UNIVERZA V LJUBLJANI
FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Rok Češnovar

Rezanje šivov na grafičnih procesnih enotah z arhitekturo CUDA

DIPLOMSKO DELO
NA UNIVERZITETNEM ŠTUDIJU

Mentor: doc. dr. Patricio Bulić

Somentor: doc. dr. Tomaž Dobravec

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: 01715/2010
Datum: 02.11.2010

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

Kandidat: **ROK ČEŠNOVAR**

Naslov: **REZANJE ŠIVOV NA GRAFIČNIH PROCESNIH ENOTAH Z ARHITEKTURO CUDA**

SEAM CARVING ON GRAPHICS PROCESSING UNITS WITH CUDA ARCHITECTURE

Vrsta naloge: Diplomsko delo univerzitetnega študija

Tematika naloge:

Implementirajte metodo rezanja šivov (Seam Carving) na grafičnih procesnih enotah (GPE) z arhitekturo CUDA (Compute Unified Device Architecture). Pri implementaciji metode rezanja šivov izberite različne postopke energetske analize slik. Implementacijo metode rezanja šivov v čimvečji meri prilagodite GPE z arhitekturo CUDa. Posebno pozornost posvetite optimalni uporabi skupnega pomnilnika za predpomnenje podatkov na procesorjih. Primerjajte čase izvajanja za različno zahtevene energetske funkcije ter skušajte oceniti, kako vpliva uporaba predpomnilnika na hitrost računanja posameznih energetskih funkcij. Primerjajte čase izvajanja metode na GPE in CPE ter poskusite ugotoviti pri katerih velikostih slik je metodo bolje izvajati na GPE. Ocenite kolikšen delež skupnega časa predstavljajo posamezni koraki metode rezanja šivov.

Mentor:

doc. dr. Patricio Bulić

Dekan:

prof. dr. Nikolaj Zimic

Somentor:

doc. dr. Tomaž Dobravec

Fakulteta za računalništvo in informatiko
UNIVERZA V LJUBLJANI
2

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Namesto te strani **vstavite** original izdane teme diplomskega dela s podpisom mentorja in dekana ter žigom fakultete, ki ga diplomant dvigne v študentskem referatu, preden odda izdelek v vezavo!

IZJAVA O AVTORSTVU

diplomskega dela

Spodaj podpisani/-a Rok Češnovar,

z vpisno tevilko 63060034,

sem avtor/-ica diplomskega dela z naslovom:

Rezanje šivov na grafičnih procesnih enotah z arhitekturo CUDA

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal/-a samostojno pod mentorstvom
  doc. dr. Patricia Bulića
  in somentorstvom
  doc. dr. Tomaža Dobravca
- 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 30.12.2010 Podpis avtorja/-ice:

Zahvala

Na tem mestu bi se zahvalil mentorjema, doc. dr. Patriciu Buliću in doc. dr. Tomažu Dobravcu, za mentorstvo, ter družini za podporo pri študiju.

Bojani in Martinu

Kazalo

Seznam uporabljenih kratic in simbolov

GPE - grafična procesna enota
CPE - centralna procesna enota
CUDA - Compute Unified Device Architecture
SM - multiprocesor (Streaming multiprocessor)
SP - skupni pomnilnik

Povzetek

Cilj tega dela je predstaviti računske zmogljivosti grafičnih procesorjev arhitekture CUDA (Compute Unified Device Architecture) na primeru algoritma za rezanje šivov. Ugotoviti želimo, če je algoritem smiselno implementirati na grafično procesni enoti. Poleg tega želimo prikazati, kako je možno izvajanje algoritma optimizirati in kakšne so razlike v zmogljivosti, če spreminjamo zahtevnosti določenega dela algoritma.

Algoritem za rezanje šivov (angl. Seam Carving), je algoritem za spreminjanje dimenzij slike, pri čemer se upošteva vsebina slike. Spreminjanje velikosti slike izvršujemo tako, da odstranjujemo t.i. optimalen šiv, to je povezana pot čez sliko, ki nosi najmanj informacije. Ugotovimo, da je uspešnost rezultatov algoritma pri spreminjanju dimenzij slik odvisna od veliko dejavnikov: števila predmetov na sliki, velikosti monotonega ozadja ter izbire energijske funkcije.

CUDA je odprta paralelna arhitektura, ki jo je leta 2007 razvilo podjetje nVIDIA. Poleg osnovne funkcije, torej grafičnega prikazovanja, lahko tiste grafično procesne enote, ki podpirajo ta sistem, izvajajo tudi splošno namensko računanje. Največja prednost izvajanja algoritmov s pomočjo GPE, je uporaba masovnega paralelizma. Poleg hitrejšega izvajanja v primeru pravilne paralelizacije nam uporaba grafično procesnih enot (v nadaljevanju GPE) omogoča tudi razbremenitev centralno procesnih enot (v nadaljevanju CPE).

Ugotovimo, da je izvedba algoritma na grafičnih procesorjih smiselna od določene velikosti slike naprej. Slednje velja tako pri povečevanju kot zmanjševanju dimenzij. Za optimizacijo izvajanja postopka je potrebno uporabiti skupni pomnilnik s predpomnenjem ter določiti optimalno velikost bloka za problem. Skupni pomnilnik nam, poleg hitrejšega dostopa do podatkov, omogoči tudi zmanjšanje števila vejitev v programu.

Ključne besede:

splošno namensko računanje na GPE, CUDA, algoritem rezanja šivov

Abstract

The purpose of this thesis is to present the computational performances of graphical processing units with CUDA (Compute Unified Device Architecture) architecture on the Seam Carving algorithm. We want to show if implementing this algorithm on graphical processing units (GPUs) is effective. We also want to show some ways of optimizing the algorithm,the results of doing so, and what are the performance differences if we change the complexity of a part of the algorithm.

Seam Carving algorithm is a content-aware image resizing algorithm. With this algorithm, resizing is done with removing the optimal seam. An optimal seam is path through the picture that carries the least information. We determine that the success of this algorithm depends on a lot of factors: the number of objects in the picture, the size of monotonous background and the energy function. CUDA is an open parallel architecture, developed in the year 2007 by the nVIDIA corporation. Besides their main function, which is the display of graphics, GPUs that support CUDA, can also be used for general purpose computation. The biggest advantage of running an algorithm on a CUDA GPU is the use of the massive parallelism. If an algorithm can be made parallel, the use of GPUs significantly improves the performance and reduces the load of the central processing units (CPUs):

We determine that implementing the Seam Carving algorithm on GPUs is effective from certain picture sizes on. The latter is true for both increasing and reducing the size of the picture. But the rate of the speed-up is not the same in both cases. We also determine that in order to optimize the performance of this algorithm we need to do the following: use the shared memory with caching and determine the optimal block size. With the use of shared memory we can access our data faster. Beside that it also enables us to remove a significant amount of branches.

Key words:
general purpose computation on GPU, CUDA, Seam Carving algorithm

Poglavje 1

Uvod

Področje procesiranja multimedije je danes eno izmed najpomembnejših področij v računalništvu. Vsakodnevne uporabe računalnika in mobilnih naprav si ne znamo več predstavljati brez funkcij prikaza in obdelave slik, videa in glasbe. Hkrati s čedalje večjim povpraševanjem torej narašča količina podatkov, ki jih mora naša naprava znati obdelati v nekem časovnem okviru. Narašča pa tudi kvaliteta multimedijskih informacij. Povprečna slika je danes predstavljena z od 5-10 miljoni slikovnih točk, v t.i. HD videu (High-Definition video, video z večjo resolucijo kot standardni video) je vsaka izmed 25 slik v sekundi predstavljena z okoli 2 miljona slikovnih točk, kar pomeni 50 miljonov le-teh na sekundo.

Vse te nove količine podatkov zahtevajo večjo procesno moč naših naprav. Že samo prikazovanje potrebuje veliko procesne moči, še bolj pa je ta potreba opazna pri obdelavi. Preden se odločimo nadgraditi strojno opremo naših naprav, se moramo vprašati, če smo izkoristili vse njihove potenciale.

V tem diplomskem delu sem se osredotočil na možnosti izkoriščanja računske moči grafično procesnih enot. Te večino časa niso polno obremenjene, njihova računska moč pa nezanemarljiva. Za prikaz primerjave računskih zmožnosti sem izbral algoritem rezanja šivov (Seam Carving). Algoritem predstavlja enega novejših pristopov za obdelavo slik, poleg tega pa v primerjavi z nekaterimi popularnimi algoritmi za obdelavo slik ni povsem trivialen. Prav tako visoka stopnja paralelizma, ki je ključna za izkoriščanje računske moči grafičnih kartic, na prvi pogled ni tako očitna.

V nadaljevanju je najprej predstavljen sam algoritem, nato okolje CUDA, ki nam omogoča pisanje programov za izvajanje na GPE, na koncu pa še implementacija samega algoritma in izmerjene zmogljivosti implementacije.

Poglavje 2

Algoritem rezanja šivov

Algoritem rezanja šivov je namenjen spreminjanju dimenzij slik, pri čemer se upošteva vsebina le-teh. Vsak dan uporabljamo veliko število naprav, ki imajo vsaka svoje dimenzije prikazovalnikov in poleg teksta je potrebno novim dimenzijam prilagoditi tudi slike. Za take prilagoditve se največ uporablja enostavno rezanje slik (angl. crop), kjer sliki enostavno odrežemo robove in prikažemo le del slike. Druga možnost pa je, da sliko zmanjšujemo tako, da ohranjamo razmerje dimenzij dokler ena izmed dimenzij ne ustreza prikazovalniku. Na ta način pa zaslon ni popolnoma izkoriščen. Slabosti teh dveh algoritmov in dobre lastnosti rezanja šivov so prikazane na sliki 2.1. V tem poglavju prikazan algoritem je povzet po [1] in [2].

Kot vidimo na sliki 2.1 algoritem rezanja šivov pri spreminjanju velikosti upošteva vsebino slike. Odstranjeno je bilo namreč monotono ozadje in ne glavni deli slike, to so veje, drevesa in odsev v jezeru. V nadaljevanju je razloženo delovanje algoritma, ki pripelje do takšnih rezultatov.

2.1 Ozadje

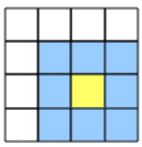
Pri krčenju ali povečevanju slik se pojavi vprašanje, katere slikovne točke izbrati za odstranitev, oziroma, kam dodati nove ob povečevanju slike. Želimo izbrati tiste slikovne točke, katerih dodajanje/odstranitev povzroči najmanjšo spremembo na sliki, torej tiste, ki imajo najmanjšo "energijo".
Energijo slikovnih točk izračunamo s pomočjo ocenjevalne funkcije. Največkrat je le-ta iz nabora gradientnih funkcij. Pri izbiri slikovnih točk je potrebno upoštevati še nekatere omejitve. Odstraniti je potrebno enako število točk v vsaki vrstici(stolpcu), da ne podremo pravokotne strukture slike. Poleg tega morajo točke biti povezane, da s tem omejimo deformacije na sliki. Ti dve

Slika 2.1: Primerjava algoritmov za spreminjanje velikosti slik. Zgoraj je prikazan izvorna slika, ki jo želimo prikazati na zaslonu manjše velikost. Levo je prikazana rešitev z enostavnim rezanjem slike, na sredini je rezultat zmanjševanja v merilu, na desni pa rezultat po izvajanju algoritma Seam Carving

omejitvi nas pripeljeta do uporabe pojmov 8-sosednosti in šivov.

8-sosednost

Dva piksla sta 8-soseda, če sta povezana preko ene izmed stranic ali vogala. Na sliki 2.2 je prikazan primer takšne sosednosti. Rumeno obarvani piksel je 8-sosed z modro obarvanimi.



Slika 2.2: Prikaz 8-sosednosti

Šiv

Šiv je povezana pot čez sliko, od točke na enemu robu slike do točke na drugemu robu. Pot med piksloma (i_0, j_0) in (i_n, j_n) nam predstavlja množica slikovnih točk:

$$\{(i_0, j_0), (i_1, j_1), ..., (i_k, j_k), (i_{k+1}, j_{k+1}), ..., (i_n, j_n)\}, \tag{2.1}$$

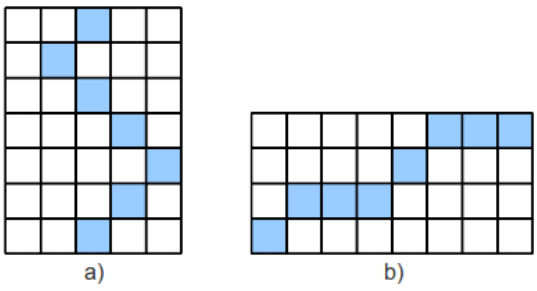
tako da sta (i_k, j_k) in (i_{k+1}, j_{k+1}) soseda za $k = 1, ..., n-1$.
Obstajata dve možnosti, navpični in vodoravni šiv. Če imamo sliko I, ki je velikosti $n \times m$, potem navpični šiv zapišemo kot:

$$S^x = \{(x(i), i)\}_{i=1}^{n}, \forall i, |x(i) - x(i-1)| \leq 1, \tag{2.2}$$

kjer je x preslikava $x : [1, ..., n] \rightarrow [1, ..., m]$. Pogoji nam določajo, da sta (i_k, j_k) in (i_{k+1}, j_{k+1}) v poti 8-soseda, in da je v vsaki vrstici izbrana samo 1 točka.
Podobno kot za navpični šiv velja tudi za vodoravnega, tega zapišemo kot:

$$S^y = \{(j, y(j)\}_{j=1}^{m}, \forall j, |y(j) - y(j-1)| \leq 1, \tag{2.3}$$

kjer je $y(j)$ preslikava $y : [1, ..., m] \rightarrow [1, ..., n]$. Sedaj ko imamo formalizirano



Slika 2.3: Prikaz navpičnega(a) in vodoravnega(b) šiva.

zapisan šiv si poglejmo še ceno šiva, torej parameter s katerim bomo določili kateri izmed vseh šivov na sliki ima najmanjšo energijo. Ceno šiva lahko zapišemo kot

$$E(s) = E(I_s) = \sum_{i=1}^{n} e(I(s_i)), \tag{2.4}$$

kjer je $e(I(s_i))$ energija šiva i-te točke šiva. Iskanje šiva z minimalno energijo je torej minimizacija funkcije:

$$s^* = \min_s E(s) = \min_s \sum_{i=1}^{n} e(I(s_i)), \tag{2.5}$$

2.2 Iskanje optimalnega šiva z dinamičnim programiranjem

Problem minimizacije funkcije (2.5) rešimo z uporabo dinamičnega programiranja. Algoritem razdelimo na 3 dele:

- energijska analiza slike
- izračun kumulativnih energij
- določitev optimalnega šiva

V podpoglavjih je uporaba dinamičnega programiranja razložena na primeru iskanja navpičnega šiva na sliki dimenzije $n \times m$, povsem enak postopek velja za iskanje vodoravnega šiva, vse je le obrnjeno za 90°.

2.2.1 Energijska analiza slike

Za izvajanje energijske analize potrebujemo ocenjevalno funkcijo. Z njo za vsako točko na sliki izračunamo kako "pomemben"del predstavlja. Pomemben del pomeni, da bi odstranitev pomenila velike spremembe na sliki. Najmanj pomembne točke so tiste, ki so del večjega monotonega predela slike, najbolj pomembne pa tiste, kjer prihaja do večjih sprememb(robovi objektov). Zato na tem mestu uporabimo gradientno analizo slike, oz. iskanje robov z operatorji prvega reda. Operatorji prvega reda določijo visoko pozitivno energijo točki, ki je del robov, in energijo okoli 0 tistim točkam, ki so del monotonega področja. Pri tem delu algoritma torej za vsako točko izračunamo novo vrednost po določeni funkciji. Časovno zahtevnost je $O(n \cdot m)$.

Ocenjevalne funkcije

Prva ocenjevalna funkcija, ki sem jo uporabil pri nastajanju tega dela, je zapisana kot matematična funkcija, ostale funkcije pa so zapisane v obliki matrik, s katerimi izvajamo konvolucijo nad sliko.

- enostavna energijska analiza

$$e(i,j) = \frac{|I(i,j) - I(i,j+1)| - |I(i,j) - I(i+1,j)| - \frac{|I(i,j) - I(i+1,j+1)|}{\sqrt{2}}}{3}, \tag{2.6}$$

kjer je $I(i,j)$ osnovna vrednost slikovne točke(recimo 0...255, če gre za sivinsko sliko), $e(i,j)$ pa je energija posamezne slikovne točke.

- Sobelov operator 3x3

$$G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} * A,\ G_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} * A \tag{2.7}$$

$$G = \sqrt{G_x^2 + G_y^2} \tag{2.8}$$

- Sobelov operator 5x5

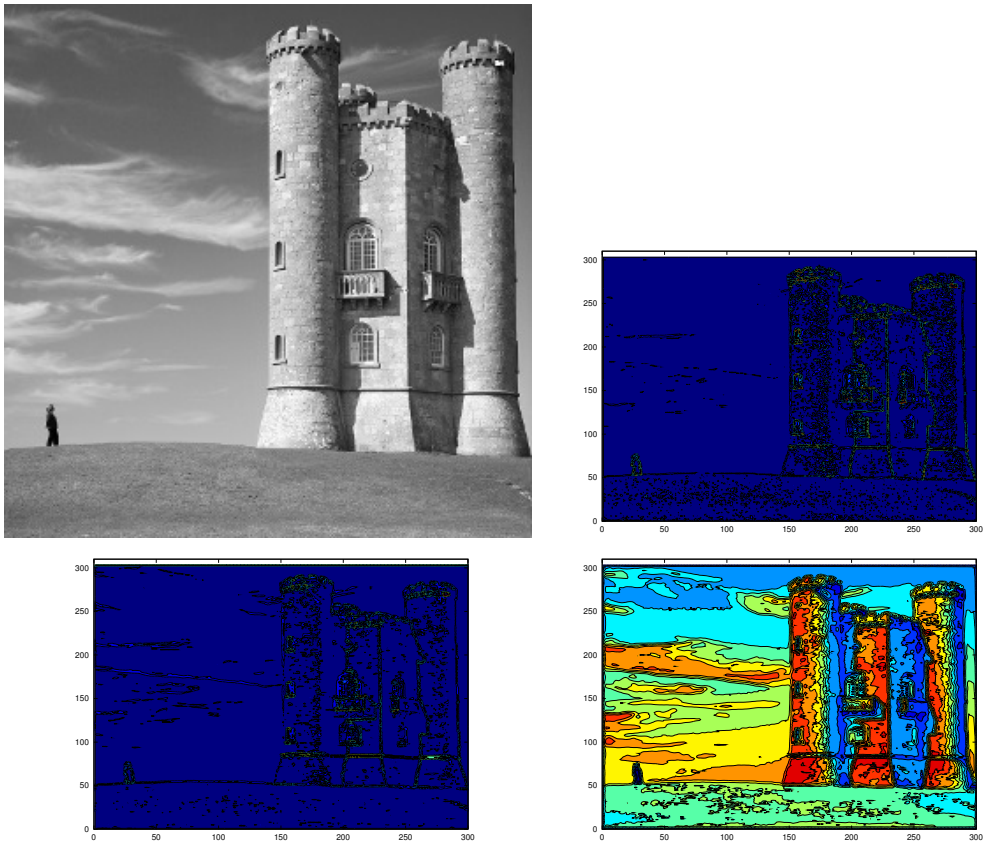
$$G_y = \begin{bmatrix} -1 & -4 & -6 & -4 & -1 \\ -2 & -8 & -12 & -8 & -2 \\ 0 & 0 & 0 & 0 & 0 \\ 2 & 8 & 12 & 8 & 2 \\ 1 & 4 & 6 & 4 & 1 \end{bmatrix} * A,\ G_x = \begin{bmatrix} 1 & 2 & 0 & -2 & -1 \\ 4 & 8 & 0 & -8 & -4 \\ 6 & 12 & 0 & -12 & -6 \\ 4 & 8 & 0 & -8 & -4 \\ 1 & 2 & 0 & -2 & -1 \end{bmatrix} * A \tag{2.9}$$

$$G = \sqrt{G_x^2 + G_y^2} \tag{2.10}$$

V slednjih primerih $*$ predstavlja konvolucijo, A vhodno sliko,G pa energijsko analizirano sliko. Primerjava ocenjevalnih funkcij sicer ni glavna tema tega dela, vendar ob razlagi samega algoritma velja omeniti, da lahko izvajamo energijsko analizo z veliko različnimi funkcijami. V tem delu so različne ocenjevalne funkcije pomembne bolj iz stališča časovne zahtevnosti. Enostavna energijska analiza zahteva samo 5 odštevanj in 2 deljenja, medtem ko pri Sobelovem operatorju 3x3 konvolucija zahteva 4 množenja(množenje z 1/-1 štejemo kot seštevanje/odštevanje), pri operatorju 5x5 pa 32 kar množenj.

Na sliki 2.4 so prikazani rezultati energijskih analiz za posamezne ocenjevalne funkcije. Kot vidimo sta rezultata, ki jih pridobimo z osnovno energijsko analizo in Sobelovim operatorjem 3x3 zelo podobni. Podobni so tudi končni rezultati celotnega algoritma, saj se šiva, pridobljena s tema analizama, razlikujeta v manj kot 10% pikslov.

Izstopa pa rezultat pri uporabi Sobelovega operatorja 5x5. Tu dobi monotono ozadje bistveno večje vrednosti v primerjavi s prejšnjima. To ima za posledico dejstvo, da je Sobelov operator primeren za uporabo v izredno redkih primerih. Kot je že bilo navedeno, je bil vključen v to delo zaradi bistveno večje časovne zahtevnosti kot prva dva.



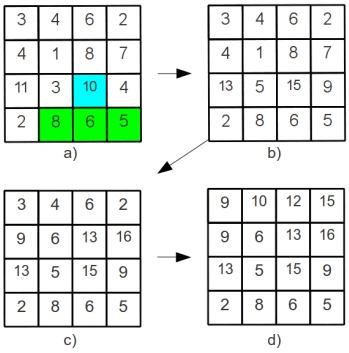
Slika 2.4: Primerjava energijskih analiz različnih operatorjev. Na sliki levo zgoraj je predstavljena izvorna slika, desno zgoraj rezultat enostavne energijske analize, levo spodaj je energijska analiza s Sobelovim operatorjem 3x3, desno spodaj pa s Sobelovim operatorjem 5x5.

2.2.2 Izračun minimalnih kumulativnih energij

Minimalna kumulativna energija posamezne slikovne točke je enaka:

$$e_k(i,j) = e(i,j) + \min(e_k(i-1,j+1), e_k(i,j+1), e_k(i+1,j+1)), \quad (2.11)$$

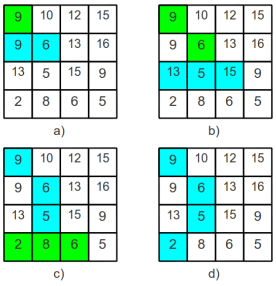
kjer je e_k minimalna kumulativna energija slikovne točke in velja $e_k(n,j) = e(n,j)$. Izračun začnemo v vrstici $n-1$ in ponavljamo vse do prve vrstice. Primer izračuna je prikazan na sliki 2.5. Časovna zahtevnost izračuna minimalnih kumulativnih energij je enaka številu točk na sliki, torej $O(n \cdot m)$.



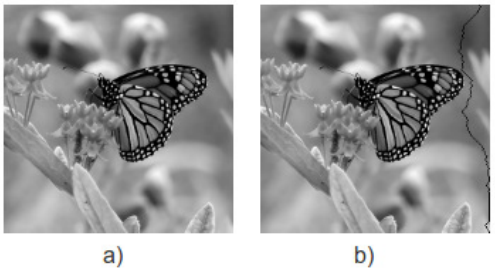
Slika 2.5: Izračun minimalnih kumulativnih energij. a) prikazuje kako se izračuna min. kumulativna energija za točko (3,3). Pri tej se vzame minimalno vrednost izmed zeleno obarvanih slikovnih točk. Kot vidimo na b), je bila izbrana vrednost točke (4,4). Na slikah c) in d) so vidni rezultati izračuna minimalnih kumulativnih energij v zgornjih dveh vrsticah.

2.2.3 Določitev optimalnega šiva

Kot vhod za določitev optimalnega šiva nam služijo vrednosti kumulativnih minimalnih energij. Najprej v n-ti vrstici najdemo najmanjšo vrednost kumulativne energije in točko s to vrednostjo označimo za del našega šiva. Šiv nato iščemo po vrsticah navzdol, od vrstice $n-1$ do 1. V vrsticah $n-1$ do 1 poiščemo vse 8-sosede tiste točke, ki smo jo določili za šiv v prejšnji vrstici. Izmed teh sosedov izberemo točko z najmanjšo vrednostjo in jo določimo za novo točko na šivu. Nato ponovimo postopek za naslednjo vrstico. Primer določitve je prikazan na sliki 2.6, končni rezultati iskanja optimalnega šiva na realni sliki, pa na slikah 2.7.



Slika 2.6: Določitev optimalnega šiva. Na sliki a) vidimo, da smo v prvi vrstici izbrali točko z minimalno vrednostijo. V prvi vrstici tako izbiramo med 8-sosedi, ki so obarvani modro. na slikah b), c) in d) je prikazano nadaljevanje postopka. Modro obarvane točke predstavljajo naš optimalni šiv.



Slika 2.7: Primer optimalnega šiva. Slika a) prikazuje vhodno sliko, slika b) pa vhodno sliko z označenim optimalnim šivom. Vidimo, da šiv potek po najbolj monotonem delu slike.

2.3 Zmanjševanje dimenzij slike

Poglejmo sedaj uporabo algoritma za iskanje optimalnega šiva pri zmanjševanju dimenzij slike. Ko enkrat poznamo optimalni šiv, sliko zmanjšamo po eni izmed dimenzij tako, da odstranimo točke, ki se nahajajo na šivu. Če želimo sliko zmanjšati za k točk, postopek k-krat ponovimo.

V primeru, da želimo sliko zmanjšati po obeh dimenzijah v vsakem koraku izračunamo kumulativne energije tako v vodoravni kot navpični smeri. Ustvarimo 2 kopiji rezultatov energijske analize, na prvi izvedemo izračun minimalne kumulativne energije za navpični šiv, na drugi kopiji pa za vodoravni šiv.
Glede na to, v kateri kopiji dobimo manjšo minimalno kumulativno energijo, se odločimo kateri izmed šivov bomo odstranili.

Slika 2.8: Primer zmanjševanja slike po obeh dimenzijah. Kot vidimo, metulj ostaja enake velikosti, odstrani se predvsem ozadje.

V primeru da želimo zmanjšati širino slike za k in višino za p točk, ni potrebno $k \cdot p$-krat računati oba tipa kumulativnih energij, pač pa do trenutka, ko odstranimo bodisi k navpičnih, bodisi p vodoravnih šivov. Potem nadaljujemo po postopku za zmanjševanje po eni dimenziji.

2.4 Povečevanje dimenzij slike

Pri povečavanju slike dodajamo nove slikovne točke v sliko. Pri širjenju slike za 1 izvedemo iskanje optimalnega šiva, nato točke desno od šiva pomaknemo za 1 v desno in vstavimo nove vrednosti. Vrednost novih točk je povprečna vrednost točke na šivu ter točk levo in desno od nje. Enako velja za povečevanje slike po višini za 1.

Če želimo sliko povečati za k, ne smemo samo ponavljati zgoraj opisanih korakov, ker bi ob ponovnem računaju optimalnega šiva dobili enak šiv kot v prejšnjem koraku. To bi pomenilo, da vseskozi vstavljamo enake(oz. zelo podobne) točke. Zato pri širjenju za k poiščemo k najbolj optimalnih šivov in nato v istem koraku izvedemo povprečenje in vstavljanje novih pikslov. Za širjenje v več korakih se odločimo le v primeru, da želimo sliko razširiti za veliko točk, recimo za več kot 50% osnovne velikosti. Takrat najprej v prvem koraku izvedemo širjenje za $k/2$, ponovno izvedemo energijsko analizo in poiščemo preostalih $k/2$ optimalnih šivov.

Poglavje 3

CUDA

V tem poglavju so razloženi principi sistema CUDA in GPE, ki sem jo uporabljal pri testiranjih v nadaljevanju. Podanih je toliko podrobnosti, kot jih je potrebno razumeti ob branju tega diplomskega dela, predvsem rezultatov meritev.

nVIDIA CUDA (Compute Unified Device Architecture) je odprta paralelna arhitektura, ki jo je leta 2007 razvilo podjetje nVIDIA. Gre za računski pogon, ki je uporabljen v novejših grafičnih procesorjih tega podjetja in je dostopen razvijalcem programske opreme preko standardnih programskih jezikov. Večinoma se uporablja *C for CUDA*, to je programski jezik C z razširitvami Nvidie. Obstajajo še ovoji (angl. wrappers) za druge jezike, kot so Java, Python, Fortran ter za okolje Matlab.

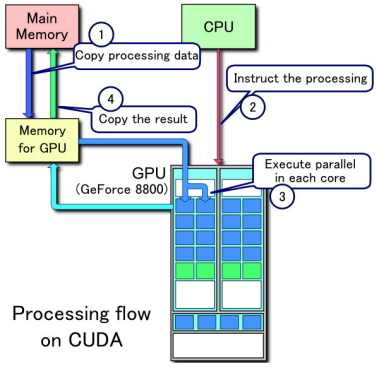
CUDA daje razvijalcem dostop do omejenega ukaznega nabora in pomnilniških resursov GPE. Izkorišča prednosti novejših grafičnih procesnih enot proti centralnim procesnim enotam današnjih računalnikov. Glavni razliki v arhitekturi CPE in GPE sta v deležu tranzistorjev, ki so namenjeni procesiranju, ter v številu opravil, ki jih lahko izvaja vzporedno. Prva razlika je vidna na sliki 3.1. Z oranžno barvo je označen pomnilniški del, z rumeno kontrolni, zelena pa predstavlja aritmetčino logični del procesne enote.

Slika 3.1: Delež tranzistorjev namenjen procesiranju

3.1 Izvajanje programa z vidika CPU

Na sliki 3.2 je prikazan primer uporabe grafičnega procesorja. Izvajanje je razdeljeno v štiri faze:

1. Prenos podatkov iz glavnega pomnilnika v pomnilnik GPE
2. CPE ukaže GPE naj izvrši določena opravila
3. Grafična procesna enota izvrši procesiranje
4. Prenos podatkov z pomnilnika GPE v glavni pomnilnik



Slika 3.2: Zaporedje dogodkov pri uporabi GPE

3.2 Izvajalni model

Programska koda, napisana za sistem CUDA, je razdeljena v 2 enoti, ščepce in serijsko kodo. Serijska koda se izvaja na CPE in vsebuje predvsem ukaze za prenos podatkov na(iz) GPE ter klice ščepcev. Sicer pa lahko serijska koda vsebuje tudi dele algoritma, ki jih želimo izvajati na CPE.

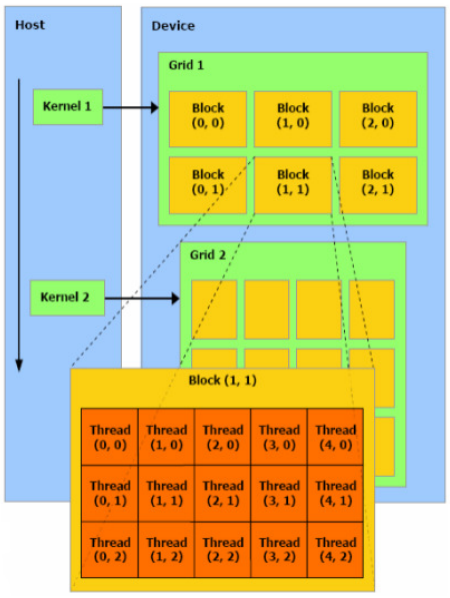
3.2.1 Ščepec(kernel) in organizacija niti

Ščepci(angl. kernels) so deli programske kode, ki jih želimo izvajati na GPE(glej programsko kodo 1). Naenkrat se na GPE izvaja samo eden. Sestavlja ga veliko število niti. Te so organizirane v bloke (do največ 512 niti na blok), ki so 1, 2 ali 3D strukture, in med seboj komunicirajo preko skupnega pomnilnika. Bloki so nadalje razporejeni v 1 ali 2D strukturo, imenovano mreža(angl.

grid). Celotna razdelitev je prikazana na sliki 3.3. Niti običajno organiziramo tako, da je velikost bloka konstanta, število blokov in njihova organizacija pa odvisna od velikosti podatkov(problema).

```
1  __global__ void izvediNaGPU(float* spr1, int spr2){
2      ....
3      //izvajaj  operacije nad podatki na GPU
4      ...
5  }
6  int main()
7  {
8      //definirali smo 3D strukturo niti v bloku 2x2x4
9      //in 2D strukturo blokov v mrezi
10     dim3 dimthreads(2,2,4);
11     dim3 dimblocks(5,5);
12     izvediNaGPU<<<dimblocks, dimthreads>>>(vhod1,vhod2);
13 }
```

Listing 1: Primer ščepca in njegovega klica

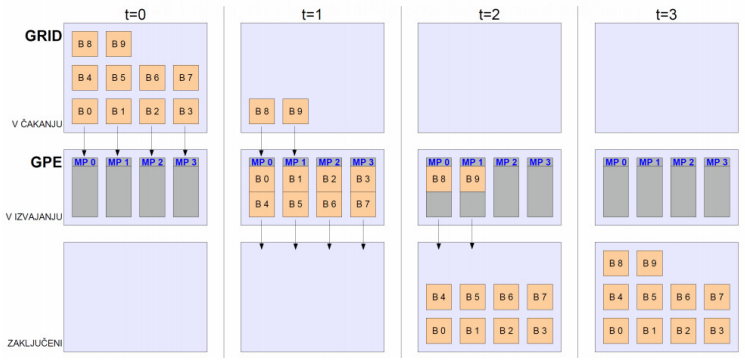


Slika 3.3: Organizacija niti

3.2.2 Zgradba GPE in porazdelitev izvajanja niti

Vsaka grafično procesna enota je fizično zgrajena iz več multiprocesorjev SM (Streaming multiprocessors), te pa se delijo na večje število izvajalnih enot SP(Streaming processors, včasih tudi jedra). Med SM se pri izvajanju ščepca porazdelijo bloki. Vsak blok se izvaja samo v enem SM. Niti znotraj bloka se razdelijo med izvajalne enote(SP). Najpomembneje pri dej delitvi je omeniti, da imajo niti znotraj bloka skupen pomnilnik, preko katerega komunicirajo, ter, da je niti možno sinhronizirati le na nivoju posameznega bloka, blokov med seboj pa ni možno.

Na sliki 3.4 je prikazano izvajanja blokov na enostaven način. Kot vidimo se v danem trenutku hkrati izvaja določeno število blokov. Nekateri čakajo na proste SM, ostali pa so se že zaključili.



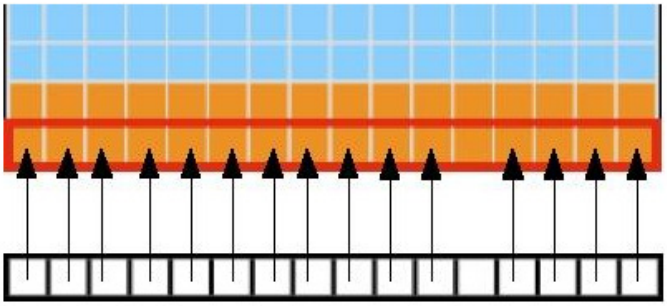
Slika 3.4: Izvajanje blokov

3.3 Pomnilniška hierarhija

Na GPE srečamo več vrst pomnilnikov, za razumevanje tega diplomskega dela je potrebno predvsem razlikovati med glavnim pomnilnikom in skupnim pomnilnikom.

Do glavnega pomnilnika lahko dostopajo vse niti, preko njega tudi komunicirajo z nitmi drugih blokov. Vsak blok pa za komunikacijo med nitmi znotraj njega uporablja skupni pomnilnik.

Še pomembnejša razlika med tema dvema tipoma pomnilnika je v časih dostopa, tej so bistveno krajši, če dostopamo do skupnega pomnilnika, saj je le-ta predpomnjen in se nahaja v samem čipu GPE, medtem ko je globalni pomnilnik zunaj. Ko niti v bloku dostopajo do globalnega pomnilnika, jih



Slika 3.5: Dostop niti do zaporednih lokacij globalnega pomnilnika

GPE razdeli v t.i. polsnope(angl. half-warps). Polsnopi pri večini različic GPE vsebujejo 16 niti. Če niti v polsnopu dostopajo do 16 zaporednih lokacij(primer na sliki 3.5) se le ta opravi z enim branjem, tudi v primeru da so lokacije med nitmi premešane. Slednje sicer velja le pri novejših različicah GPE, starejše bi v primeru permutiranih dostopov do 16 zaporednih lokacij dostop razdelile v 16 samostojnih dostopov, kar seveda bistveno upočasni delovanje.

3.4 Tesla C1060

Tesla C1060 je izdelek podjetja nVIDIA, ki sem ga uporabljal pri testiranjih in meritvah, ki so prikazana v nadalnjih poglavjih. Na tem mestu so naštete bistvene lastnosti te grafične kartice:

- vsebuje 30 multiprocesorjev(SM) s frekvenco delovanja 1.3GHz
- skupno število izvajalnih enot(SP) je 240
- vsak blok ima na voljo 16384B skupnega pomnilnika in 16384 registrov
- polsnop vsebuje 16 niti
- število niti na blok je 512
- velikost glavnega pomnilnika je 4GB

Poglavje 4

Implementacija

V tem poglavju se nahaja razlaga implementacije algoritma rezanja šivov za okolje CUDA. Programska koda, tako za CPE kot GPU, je bila napisana v programskem jeziku C in razširitvijo *C for CUDA* za del programske kode, ki je namenjen izvajanju na GPU. Oba programa, torej tako iterativni za CPE kot paralelni za GPU, sprejemata enake vhode, to so:

- pot do vhodne slike
- pot za izhodno sliko
- sprememba po širini (v št. slikovnih točk)
- sprememba po višini (v št. slikovnih točk)

Vsak podproblem algoritma in njegova realizacija za sistem CUDA je zapisana v posameznem podpoglavju. Tam, kjer je izvorna koda ščepca pomembna za razumevanje napisanega, je le-ta zapisana, sicer je izpuščena. V nadaljevanju poglavja višino slike predstavlja n, širino pa m.

4.1 Alokacija pomnilnika in prenos podatkov na/iz grafične kartice

Preden lahko začnemo na GPE izvajati algoritem, je potrebno alocirati pomnilnik za podatkovne strukture, ki jih bomo potrebovali na GPE.

Uporabljene strukture so:

- vhodna/izhodna slika - vektor celoštevilskih vrednosti

  Tabela nam predstavlja dejanske vrednosti slikovnih točk. Najprej iz nje črpamo podatke za energijsko analizo, ob zaključku pa v njej popravljamo podatke o sliki, torej odstranjujemo/dodajamo točke.

- energijske vrednosti - vektor števil v plavajoči vejici

  V to tabelo shranimo rezultate energijske analize, prav tako nad temi podatki izvajamo izračune kumulativnih energij. V primeru ko izvajamo zmanjševanje po obeh dimenzijah imamo 2 taki strukturi.

- šiv - vektor celoštevilskih vrednosti

  V to tabelo zapisujemo indekse točk, ki so del optimalnega šiva oz. enega izmed optimalnih šivov. Uporabimo jo ob popravljanju prvo navedene strukture.

Na začetku našega programa alociramo pomnilnik za vse zgoraj navedene strukture in izvedemo prenos vhodne slike na GPE. Ob koncu algoritma prenesemo izhodno sliko nazaj v glavni pomnilnik, da jih lahko CPE shrani na disk, nato pa sprostimo pomnilnik, ki so ga alocirale te strukture.

4.2 Energijska analiza

Energijska analiza predstavljan prvi ščepec. Pred klicem moramo določiti število blokov in niti v posameznem bloku, ter njihovo razporeditev. Ker je slika dvodimenzionalna struktura, je zaradi lažjega programiranja najbolje tudi niti in bloke razdeliti v dvodimenzionalno strukturo.

Vsakemu bloku določimo $p \times r$ niti, kakšne so vrednosti p in r je odvisno od narave reševanja problema, število blokov pa določimo glede na velikost slike. V x-osi mreže določimo $\lceil \frac{m}{r} \rceil$ blokov, v y-osi pa $\lceil \frac{n}{p} \rceil$. Skupno število niti, ki jih ustvarimo je torej $\lceil \frac{m}{r} \rceil * r + \lceil \frac{n}{p} \rceil * p$.

Ker zaokrožujemo navzgor, bomo v primeru, da dimenzije slike ne bodo deljive z p in r ustvarili več niti kot je točk v sliki. Za te niti moramo v ščepcu poskrbeti, da nam ne vplivajo na izračune. "Pravim"nitim najprej, glede na njihovo pozicijo v strukturi, določimo točko za katero bodo izračunale energijo. Nato jim zapišemo način, na katerega izračunajo energijo. Paziti je potrebno še na niti, ki računajo energijo za robne piksle, da ne dostopajo do neobstoječih lokacij. Spodaj je prikazan celoten ščepec za izračun enostavne

energijske analize. Zaradi preglednosti so izpuščene vejitve za preverjanje niti robnih pikslov.

```
1  __global__ void IzracunEnergije(const int* Input, float* Output,
       int width,int height){
2      //Input je vhodna slika, Output izhodna slika, width in height
           pa dimenzije slike
3
4      //dolocimo piksel za katerega nit izracuna energijo
5      int j =blockIdx.x*blockDim.x+threadIdx.x;
6      int i =blockIdx.y*blockDim.y+threadIdx.y;
7      int id=i*width+j;
8
9      //vejitev da odvecno ustvarjene niti ne izracunavajo vrednosti
10     if(i<height && j<width){
11         Output[id]=(abs(Input[id]-Input[id+width])+abs(Input[id]-
               Input[id+1])+abs(Input[id]-Input[id+width+1])/sqrt(2.0))
               /3;
12     }
13 }
```

Listing 2: Ščepec za izračun energije

4.2.1 Uporaba skupnega pomnilnika

Pri izračunavanju energije lahko izrabimo skupni pomnilnik, ki nam omogoča hitrejše dostopanje do podatkov. Pred samim izračunom vsaka nit v bloku prenese vrednost pripadajoče točke v skupni predpomnilnik. To lahko potem berejo preostali niti v bloku. Preden začnemo brati, je potrebno niti znotraj posameznega bloka sinhronizirati. S tem zagotovimo prisotnost podatkov v skupnem pomnilniku v nadaljevanju ščepca. Sledi izračun, ta je podoben kot prej, le da podatke pridobivamo iz skupnega pomnilnika.

Če pogledamo uporabo enostavne energijske funkcije, vidimo, da vsaka nit poleg vrednosti pripadajoče točke potrebuje tudi vrednosti nekaterih sosednjih točk. Če želimo, da tudi niti na robu bloka uporabljajo skupni pomnilnik, je potrebno ustvariti dodatne niti v bloku, ki bodo skrbele za to, da se prenesejo tudi vrednosti točk, za katere se energija sicer ne računa v tem bloku. V primeru osnovne energijske funkcije vsak blok potrebuje dodatno vrstico spodaj in stolpec desno. Definiramo $(p+1)*(r+1)$ niti na blok, kjer $p+r-1$ niti skrbi le za prenos v skupni pomnilnik. S tem smo sicer ustvarili dodatno število niti, ki nam zasedajo mesta v blokih, vendar večja hitrost dostopa do skupnega pomnilnika izniči morebitne upočasnitve zaradi dodatnih niti.

V primeru uporabe Sobelovega operatorja 3x3 je potrebno blok razširiti v vse 4 smeri za 1, pri operatorju 5x5 pa za 2.

```
1  __global__ void IzracunEnergijePP(const int* Input, float* Output,
       int width,int height){
2     //dolocimo piksel za katerega nit izracuna energijo
3     //izracun id se razlikuje zaradi uporabe vecjih blokov
4     //in uporabo nekaterih niti samo za prenos
5
6     //dolocitev indeksa slikovne tocke, kateri pripada nit
7     //vidna je razlika z scepcem brez uporabe skupnega pomnilnika (
          odstevanje blockIdx.x)
8     // pri Sobelovem operatorju 3x3 odstevamo -2*blockIdx.x - 1
9     // pri Sobelovem operatorju 5x5 odstevamo -4*blockIdx.x - 2
10    int j =blockIdx.x*blockDim.x+threadIdx.x-blockIdx.x;
11    int i =blockIdx.y*blockDim.y+threadIdx.y-blockIdx.y;
12    int id=i*width+j;
13
14    int row=threadIdx.y;
15    int col=threadIdx.x;
16    __shared__ int InpSh[18][18];
17
18    //niti ki ne smejo prenesti nicesar vpisejo 0
19    //(niti na robovih mreze nimajo sosedov)
20
21    if(j!=-1 && j!=width && i!=-1 && i!=height)
22       InpSh[row][col]=Input[id];
23    else
24       InpSh[row][col]=0;
25
26    //pocakamo da vse niti prenesejo svoje vrednosti
27    __syncthreads();
28
29    //vejitev da odvecno ustvarjene niti ne izracunavajo vrednosti
30    if(i<height && j<width && col!=0 && col!=17 && row!=0 && row
          !=17){
31       Output[id]=(abs(InpSh[row][col]-InpSh[row+1][col])
32       +abs(InpSh[row][col]-InpSh[row][col+1])
33       +abs(InpSh[row][col]-InpSh[row+1][col+1])/sqrt(2.0))/3;
34    }
35
36 }
```

Listing 3: Ščepec za izračun osnovne energijske analize z uporabo skupneg pomnilnika

4.3 Izračun minimalnih kumulativnih energij

Pri izračunu minimalnih kumulativnih energij se postopoma pomikamo po vrsticah navzgor in za vsako točko izračunamo njegovo minimalno kumulativno energijo. Na prvi pogled bi torej ustvarili bloke z r nitmi ,$\lceil \frac{m}{r} \rceil$ blokov ter vsaki niti določili pripadajoč stolpec, za katerega bi računala vrednosti. Zadostovalo bi 1 nalaganje in izvajanje ščepca.
Vendar tega ne smemo storiti, saj nimamo mehanizma, s katerim bi lahko sinhronizirali niti, ki so v različnih bloki. Sinhronizirati namreč moramo vse niti v mreži in to po vsakem izračunu, saj za izračun v i-ti vrstic potrebujemo vse rezultate iz vrstice $i + 1$.

V prejšnjem poglavju smo spoznali da se na GPE naenkrat izvaja samo 1 ščepec. Naslednji se lahko začne izvajati šele, ko vse niti izvedejo prejšnjega. Na ta način lahko dosežemo sinhronizacijo vseh niti. Rešitev je torej, da ščepec napišemo tako, da izvede izračun samo za eno vrstico. Kličemo ga $(m-1)$-krat, vsakič pa mu kot argument podamo številko vrstice za katero naj računa. Isto velja, ko izvajamo izračun vodoravnih kumulativnih energij, le da zamenjamo vlogo stolpcev in vrstic.

```
1  __global__ void kumulativaNavpicno(float* Output, int width,int
       height,int rowNum){
2          //spremenljivka i doloca vrstico za katero racunamo
3
4          int j=blockIdx.x*blockDim.x+threadIdx.x;
5          int id=rowNum*width+j;
6
7          //indeks spodnjega piksla
8          int idS=id+width;
9
10         if(j<width){
11             //izbira najmanjsega izmed
12             //8-sosedov v spodnji vrstici
13             int min=Output[idS];
14             if(j!=0)
15                 if(Output[idS-1]<min)
16                     min=Output[idS-1];
17             if(j!=(width-1))
18                 if(Output[idS+1]<min)
19                     min=Output[idS+1];
20             Output[id]=Output[id]+min;
21         }
22 }
```

Listing 4: Ščepec za izračun navpičnih minimalnih kumulativnih energij

4.3.1 Hkratni izračun vodoravnih in navpičnih kumulativnih energij

Če zmanjšujemo sliko po obeh dimenzijah, v vsakem koraku računamo oba tipa kumulativnih energij. V iterativnem programu to predstavlja bistveno upočasnitev izvajanja, v paralelnem programu pa ju lahko računamo hkrati. Pri običajnem izračunu kumulativnih energij imamo niti organizirane v eno dimenzijo, za hkratno računanje dodamo še drugo dimenzijo. Bloki v prvi vrstici mreže računajo navpično, bloki v drugi vrstici pa vodoravno kumulativno energijo. Ščepec za hkratno računanje vsebuje kodo za računaje obeh tipov kumulativnih energij, z enostavno vejitvijo niti glede na indeks bloka določimo katero izmed obeh naj izvaja.

4.4 Določitev optimalnega šiva

Ta del algoritma ni mogoče paralelizirati, vseeno ga izvedemo na GPE, saj bi drugače energijsko sliko prenašali v glavni pomnilnik, določili šiv ter sliko znova prenesli nazaj, zato da bi izvedli odstranitev šiva iz slike. Ker pa je prenos podatkov časovno zahteven tak pristop odpade. Ustvarimo le eno nit, ki zapiše indekse pikslov ki pripadajo šivu.

Pri povečevanju slik za k po eni izmed dimenzij določimo k najoptimalnejših šivov. Tudi tu ustvarimo le eno nit.

4.5 Odstranitev optimalnega šiva

Pri odstranitvi navpičnega šiva premikamo točke, ki so desno od šiva, za (i+1) pomnilniških lokacijo nazaj, kjer je i številka vrstice piksla(0..n), ostale pa premikamo za i lokacij. Pri vodoravnem šivu točke, ki so pod šivom, premaknemo za m pomnilniških lokacij nazaj. Tu ustvarimo enako število niti kot pri energijski analizi, torej $\lceil \frac{m}{r} \rceil * r + \lceil \frac{n}{p} \rceil * p$. Ker tu obstaja nevarnost, da neka nit prepiše vrednost točke, ki še ni bila prenešena imamo na GPE shranjeni dve kopiji slike, ki jih v primeru več zaporednih odstranitev uporabimo izmenično. Prvič zapišemo popravljeno sliko v kopijo št. 2, naslednjič v kopijo št. 1 in tako dalje.

4.6 Povečevanje slike - dodajanje novih pikslov

Za izvedbo tega dela algoritma potrebujemo dva ščepca. S prvim opravimo premestitev pikslov na nove pomnilniške lokacije(piksle desno od šiva oz. pod šivom prestavimo). Z drugim pa opravimo povprečenje točk okoli šiva in zapis nove vrednosti. Za prvi ščepec velja enaka razporeditev niti kot pri odstranitvi optimalnega šiva, kot tudi potreba po dveh kopijah, ki smo jo omenili. Za drugi ščepec pa potrebujemo toliko niti, kot je dolžina šiva, te niti nato razporedimo v $\lceil \frac{s}{p} \rceil$ blokov po eni dimenziji, kjer je s dolžina šiva.

Poglavje 5

Rezultati

Slikovne rezultate algoritma rezanja šivov sem prikazal že med razlago algoritma, v tem poglavju pa se osredotočimo na rezultate časovnih meritev za iterativno in paralelno različico algoritma. Prikazani so rezultati najzanimivejših meritev, predvsem tiste, na podlagi katerih sem prišel do pomembnejših zaključkov.

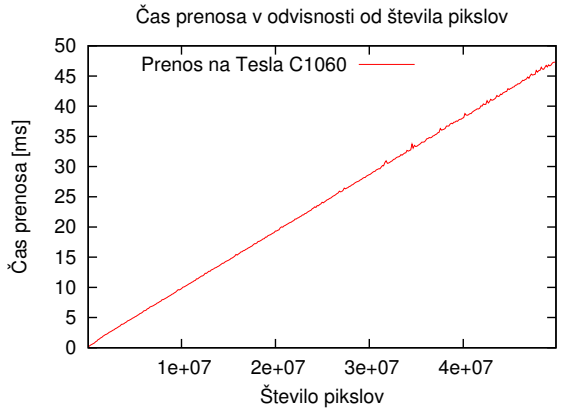
Najprej so prikazani rezultati meritev za posamezne dele paralelne izvedbe (prenos, energijska analiza, izračun kumulativnih energij, ...) ter primerjave med različnimi načini reševanja (različne ocenjevalne funkcije,uporaba skupnega pomnilnika ter različne velikosti blokov). Na podlagi teh meritev sem določil optimizirano različico paralelne izvedbe. To sem nato primerjal z iterativno izvedbo algoritma.

5.1 Prenos podatkov

Prvi del meritev, ki sem jih opravil, se nanaša na prenos podatkov v pomnilnik GPE. Graf na sliki 5.1 prikazuje čas prenosa v odvisnosti od števila točk, ki jih moramo prenesti. Kot smo že omenili so točke predstavljene z celimi števili(integer). Na grafu se vidi linearno odvisnost časa od števila točk. Za primerjavo, ob velikosti slike 256x256 je za prenos potrebnih 0.12 ms, ob velikosti 7000x7000 točk pa 47ms.

5.2 Energijska analiza

Pri energijski analizi sem najprej poskušal ugotoviti, kako velikost slike vpliva na izvajanje osnovne energijske analize. Prve meritve so nakazovale veliko

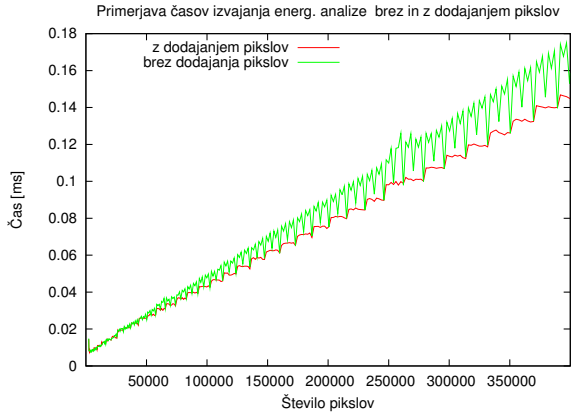


Slika 5.1: Čas prenosa podatkov v odvisnosti od števila točk na sliki.

nihanje v času izvajanja tega ščepca. Ugotovil sem, da je to v povezavi z deljivostjo dimenzij slike z 16.

5.2.1 Vpliv deljivosti dimenzij s 16 na čas izvajanja

Nihanja, ki sem jih omenil, so prikazana na grafu na sliki 5.2. Opazimo, da je čas najmanjši takrat, ko so dimenzije slike deljive z 16. V primeru ko dimenzije niso deljive s 16, so časi izvajanja večji, čeprav je število točk manjše. To nas



Slika 5.2: Čas izvajanja osnovne energijske analize ob dodajanju ničelnih točk.

pripelje do razmišljanja o tem, da bi morali sliki dodajati točke, da bi dosegli dimenzije, ki bi bile deljive z 16. Ob koncu pa teh točk nebi več upoštevali.

Ker pa je teh dodatnih točk pri velikih dimenzijah slik veliko,to pomeni, da ne smemo slike razširiti preden jo pošljemo na GPE. Potrebno je le alocirati večji del pomnilnika in pustiti ščepcem da računajo vrednosti in dostopajo do pomnilnika tudi za točke, ki nam nič ne pomenijo pri izračunu. Seveda moramo ob tem paziti, da nam dodatne niti, ki izvajajo računanje za te točke, ne vplivajo na pravilnost rezultatov. Kako se gibajo časi če sliki dodajamo, recimo jim ničelne točke, vidimo na grafu na sliki 5.2. Opazimo da so časi izrazito krajši.

To ugotovitev sem nato upošteval pri izdelavi vseh nadaljnih ščepcev.

5.2.2 Vpliv ocenjevalne funkcije na čas energijske analize

Nadalje sem želel preveriti kakšen vpliv ima kompleksnost energijske analize na čas izvajanja le-te. Vsako izmed energijskih analiz sem optimiziral z uporabo skupnega pomnilnika, odstranitvijo odvečnih vejitev ob uporabi SP in izbiro velikosti bloka. Optimizirane različice sem nato primerjal med seboj.

V nadaljevanju so, v legendah grafov, zaradi preglednosti uporabljane določene okrajšave. SP pomeni uporabo skupnega pomnilnika, črka A pa pomeni da so bile odstranjene vejitve.

Vejitve, ki jih odstranjujemo se nanašajo na robne primere, torej za niti, ki pripadajo točkam na robovih slik. Te nimajo sosedov, katerih vrednosti potrebujejo, zato jim z vejitvijo določimo, da so vrednosti sosednjih točk 0.

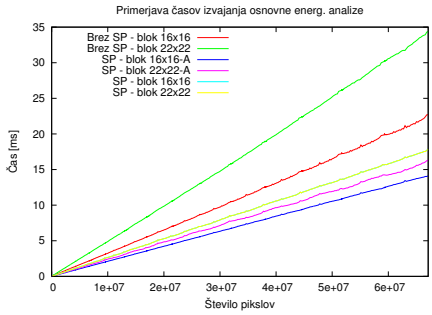
Kot smo spoznali v poglavju o implementaciji, imamo v primeru uporabe skupnega pomnilnika razširjene bloke, ki služijo za prenos potrebnih točk. Ko gre za robne bloke(bloke zadolžene za robove slike) pa nam dodatne niti v bloku ne služijo za prenos vrednosti, saj robne točke slike nimajo nekaterih sosedov. Tem nitim določimo, da v skupni pomnilnik na njihovo lokacijo vpišejo 0. To pomeni, da okoli slike dobimo dodatne točke, s katerimi se znebimo potrebe po preverjanju robnih pogojev za vsako točko. Te dodatne točke niso prisotne v globalnem pomnilniku, pač pa le v skupnih pomnilnikih blokov na robovih slike. V nadaljevanju je na grafih prikazan vpliv teh vejitev na čase izvajanja ščepcev.

Najprej si poglejmo osnovno energijsko analizo. Graf na sliki 5.3 prikazuje čase osnovne energijske analize pri uporabi pred tem naštetih optimizacij. Ugotovitve do katerih sem prišel na podlagi meritev pri osnovni energijski analizi so:

- uporaba SP je smiselna kljub majhnem številu dostopov do pomnilnika
- če ne uporabimo SP je optimalna velikost bloka 16x16
- če uporabimo SP je optimalna velikost bloka 16x16
- vpliv odstranitve vejitev na izvajanje je velik

Najprej sem opazoval vpliv velikosti bloka na čase izvajanja. Ko ne uporabimo skupnega pomnilnika, je optimalna velikost bloka 16x16. Če pa uporabimo velikost bloka 22x22, opazimo kar 50% povečanje časov izvajanja. Velik vpliv na optimalno velikost bloka ima pri osnovni energijski analizi hkratni prenos pri polsnopih in optimalna zasedenost izvajalnih enot. Ko velikosti bloka ni 16x16 to dvoje ni optimalno izkoriščeno. Če naš ščepec vsebuje vejitve za robne pogoje, se prednost hkratnega prenosa in optimalne zasednosti izvajalnih enot ne pozna na časih(časi so identični, vidimo pokrivanje na grafu), se pa pozna če vejitve odstranimo.

Samo uporaba skupnega pomnilnika, brez odstranitve vejitev, nam omogoči zmanjšanje časa izvajanja za 20%. Če odstranimo še vejitve to skupaj prinese 38 odstotno zmanjšanje časov izvajanja ščepca.



Slika 5.3: Čas izvajanja osnovne energ. analize.

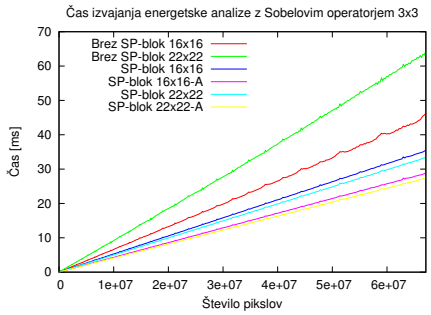
Naslednja meritev se nanaša na energijsko analizo s Sobelovim operatorjem 3x3. Graf na sliki 5.4 prikazuje čase izvajanja optimiziranih in neoptimiziranih ščepcev. Glavni razliki glede na osn. energijsko analizo so: večji delež pohitritve pri optimalni uporabi SP, ter drugačna optimalna velikost bloka (22x22). Razlog za to je, da uporaba SP pri Sobelovem operaterju zahteva večji delež niti samo za prenos, kot pri osnovni energ. analizi. Te dodatne niti pa vplivajo na izvajanje bolj, kot neoptimizirana izraba izvajalnih enot in hkratnega prenosa pri polsnopih.

Neoptimalna uporaba skupnega pomnilnika nam tu prinese cca. 20% zmanjšanje časov izvajanja. Če uporabimo optimalno velikost bloka zmanjšanje naraste na 25%. Z odstranitvijo vejitev pa dosežemo končno, 40% pohitritev. Če primerjamo to s pohitritvami, ki smo jih dobili pri osnovni energijski analizi, vidimo, da pravilna uporaba SP prinese večje pohitritve pri Sobelovem operatorju 3x3. Tudi vpliv vejitev je tu večji, kar je sicer bilo pričakovati, saj je robnih pogojev več kot pri osnovni energijski analizi.

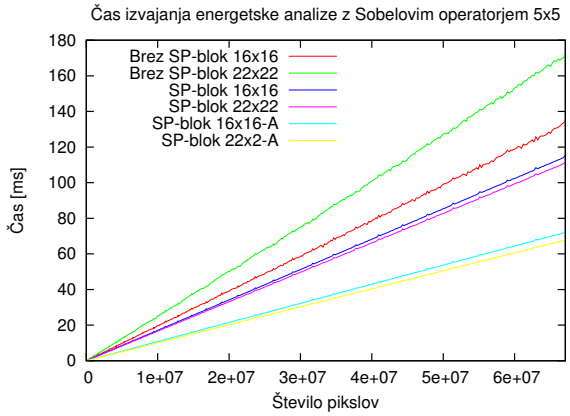
Pri Sobelovem operatorju 5x5 (časi na grafu 5.5) se izmed vseh treh energijskih analiz najbolj opazi potreba po minimalnem številu vejitev.

Neoptimalna uporaba SP nam prinese le 13% pohitritev, optimalna velikost bloka prinese še dodatna 2 odstotka. Če pa uporabimo skupni pomnilnik najoptimalneje, torej brez vejitev, vidimo zmanjšanje časov izvajanja za 50%.

Povečan vpliv vejitev je sicer pričakovan, saj je, pri uporabi Sobelovega operatorja 5x5, 3x več robnih primerov kot pri operatorju 3x3, oz. $4,8$x več v primerjavi z osnovno energijsko analizo. Optimalna velikost bloka je tudi pri tej energijskih analiz 22x22 niti.



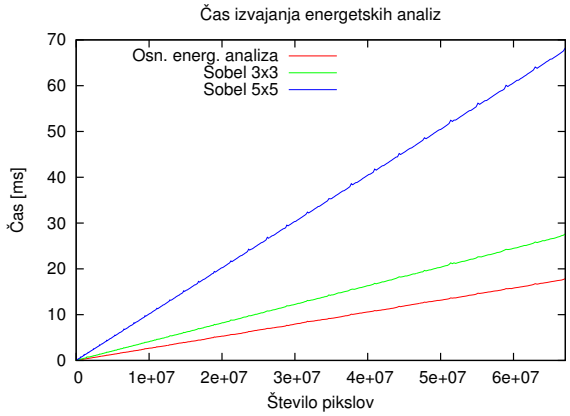
Slika 5.4: Čas izvajanja energ. analize s Sobelovim operatorjem 3x3



Slika 5.5: Čas izvajanja energ. analize s Sobelovim operatorjem 5x5

Primerjava optimalnih različic

Graf na sliki 5.6 prikazuje primerjavo časov izvajanja optimalnih različic posameznih energijskih analiz. Za najhitrejšo se po pričakovanjih izkaže osnovna energijska analiza. Za najpočasnejšo različico se izkaže uporaba operatorja 5x5. Pri velikosti slike 8192*8192 osnovna energijska analiza traja 17ms, uporaba 3x3 operatorja 27ms, uporaba 5x5 operatorja pa 68ms. Razmerje je torej $1 : 1.6 : 4$. Pri velikosti slike 64x64 pa je razmerje med časi $1 : 1.22 : 1.78$, kjer osnovna energ. analiza traja 0.09 ms. V nadaljevanju je v testiranjih bila vključena



Slika 5.6: Čas izvajanja optimiziranih energijskih analiz.

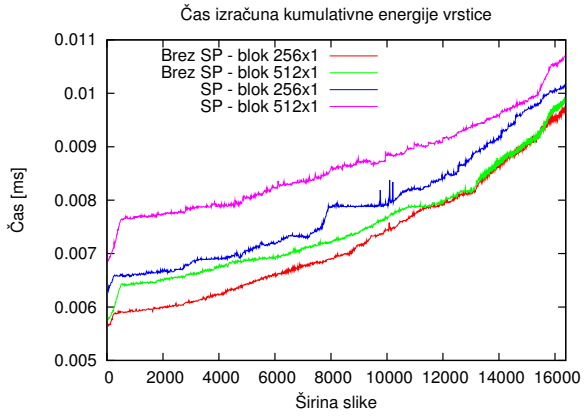
osnovna energijska funkcija. Ta se je izkazala za najhitrejšo, prav tako so njeni rezultati na slikah bili najbolj zadovoljivi.

5.3 Izračun kumulativne energije

Kot smo ugotovili v prejšnjem poglavju, posamezen ščepec za izračun kumulativne energije ni računsko zahteven. Vsaka nit prebere 3 vrednosti, naredi 2 primerjanji in seštevanje.

Pri izdelavi ščepca za izračun kumulativne energije sem izhajal iz ugotoviev, do katerih sem prišel pri prejšnjih meritvah. V različici brez uporabe skupnega pomnilnika sem najprej, za velikost bloka, vzel 256x1, enako v primeru uporabe SP, saj nam uporaba SP doda le 2 dodatni niti na 254 "pravih". Če bi bilo število dodatnih niti večje bi bila smotrna uporaba večjih blokov. Prav tako sem odstranil vse vejitve nepotrebne ob uporabi SP.

Za kontrolo sem izmeril čas tudi pri velikosti bloka 512x1. Izkaže se da je velikost 256x1 res optimalna. Graf na sliki 5.7 prikazuje čase izračuna kumulativne energije za 1 vrstico ob uporabi SP in brez. Vidimo da uporaba skupnega pomnilnika pri izračunu kumulativnih energij ni smiselna. Očitno je torej število prenosov v ščepcu premajhno, da bi njihova pohitritev odtehtala potrebo po sinhronizaciji takšnega števila niti. Ob tem pomislimo, da je morda manjša velikost bloka uspešnejša. Izkaže se da tudi takrat uporaba skupnega pomnilnika ni smiselna, časi so v tem primeru le večji, saj je izraba izvajalnih enot (Streaming processors) slabša.



Slika 5.7: Čas izvajanja izračuna kumulativne energije.

Na podlagi teh meritev se kot optimalna različica izračuna kumulativne energije pokaže izračun brez uporabe skupnega pomnilnika in velikostjo bloka 256x1.

Na podlagi zgornjih meritev seadaj lahko določimo optimalno različico implementacije rezanja šivov na GPE. Za energijski analizo uporabimo osnovno

energijsko funkcijo z uporabo skupnega pomnilnika in velikostjo bloka 16x16. Pri izračunu kumulativne energije pa uporabimo bloke velikosti 256x1 in ne uporabimo skupnega pomnilnika. Pri določitvi šiva uporabimo le eno nit, zato imamo le eno možno različico izvajanja tega dela algoritma. Preostane nam le še odstranitev šiva. Pri odstranitvi šiva posamezna nit opravi le prestavitev vrednosti iz ene v drugo lokacijo, tudi tu torej uporaba skupnega pomnilnika ni smiselna, velikost blokov pa določimo na podlagi ugotovitev dosedanjih meritev. Uporabimo velikost bloka 16x16, ki se, v primeru, da ne uporabljamo skupnega pomnilnika, vedno izkaže za najboljšo.

5.4 Deleži posameznih korakov algoritma v celotnem času izvajanja

V tem podpoglavju si bomo pogledali, kako se spreminjajo deleži posameznih korakov algoritma v celotnem času izvajanja. Tako bomo najbolje videli, kateri koraki nam predstavljajo ozko grlo. Spremembe deležev si bomo pogledali na dveh primerih, ko je dolžina šiva konstanta in ko se le-ta spreminja(spremnija se širina slike).

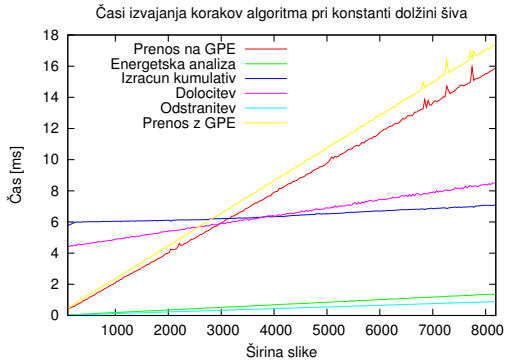
5.4.1 Konstanta dolžina šiva

Pri tej meritvi sem izvajal odstranitev navpičnega šiva pri konstanti višini in spreminajoči širini. Za višino slike sem vzel 2048 točk. Graf na sliki 5.8 prikazuje izmerjene čase izvajanja posameznih korakov za tak primer. Graf na sliki 5.11 pa prikazuje gibanje deležev posameznih korakov v celotnem času izvajanja.

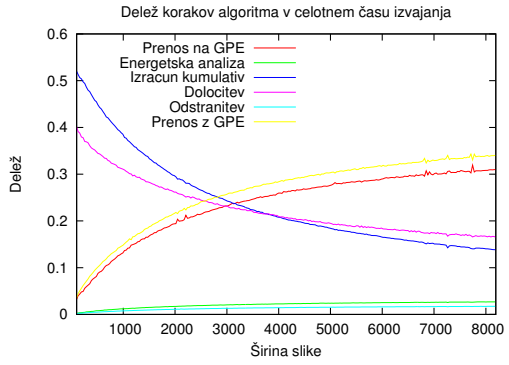
Vidimo da pri manjših širinah največ časa porabimo za izračun kumalativnih energij in samo določitev šiva. S povečevanjem začne njun delež padati. To gre na račun tega, da čas izvajanja teh dveh korakov narašča počasneje kot čas izvajanja ostalih korakih, kar je vidno na sliki 5.8.

S povečevanjem širine slike začneta prevladovati prenosa slik. Pri širini slike nad 3000 točk postane prenos največji porabnik časa paralelne izvedbe algoritma.

Delež prenosov v skupnem času se zmanjša ko izvajamo odstranitev k šivov. Takrat namreč vse korake algoritma, razen prenosov, ponovimo k krat.



Slika 5.8: Časi izvajanja korakov algoritma pri konstanti dolžini navpičnega šiva. Višina slike je 2048 točk.

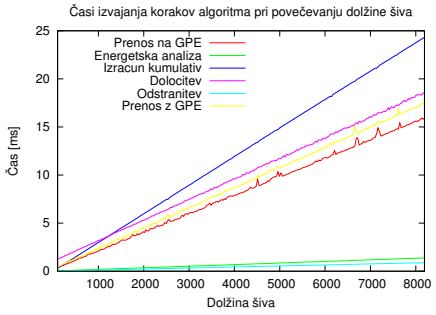


Slika 5.9: Deleži korakov algoritma v celotnem času pri konstanti dolžini navpičnega šiva. Višina slike je 2048 točk.

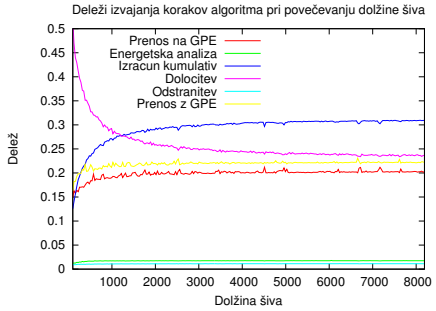
5.4.2 Povečevanje dolžine šiva

Tu sem obrnil situacijo iz prešnjega podpoglavja. Še vedno gre za odstranjevanje navpičnega šiva, tokrat pri povečevanju dolžine šiva in konstanti širini slike. Kot prikazuje graf na sliki 5.10, največ časa porabimo za določitev šiva in izračun kumulativnih energij. Na osnovi grafov na slikah 5.8 in 5.10 pridemo do ugotovitve, da sta ta dva koraka bolj odvisna od dolžine šiva(oz. višine slike), saj časi izvajanja naraščajo hitreje pri povečevanju dolžine šiva, kot pri povečevanju širine.

Za določitev šiva to dejstvo velja že za iterativno različico. Pri izračunu kumulativne energije pa je to posledica dejstva, da pri povečevanju dolžine šiva in konstanti širini dosežemo maksimalno raven paralelizacije pri dolžini šiva 1.



Slika 5.10: Časi izvajanja korakov algoritma pri povečevanju dolžine napvičnega šiva. Širina slike je 2048 točk.



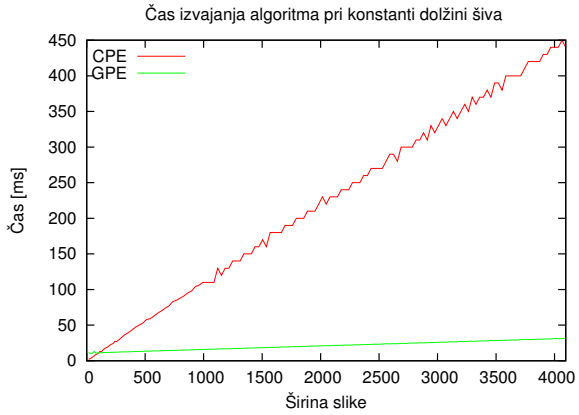
Slika 5.11: Deleži korakov algoritma v celotnem času pri povečevanju dolžine napvičnega šiva. Širina slike je 2048 točk.

5.5 Primerjava med izvajanjem na CPE in GPE

V zadnjem sklopu meritev sem izvajal primerjave med časi izvajanja algoritma za rezanje šivov na CPE in GPE. Pri tem sem prišel do ugotovitev glede dimenizij, pri katerih je paralelizacija smiselna. Primerjavo med izvajanjem algoritma na CPE in GPE sem izvedel na štiri načine: primerjava ob povečevanju dolžine šiva, širine slike, dimenzij kvadratne slike in števila odstranjenih šivov.

Če ni drugače omenjeno so bile meritve izvajane na primeru navpičnega šiva. Algoritem za CPE sem izvajal na procesorju Intel Core i5 s frekvenco ure 3.2 GHz.

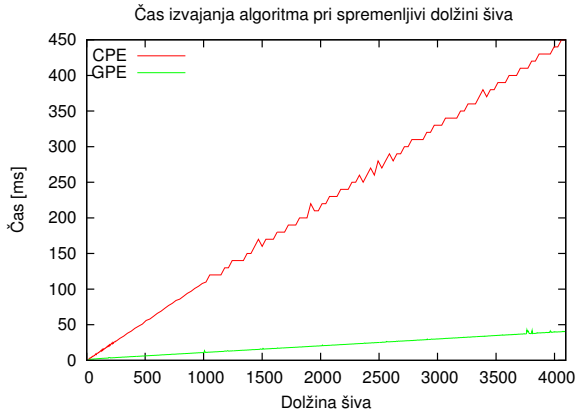
Graf, ki je prikazan na sliki 5.12 prikazuje čase izvajanja ob povečevanju širine slike in višini slike 2048 točk. Kot je vidno na sliki je pohitritev ob uporabi GPE velika. Izvajanje na GPE je, v primeru da je višina slike 2048, hitrejše od izvajanja na CPE, že pri širini slike 112 točk.



Slika 5.12: Primerjava časov izvajanja na CPE in GPE pri konstanti dolžini šiva.

Graf na sliki 5.13 prikazuje gibanje časov izvajanja če imamo, namesto konstante dolžine šiva, konstanto širino slike(v tem primeru 2048 točk), povečujemo pa dolžino šiva. Tu je uporaba GPE smiselna že pri višini slike 32 točk, torej že pri majhnih višinah slike. To gre na račun dejstva, da je maksimalna stopnja paralelizacije pri izračunu kumulativne energije dosežena že prvi višini slike 1, kar smo ga omenili že pri poglavju o deležih časov posameznih korakov.

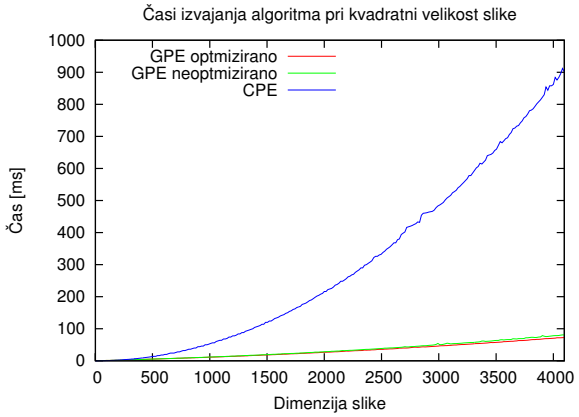
Opazimo tudi, da čas izvajanja na GPE narašča hitreje kot pri grafu na sliki 5.12. Ugotovimo da je, če velja $n > m$, pohitritev paralelne implementacije večja kot če velja $n < m$.



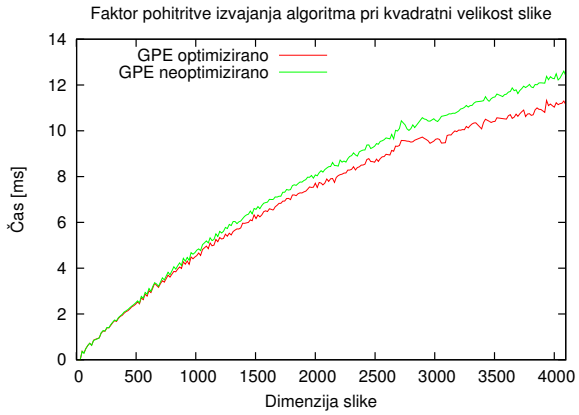
Slika 5.13: Primerjava časov izvajanja na CPE in GPE pri spremenljivi dolžini šiva.

Pri naslednji meritvi gre za primerjavo časov izvajanja v primeru da velja $n = m$. Povečujemo torej višino in širino slike. Poleg primerjave med časi izvajanja na GPE in CPE, so prikazani še časi izvajanja algoritma na GPE pri neoptimizirani implementaciji osnovne energijske analize(brez SP in vel. bloka 22x22). V primeru kvadratne dimenzije slike je uporaba GPE za izvajanje algoritma, smiselna pri velikosti slike 150x150 točk.

Graf 5.15 prikazuje faktor pohitritve obeh paralelnih implementacij glede na iterativno implementacijo algoritma. Vidimo da pohitritev pri neoptimizirani implementaciji narašča počasneje kot pri optimizirani. Pri velikosti slike 2000x2000 točk je faktor pohitritve 8.07 pri optimizirani različici in 7.71 pri neoptimizirani. Pri velikost 4000x4000 točk faktorja narasteta do 12.27 in 11.03.

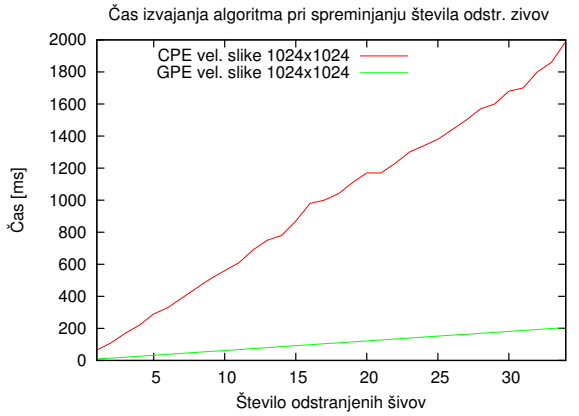


Slika 5.14: Primerjava časov izvajanja na CPE in GPE pri kvadratnih dimenzijah slike.



Slika 5.15: Faktor pohitritve paralelnih implementacij glede na CPE implementacijo

Kot zadnje si poglejmo različico algoritma, kjer se najbolj pokaže dejanska zmogljivost GPE. To je, pri odstranjevanju večjega števila šivov. Graf na sliki 5.16 prikazuje gibanje časov ob povečevanju števila odstranjenih šivov za primer velikosti slike 1024x1024. Vidimo lahko, da odstranjevanje 35ih šivov na CPE traja kar 2 sekundi, na GPE pa za enak postopek potrebujemo 200ms.



Slika 5.16: Primerjava časov izvajanja na CPE in GPE pri povečevanju št. odstranjenih šivov.

Poglavje 6

Zaključek

Implementacija algoritma za rezanje šivov na grafično procesnih enotah je, glede na opravljene meritve, smiselna. Dimenzije, od katerih dalje, je smiselno uporabiti GPE, se razlikujejo glede na način uporabe algoritma za rezanje šivov(ali odstranjujemo več šivov, dolžina šiva in drugo). Za vse variacije pa velja, da je uporaba zagotovo smiselna najmanj od slik velikosti 150x150 slikovnih točk naprej.

Opazimo da je čas izvajanja algoritma na GPE hitrejši tudi do 10 krat, kot je primer pri odsranitvi šiva pri velikostih slike 2048x4096 ali 4096x2048.

Ob preučevanju časov izvajanja ščepcev sem prišel do spoznanja, da je pri uporabi GPE potrebno biti pozoren na deljivost dimenzij slike z 16. Če dimenzije niso deljive z 16, je potrebna razširitev slike. Slednjo, zaradi časovne potratnosti prenosa, opravimo šele v pomnilniku GPE.

Ob optimizaciji posameznih ščepcev sem ugotovil, da uporaba skupnega pomnilnika ni vedno smiselna. Pri implementaciji algoritma rezanja šivov je uporaba skupnega pomnilnika smiselna le pri energijski analizi. Tu nam prinese od 40% do 50% zmanjšanje časov izvajanja ščepca.

Uporaba skupnega pomnilnika nam sicer, poleg hitrejših dostop do podatkov, omogoči tudi izločitev vejitev za robne pogoje. Ugotovimo, da je slednje ključnega pomena pri optimizaciji ščepcev. Prinese nam namreč enake ali večje pohitritve, kot sam hitrejši dostop do podatkov. Pri Sobelovem operatorju 5x5, ki zahteva največ vejitev za robne pogoje opazimo 50% manjše čase izvajanja ščepca, od tega izločitev vejitev doprinese 35 odstotkov zmanjšanja, sam hitrejši dostop pa preostalih 15.

Posebno pozornost je, poleg naštetega, potrebno posvetiti tudi optimalni velikosti blokov, saj nam neoptimalna velikost blokov povzroči tudi do 50% večje čase izvajanja ščepca.

Nadalje sem ugotovil, da največji delež časa, pri izvajanju algoritma na GPE, porabimo pri prenosu podatkov, izračunu kumulativnih energij in določitvi šiva. Pri posamezni velikosti so deleži korakov algoritma v času izvajanja, odvisni predvsem od razmerja dimenzij slike.

Največja ugotovitev, do katere sem prišel med nastajanjem tega dela, je spoznanje, da je pri izdelavi optimalnega paralelnega algoritma rezanja šivov za sistem CUDA, potrebno podrobno poznavanje arhitekture grafično procesnih enot. To ugotovitev sicer lahko razširimo na optimizacijo kateregakoli algoritma za izvajanje na GPE.

Slike

Literatura

[1] S. Avidan, M. Rubinstein, A. Shamir, "Seam Carving for Content-Aware Image Resizing" v *ACM Transactions on Graphics*, vol. 26, no. 3, 2007

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

[3] nVIDIA CUDA, *Programming Guide, Version 3.1.0*, ZDA, 2010

[4] J. Sanders, E. Kandrot, "CUDA by Example: An introduction to general-purpose GPU programming", Addison-Wesley, ZDA, 2010

[5] D. B. Kirk, W Hwu, "Programming Massively Parallel Processors", Morgan Kaufmann, ZDA, 2010