. Slednjo uporabimo za vhod pri računanju tabele (pod)šivov pri brisanju naslednjega šiva. Tabela slike pa po končanem rezanju vseh šivov služi za vhod v funkcijo, ki zapiše novo sliko.

Funkcija pisanja slike ima torej na razpolago enodimenzijsko polje, v katerem so izbrisane vrednosti predstavljene z negativno celoštevilsko vrednostjo ter seveda nove in stare dimenzije slike.

Slika 3.5 [12]: rezultati implementacije metode rezanja šivov na CUDA. Desno vidimo originalni sliki, levo pa sliki za tem, ko smo ju zmanjšali z našo implementacijo metode rezanja šivov na CUDA

3.2 PARALELIZACIJA ALGORITMA

Opisan algoritem rezanja šivov je bil najprej razvit v C++, nato pa paraleliziran s pomočjo arhitekture CUDA (poglavje 2.2). Pri razvoju sta bila glavna cilja identični rezultati (slika energetske analize ter nova slika) ter čim večja paralelizacija prvotnega algoritma. Tak pristop nam omogoča učinkovito primerjavo obeh izvedb, ki je na voljo v poglavju 4.

Skozi algoritem se izkaže, da je večino problemov mogoče rešiti z vsaj delno paralelnim izvajanjem. Obstaja le en del, ki ga je zaradi medsebojne odvisnosti korakov potrebno izvesti zaporedno.

V nadaljevanju si bomo ogledali algoritem v istem vrstnem redu, kot je opisan v poglavju 3.1. Ob tem samega algoritma ne bomo ponovno opisovali, temveč se bomo osredotočili na vzporedne operacije znotraj algoritma.

3.2.1 Alokacija pomnilnika na napravi

Vzporedne funkcije se v CUDA izvajajo na multiprocesorjih grafične enote. Izvajajo jih posebne funkcije imenovane jedra. Več o jedrih si lahko preberemo v poglavju 2.2.1. Polja, ki jih želimo uporabiti v teh funkcijah, moramo predhodno prekopirati v napravo. Za polja namenjena rabi na napravi se uporabljajo imena oblike d_imePolja. Tako se jih loči od ostalih polj. Več o jedrnih funkcijah in CUDA si lahko preberemo v poglavju 2.2.

Algoritem najprej rezervira prostor za vse tabele na napravi. Cilj je izvajanje celotnega algoritma znotraj naprave, saj je kopiranje podatkov v napravo in nazaj relativno zamuden proces, ki postane problematičen, ker bi ga morali ponavljati ob vsakem rezanem šivu. Tako bi ob kombiniranju dela CPE ter GPE morali na vsakem koraku (novem šivu) kopirati uporabljane tabele iz grafične enote v pomnilnik ter po končanem delu CPE nazaj v grafično enoto.

Čas kopiranja posamezne tabele tipa int dimenzije x=2050 pik in y=1530 pik v napravo znaša na uporabljenem sistemu (poglavje 4.1) 8 ms, čas kopiranja iz naprave pa 9 ms. Ob zožanju širine slike za 500 pik nam zakasnitev zaradi kopiranja znaša: (8+9)*500 = 8500 ms oz. 17 ms na korak. To pa presega prednosti, ki bi jih pridobili z računanjem daljšega niza zaporednih operacij na CPU. Prav tako se s izvajanjem vseh procesov na grafični enoti zmanjša število tabel v pomnilniku ter obremenitev procesne enote.

V praksi se izkaže, da uporabljen pristop zahteva le dve kopiranji. V začetku se v napravo kopira prebrana slika, na koncu pa se ustrezno 'zmanjšana' (izrezane točke na desni strani slike so predstavljene z negativnimi vrednostmi) slika prebere iz naprave in je na voljo funkciji za pisanje datoteke.

3.2.2 Porazdelitev niti

Pred izvajanjem jeder jim moramo določiti dimenzijo bloka ter dimenzijo mreže na kateri se bodo izvajali. Več o nitih in jedrih si lahko preberemo v poglavju 2.2.1.

Uporabimo dvodimenzionalno mrežo. Ker število niti določa količino vzporedno izvedenih operacij, njihovi indeksi pa položaj znotraj polja, seveda želimo, da število niti sovpada s številom elementov, ki jih želimo izračunati vzporedno. Indekse niti uporabljamo za orientacijo znotraj polja, zato moramo biti pozorni tudi na to, da ne pišemo izven rezerviranega prostora.

Tako moramo zagotoviti, da sta širina in višina bloka delitelja višine in širine slike. Dimenzija mreže (število blokov) pa se določi tako, da se z bloki pokrije celotno polje. Razmerja med dimenzijo bloka, mreže in slike podaja formula (6). Vidimo lahko, da števila niti na blok ne moremo fiksno določiti vnaprej ampak ga moramo prilagoditi dimenzijam slike.

$$dimBloka * dimMreze = dimSlike \quad (6)$$

Število niti na začetku postavimo na določene vrednosti, nato pa jih odštevamo za vsako dimenzijo bloka (število niti na blok je omejeno na 512 [6]) dokler ne pridemo do delitelja za sovpadajočo dimenzijo slike.

3.2.3 Analiza slike in brisanje šivov

Na prebrani sliki se najprej izvede energetska analiza. Iz opisa postopka (poglavje 3.1.2) je razvidno, da se vsak element izračuna neodvisno od ostalih. Energetsko analizo slike lahko z eno nitjo opravimo v O(X*Y). Če imamo na razpolago N jeder in niti, pa v O(X*Y/N).

Računanje (pod)šivov (poglavje 3.1.2) se izvaja od spodaj navzgor. Pri tem je iz opisa algoritma razvidno, da je vrednost korena (pod)drevesa odvisna od vrednosti korenov njegovih poddreves. V praksi to pomeni, da je rezultat vrstice i odvisen od rezultata vrstice i-1 in zato ne moremo naenkrat izračunati celotne tabele. Izračun tabele prikazuje slika 3.3.

Med sabo neodvisna pa so posamezna drevesa iste globine. Torej lahko vzporedno izračunamo vrednosti točk za posamezno vrstico tabele. Izračun (pod)šivov z eno nitjo ima časovno zahtevnost O(X*Y). Če imamo na razpolago N jeder in niti, pa v O(Y/N).

Rezanje optimalnega šiva razdelimo na dve funkciji. Določanje in brisanje optimalnega šiva. Določanja optimalnega šiva ne moremo izvesti vzporedno, saj so vsi koraki medsebojno odvisni. Najprej moramo izračunati minimum prve vrstice, ki je naš prvi izbrani element. Nato se na vsakem koraku premaknemo na minimalnega izmed spodnjih treh sosedov

trenutno izbranega elementa. Tako je vsak korak odvisen od rezultata prejšnjega koraka. Funkcijo sicer še vedno izvajamo na grafični enoti, da se izognemo nepotrebnemu kopiranju polj, vendar ta del izvaja ena sama nit. Kot bomo videli v nadaljevanju, gre k sreči za časovno precej nezahtevno operacijo. Za vsako vrstico pregledamo le tri elemente ter se premaknemo na najmanjšega izmed njih. Seveda pa moramo najprej poiskati minimum prve vrstice. Časovna zahtevnost iskanja minimuma prve vrstice in določanja šiva čez vse vrstice je torej O(X+Y).

Pri brisanju šiva prepišemo vse elemente tabele. Elementi pred šivom se ne spremenijo, elemente za šivom pa prepišemo z vrednostmi na njihovi desni. Ker ne moremo nadzirati vrstnega reda izvajanj niti, obstaja nevarnost, da se prepis (x+1) = (x+2) zgodi pred prepisom (x) = (x+1) in priredimo (x) = (x+2), kar pa ni pravilno. Določanje in brisanje šiva prikazuje slika 3.4.

Neodvisnost za vse celice zagotovimo z uvedbo dodatnih tabel, v katere pišemo nove vrednosti. V zgoraj opisanem primeru torej velja (x+1)' = (x+2) ter (x)'=(x+1) kar pa je pravilno, saj se vrednosti (x+1)' ter (x+1) nahajata v ločenih tabelah. Po rezanju šiva moramo seveda nove tabele prepisati nazaj v stare (x)=(x'), da se pripravimo na naslednji šiv. Obe operaciji sta pri izvajanju z eno nitjo reda O(X*Y) torej je tudi celotno brisanje šiva reda O(X*Y). Če imamo na razpolago N jeder in niti, pa v O(X*Y/N).

3.2.4 Časovna zahtevnost posameznega koraka

Po teoretični obravnavi vsake izmed glavnih funkcij si oglejmo še njihovo realno obnašanje. Izmerimo povprečno porabo časa posamezne funkcije znotraj algoritma. Za meritev smo vzeli blok niti dimenzije 182x2 ter sliko dimenzij X=2048, Y=1536. Izmerili smo čas delovanja posamezne funkcije na vsakem izmed 200 rezanih šivov ter izpisali povprečne vrednosti. Energetsko analizo smo pognali dvesto in štiristokrat v ločeni zanki. Podatki o sistemu na katerem potekajo meritve so na voljo v poglavju 4.1. Rezultate meritev nam prikazuje tabela 3.1.

tabela 3.1: povprečna časovna poraba posamezne funkcije. Za meritev smo vzeli blok niti dimenzije 182x2 ter sliko dimenzij X=2048, Y=1536.

funkcija	**čas pri rezanju 200 šivov (ms)**	**čas izvajanja na šiv (ms)**
en. analiza	~ 1	<< 1
računanje tabele (pod)šivov	13416	67
iskanje in rezanje šiva	6	< 1

Vidimo, da nam daleč največ časa pobere preračunavanje tabele (pod)šivov in ne iskanje optimalnega šiva, ki je zaporedna operacija. Pri prihodnjih izboljšavah se je torej potrebno usmeriti na pohitritev te funkcije.

Razlog leži v tem, da je iskanje optimalnega šiva relativno nezahteven postopek. Računanje tabele (pod)šivov vsebuje več pogojnih stavkov, ki služijo določanju posameznega elementa. Prav tako večkrat pišemo v globalni pomnilnik. Poleg tega se jedro kliče znotraj zanke, saj lahko naenkrat izračunamo le eno vrstico. Vse skupaj nas pripelje do tega, da na enem koraku večkrat kličemo jedro, ki vsebuje pogojne stavke ter ima relativno majhno gostoto aritmetičnih operacij, kar pa je oboje v nasprotju z načeli optimalnega programiranja za CUDA.

3.2.5 Zakasnitve in odstopanja

Paralelizacija algoritma nam poleg opisanih prednosti prinese tudi nove zamike pri izvajanju. Kot prvo je tukaj že omenjeno kopiranje tabel v naprave, ki pa smo ga zmanjšali na minimum. Poleg tega je potrebno alocirati pomnilnik za tabele na napravi (poglavje 4.2), ter ustvariti par novih tabel, ki zagotavljajo neodvisnost operacij.

Pri oceni časovne zahtevnosti smo že omenili, da je optimalno, če se nam lahko vse vzporedne operacije izvedejo v enem koraku. Pri večjih slikah to zaradi strojnih omejitev ne drži. Vsaka CUDA kartica ima omejeno število multiprocesorjev (z omejeno količino skupnega pomnilnika) in lahko posledično vzporedno izvaja omejeno število niti. Več o nitih si lahko preberemo v poglavju 2.2.2. Na izvajalne čase zato vpliva tudi število niti na blok ter omejitve strojne opreme.

Omenili bi tudi, da procesorji na GPE praviloma delujejo z manjšo frekvenco kot CPE ter da NVIDIA pri opisu arhitekture priporoča veliko količino aritmetičnih operacij v razmerju s pomnilniškimi (dostopni časi do globalnega pomnilnika), kar pa pri nas ni vedno realizirano (več je prepisovanj vrednosti v tabele in manj računanja).

Kot pri več paralelnih rešitvah problemov se tudi tukaj izkaže, da je zaradi zgoraj omenjenih zamikov paralelni pristop primernejši pri večjih slikah, kjer vzporedno izvajanje ukazov nad celotnimi tabelami pride do večjega izraza. Pri manjših slikah so izvajalni časi 'navadnega' zaporednega algoritma dovolj hitri, da omenjeni zamiki pomenijo daljši čas izvajanja. Več o tem pa si bomo pogledali v naslednjem poglavju, kjer bomo primerjali obe rešitvi ter si ogledali par dodatnih omejitev, ki se pojavijo pri delovanju CUDA algoritmov.

4. PRIMERJAVA IMPLEMENTACIJ

Že v prejšnjem poglavju smo nakazali na dejstvo, da čistega odgovora na vprašanje, katera izmed implementacij je boljša, ne bomo mogli podati. Vse je namreč odvisno od velikosti slik, ki jim želimo spremeniti velikost. Priprave na paralelno izvajanje ter klici funkcije na napravi poberejo svoj čas, zato se prednosti vzporednega računanja pokažejo šele pri večjih slikah.

Prav tako velja omeniti, da je ocena primernosti odvisna tudi od sistema, na katerem se bo algoritem uporabljal. Vse naprave, ki podpirajo CUDA namreč nimajo enakih zmogljivosti. Razlikujejo se tako v hitrosti GPE kot v hitrosti prenosa podatkov ter številu jeder. Razlike med tremi izbranimi modeli prikazuje tabela 4.1.

Podobno bi lahko povedali za ostale komponente sistema. Tako se npr. paralelni algoritem hitreje izkaže za boljšega na sistemu s šibkejšo procesno enoto. To omenjamo zato, ker pri algoritmih ki se izvajanju vzporedno na procesni enoti z več jedri podobne ugotovitve ne veljajo.

tabela 4.1: lastnosti več grafičnih kartic, ki podpirajo CUDA.
Izbrane so kartice različnih cenovnih razredov in starosti. Vir [6] in [7].

MODEL	**Frekvenca ure jeder (MHz)**	**št. multiprocesjev (x 8 = št. jeder)**	**Delovni spomin**	**število CUDA jeder**
Tesla S1070-500 (štiri GPE)	1440	120 (30 na GPE)	16GB (4GB na GPE)	940 (240 na GPE)
GeForce GTX 295 (dve GPE)	576	60 (30 na GPE)	1792 MB (896 na GPE)	480 (240 na GPE)
GeForce GTX 260	576	24	896 MB	192
GeForce 8800 GT	600	14	512MB	112

Pred izvedbo primerjalnih meritev si oglejmo dva problema, ki pestita paralelno izvedbo algoritma. To sta odvisnost dolžine izvajanja algoritma glede na porazdelitev niti v blokih ter omejitve glede na pomnilnik, ki se nahaja na grafični kartici. Odveč je omenjati, da čiste iterativne izvedbe ti problemi ne obremenjujejo. Kot bomo videli v nadaljevanju pa se CUDA izvedba pri večjih problemih kljub temu izkaže za hitrejšo.

4.1 SPECIFIKACIJE SISTEMA

Da bodo meritve v tem poglavju jasneje umeščene, si najprej poglejmo osnovne specifikacije sistema, na katerem jih bomo izvajali ter orodja, ki jih bomo pri tem uporabili.

- CPE: Intel E8200 (2.66GHz)
- pomnilnik: 4GB DDR2
- grafična kartica: GeForce 8800 GT
- sistem: 64bit Windows 7

Uporabljeni merilni pripomočki pa so:

- za merjenje časa: C++ funkcija clock()
- za analizo obremenjenosti CPE: Windows Performance Monitor [11]
- za primerjavo izvedbe: GIMP za Windows 2.6.8 [10]

4.2 UPRAVLJANJE S POMNILNIKOM

V tabeli 4.1 lahko vidimo, da starejše grafične kartice razpolagajo z relativno majhno količino pomnilnika. V opisanem algoritmu najprej na napravi alociramo ves pomnilnik, da se s tem ognemo izvajanju te operacije za vsak rezan šiv. To zagotavlja hitrejše izvajanje, vendar nam večje slike povzročijo probleme, saj prostora za vsa polja lahko zmanjka.

Kot primer si poglejmo sliko dimenzije X=5000 in Y=5000. Privzemimo, da je int velikosti 16bit = 2B. Izračunamo si lahko, da velikost primerne int tabele (X*Y*sizeof(int)) znaša 122MB, torej imamo na GF8800GT prostora le za 4 take tabele. Če pa za velikost tipa int vzamemo 4B, lahko uporabimo le 2 tabeli. Rešitev predstavlja rezerviranje in sproščanje pomnilnika znotraj funkcij, kar pa nam prinese nove časovne zamike.

Z meritvami ugotovimo, da alokacija pomnilnika za šest tabel traja 47ms. Iz tega lahko ugotovimo, da čas rezervacije prostora ene tabele znaša 7.8ms. Pri rezanju 300 šivov nam to pomeni 2350ms na tabelo. Trenutna implementacija zato uporablja alokacijo pomnilnika pred začetkom rezanja šivov.

Prav tako bi večje slike lahko obravnavali z razrezom na manjše, vendar nam to prinese še večje zamike pri izvajanju. Predvidevamo lahko, da se v dobro sestavljenih, zmogljivejših sistemih, na katerih teče algoritem na res velikih slikah, nahaja ustrezna strojna oprema.

4.3 ŠTEVILO NITI NA BLOK

Na čas izvajanja CUDA jeder močno vpliva število niti na blok. V splošnem je priporočljivo, da je število niti večkratnik 32 (velikost snopa), saj so takrat mikroprocesorji polno izkoriščeni. Na izvedbo prav tako vpliva količina skupnega pomnilnika ter število registrov, ki jih jedro zahteva na blok. [6] Ta dva parametra omejujeta število aktivnih blokov na multiprocesor, saj sta količina skupnega pomnilnika in število registrov na multiprocesor omejena [6]. V vsakem primeru pa je maksimalno število niti na blok 512.

Optimalno število niti na blok je torej odvisno od obravnavanega problema. V našem algoritmu tukaj naletimo na težavo, saj je število niti na blok odvisno od dimenzij obravnavane slike in jih zato ne moremo vnaprej optimalno določiti. Sklepamo lahko, da bomo boljše rezultate dosegli, če več niti porazdelimo v širino bloka, saj najpomembnejše jedro (glede na čas izvajanja) izvajamo le nad posameznimi vrsticami (višina bloka je v tem primeru vedno 1). Zato lahko sklepamo tudi, da bo spreminjanje širine bloka bolj vplivalo na izvajalne čase.

tabela 4.2: izvajalni časi glede na število niti na blok.
Algoritem izvajamo na dveh različnih slikah. Število odstranjenih šivov je 200.

	slika dimenzije 400x300		**slika dimenzije 2048x1635**	
vhod	**dejansko št. niti**	**čas (ms)**	**dejansko št. niti**	**čas (ms)**
8x8	8x6	908	8x8	33117
16x16	16x15	891	16x16	42392
23x22	20x20	916	16x16	42392
32x8	25x6	971	32x8	32636
64x8	50x6	897	64x8	32475
128x4	100x4	912	128x4	14039
256x2	200x2	979	256x2	10222

Tabela 4.2 nam pokaže, da točnega števila šivov ne moremo predvideti. Vhod nam predstavlja začetno število niti, ki ga v programu zmanjšamo tako, da se ujema z dimenzijo obravnavane slike, ki pa je vnaprej ne poznamo. Glede na več izvedenih meritev smo kot začetni faktor izbrali razmerje niti 128x4.

Bolj predvidljivo se obnaša druga slika, katere dimenzije ohranjajo število niti, ki je faktor 32. Prav tako lahko vidimo, da na časovno izvedbo vplivajo dimenzije bloka in ne le število niti na blok. Rezultati kažejo, da je optimizacija števila niti na blok kritičen faktor in kot tak eden izmed prihodnjih ciljev optimizacije.

Paralelna izvedba je torej hitrejša, vendar ne za vse dimenzije slik. Kaj naj storimo?

Iz dosedanjih ugotovitev lahko sklepamo, da pametna uporaba algoritma vsebuje pogojni stavek, prek katerega preveri vhodne pogoje funkcije ter se glede na njihove vrednosti odloči, katero implementacijo algoritma izbere. Tukaj se nam seveda zastavi vprašanje, kakšne so določene vrednosti pri katerih se izplača oz. je še mogoče pravilno uporabljati določeno izvedbo algoritma. Seveda je tudi tukaj veliko odvisno od specifikacij sistema in naših želja. V nadaljevanju si bomo ogledali primerjavo odločilnejših kriterijev pri tej izbiri.

4.4 MERITVE IN PRIMERJAVA

Cilj meritev je medsebojna primerjava obeh zgoraj opisanih izvedb algoritma. Kot glavne kriterije bomo definirali:

- čas izvajanja algoritma na slikah različne velikosti
- čas izvajanja algoritma glede na število rezanih šivov
- obremenitev procesorja
- primerjava spremenjenih slik z obstoječimi izvedbami algoritma

4.4.1 Čas izvajanja algoritma na slikah različne velikosti

tabela 4.3: izvajalni časi glede na število niti na blok.
Meritev se je izvajala z rezanjem 200 šivov in dimenziji blokov 128x4.

	iterativna	**paralelna**
slika dimenzije	**čas (ms)**	**čas (ms)**
400x300	537	928
640x480	1607	1595
800x600	2652	2282
1024x768	4547	4110
1500x1000	9122	5522
2048x1635	19619	14014
3000x3000	58089	28543

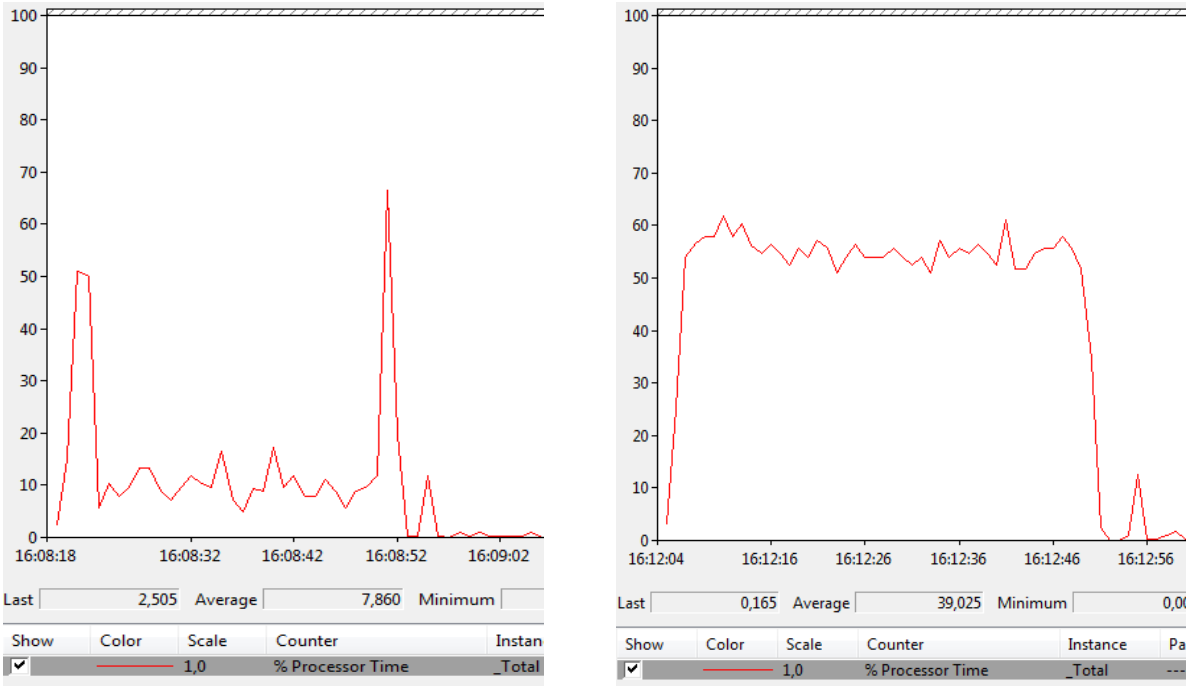
Kot smo predvidevali, se paralelna izvedba na manjših slikah izvaja dlje, medtem ko so na velikih slikah prednosti očitne. Dimenzija, pri kateri pride do prehoda, je odvisna od strojne konfiguracije sistema, na katerem poganjamo algoritma.

4.4.2 Čas izvajanja algoritma glede na število rezanih šivov

tabela 4.4: izvajalni časi glede na število niti na blok.
Vidimo, da se razlika med izvedbama s številom rezanih šivov veča.
Test smo izvajali na sliki dimenzije X=2048 Y=1536 in dimenziji blokov 128x4.

slika dimenzije	**iterativna**	**paralelna**
Izrisano število šivov	**čas (ms)**	**čas (ms)**
10	1357	814
50	5308	3603
100	10129	7086
200	19468	14003
300	28270	20846
400	36714	27643
500	44912	34408

4.4.3 Obremenitev procesorja



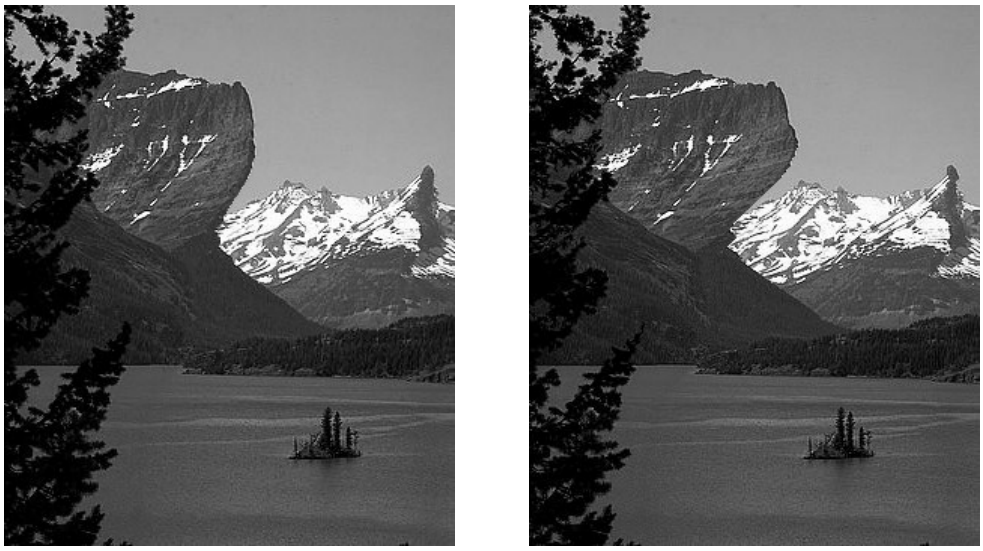
slika 4.1: primerjava obremenitve procesorja
levo: izvajanje paralelnega algoritma s CUDA, desno: izvajanje C algoritma

Test smo izvajali na sliki velikosti X=2048 Y=1536 ter blokom niti 128x4. Odstranili smo 400 šivov. Vidimo lahko, da poganjanje algoritma na CUDA kartici močno razbremeni procesor. Ker se pri izvedbi s pomočjo CUDA vse tabele, razen ene, nahajajo na grafični kartici lahko podobno ugotovitev postavimo tudi za porabo pomnilnika.

4.4.4 Primerjava spremenjenih slik z obstoječimi izvedbami algoritma

Ogledali smo si že, kako posamezen algoritem vpliva na delovanje sistema. Nismo pa še primerjali rezultatov (tj. izhodnih slik) naše izvedbe z obstoječimi rešitvami. V ta namen bomo implementacijo primerjali z Liquid Rescale filtrom programa GIMP [10]. Zanima nas podobnost izhodnih slik. V ta namen si bomo pogledali dve sliki. Prva prikazuje 'pravilno' delovanje algoritma, na drugi pa bomo primerjali slike, v katerih se pojavijo neželene distorzije. Primerjavo nam prikazujeta sliki 4.2 in 4.3.

slika 4.2 [12]: primer s slike 3.5, levo prikaz filtra Liquid Rescale, desno naša izvedba.



Slika 4.3 [12]: primer s slike 2.5, levo: Liquid Rescale, desno: naša izvedba. Vidimo, da se deformacija gorovja pojavi pri obeh različicah. Prav tako je oblika deformacije na obeh slikah podobna, vendar je na levi sliki manjša.

Vidimo lahko, da se izvedena implementacija obnaša pravilno. Manjše razlike so vidne, saj Liquid Rescale omogoča nastavljanje več parametrov (ob izdelavi slik smo jih pustili na privzetih nastavitvah). Prav tako se algoritma izvajata na podlagi različnih energetskih analiz.

5. ZAKLJUČEK

Pri obravnavi metode rezanja šivov smo ugotovili, da je rezultat metode močno odvisen od vsebine slike ter količine zmanjšanja. Boljše rezultate dobimo, ko lahko iz slike režemo ozadje, slabše pa, ko moramo za večji faktor pomanjšati glavne objekte. Prav tako so rezultati slabši, če imamo močno ozadje ali pa so v ozadju prisotne pravilne oblike (črte, krogi …), ki jih metoda popači.

Kot primerno uporabo za zmanjšanje velikih slik smo zato predlagali, da se slike za določen faktor zmanjšajo s klasičnimi metodami in pri tem ohranjajo originalno razmerje med stranicama. Nato se jih zmanjša še z metodo rezanja šivov, ki spremeni tudi razmerje stranic. Razmerje, kjer pri zmanjšanju lahko optimalno preklopimo iz ene metode, na drugo pa je zopet odvisno od slike same. Tako iz obeh metod potegnemo najboljše lastnosti in se ognemo deformacijam, ki so tipične za posamezno metodo. Izboljšano delovanje se lahko doseže tudi z uporabo nadgrajene metode rezanja šivov (improved seam carving).

Za tem smo metodo implementirali s pomočjo paralelne arhitekture CUDA, ki teče na grafičnih karticah. Ugotovili smo, da je taka izvedba hitrejša pri velikih slikah ali pri brisanju večjega števila šivov. Pri manjših slikah je sicer počasnejša, vendar so izvajalni časi dovolj majhni, da te razlike ne prekašajo precej hitrejših izvajalnih časov, ki jih dobimo na večjih slikah.

Pri tem smo izpostavili, da je hitrost izvajanja na CUDA odvisna od porazdelitve blokov na multiprocesorje, ta pa od porazdelitve niti na blok. Prav tako smo izvedli časovno analizo ter ugotovili, da večino izvajalnega časa pripada eni funkciji. Ti dve ugotovitvi smo izpostavili kot predmet prihodnje optimizacije.

Ugotovili smo tudi, da je razlika med klasično in paralelno izvedbo algoritma odvisna od uporabljane strojne opreme. Na tej podlagi predlagamo, da optimalna izvedba uporablja pogojni stavek, ki glede na velikost obravnavane slike in uporabljane strojne opreme določi katera izmed implementacij se uporabi pri izvedbi algoritma. Pri tem opozarjamo, da zaradi omejenega pomnilnika (predvsem na starejših grafičnih karticah) izvedba s pomočjo CUDA pri zelo velikih slikah ni mogoča.

VIRI

[1] S. Avidan, M. Rubinstein, A. Shamir, „ Improved Seam Carving for Video Retargeting,“ v: *ACM Transactions on Graphics*, vol. 27, no. 3, 2008.

[2] S. Avidan, A. Shamir, „Seam Carving for Content-Aware Image Resizing,“ v: *ACM Transactions on Graphics*, vol. 26, no. 3, 2007.

[3] S. Isaković, *Rezanje Šivov, Seam Carving: diplomsko delo*, Ljubljana, Fakulteta za računalništvo in informatiko, 2009.

SPLETNI VIRI

[4] (25.03.2010) CAIR – Content Aware Image Resizer. Dostopno na: http://sites.google.com/site/brainrecall/cair

[5] (22.04.2001) T.R. Halfhill, Parallel Processing with CUDA. Dostopno na: http://www.nvidia.com/docs/IO/55972/220401_Reprint.pdf

[6] NVIDIA CUDA™, *Programming Guide, Version 2.3.1,* USA, 2009. Dostopno na: http://developer.nvidia.com/object/gpucomputing.html

[7] (2010) NVIDIA spletna stran. Dostopno na: http://www.nvidia.com

[8] (2005) G. Torres, *DDR vs. GDDR Memories* . Dostopno na: http://www.hardwaresecrets.com/article/168

PROGRAMI

[9] Photoshop - http://www.photoshop.com

[10] GIMP za Windows - http://www.gimp.org/windows/

[11] Windows Performance Monitor - http://technet.microsoft.com/en-us/library/cc749249.aspx

FOTOGRAFIJE

[12] (2010) Public domain photos.com. Dostopno na http://public-domain-photos.com