Timo Matti Iisakki Pitkänen

Heterogeenisten laskenta-alustojen käyttö kuvien segmentoinnissa

Tietotekniikan pro gradu -tutkielma

7. tammikuuta 2015

Jyväskylän yliopisto

Tietotekniikan laitos

Tekijä: Timo Matti Iisakki Pitkänen

Yhteystiedot: timo.pitkanen@iki.fi

Ohjaajat: Tuomo Rossi, Ville Tirronen ja Matti Eskelinen

Työn nimi: Heterogeenisten laskenta-alustojen käyttö kuvien segmentoinnissa

Title in English: Using heterogeneous platforms in image segmentation

Työ: Pro gradu -tutkielma

Suuntautumisvaihtoehto: Ohjelmistotekniikka

Sivumäärä: 70+26

Tiivistelmä: Kuvien segmentointi on merkittävä konenäön osa-alue. Nykyisin heterogeenisia laskenta-alustoja käytetään yhä kasvavassa määrin konenäössä. Asiasta on jo paljon tutkimusta, mutta tämä tutkimus käsittelee ongelmaa yleisesti ja liittyen uuteen nelipuumetsäsegmentointialgoritmiin, jonka rinnakkaistamisesta ei ole vielä aikaisempaa tutkimusta. Tutkimuksessa tutustutaan OpenGL:n, CUDA:n ja OpenCL:n käyttöön kuvien segmentoinnissa ja toteutetaan niillä kynnystysalgoritmi. Lisäksi toteutetaan integraalikuvien laskenta CUDA:lla.

Avainsanat: GPU, rinnakkaislaskenta, integraalikuva, kynnystys, OpenGL, OpenCL, CUDA

Abstract: Image segmentation is an important part of the computer vision research. Today, heterogeneous computing platforms are increasingly used in computer vision. There has been a lot of research on the subject, but this research deals with the problem in general and in relation to the new quad tree forest segmentation algorithm, which has not yet been parallelized. The study introduces usage of OpenGL, CUDA and OpenCL in the segmentation of images and the implementation of thresholding algorithm with them. In addition, calculation of integral images is implemented on CUDA.

Keywords: GPU, integral image, thresholding, OpenGL, OpenCL, CUDA

Termiluettelo

GPU	Graphics Processing Unit, näytönohjaimen grafiikkasuoritin, jolla suoritetaan varjostimia
GPGPU	General Purpose Graphics Processing Unit, grafiikkasuoritin, jolla voidaan ajaa myös muuta koodia
OpenCL	Open Computing Language, kirjasto, jolla voidaan tehdä rinnakkaislaskentaa useilla erilaisilla alustoilla
CUDA™	Compute Unified Device Architecture, Nvidian kirjasto, jolla voidaan ajaa rinnakkaista koodia grafiikkasuorittimilla
OpenGL	Open Graphics Library, rajapinta, jota käytetään grafiikan renderöimiseen, usein näytönohjaimella kiihdytettynä
OpenGL ES	Open Graphics Library for Embedded Systems, rajapinta, jota käytetään grafiikan renderöimiseen sulautetuilla laitteilla
OpenCV	Open Computer Vision, suosittu konenäkökirjasto
SISD	Single Instruction, Single Data, Flynnin taksonomian mukainen määrittely tavalliselle laskennalle
MISD	Multiple Instruction, Single Data, Flynnin taksonomian mukainen määrittely rinnakkaislaskennalle, jossa samalle datalle suoritetaan rinnakkain erilaisia operaatioita
SIMD	Single Instruction, Multiple Data, Flynnin taksonomian mukainen määrittely rinnakkaislaskennalle, jossa suurelle määrälle dataa suoritetaan rinnakkain samoja operaatioita
MIMD	Multiple Instruction, Multiple Data, Flynnin taksonomian mukainen määrittely rinnakkaislaskennalle, jossa suurelle määrälle dataa suoritetaan rinnakkain erilaisia operaatioita

Kuviot

Taulukot

Sisältö

1 Johdanto

> *"There is nothing more difficult to take in hand, more perilous to conduct, or more uncertain in its success, than to take the lead in the introduction of a new order of things."*
>
> *— Niccolo Macchiavelli*

Tämä tutkielma käsittelee heterogeenisten laskenta-alustojen käyttöä kuvien segmentoinnissa. Toisin sanoen tutkitaan, miten näytönohjaimella ja keskussuorittimella voidaan suorittaa osa segmentoinnin operaatioista kuvankäsittelyssä. Tutkielmassa käydään läpi yleisimmät viitekehykset ja kirjastot grafiikkasuorittimella tapahtuvalle laskennalle sekä vertaillaan niiden ominaisuuksia ja käyttötapoja. Tutkimuksessa käytetyt grafiikkasuoritinlaskennan viitekehykset ovat OpenGL, CUDA ja OpenCL, joilla luodaan pieni esimerkkiohjelma kynnystämällä tapahtuvasta segmentoinnista, ja vertaillaan viitekehysten nopeuksia kyseisessä esimerkissä. Viitekehyksistä CUDA osoittautui odotetustikin nopeimmaksi alustaksi, mikä oli aikaisempien tutkimusten perusteella odotettavissa (Karimi, Dickson ja Hamze 2010).

Käytännön sovelluksena tutkielmassa käsitellään Eskelinen, Tirronen ja Rossi (2013) esittämän algoritmin osittaista rinnakkaistamista muuntamalla algoritmissa tehty integraalikuvan laskenta rinnakkaistetuksi. Sen rinnakkaistaminen osoittautui ainakin käytetyillä testialustoilla hitaammaksi kuin alkuperäinen rinnakkaistamaton koodi. Syitä tähän pohditaan ja arvioidaan myös algoritmin loppuosan rinnakkaistamiseen vaadittavia toimenpiteitä.

1.1 Tarkasteltava algoritmi

Tässä tutkimuksessa tarkastellaan nelipuumetsäsegmentointialgoritmia (Eskelinen, Tirronen ja Rossi 2013), joka on suunniteltu matalan kontrastin omaavien kuvien segmentointiin. Algoritmi käyttää integraalikuvia intensiteetin keskiarvojen ja varianssin laskemiseksi tehokkaasti kuvan osille. Niitä käytetään luomaan nelipuumetsä, jota käytetään segmentoinnissa yhdistelemällä samankaltaiset ja yhtenevät osat.

Algoritmia on käytetty tunnistamaan autonrenkaiden kyljessä olevaa tekstiä, joka on alunperin tehty lähinnä ihmisten luettavaksi.

1.2 Tutkimuskysymys

Tutkimuskysymyksenä tässä tutkimuksessa on seuraava. Voiko Eskelinen, Tirronen ja Rossi (2013) esittämän segmentointialgoritmin rinnakkaistaa ja toteuttaa GPU-laskennalla? Tämä voidaan jakaa kolmeen osakysymykseen seuraavasti.

1. Miten kyseisen segmentointialgoritmin voi rinnakkaistaa ja toteuttaa grafiikkasuorittimella?
2. Jos algoritmin voi toteuttaa,kuinka paljon nopeampi tai hitaampi se on grafiikkasuoritinlaskennalla toteutettuna kuin rinnakkaistamattomana?
3. Jos algoritmia ei voi toteuttaa, miksi sitä ei voi toteuttaa?

Tutkimuskysymykseen pyritään vastaamaan tekemällä ensin kattava selvitys taustatiedoista ja käytettävissä olevista heterogeenisiin laskentaympäristöihin käytetyistä kirjastoista ja alustoista. Alustoina käytetään OpenCL-, CUDA- ja OpenGL-kirjastoja, joiden nopeutta ja helppokäyttöisyyttä testataan ensin rinnakkaistamalla yksinkertainen kynnystysalgoritmi. Näistä alustoista valitaan toteutuksen kannalta paras alusta. Tarkastellaan algoritmia ja tutkitaan mitkä osat siitä voi rinnakkaistaa jo olemassa olevilla menetelmillä. Tutkitaan mitä loppuosan rinnakkaistaminen vaatisi ja olisiko siitä hyötyä. Lopuksi pyritään tarkastelemaan onko algoritmin rinnakkaistaminen onnistunut ja mitä voitaisiin tehdä paremmin.

2 Taustaa

> *"The theory that can absorb the greatest number of facts, and persist in doing so, generation after generation, through all changes of opinion and detail, is the one that must rule all observation."*
>
> *— Adam Smith*

Tässä osassa käydään läpi ja esitellään tutkimuksessa tarvittava taustatieto ja tutkimukset, joihin tämä tutkimus perustuu. Tarkoituksena olisi esittää kaikki tarvittava taustatieto, jota tutkimuksen ymmärtämiseksi tarvitaan. Aluksi käsitellään rinnakkaislaskentaa yleisesti, sitten grafiikkasuoritinlaskentaa ja lopuksi konenäköä.

2.1 Rinnakkaislaskenta

Rinnakkaislaskenta tarkoittaa laskentatehtävän suorittamista tavalla, jossa osa tehtävästä lasketaan useammassa osassa samaan aikaa eikä peräkkäin, kuten tavallisessa laskennassa. Rinnakkaislaskentaa käyttävän ohjelmakoodin kirjoittaminen on yleensä vaikeampaa kuin tavallisen peräkkäislaskentaa sisältävän ohjelmakoodin. Läheskään kaikki algoritmin osat ei rinnakkaistu, koska se sisältää välttämättä peräkkäin suoritettavia osioita. Lisäksi monta yhtä aikaista prosessia on vaikea synkronoida keskenään. Usein rinnakkaislaskennan prosessit eivät valmistu samassa järjestyksessä vaan suoritusnopeus saattaa vaihdella ympäristön mukaan.

Myös *deadlock-*, *livelock-* ja *race condition* -tilanteet tuottavat perinteisesti ongelmia. Deadlock tarkoittaa tilannetta, jossa kaksi tai useampi prosessi jää jumiin tilaan, jossa esimerkiksi molemmat tarvitsevat jotain toisen käyttämää resurssia ja jäävät odottamaan toistensa päättymistä. Livelock tarkoittaa tilannetta, jossa kaksi tai useampi prosessi jää vaihtamaan tilaansa toisen prosessin vuoksi ja suoritus ei etene.

Race condition tarkoittaa tilannetta, jossa laskennan tulos riippuu siitä, kumpi prosessi valmistuu ennemmin, ja mikä ei ole ohjelmoijan suunnittelema toiminta. Rinnakkaisen koodin debuggaus on myös hankalampaa, koska monta asiaa tapahtuu

samanaikaisesti eikä ole välttämättä selvää, missä virhe tapahtuu. Virhe ei aina tapahdu samaan aikaan, riippuen prosessien valmistumisjärjestyksestä.

Rinnakkaislaskentaa voidaan suorittaa usealla eri tavalla ja se voidaan jakaa seuraavassa luettelossa oleviin tyyppeihin tai myös Flynnin taksonomian mukaan jaettuihin tyyppeihin:

- bittitason rinnakkaislaskenta (engl. *bitlevel parallel computing*), jossa suoritetaan samoja operaatioita usealle bitille,
- käskytason rinnakkaislaskenta (engl. *instruction level parallel computing*), jossa suoritetaan useita käskyjä yhtä aikaa,
- datatason rinnakkaislaskenta (engl. *data level parallel computing*), jossa suoritetaan samoja käskyjä eri datalle ja
- tehtävätason rinnakkaislaskenta (engl. *tasklevel parallel computing*), jossa suoritetaan useita tehtäviä yhtä aikaa.

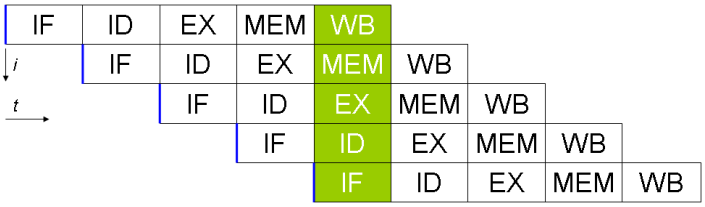
Bittitason ja käskytason rinnakkaislaskentaa suoritetaan automaattisesti nykykoneissa. Useimmat nykyiset koneet ovat 64-bittisiä ja suorittavat laskentaa 64-bittisellä ytimellä. Myös käskytason rinnakkaislaskentaa suoritetaan automaattisesti nykykoneissa, joissa monta peräkkäistä käskyä suoritetaan prosessorin liukuhihnalla. (Bernstein 1966; Barney 2010; Asanovic ym. 2006).

2.1.1 Bittitason rinnakkaislaskenta

Bittitason rinnakkaislaskennassa useita bittitason operaatioita suoritetaan rinnakkain. Nykyisissä pöytäkoneissa on usein 64-bittinen prosessori, jossa yksi käsky käsittelee rinnakkain 64 bitin verran dataa. Esimerkkinä voidaan mainita vaikka kahden 64-bittisen luvun yhteenlaskenta, joka tapahtuu nykykoneissa yhdellä konekielisellä käskyllä. Vanhemmissa koneissa on ollut 32-, 16-, 8- ja 4-bittisiä prosessoreita.

2.1.2 Käskytason rinnakkaislaskenta

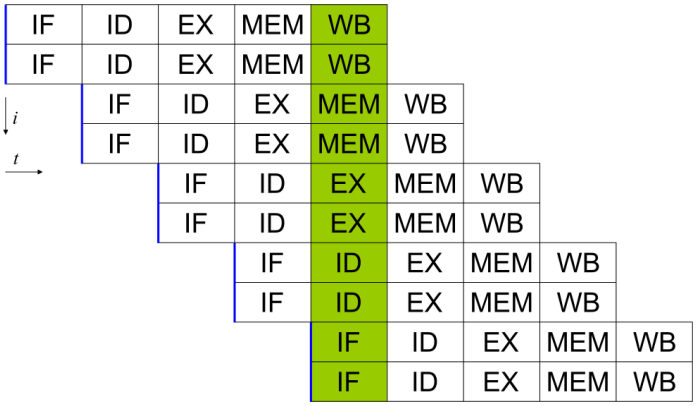
Käskytason rinnakkaislaskentaa suoritetaan nykykoneissa automaattisesti. Prosessorit voivat suorittaa liukuhihnalla olevia käskyjä rinnakkain, jos niiden laskenta on toisistaan riippumatonta. Myös useat nykykääntäjät pyrkivät tähän järjestelemällä käskyjä siten, että niitä voidaan suorittaa mahdollisimman paljon rinnakkain. Kuvio 1 esittää viiden eri käskyn suoritusta skalaarisen prosessorin liukuhihnalla. Kuvio 2 puolestaan esittää superskalaarisen prosessorin liukuhihnalla olevia kymmentä käskyä, joista aina kaksi suoritetaan rinnakkain samalla syklillä ja muut etenevät eri suoritusvaiheissa. Käskyjä voi suorittaa liukuhihnalla peräkkäin ja rinnakkain vain jos ne eivät käsittele samaa dataa.



Kuvio 1: Viisi suoritettavaa käskyä skalaarisen prosessorin liukuhihnalla, jossa käskyt ovat eri suoritustilassa (IF = Instruction Fetch, ID = Instruction Decode, EX = Execute, MEM = Memory access, WB = Register write back) Lainattu Wikipediasta Poil (2005)

2.1.3 Datatason rinnakkaislaskenta

Datatason rinnakkaislaskenta(engl. *Data level parallel computing* tarkoittaa tapaa suorittaa rinnakkaislaskentaa, jossa kukin prosessi suorittaa rinnakkain samoja käskyjä eri osalle datasta. Tällaisen laskennan edellytyksenä on tietenkin se, että dataa, jolle pitää suorittaa samat operaatiot on olemassa, kuten kuva, jossa jokaiselle pikselille suoritetaan samat operaatiot. Jos dataa jolle suoritetaan samat käskyt ei ole, ei tämän tyyppistä rinnakkaislaskentaa voida suorittaa. Tätä rinnakkaislaskennan tapaa käytetään tässä tutkimuksessa integraalikuvan laskennassa.



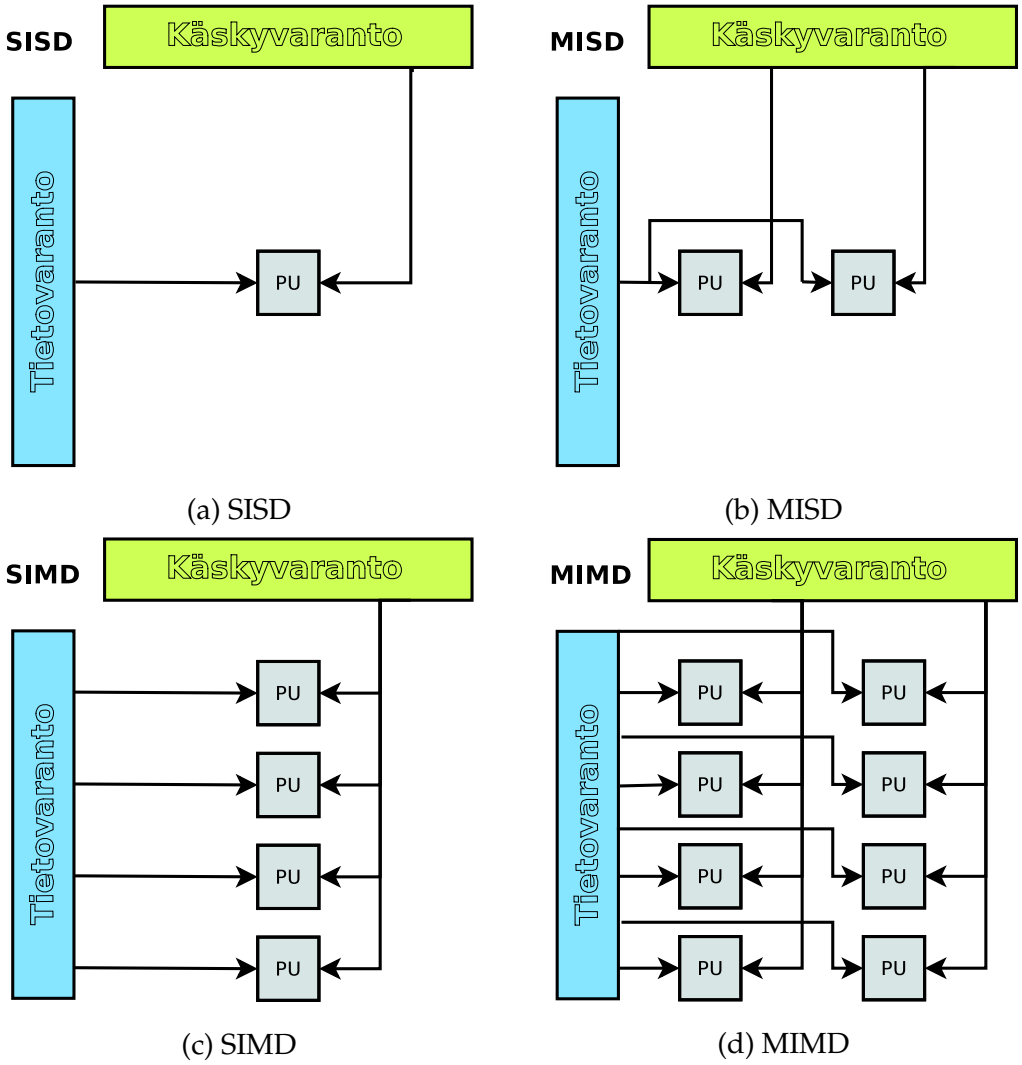
Kuvio 2: Viisi käskyä superskalaarisen prosessorin liukuhihnalla, joka suorittaa kaksi käskyä per sykli, käskyt ovat samat kuin edellisessä kuvassa 1. Lainattu Wikipediasta Poil (2010)

2.1.4 Tehtävätason rinnakkaislaskenta

Tehtävätason rinnakkaislaskenta (engl. *Task level parallel computing*) tarkoittaa tapaa suorittaa rinnakkaislaskentaa, jossa kukin prosessi saattaa suorittaa eri tehtävää. Tässä tutkimuksessa ei tulla käyttämään tehtävätason rinnakkaislaskentaa, koska tehtävätason rinnakkaislaskentaa on hyvin vaikea suorittaa grafiikkasuorittimilla käyttäen OpenGL-varjostimia (engl. *shader*). Grafiikkasuorittimella kaikki ytimet saavat yleensä saman ohjelman.

Tehtävätason rinnakkaislaskentaa voi suorittaa CUDA:lla käyttämällä CUDA streameja, mutta kaikki laitteet eivät välttämättä tue *device overlap* -ominaisuutta, joka mahdollistaa streamien käytön. Lisäksi tehtävätason rinnakkaislaskentaa voidaan suorittaa CUDA:lla, jos käytössä on useampia grafiikkasuorittimia. Tehtävätason rinnakkaislaskentaa voitaisiin suorittaa myös käyttämällä yhtä aikaa keskusprosessoria ja grafiikkaprosessoria.

2.1.5 Flynnin taksonomia



(a) SISD (b) MISD

(c) SIMD (d) MIMD

Kuvio 3: Flynn (1972) esittää erilaisia malleja, joissa osassa käytetään rinnakkaislaskentaa. Kuvissa lyhenne PU tarkoittaa suoritusyksikköä (engl. *processing unit*)

Flynnin taksonomia määrittelee (Flynn 1972) myös tavan luokitella rinnakkaislaskentaa. Se määrittelee neljä eri luokkaa:

- SISD (engl. *Single Instruction, Single Data*) yksi käsky, yksi data,
- MISD (engl. *Multiple Instructions, Single Data*) monta käskyä, yksi data,
- SIMD (engl. *Single Instruction, Multiple Data*) yksi käsky, monta dataa ja
- MIMD (engl. *Multiple Instructions, Multiple Data*) monta käskyä, monta dataa.

SIMD on datatason rinnakkaislaskentaa ja MIMD on tehtävätason rinnakkaislaskentaa. SISD ei ole rinnakkaislaskentaa ollenkaan ja MISD määrittelee hiukan oudon rinnakkaislaskennan muodon, jossa samalle datalle suoritetaan erilaisia käskyjä. Kuvio 3 esittää Flynnin taksonomian mukaiset rinnakkaislaskentatavat kaavioina. Sitä käytetään "MISD - Wikipedia, the free encyclopedia" (2013, Wikipedian sivu) mukaan esimerkiksi virheentunnistavissa tietokoneissa ja avaruussukkulan lentotietokoneessa. Käytännössä voidaan sanoa nykyaikaisen prosessorin liukuhihnan toimivan tällä tavalla, vaikka siinä käytännössä suoritetaan eri käskyjä. Sitä ei käsitellä tässä tutkimuksessa sen tarkemmin.

2.1.6 Työssä käytetyt rinnakkaislaskennan tyypit

Tässä tutkimuksessa pyritään hyödyntämään lähinnä datatason rinnakkaislaskentaa, koska sitä on helpointa suorittaa grafiikkasuorittimella. Grafiikkasuoritin on suunniteltu käyttämään juuri tämän tyyppistä rinnakkaislaskentaa. Grafiikkaprosessori suorittaa samaa ohjelmakoodia eri datalle jokaisella ytimellä. Datatason rinnakkaislaskentaa käytetään tässä tutkimuksessa rinnakkaistamaan integraalikuvan laskenta. Lisäksi tutkitaan mahdollisuutta hyödyntää tehtävätason rinnakkaislaskentaa nelipuumetsien käsittelyssä. Käskytason ja bittitason rinnakkaislaskentaa käytetään myös, mutta se tapahtuu kääntäjä- ja laitetasolla automaattisesti.

2.1.7 Amdahlin laki

Amdahlin lain määritteli Gene Amdahl jo 1967 julkaistussa tutkimuksessaan (Amdahl 1967). Amdahlin laki kertoo maksimaalisen nopeuden lisäyksen, jonka algoritmin rinnakkaistamisella voidaan saada. Nopeuden lisäystä rajoittaa osa algoritmista, jota ei voida suorittaa rinnakkain. Rinnakkaistettu osa ei siksi lisää koko algoritmin nopeutta vaan sillä on olemassa teoreettinen yläraja.

Amdahlin lain mukaan laskettu maksimaalinen nopeuden lisäys saadaan "Amdahl's law - Wikipedia, the free encyclopedia" (2013) kaavalla

$$nopeutus = \frac{1}{r_s + \frac{r_p}{n}}, \quad \text{jossa} \quad r_s + r_p = 1. \tag{2.1}$$

Kaavassa r_s on se osuus algoritmista, jota ei voi rinnakkaistaa, r_p on rinnakkaistuva osuus algoritmista ja n on rinnakkaisten prosessien lukumäärä. Kaavasta saadaan myös maksimaalinen nopeuden lisäys, jos rinnakkaisia prosesseja olisi käytettävissä ääretön määrä. Maksimaalinen nopeutus on $\frac{1}{r_s + \frac{r_p}{n}}$ ja raja-arvo on $\frac{1}{r_s}$, kun n lähestyy ääretöntä.

2.1.8 Gustafsonin laki

Gustafsonin laki (ks. Gustafson 1988) ottaa toisenlaisen näkökulman rinnakkaistamisella saatavaan hyötyyn. Sitä käytetään laskemaan kuinka suuri ongelma pystytään ratkaisemaan kohtuullisessa ajassa. Gustafsonin lain voi esittää seuraavalla tavalla (“Gustafson’s law - Wikipedia, the free encyclopedia” 2013)

$$skaalattu\,nopeutus = \frac{r_s + r_p * n}{r_s + r_p} = r_s + r_p * n = n + (1 - n) * r_s. \tag{2.2}$$

Kaavassa r_s on se osuus algoritmista, jota ei voi rinnakkaistaa, r_p on rinnakkaistuva osuus algoritmista ja n rinnakkaisten prosessien lukumäärä.

2.1.9 ”Embarrassingly parallel” -algoritmit

”Embarrassingly parallel” -algoritmi tarkoittaa algoritmia, joka on hyvin helposti rinnakkaistettavissa. Olisi suorastaan noloa jättää se rinnakkaistamatta alustoilla, joissa se lisää ohjelmiston suorituskykyä. Tämän vastakohtana ovat algoritmit, joita on hyvin vaikea tai mahdoton rinnakkaistaa.

(ks. “Embarrassingly parallel - Wikipedia, the free encyclopedia” 2014). Termi on alunperin peräisin Cleve Molerin tutkimuksesta (ks. Moler 1986). ”Embarrassingly parallel” -algoritmeja kutsutaan englanninkielisessä kirjallisuudessa myös nimillä *”perfectly parallel”* ja *”pleasingly parallel”*. ”Pleasingly parallel” -termiä on alettu viime

aikoina käyttää enemmän, koska se ei kuulosta niin negatiiviselta. Varsinaisestihan näissä algoritmeissa ei ole mitään noloa.

Tällaiset algoritmit rinnakkaistuvat yleensä siten, että ne on helppo jakaa osiin, jotka suoritetaan eri prosesseissa. Lisäksi eri prosessien tuloksien yhdistäminen ei vaadi suurta käsittelyä ja prosessien ei tarvitse kommunikoida keskenään esimerkiksi jakamalla välituloksia(ks. luku 1.4.4 Foster 1995).

Muutamia esimerkkejä ”Embarrassingly parallel” -ongelmista voidaan mainita:

- raakaa voimaa käyttävä etsintä kryptografiassa,
- Mandelbrot-fraktaalin renderöinti,
- tietokoneanimaatioiden renderöinti, jossa jokainen ruutu voidaan renderöidä erikseen,
- sääennustusten laskenta,
- geneettiset algoritmit ja
- kasvojentunnistus, jossa joukkoa tunnistettavia kasvoja vertaillaan toiseen suureen joukkoon kasvoja.

2.2 Grafiikkasuoritinlaskenta

Grafiikkasuoritinlaskenta tarkoittaa yleistä laskentaa, joka suoritetaan grafiikkasuorittimella (engl. *Graphics Processing Unit*) keskussuorittimen (engl. *Central Processing Unit*) sijaan. Pääasiassa grafiikkasuorittimella suoritetaan SIMD-rinnakkaislaskentaa (Stone, Gohara ja Shi 2010). Grafiikkasuorittimella tehtävää yleistä laskentaa kutsutaan usein englanninkielisellä nimellä *General-Purpose computation on Graphics Processing Units*, lyhennettynä GPGPU.

Vielä noin 20 vuotta sitten grafiikkasuorittimet oli suunniteltu laskemaan vain etukäteen määrättyjä laskutoimitoimituksia, joita tarvittiin OpenGL- ja Direct3D-renderöinnissä. Nykyiset grafiikkasuorittimet ovat kuitenkin kehittyneet sisältämään ohjelmoitavia ominaisuuksia, joita voidaan käyttää yleisessä laskennassa. Nykyisiä grafiikkasuorittimia kutsutaankin yleiskäyttöön soveltuviksi grafiikkasuorittimiksi

(engl. *General Purpose Graphics Processing Unit*, lyhennettynä myös GPGPU) (Cohen ja Garland 2009).

Grafiikkasuorittimilla suoritettavaa SIMD-rinnakkaislaskentaa suoritetaan periaatteessa kaikilla alustoilla hyvin samalla lailla. OpenGL eroaa muista hiukan sen vuoksi, että siinä ei ole suoraa tapaa suorittaa laskentaa, vaan siinä on käytettävä grafiikanpiirtorajapintoja. Tapa, jolla algoritmit rinnakkaistetaan, on yleensä silmukoiden muuntaminen kernel-funktioiksi, joista jokainen suorittaa saman laskennan kuin silmukan yksi iteraatiokerta. OpenGL:ssä näitä kernel-funktioita vastaavat varjostin-funktiot. Tietysti kernel-funktioissa voidaan käyttää apufunktioita, jotka myös lasketaan grafiikkaprosessorilla.

Aina silmukan rinnakkaistaminen ei ole näin suoraviivaista, koska silmukan aiemmat iteraatiokerrat voivat vaikuttaa myöhempien iteraatioiden tuloksiin, mikä voi monimutkaistaa rinnakkaistettavaa koodia hyvinkin paljon. Tästä esimerkkinä voidaan mainita myöhemmin tutkielmassa esiteltävä rinnakkaistettu integraalikuvan laskenta, mutta myös sen rinnakkaistaminen sisältää silmukan muuttamisen kernel-funktioilla laskettavaksi.

Grafiikkasuoritinlaskentaan on olemassa erilaisia alustoja ja kirjastoja, joilla ohjelmiston tai algoritmin voi rinnakkaistaa käyttäen grafiikkasuoritinta. Seuraavaksi esitellään tärkeimmät ja tähän tutkimukseen olennaisesti liittyvät.(Owens ym. 2008)

2.2.1 CUDA

CUDA:sta kerrotaan “What Is Cuda” (2013) verkkosivulla vapaasti käännettynä seuraavaa. "CUDA™ on rinnakkaislaskenta-alusta ja ohjelmointimalli, joka mahdollistaa dramaattisen lisäyksen tietokoneen suorituskyvyssä ottamalla käyttöön grafiikkasuorittimen(engl. *GPU*)." CUDA[1] on ehkä helppokäyttöisin tällä hetkellä saatavilla olevista GPU-laskentakirjastoista. Ennen CUDA:a grafiikkasuoritinlaskentaa joutui toteuttamaan grafiikkaohjelmointirajapintojen kautta, mikä oli usein hyvin vaikeaa (Nickolls ym. 2008).

1. CUDA on Nvidian rekisteröity tavaramerkki.

Sanders ja Kandrot (2010) mukaan CUDA:n historia sai alkunsa 2006, kun Nvidia julkaisi ensimmäiset grafiikkakortit, jotka tukivat yleistä laskentaa grafiikkasuorittimella. Tämän mahdollisti se, että julkaistut grafiikkakortit tukivat IEEE:n standardia liukuluvuille ja tulosten ja laskujen tarkkuus vastasi muita standardin täyttäviä prosessoreita (Blythe 2008, s.776). CUDA tarjoaa valmiita ohjelmointimalleja ja ohjelmoijan ei tarvitse tehdä kaikkea itse alusta alkaen.

CUDA:lla voidaan helposti rinnakkaistaa silmukat käyttämällä rinnakkaisia säikeitä laskemaan kukin oman yhtä iteraatiokierrosta vastaavan osan silmukasta. Säikeet suorittavat koodia, jota kutsutaan kernel-funktioksi. Silmukkaa vastaava osa koodista on kulmasulkusyntaksin avulla kutsuttava kernel-funktio, josta myöhemmin pieni esimerkki. Kulmasulkusyntaksi ottaa parametreikseen hilan ja blokkien koot, jotka vastaavat ikäänkuin silmukan kierrosten määriä .(Sanders ja Kandrot 2010)

CUDA:ssa kutsutaan tietokoneen keskussuoritinta isännäksi (engl. *host*) ja näytönohjainta laitteeksi (engl. *device*). CUDA:sta on useita eri versioita, joista uudempia tukevat vain viimeisimmät grafiikkakortit. Uudemmat versiot kuitenkin sisältävät vanhempien versioiden ominaisuudet. Versioista 1.0 ei tue mitään atomisia operaatioita. 1.1 tukee globaaleja atomisia operaatioita. 1.2 tukee niiden lisäksi jaetun muistin atomisia operaatioita. 1.3 tukee kaksoistarkkuuden liukulukuoperaatioita ja 2.0 muun muassa atomisia funktioita 64 bittisillä kokonaisluvuilla jaetussa muistissa. 3.0 tukee warpin sekoitus (engl. *warp shuffle*) funktioita Wilt (2013) mukaan.

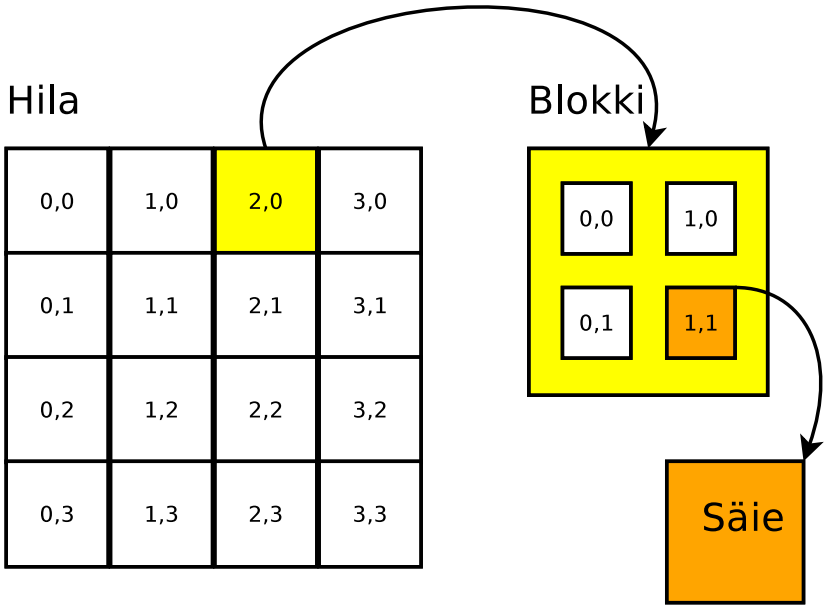
Atomiset operaatiot tarkoittavat operaatioita, joita muut säikeet eivät häiritse. Niitä käytetään estämään race condition -tilanteita siten, että vain yksi säie kerrallaan voi päästä käsiksi tiettyyn muistialueeseen, ja muut säikeet odottavat kunnes operaatio on suoritettu, jolloin seuraava säie pääsee käsiksi muistialueeseen. Globaalit atomiset operaatiot tapahtuvat globaalin muistin tasolla ja jaetun muistin atomiset operaatiot tapahtuvat jaetun muistin alueella.

Warpin sekoitus -funktio mahdollistaa muuttujien vaihtamisen warpin sisältämien säikeiden välillä. Warp on joukko säikeitä, jotka suoritetaan aina kerralla samaan aikaan grafiikkaprosessorilla yhdessä kellojaksossa . Warpin koko voi vaihdella lait-

teiden välillä, mutta on nykyisissä laitteissa aina 32. Kaikki warpin säikeet suorittavat samoja käskyjä eri datalle. Warpin säikeet suorittavat joka käskyn ohjelman haaroista, jonka niistä joku joutuu suorittamaan. Siten osa säikeistä laskee tyhjää laskentaa, kun ollaan suorittamassa koodin haaraa, jota ei suoriteta kyseisessä säikeessä.

Näytönohjaimella ajettavaa koodia kutsutaan kernel-funktioksi. Kernel-funktio ja siitä kutsutut apufunktiot suoritetaan näytönohjaimen suoritusytimillä. Kernel-funktiota suoritetaan rinnakkain useissa säikeissä siten, että jokainen säie suorittaa samaa koodia. Suoritettaessa kernel-funktiota se suoritetaan määritellyllä hilalla (engl. *grid*), joka on jaettu blokkeihin (engl. *block*) . Jokainen blokki sisältää määritellyn määrän säikeitä. Hila voi sisältää blokkeja kolmessa ulottuvuudessa, mutta kaikki laitteet eivät tue kuin kahta ulottuvuutta. Blokki taas voi sisältää säikeitä jaettuna kolmeen ulottuvuuteen. Kuvio 4 esittää tätä rakennetta.

Hilan ja blokkien koko valitaan ratkaistavan ongelman koon mukaan. Esimerkiksi kynnystysalgoritmissä hilan ja blokkien kokoon vaikuttavat kuvan mittasuhteet. Yleensä optimoidaan blokkien kokoa, koska se voi vaikuttaa suuresti ohjelman nopeuteen. Yhtä blokkia suorittaa kerrallaan vain yksi multiprosessointiyksikkö. Hilan koko valitaan taas siten, että se kattaa tietyllä blokkien koolla koko ongelman. Jos ongelman koko ei ole jaollinen blokkien koolla, usein lasketaan ongelman kokoa vähän suuremmalla hilalla ja ei suoriteta laskentaa säikeillä, jotka jäävät ulkopuolelle.



Kuvio 4: CUDA:n hila, blokit ja säikeet

Kernel-funktiosta käytetään osoittimia, jotka on varattu näytönohjaimen muistiin. Lisäksi siitä pääsee käsiksi parametreiksi annettuihin muuttujiin ja paikallisesti kernel-funktiossa tai sen kutsumissa funktioissa käytettäviin muuttujiin. Lisäksi on käytettävissä joitain säie- ja blokkikohtaisia muuttujia. Kernel-funktiota kutsutaan isännästä ja se suoritetaan laitteella.

Kernel-funktiota kutsutaan käyttäen kulmasulkusyntaksia. Funktio saa parametrit aivan kuin normaalit funktiot, mutta sille annettavat osoittimet on alustettava näytönohjaimen muistiin. Kulmasulun sisälle annetaan blokkien ja säikeiden määrät, joilla kernel-funktiota suoritetaan. Alla esimerkki kulmasulkusyntaksilla kutsutusta kernel-funktiosta. Alla olevassa esimerkissä blokkeja on N ja säikeitä blokissa on myös N, joka saa arvon 1024.

```
1 #define N 1024
2 int main(void){
3
4 int* dev_imgA;
5 int* dev_imgB;
6
```

```
7  cudaMalloc((void**)&dev_imgA, N*N* sizeof(int))
8  cudaMalloc((void**)&dev_imgB, N*N* sizeof(int))
9  threshold<<<N,N>>>(dev_imgA, dev_imgB);
10 }
```

Kernel-funktiossa päästään käsiksi muutamiin paikallisiin muuttujiin, joista saadaan esimerkiksi blokin indeksi ja säikeen indeksi. Esimerkiksi yksinkertainen silmukka voidaan rinnakkaistaa käyttäen säikeitä siten, että jokainen säie suorittaa yhden iteraation silmukasta. Tämä edellyttää sitä, että silmukan iteraatiot eivät ole riippuvaisia aiemmista.

CUDA:n kääntämiseen tarvitaan erillinen kääntäjä nvcc. Se tukee kulmasulkusyntaksin lisäksi muita CUDA:n ominaisuuksia. Sillä voi kääntää myös koko ohjelman, mutta sitä ei suositella, sillä nvcc roskaa globaalia nimiavaruutta. Yleisesti ottaen CUDA:n kääntäminen mukaan projektiin ei ole aina läheskään yksinkertaista, koska se vaatii aika monimutkaisen rakenteen Makefile-tiedostoihin. Tutkimusta tehdessä Makefile-tiedostojen rakenteen säätöön käytettiin paljon aikaa.

Tässä tutkimuksessa tarvitaan integraalikuvan laskemiseen vähintään versiota 1.3 CUDA:sta, koska se tukee kaksoistarkkuuden liukulukuja. Kynnystysalgoritmin toteutukseen riittää varhaisempikin versio. Yksi käytettävissä olleista näytönohjaimista tuki vain versiota 1.2, joten sitä ei käytetty integraalikuvan laskentaan.

2.2.2 OpenCL

OpenCL-spesifikaatio (ks. OpenCL Working Group ym. 2012) määrittelee mitä kaikkea OpenCL sisältää. OpenCL:llä laskenta voidaan rinnakkaistaa käyttäen erilaisia alustoja, kuten useaa CPU:ta tai GPU:ta tai jopa DSP-prosessoreita (Stone, Gohara ja Shi 2010). OpenCL on standardi tehtävä-rinnakkaiselle ja data-rinnakkaiselle heterogeenisella alustalla tapahtuvalle laskennalle. OpenCL yrittää täyttää tarpeen kirjastolle, joka suorittaa laskentaa heterogeenisessä ympäristössä eri alustoilla käyttäen samaa lähdekoodia.

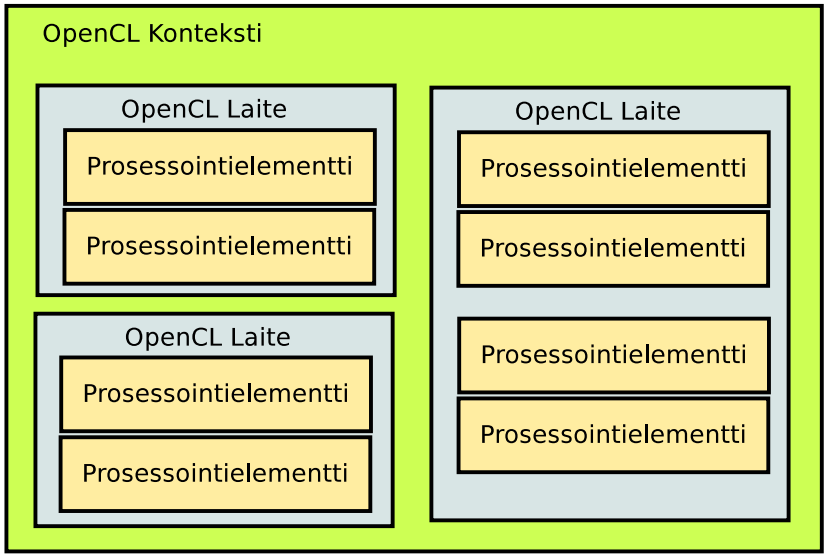
Aiemmat rinnakkaislaskentakirjastot eivät ole toimineet heterogeenisissä laskentaympäristöissä vaan ovat keskittyneet vain yhteen alustaan. OpenCL sisältää ydintoiminnot, joita tuetaan kaikilla alustoilla, ja lisäksi valinnaisia toiminnallisuuksia, joita tuetaan vain tietyillä alustoilla. Lisäksi OpenCL:ssä on laajennusmekanismi, jonka avulla laitetoimittajat voivat lisätä heidän laitteidensa tukemia uusia ominaisuuksia. OpenCL ei kuitenkaan voi täysin piilottaa laitteiden eroja eikä taata ohjelmien sovitettavuutta kaikille alustoille.(Stone, Gohara ja Shi 2010)

OpenCL-ohjelmointimalli sisältää yhteisen kielen, ohjelmointirajapinnan ja laiteabstraktion heterogeeniselle laitealustalle. Mallilla voidaan suorittaa tehtävä-rinnakkaista ja data-rinnakkaista laskentaa. Heterogeeninen laskentaympäristö koostuu isäntä-CPU:sta ja liitetyistä OpenCL-laitteista. Laitteet eivät välttämättä jaa muistia isäntäprosessorin kanssa. Lisäksi laitteilla saattaa olla erilainen käskykanta kuin isäntäprosessorilla. OpenCL tarjoaa kontekstirajapinnan (engl. *context*) laitteille, jolla voi käydä läpi käytettävissä olevia laitteita, ja rajapinnat muistin varaukseen, isäntälaitteella muistinsiirtoon, OpenCL-ohjelmien kääntämiseen ja kernel-funktioiden suorittamiseen laitteilla. Lisäksi OpenCL tarjoaa rajapinnan virhetilanteiden tarkastamiselle.

OpenCL-koodin voi kääntää ja linkittää binääritiedostoksi tavalliseen tapaan, mutta OpenCL tukee myös ajonaikaista kääntämistä. Ajonaikainen kääntäminen mahdollistaa saman ohjelmakoodin ajamisen erilaisilla alustoilla ja mahdollistaa alustojen, ajureiden ja tukikirjastojen muutokset muuttamatta alkuperäistä koodia. Ajonaikaista käännöstä käyttävät ohjelmat hyödyntävät aina viimeisimpiä ohjelmisto- ja laiteominaisuuksia ilman uudelleenkääntämistä.(Feng ym. 2012)

Vaikka OpenCL tukee valtavaa määrää erilaisia alustoja, ei ole mitenkään taattua, että OpenCL:llä kirjoitettu kernel toimisi kaikilla alustoille optimaalisesti. Joillakin alustoilla tietyn tyyppiset kernelit, jotka käyttävät kyseisen laitteiston parhaimpia ominaisuuksia, toimivat parhaiten. OpenCL:n ominaisuutta kysellä alustan ominaisuuksia voi käyttää ajonaikana valitsemaan optimaalisen kernelin kyseiselle alustalle.(Komatsu ym. 2010)

OpenCL-ohjelmointimalli antaa abstraktin rajapinnan keskussuorittimelle, grafiikka-



Kuvio 5: OpenCL Kontekstin rakenne

suorittimille ja muille laitteille. Nämä näkyvät niin sanottuina laitteina (engl. *device*), jotka sisältävät yhden tai useamman ytimen (engl. *core*). Ytimet taas koostuvat yhdestä tai useammasta prosessointielementistä (engl. *processing element*), jotka laskevat SIMD-tyyppistä rinnakkaislaskentaa. Ohjelmointimallin rakennetta esittää myös kuva 5. OpenCL tarjoaa neljää erityyppistä muistia. Laite voi sisältää jotain näistä tyypeistä tai vaikka kaikkia. (Lee ym. 2010)

Muistin erityyppiset luokat laitteille OpenCL:ssä ovat:

- global yleensä paljon ja suurella latenssilla,
- constant pienen latenssin vain luettavaa muistia,
- local saman ytimen prosessointielementtien jakamaa muistia ja
- private yksittäisen prosessointielementin omaa muistia.

Lisäksi on käytettävissä niin kutsuttua *host*-muistia, joka on isäntälaitteen omaa muistia ja sitä ei voi käyttää muista laitteista. OpenCL tarjoaa näistä muistiluokista puskuriobjekteja ja kuvaobjekteja. Puskuriobjekti on kernelien käytössä oleva yhtenäinen muistialue, johon voi sijoittaa halutunlaisia muistirakenteita. Kuvaobjektit taas ovat vain kuvien käsittelyä varten. Kuvaobjektien sisäinen formaatti voi olla erityyppinen eri laitteilla ja OpenCL tarjoaa vain rajapinnan kuvaobjektin käsittelyyn,

mutta muuten datan tyyppi on kätketty laitteelta.(Munshi ym. 2012)

Isäntälaitteen ja muiden OpenCL-laitteiden muistit ovat siis enimmäkseen erilliset. Tämä johtuu tietysti siitä, että isäntälaite on määritelty OpenCL-kontekstin ulkopuolella. Muistia voidaan kuitenkin siirtää ja kartoittaa isäntälaitteen ja muiden laitteiden välillä. Muistia voidaan siirtää OpenCL-komennoilla isäntälaitteen ja muistiobjektien välillä, joko blokkaavasti tai ei-blokkaavasti. Blokkaava siirto on synkroninen ja muistia voi käyttää suoraan siirron jälkeen. Ei-blokkaava on asynkroninen ja muistin siirto vain lisätään jonoon ilman varsinaista välitöntä suorittamista. Silloin siirrettävää muistia ei voida vapauttaa isäntälaitteessa välittämästi kutsun palaamisen jälkeen, koska sitä ei ole vielä välttämättä kopioitu laitteelle.

Muistin kartoittaminen taas toimii siten, että isäntälaite voi kartoittaa OpenCL-muistiobjektin omaan muistiavaruuteensa. Muistin kartoittaminen voi olla myös blokkaavaa ja ei-blokkaavaa. Kun muistiobjekti on kartoitettu, isäntälaite voi lukea ja kirjoittaa muistia. Kun isäntälaite on käsitellyt dataa tarpeen mukaan, täytyy muistialueen kartoitus vapauttaa.

Ensimmäiseksi OpenCL:ssä pitää aina luoda konteksti (engl. *context*), joka liittyy yhteen tai useampaan laitteeseen. Konteksti sisältää periaattessa koko laskentaympäristön. Kontekstiin varataan käytetyt laitteet ja kaikki laskenta laitteilla tapahtuu kontekstin sisällä. Muistin varaukset liittyvät aina yksittäiseen kontekstiin ja muistin määrä rajautuu aina sen laitteen mukaan, jossa on vähiten muistia. Jos halutaan käyttää joillakin laitteilla enemmän muistia tai ominaisuuksia, joita joistakin laitteista puuttuu, täytyy vähemmän muistia tai ominaisuuksia sisältävät laitteet jättää pois käytetystä kontekstista. Kontekstin luonnin jälkeen OpenCL-ohjelma voidaan kääntää ajonaikaisesti ja käännöksestä saadaan kernel-funktiot (engl. *kernels*), joita voidaan käyttää kyseisessä kontekstissa. (Stone, Gohara ja Shi 2010)

Kernel voidaan tämän jälkeen käynnistää kontekstin laitteille ja antaa samalla parametreinä ongelman koko. Kernel-funktio toimii hyvin samalla tapaa kuin CUDA:ssa. Se sisältää yhtä silmukan iteraatiota vastaavan koodin. Siinä pääsee käsiksi muuttujiin, joissa ovat suoritettavan funktion globaali- ja lokaali-indeksi. Globaali-indeksi

vastaa silmukan laskuria ja lokaali-indeksi taas vastaa CUDA:n säikeen indeksiä blokissa. Lisäksi käytettävissä on workgroup-indeksi, joka vastaa CUDA:n blokin indeksiä. OpenCL tukee 1-, 2- ja 3-ulotteisia indeksejä.

Ohjelman rinnakkaistaminen tapahtuu myös samaan tapaan kuin CUDA:ssa. Yleisesti silmukkarakenteet muutetaan kernel-funktioiksi ja kernel-funktio suoritetaan yhdellä tai useammalla laitteella. Funktioiden suorittaminen tapahtuu asynkronisesti ja funktion kutsu vain käynnistää ne. Funktion paluun odottamiseen on toinen käsky. Isäntälaitteella voidaan myös suorittaa laskentaa sillä välin kun yhden tai useamman kernel-funktion ajo on menossa.

OpenCL tarjoaa rajapinnan modernien moniytimisten keskussuorittimien SIMD-yksikköjen käyttöön. Useimmat ohjelmointikielet eivät ole tarjonneet suoraa rajapintaa näihin, vaan niitä käytettäessä on täytynyt hyödyntää erillisiä ohjelmakirjastoja tai luottaa siihen, että kääntäjä on optimoinut käännettävän koodin siten, että se käyttää hyväksi SIMD-laskentayksikköjä. Moniytimisillä keskussuorittimilla kannattaa käyttää sellaisia kerneleitä, jotka käyttävät *global*-tyyppistä muistia *constant*- ja *local*-tyyppisen sijaan, koska ne kaikki sijaitsevat näillä alustoilla samassa fyysisessä välimuistissa. (Munshi ym. 2012; Komatsu ym. 2010).

2.2.3 OpenGL

OpenGL on laajasti käytetty grafiikkakirjasto, johon on olemassa ajurit useimmille grafiikkakorteille. Se on tarkoitettu pääasiassa suoraan grafiikan tekemiseen, mutta sillä voidaan pientä vaivaa nähden suorittaa erilaisia rinnakkaislaskentatehtäviä. Esimerkiksi Hensley ym. (2005) käyttää OpenGL:ää integraalikuvan laskemiseen. Tähän kuitenkin tarvitaan vähintään OpenGL:n versiota 1.5. Tässä tutkimuksessa on käytetty lähinnä OpenGL:n versiota 2.0 ja uudempien versioiden ominaisuuksia ei käsitellä.(Shreiner ym. 2007)

OpenGL 1.5 mahdollistaa rinnakkaislaskennan suorittamisen grafiikkaprosessorilla käyttämällä vertex-varjostimia (engl. *vertex shader*) ja fragment-varjostimia (engl. *fragment shader*.) Vertex-varjostin on pieni C-ohjelma, joka suoritetaan ennen jokaisen

verteksin renderöintiä. Se määrittää mihin kohtaan kyseinen verteksi piirretään. Fragment-varjostin on myös pieni C-ohjelma joka suoritetaan ennen jokaisen pikselin renderöinti, ja jolla päätetään kyseisen pikselin väri. Molemmilla varjostimilla on käytettävissä joitakin syötteitä ja joitakin tulosteita. Kun näitä ohjelmia ja renderöintiä käytetään luovasti voidaan suorittaa yleistä laskentaa grafiikkaprosessorilla.(Rost ym. 2007)

Käytännössä silmukan rinnakkaistaminen tehdään OpenGL:llä siten, että renderöidään geometriaa, jossa jokainen varjostimille välitettävä pikseli tai verteksi voi vastata rinnakkaistettavan silmukan yhtä iteraatiokierrosta. Varjostimet suorittavat silmukan varsinaisen laskennan. Varjostimille voidaan välittää erilaisia parametreja, kuten tekstuureja, jotka voivat sisältävät erityyppistä dataa. Kun sitten renderöidään tekstuuria vastaava alue, grafiikkaprosessori suorittaa varjostimien koodia rinnakkain. Lopuksi tulokset luetaan tekstuureista tai framebufferista johon renderöinti suoritettiin.

OpenGL:ää käyttäen voi tehdä monenlaista laskentaa, mutta se, että sitä ei ole alunperin suunniteltu yleiselle laskennalle rajoittaa sen käyttöä jonkun verran. Laskennan syötteenä voi käyttää monenlaisia ympäristömuuttujia ja arvoja sekä tekstuureita, joissa on laskettava data. Laskennan tulosten saaminen on kuitenkin vaativampaa. Tulokset saa vain tekstuureita ja puskureita lukemalla ja muuntamalla kuvainformaation sopivaksi dataksi. Lisäksi laskennassa ei voida käyttää osoittimia dataan. Siksi OpenGL ei sovellu kaikenlaiseen laskentaan, mutta tietynlaiseen segmentointiin se sopii kohtalaisen hyvin.(Shreiner ym. 2007)

2.2.4 OpenGL ES 2.0

OpenGL ES -varjostimilla voi suorittaa GPU-laskentaa, mutta koska se on riisuttu versio OpenGL:stä, sen käyttäminen vähänkin monimutkaisemman laskennan tekemiseen on vaikeaa. Tässä tutkimuksessa suunniteltiin alunperin,että integraalikuvan laskenta toteutettaisiin käyttäen OpenGL ES - järjestelmää, mutta se osoittautui liian työlääksi tehtäväksi. Lisäksi ongelmana oli, että OpenGL ES ei sisällä kaksoistark-

kuuden liukulukuja.

OpenGL ES sisältää kuitenkin vertex- ja fragment-varjostimet, jotka suorittavat pienen ohjelman jokaista piirrettyä verteksiä ja pikseliä kohti. Näillä varjostimilla voidaan suorittaa jonkin verran rinnakkaista laskentaa grafiikkaprosessorilla. Tämä on kuitenkin yleensä jossain määrin rajatumpia verrattuna tavallisen OpenGL:n ominaisuuksiin, esimerkiksi laskentatarkkuus on rajatumpi.(Rost ym. 2007)

2.3 Konenäkö

Konenäkö on tietokoneen suorittamaa automaattista kuvan analysointia. Se käyttää apunaan kuvantunnistusta ja kuvankäsittelyalgoritmeja. Siihen sisältyvät kuvan hankinta -prosessointi ja analysointimenetelmät sekä kuvan ymmärtämiseen käytetyt menetelmät. Tässä luvussa käydään läpi konenäön menetelmiä, jotka liittyvät olennaisesti tähän tutkimukseen.

2.3.1 Segmentointi

Kuvan segmentointi tarkoittaa kuvan jakamista yhtenäisiin alueisiin esimerkiksi värin, tekstuurin tai objektin perusteella. Segmentointimenetelmiä käytetään pilkkomaan kuvasta osia ja objekteja. Yksinkertaisimmillaan segmentointimenetelmä on esimerkiksi kynnystysmenetelmä, joka muuttaa kuvan pikselit mustiksi, jos niiden voimakkuus on pienempi kuin annettu raja-arv,o ja muulloin valkeaksi. Monimutkaisimmillaan kuvan segmentointialgoritmit ovat todella vaativia. Monimutkaisen kuvan segmentointi on Gonzales ja Woods (2002, sivu 567) mukaan yksi vaikeimmista digitaalisen kuvankäsittelyn tehtävistä.

Segmentoinnin onnistuminen määrää yleensä onnistuuko konenäkötehtävän ratkaiseminen. Esimerkiksi tekstin tunnistuksessa segmentoinnin onnistuminen tekee mahdolliseksi koko operaation onnistumisen. Kuvan segmentoinnissa on usein tärkeää, että segmentointi jakaa kuvan sopivan kokoisiin osiin, mutta ei sitä pienempiin. Koska sopivan kokoisten osien määrittely riippuu ratkaistavasta ongelmasta, on tämän toteuttaminen usein hyvin vaikeaa.(Gonzales ja Woods 2002)

Kuvan segmentointi perustuu tyypillisesti kuvan intensiteettiarvojen epäjatkuvuuteen ja samankaltaisuuksiin. Epäjatkuvuuteen perustuvissa menetelmissä yritetään etsiä kuva-alueen reunoja intensiteetin suureen muutokseen perustuen. Samankaltaisuuksiin perustuvissa menetelmissä kuva pyritään jakamaan alueisiin, jotka ovat jonkin ennalta määrätyn ominaisuuden perusteella samankaltaisia.(Gonzales ja Woods 2002)

2.3.2 Datan redusointi

Datan redusointi tarkoittaa datan käsittelyä siten, että se tiivistää käytettävää informaatiota. Erilaisiin redusointeihin kuuluvat esimerkiksi keskiarvon laskeminen ja vähän myöhemmin esiteltävä kumulatiivisen summan laskenta kappaleesta 2.3.4. Digitaalisen datan redusointi tarkoittaa yleensä editointia, skaalausta, koodausta, järjestämistä, kokoamista tai taulukoiden muodostamista. Pääasiana datan redusoinnissa on tuottaa suuresta määrästä dataa tiivistä tietoa, jolla on merkitystä.(“Data Reduction - Wikipedia, the free encyclopedia” 2014)

Esimerkkinä datan redusoinnista voidaan mainita Kepler-satelliitin kameran informaation redusointi (“Data Reduction - Wikipedia, the free encyclopedia” 2014). Satelliitti otti 95 megapikselin kuvia joka kuudes sekunti, jonka datan määrä oli paljon suurempi kuin käytetyn datayhteyden kapasiteetti (550 KBps). Satelliitti redusoi dataa yhdistämällä kuvia ennen siirtoa ja siirtämällä vain merkittävät osat kuvista.

2.3.3 Kynnystysalgoritmi

Kynnystäminen (engl. *thresholding*) tarkoittaa tapaa käsitellä kuvaa siten, että siitä tulee mustavalkoinen. Yksinkertaisimmassa muodossa annettua raja-arvoa pienemmät pikselit muuttuvat mustiksi ja raja-arvoa suuremmat pikselit muuttuvat valkeiksi. Monimutkaisemmista kynnystysalgoritmeista löytyy lisää tietoa kirjasta Sezgin ja Sankur (2004). Yksinkertainen kynnystysoperaatio on niin sanottu pisteoperaatio, johon vaikuttaa vain kyseessä olevan pikselin arvo. Kuviossa 6 on kuva, jota on käsitelty kynnystysoperaatiolla. (“Thresholding (image processing)- Wikipedia, the

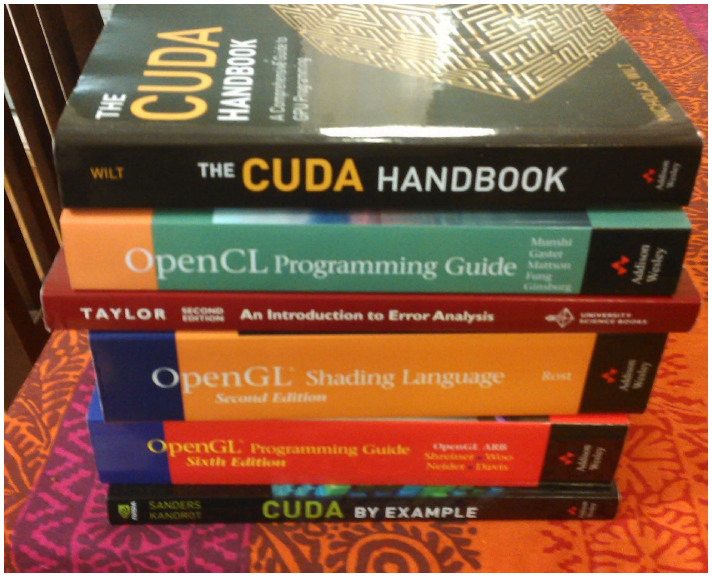
free encyclopedia" 2014; Sezgin ja Sankur 2004).

Kynnystämisen toimintaa voidaan kuvata kaavalla

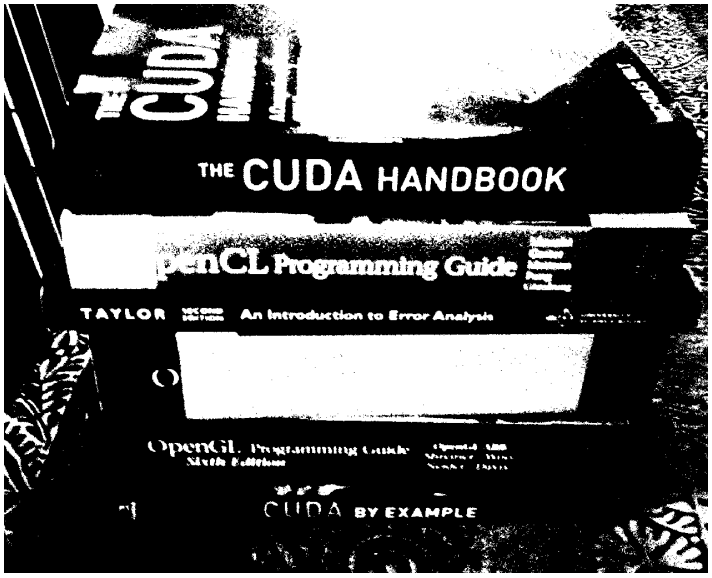
$$F(x,y) = \begin{cases} 1 \text{ jos } f(x,y) > T \\ 0 \text{ muuten,} \end{cases} \tag{2.3}$$

jossa $f(x,y)$ on alkuperäisen kuvan intensiteetti pisteessä (x,y), T on Raja-arvo ja $F(x,y)$ on käsitellyn pikselin arvo pisteessä (x,y).

Muiden pikselien arvot eivät vaikuta laskennan tulokseen. Se tekee tästä algoritmista helposti rinnakkaistuvan, koska kuva voidaan paloitella minkälaisiin osiin tahansa, käsitellä osat erikseen, ja myöhemmin yhdistää tulos yhdeksi kuvaksi. Kynnystäminen on siksi kuvankäsittelyalgoritmina *"embarassingly parallel"* -tyyppisesti rinnakkaistuva.(Sezgin ja Sankur 2004)



(a) Alkuperäinen kuva.



(b) Kynnystysalgoritmin tulos raja-arvolla 120

Kuvio 6: Kynnystysalgoritmilla segmentointi.

2.3.4 Kumulatiivinen summa

Weisstein (2014) kertoo seuraavasti: "Kumulatiivinen summa on sarja annetun sarjan osittaissummia." Osittaissumma ensimmäiselle N:lle termille sarjalle $\{a_k\}_{k=1}^{n}$ saadaan kaavalla

$$S_n = \sum_{k=1}^{N} a_k \tag{2.4}$$

“Prefix sum - Wikipedia, the free encyclopedia” (2014) kertoo, että "Kumulatiivinen summa (engl. *prefix sum*) tarkoittaa tietotekniikassa sarjaa numeroita $y_0, y_1, y_2, \ldots$, joka kuvaa toista sarjaa numeroita $x_0, x_1, x_2, \ldots$ siten, että

$y_0 = x_0$

$y_1 = x_0 + x_1$

$y_2 = x_0 + x_1 + x_2$

$y_3 = x_0 + x_1 + x_2 + x_3$

$y_4 = x_0 + x_1 + x_2 + x_3 + x_4$

…"

Esimerkkinä kumulatiivisesta summasta voidaan antaa alkulukujen kumulatiivinen summa:

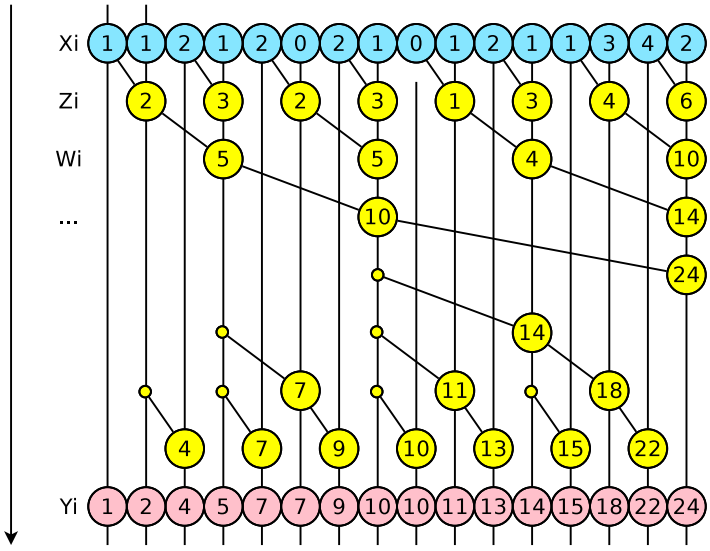
alkuluvut	1	2	3	5	7	…
alkulukujen prefix sum	1	3	6	11	18	…

Kumulatiivisen summan laskenta tunnetaan myös nimellä *prefix sum scan*, *scan* ja *cumulative sum* (Weisstein 2014). Kumulatiivinen summa voidaan hyvin helposti laskea etenemällä peräkkäin ensimmäisestä viimeiseen koko sarjalle laskemalla aina yhteen seuraavan arvo ja edellinen prefix-summa. Rinnakkaistettuna algoritmi on hiukan monimutkaisempi, ja seuraava luku käsittelee sitä. Kumulatiivisen summan laskenta on yksinkertainen mutta hyvin käyttökelpoinen menetelmä. Sen monia käyttötarkoituksia käsittelee G. E. Blelloch (1990). Yhtenä esimerkkinä voidaan mainita *grep*-komennon toteutus unix-ympäristössä.

2.3.5 Rinnakkaistettu kumulatiivisen summan laskenta

Kumulatiivisen summan laskenta voidaan rinnakkaistaa seuraavalla tavalla (ks. Ladner ja Fischer 1980). Kuva 7 esittää tapaa, jolla rinnakkainen kumulatiivisen summan laskenta toteutetaan. Lisäksi Nguyen (ks. 2007, s. 851-876) kuvaa miten rinnakkaistaminen voidaan toteuttaa tehokkaasti näytönohjaimella. Vielä yhtenä

lähteenä voidaan käyttää varhaisia tutkimusta aiheesta (G. Blelloch 1989).(Hillis ja Steele 1986)



Kuvio 7: Rinnakkainen kumulatiivisen summan laskenta. Laskenta etenee ylhäältä alaspäin. Ylhäällä sinisellä on alkuperäinen sarja. Alhaalla vaaleanpunaisella on kumulatiivinen summa. Suuret keltaiset pallot ovat yhteenlaskuoperaatioita.

Rinnakkaistus hoidetaan seuraavilla askelilla, kuten kuvassa 7 on tehty (ks. Ladner ja Fischer 1980; ks. “Prefix sum - Wikipedia, the free encyclopedia” 2014; ks. Tarjan ja Vishkin 1985).

1. Lasketaan summat peräkkäisistä pareista joiden ensimmäinen pari on parillinen indeksi: $z_0 = x_0 + x_1, z_1 = x_2 + x_3, \ldots$
2. Rekursiivisesti lasketaan prefix sum sarjalle $w_0, w_1, w_2, \ldots$ sarjasta $z_0, z_1, z_2, \ldots$ ja niin edelleen aina seuraavalle välisarjalle, kunnes termejä on vain yksi.
3. Jokainen termi lopullisessa sarjassa $y_0, y_1, y_2, \ldots$ lasketaan summana enintään kahdesta välituloksesta välitulosten sarjasta $y_0 = x_0, y_1 = z_0, y_2 = z_0 + x_2, y_3 = w_0$, jne. Ensimmäisen arvon jälkeen jokainen seuraava arvo y_i joko kopioidaan välitulosten sarjasta tai on edellinen käytetty arvo vastaavasta välitulosten sarjasta lisättynä arvolla x_i.

Algoritmin vaativuus on $O(log\, n)$, kun listassa on n alkiota ja on käytettävissä kone

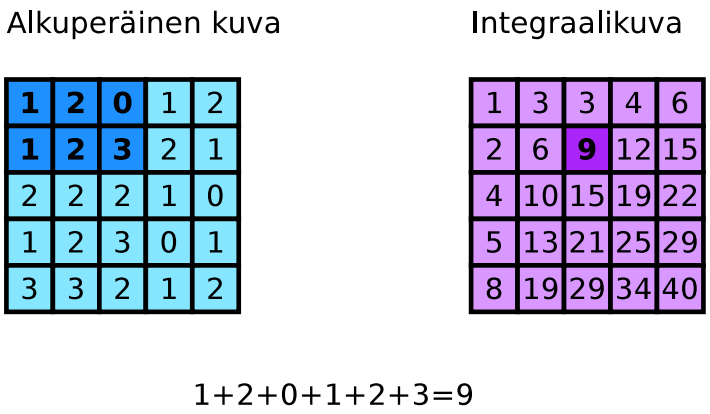
jossa on $O(n/log\,n)$ prosessoria. Algoritmi ei myöskään hidastu asymptoottisesti, vaikka elementtejä olisi enemmän kuin prosessoreja ("Prefix sum - Wikipedia, the free encyclopedia" 2014). Kumulatiivisen summan laskentaa voi käyttää esimerkiksi laskentalajittelu -algoritmissa (ks. Knuth 1997, s. 74).

2.3.6 Integraalikuva

(ks. Crow 1984) esitteli ensimmäisen kerran integraalikuvan (engl. *integral image*) käsitteen. Integraalikuva muodostetaan tavallisesta kuvasta siten, että jokaisen sarakkeen ja jokaisen rivin pikseliarvoihin lisätään niitä edeltävien pikseleiden arvot, jolloin muodostuu ikään kuin integraali alkuperäisestä kuvasta alusta kyseiseen pikseliin vaaka- ja pystysuunnassa. Tutkimuksessa Viola ja Jones (2001) käytettiin ensimmäistä kertaa integraalikuvaa konenäössä ja Shafait, Keysers ja Breuel (2008) käytti integraalikuvia keskiarvon ja keskihajonnan nopeaan laskemiseen.

Integraalikuvan pisteen arvo pisteessä (x,y) saadaan kaavalla 2.5 ("Summed area table - Wikipedia, the free encyclopedia" 2014), kuten havainnollistavassa kuviossa 8 esitetään.

$$I(x,y) = \sum_{\substack{x' \leq x \\ y' \leq y}} i(x',y') \tag{2.5}$$



Kuvio 8: Integraalikuvan pikselien arvot.

Integraalikuva saadaan laskettua alkuperäisestä kuvasta yhdellä läpikäyntikerralla. Yläreuna ja vasen reuna alustetaan arvoilla 0. Kuva käydään alusta loppuun normaalissa skannausjärjestyksessä ja integraalikuvan pisteen arvo saadaan laskemalla yhteen kyseisen pikselin arvo ja integraalin arvo edelliseltä riviltä ja sarakkeelta ja vähentämällä niistä edellisellä rivillä edellisessä sarakkeessa oleva integraalin arvo. Algoritmi 1 kuvaa tätä.

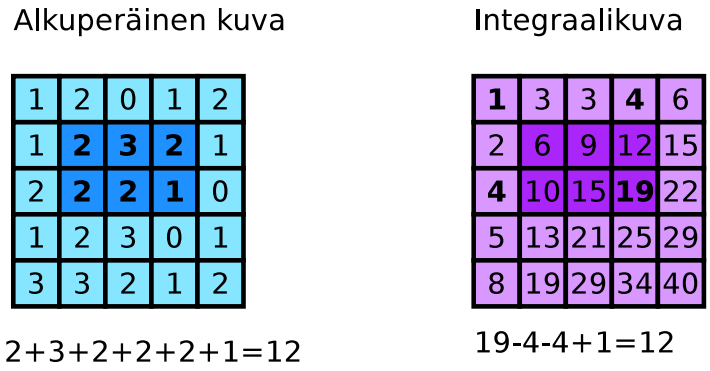
Integraalikuvasta voidaan helposti laskea tietyn alueen pikseleiden keskiarvo. Käytännössä integraalikuva on kaksiulotteinen versio kumulatiivisesta summasta. Integraalikuva tunnetaan englanninkielisessä kirjallisuudessa myös nimellä *summed area table*.

Perinteisesti keskiarvo lasketaan seuraavalla kaavalla:

$$\bar{x} = \frac{x_1 + x_2 + \cdots + x_N}{N} = \frac{\sum x_i}{N}. \tag{2.6}$$

Integraalikuvasta voi laskea summan alueelle alla olevalla kaavalla, jossa $S(A)$ on integraalikuvan arvo pisteessä suorakulmion $ABCD$ kulmapisteessä A, kuten kuvasta 9 näkee. Kuvassa piste A integraalikuvassa arvon 19 kohdalla, piste D on yläkulmassa arvon 1 kohdalla ja pisteet B ja C ovat arvojen 4 kohdalla.

$$S(ABCD) = S(A) + S(D) - S(B) - S(C) \tag{2.7}$$



Kuvio 9: Integraalikuvasta laskettava alueen summa.

Alueen summasta taas saadaan helposti alueen keskiarvo jakamalla summa alkioiden määrällä. Näin integraalikuvasta saadaan yhteenlaskulla, kahdella vähennyslaskulla ja yhdellä jakolaskulla alueen keskiarvo, ja laskennan vaativuus on O(1) kun taas perinteisesti laskettaessa keskiarvon laskemisen vaativuus on O(n).

$$\bar{x} = \frac{S(A) + S(D) - S(B) - S(C)}{N} \tag{2.8}$$

Varianssi taas lasketaan kaavalla

$$v = \frac{1}{N}\sum_i (x_i - \bar{x})^2 = \frac{1}{N}(\sum_i x_i^2 - 2\sum_i x_i\bar{x} + N\bar{x}^2) = \frac{1}{N}\sum_i x_i^2 - \bar{x}^2 \tag{2.9}$$

Jossa viimeinen muoto vastaa kuvan pikseleiden intensiteetin neliöiden integraalikuvasta laskettavaa summaa vähennettynä keskiarvon neliöllä, joka saadaan myös laskettua helposti edellä esitetyllä tavalla. Varianssista saa taas laskettua tarvittaessa keskihajonnan ottamalla siitä neliöjuuren.

2.3.7 Integraalikuvan laskenta grafiikkasuorittimella

Tässä osassa esitellään tapa, jolla tutkimuksessa Bilgic, Horn ja Masaki (2010) on laskettu integraalikuva käyttäen grafiikkasuoritinta. Integraalikuva lasketaan käyttämällä scan (prefix sum) -kerneliä ja transpose-kerneliä. Scan-kernel laskee jokaiselle riville kumulatiivisen summan ja transpose-kernel kääntää kuvan ympäri $y = x$ -akselin suuntaisesti.

Prefix sum operaatio ajetaan ensin riveittäin, jonka jälkeen kuva käännetään käyttäen transpose-kerneliä ja prefix sum -operaatio ajetaan uudelleen riveittäin. Tällä kertaa se tapahtuu kuitenkin transpoosin takia sarakkeille. Lopuksi kuva vielä käännetään oikein päin käyttäen transpose-kerneliä. Tuloksena on integraalikuva, koska ensin lasketaan riveille kumulatiivinen summa ja sitten lasketaan sarakkeille kumulatiivinen summa, mikä vastaa integraalikuvan laskennassa suoritettavia operaatioita. Myös Nguyen (2007) esittää tavan laskea integraalikuvan grafiikkasuorittimella käyttäen samaa menetelmää. Toinen lähde laskee myös integraalikuvan grafiikkasuorittimella (ks. Nehab ym. 2011).

Algorithm 1 Calculation of integral image sequentially

1: **procedure** CALCULATE INTEGRAL IMAGE(image)
2: I: Input image with size $w * h$
3: I_{int} : integral image with size $w * h$
4: Array elements are accessed in row major order
5: **for** x=0 to w-1 **do**
6: $I_{int}[x] \leftarrow 0$
7: **end for**
8: **for** y=1 to h-1 **do**
9: $I_{int}[y * w] \leftarrow 0$
10: $s \leftarrow 0$
11: **for** x=0 to w-1 **do**
12: $s \leftarrow s + I[x + (y - 1) * w]$
13: $I_{int}[x + y * w + 1] \leftarrow s +_{int} [x + (y - 1) * w + 1]$
14: **end for**
15: **end for**
16: **end procedure**

2.3.8 Nelipuumetsäsegmentointi

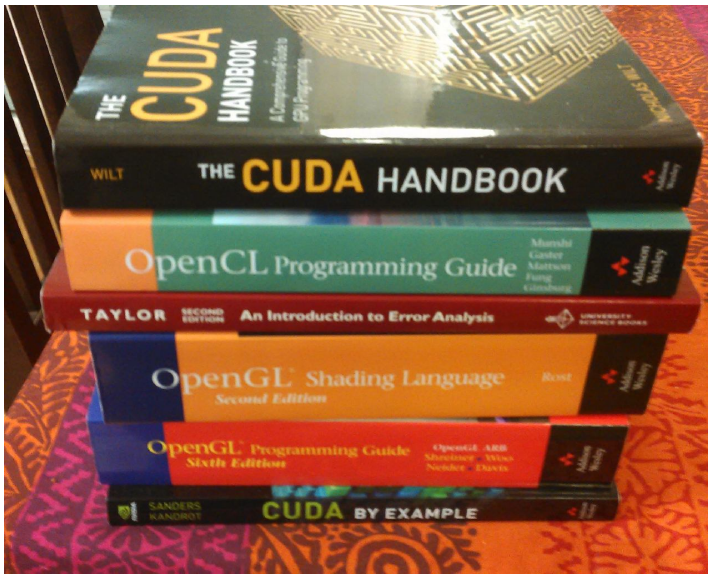
Eskelinen, Tirronen ja Rossi (2013) esittää uuden tavan segmentoida kuvan käyttäen integraalikuvia ja nelipuumetsää apuna. Menetelmä on suunniteltu segmentoimaan erityisesti teksturoituja alueita, joiden välinen kontrasti on heikko. Esimerkki kuva 10a ja siitä tietyillä parametreilla segmentoitu kuva 10b esittää segmentoinnin, jossa kuvasta erottuu samanlaisen tekstuurin alueet. Jos kuvasta olisi haluttu tunnistaa tekstiä, olisi pitänyt valita parametrit siten, että ylimmän tason neliöt olisivat olleet pienempiä.

Menetelmän perusidea on sama kuin Horowitz ja Pavlidis (1976) esittelemässä *split and merge* -algoritmissa. Se perustuu siihen, että ensin alueet jaetaan pienemmiksi, jos alueen sisällä on paljon vaihtelua. Jos taas sen jälkeen kaksi vierekkäistä aluetta ovat samankaltaisia, alueet yhdistetään. Tämän toteuttamiseen tarvitaan tapaa pitää kirjaa alueista, jotka kuuluvat yhteen. *Split and merge* etenee vaiheittain seuraavasti.

1. Aloitetaan pilkkomaan nelipuuta ylimmältä tasolta ja oletetaan koko puun kuuluvan yhteen alueeseen.
2. Pilkotaan solmuja neljäksi pienemmäksi solmuksi rekursiivisesti, jos alue ei ole yhtenevä, kunnes saavutetaan solmun tasot, joita ei tarvitse pilkkoa, tai yhden pikselin kokoiset alueet.
3. Yhdistetään vierekkäisiä alueita, jotka ovat riittävän samankaltaiset, kunnes yhdistettäviä alueita ei enää ole.

Split and merge -algoritmin alkuperäinen versio vaatii, että kuvan on oltava neliön muotoinen ja mittojen on oltava kahden potensseja. Lisäksi siinä on ongelmana, että se ei toimi hyvin matalan kontrastin kuville, koska alueiden kirkkauksia käytetään suoraan yhdisteltäessä alueita. Myös yhdestä alueesta aloittaminen aiheuttaa alussa turhaa pilkkomista, koska kuva harvemmin muodostuu yhdestä segmentistä.

Nelipuumetsäsegmentoinnissa on parannus vaatimukseen, että kuva on oltava neliön muotoinen ja mittojen oltava kahden potensseja. Siinä riittää että kuvan pituus ja korkeus ovat jollakin kakkosen potenssilla jaollisia. Tämä johtuu siitä, että yhden nelipuun sijaan kuva on jaettu useisiin nelipuihin, joita kutsutaan nelipuumetsäksi.



(a) Alkuperäinen kuva.

(b) Algoritmilla segmentoitu kuva.

Kuvio 10: Segmentointi algoritmilla Eskelinen, Tirronen ja Rossi (2013).

Tästä on myös hyötyä siinä suhteessa, että nelipuuta ei tarvitse pilkkoa useita kertoja, jos segmentoitavat alueet ovat paljon pienempiä kuin neljäsosa kuvasta. Lisäksi nelipuumetsäsegmentoinnissa yhdistämiselle on uusi kriteeri.

Algoritmi 2 kuvaa nelipuumetsäsegmentointia. Algoritmi etenee vaiheittain seuraavasti.

1. Luo kuvasta nelipuumetsä.
2. Nelipuumetsässä pilkotaan solmuja pienemmiksi rekursiivisesti kunnes solmu koostuu yhtenäisestä alueesta ja lisätään yhtenäinen alue segementtilistalle.
3. Jokainen segmentti segmenttilistalla yhdistetään käyttäen *union-find* -menetelmän *union*-toimintoa naapurisegmentteihin, jos segmentit ovat samankaltaisia.

Nelipuumetsä muodostetaan jakamalla kuva samankokoisiin neliön muotoisiin alueisiin. Jokainen alue muodostaa oman nelipuun. Nelipuumetsä on joukko nelipuita, jotka taas koostuvat juurisolmuista, joilla voi olla neljä lapsisolmua ja niillä taas voi tarvittaessa olla myös neljä lapsisolmua aina rekursiivisesti kunnes yhden solmun alue vastaa yhtä kuvanpikseliä.(Eskelinen, Tirronen ja Rossi 2013)

Nelipuumetsäsegmentoinnissa erillisistä alueista pidetään kirjaa käyttäen union-find tietorakennetta. Kuva on jaettu erillisiin joukkoihin, jotka muodostavat aina yhden segmentin. Kaikkien joukkojen unioni on koko kuva. Union-findin avulla saadaan pidettyä nopeasti kirjaa mitkä nelipuun solmut kuuluvat mihinkin joukkoon ja eri joukkoja voidaan yhdistellä vakioajassa.

Union-find on vakioajassa toimiva menetelmä erillisten joukkojen hallinnointiin. Perustoiminnot ovat union, jossa kaksi joukkoa yhdistetään, ja find, jossa tarkastetaan kuuluvatko kaksi alkiota samaan joukkoon. Union-findin toteutus perustuu linkitettyihin listoihin, jotka aina kuvaavat yhtä joukkoa. Jokainen alkio on linkitetty joukon tunnisteena toimivaan alkioon. Jos kahdessa alkiossa on linkki samaan tunnistealkioon, ne kuuluvat samaan joukkoon.(Fiorio ja Gustedt 1996)

Koska nelipuumetsäsegmentointi käyttää integraalikuvaa ja pikseleiden neliöiden integraalikuvaa, voidaan nelipuusta laskea vakioajassa neliönmuotoisen alueen kes-

kiarvo ja keskihajonta aiemmin kuvatuilla menetelmillä. Nelipuumetsän ideana on, että käytössä on eräänlainen diskreetti skaala-avaruus, jossa kuvasta on käytössä versioita erilaisilla tarkkuuksilla. Tämä antaa käyttöön Gaussin pyramidin kaltaisen esitysmuodon, jossa kuvasta on olemassa aina puolta pienempi versio, joka on sumennettu keskiarvomenetelmällä Gaussin suotimen sijaan. Keskiarvomenetelmän käyttö ei ole paras mahdollinen ratkaisu, mutta se on approksimaatio, joka on nopea laskea integraalikuvan avulla.

Alueiden yhdistämiseen käytetään Felzenszwalb ja Huttenlocher (1998) esittelemää kriteeriä. Sitä kuvaa yhtälö 2.10. Kriteeri on sama, mutta nelipuumetsäsegmentoinnissa ei verrata yksittäisiä pikseleitä vaan alueita. Alueet kuuluvat samaan ryhmään, jos poikkeama alueiden välillä on pieni verrattuna alueen sisäiseen vaihteluun. Sen lisäksi käytetään toista kriteeriä, jossa alueiden keskimääräisiä intensiteettiarvoja vertaillaan ennen yhdistämistä. Se kuvataan yhtälössä 2.11. Yhtälöissä 2.10 ja 2.11 m on alueen keskiarvo ja s on alueen keskihajonta.

Yhtälö 2.10 tuottaa pieniä arvoja, jos keskiarvojen erotuksen itseisarvo on pieni verrattuna keskihajontojen summaan vastaavilla alueilla. Tämä ehto ei kuitenkaan yksin riitä, sillä suuren vaihtelevuuden alueet voidaan yhdistää pienen vaihtelevuuden alueeseen, jos niiden keskiarvo on suunnilleen sama. Siksi tarvitaan toinen kriteeri, joka on esitetty yhtälössä 2.11.

$$D(R_i, R_2) = \frac{|m_1 - m_2|}{s_1 + s_2} \tag{2.10}$$

Yhtälössä 2.11 osoittaja kuvaa alueiden R_i arvojen päällekkäisyyttä. Nimittäjä kuvaa arvojen yhdistelmää. Parametri α määrää kuinka monta keskihajontaa käytetään tuottamaan vaihteluväli. Raja-arvona alueiden yhtenevyydelle on käytetty arvoja 0.4, 0.5 ja 0.6 hyvillä tuloksilla. Arvo 0.6 toimii erityisen hyvin matala kontrastisilla kuvilla.(Eskelinen, Tirronen ja Rossi 2013)

$$D(R) = \frac{min_i(m_i + \alpha s_i) - max_i(m_i - \alpha s_i)}{max_i(m_i + \alpha s_i) - min_i(m_i - \alpha s_i)} \tag{2.11}$$

, jossa i viittaa solmun R lapsisolmuihin

Algorithm 2 Quad Forest Segmentation

```
1: procedure QUAD FOREST SEGMENTATION(image)
2:     forest ← CREATE QUAD FOREST(image)
3:     segments ← 0
4:     for every tree in forest do
5:         ADD CONSISTENT SEGMENTS(segments,tree)
6:     end for
7:     for every segment in segments do
8:         neighbours ← all neighboring segments
9:         for every neighbor in neighbors do
10:            if SIMILAR(neighbor,segments) then
11:                UNION(neighbor,segment)
12:            end if
13:        end for
14:    end for
15: end procedure
16: procedure ADD CONSISTED SEGMENTS(segments, tree)
17:    if not CONSISTENT(tree) then
18:        children ← DIVIDE(tree)
19:        for every child in children do
20:            CREATE CONSISTENT SEGMENTS(segments, tree)
21:        end for
22:    else
23:        ADD SEGMENT(segments, tree)
24:    end if
25: end procedure
```

3 Algoritmien rinnakkaistaminen

"Coming together is a beginning. Keeping together is progress. Working together is success."

— Henry Ford

Tässä luvussa käydään läpi mitä algoritmien rinnakkaistamiseksi voidaan mahdollisesti tehdä ja mitä tässä tutkimuksessa on tehty, esitellään mahdolliset alustat ja niiden ominaispiirteet liittyen kyseiseen algoritmiin, ja kerrotaan mitä osia algoritmeista on rinnakkaistettu.

3.1 Kynnystysalgoritmin rinnakkaistaminen

Kynnystysalgoritmin rinnakkaistaminen on varsin triviaalia. Onhan se "embarrassingly parallel" -tyyppinen algoritmi. Tässä tutkimuksessa kynnystysalgoritmia on käytetty vertailukohtana erilaisille alustoille. Algoritmi on hyvin yksinkertainen ja siksi algoritmi on rinnakkaistettu kaikille alustoille.

Algoritmi sisältää kynnystysalgoritmin lisäksi muunnoksen rgb-väriavaruudesta harmaasävy-väriavaruuteen. Algoritmi ei välttämättä toimi yksittäiselle kuvalle nopeammin käyttäen grafiikkasuoritinta, koska kuvan siirto edestakaisin vaatii aikaa. Eri alustoilla käytetty lähdekoodi on esitetty kunkin alustan omassa aliosiossa.

Algoritmeissa on käytetty lähdekuvana värikuvaa, ja niissä tehdään värimuunnos rgb-väriavaruudesta intensiteettiarvoksi. Aluksi algoritmeissa käytettiin NTSC-standardin mukaista muunnosta rgb-väriavaruudesta Luma-arvoksi (luminositeetti)(Bingley 1954; Poynton 2003).

$$Luma = 0.299 * R + 0.587 * G + 0.114 * B \tag{3.1}$$

Muunnosta käytettiin sitten pikselin intensiteettiarvona kynnystysoperaatiota tehdessä, mutta kyseinen kaava aiheutti tarkkuusongelmia joillakin kuvilla ja siksi siirrettiin

käyttämään sRGB-standardin mukaista kaavaa (“Luma (video) - Wikipedia, the free encyclopedia” 2014), joka toimikin paremmin.

$$Luma = 0.2126 * R + 0.7152 * G + 0.0722 * B \tag{3.2}$$

3.1.1 Rinnakkaistamaton kynnystysalgoritmi

Rinnakkaistamaton algoritmi on mukana tässä rinnakkaistusosiossa, jotta ohjelmakoodin monimutkaisuutta voidaan helposti verrata rinnakkaistettuihin koodeihin. Käytännössä algoritmi käy läpi kahdessa sisäkkäisessä silmukassa kuvan pikselit ja suorittaa niille kynnystysoperaation.

```
1  void threshold(pixel_image *src_image,
2                 pixel_image *dst_image,
3                 unsigned char threshold)
4  {
5    uint32 x,y;
6  // Käydään kahdessa sisäkkäisessä silmukassa leveys ja
7  // korkeus läpi kuvan kaikki pikselit
8    for(x =0; x < src_image->width; x++)
9    {
10     for(y =0 ; y< src_image->height; y++)
11     {
12       //lasketaan pikselin offset datassa 3 värikanavaa kertaa
13       // koordinaatti x plus koordinaatti y kertaa kuvan leveys
14       uint32 offset =3*(x+src_image->width*y);
15       //luetaan pikselien arvot datasta
16       unsigned char b = ((unsigned char*)src_image->data)[offset];
17       unsigned char g = ((unsigned char*)src_image->data)[offset+1];
18       unsigned char r = ((unsigned char*)src_image->data)[offset+2];
19       // lasketaan intensiteetti arvo pikselille
20       float intensity = (float)r*0.2126f +
```

```
21          (float)g*0.7152f +
22          (float)b*0.0722f;
23       // verrataan intensiteettiarvoa kynnystysarvoon
24        if(intensity>(float)threshold)
25        {
26        // ja jos se on suurempi asetetaan kohde pikseli valkeaksi
27          ((unsigned char*)dst_image->data)[offset] = 255;
28          ((unsigned char*)dst_image->data)[offset+1] = 255;
29          ((unsigned char*)dst_image->data)[offset+2] = 255;
30        }
31        else
32        {
33        // muuten asetetaan kohdepikseli mustaksi
34          ((unsigned char*)dst_image->data)[offset] = 0;
35          ((unsigned char*)dst_image->data)[offset+1] = 0;
36          ((unsigned char*)dst_image->data)[offset+2] = 0;
37        }
38      }
39    }
40  }
```

3.1.2 Kynnystysalgoritmi käyttäen OpenGL:ää

Kynnystysalgoritmi käyttäen OpenGL:ää vaatii niin paljon koodia, että täydellinen ohjelmakoodi on laitettu liitteisiin. Se löytyy liitteestä A. Alla esitetään kuitenkin listauksena mitä kaikkea tarvitsee tehdä, jotta näin yksinkertainen algoritmi saadaan toteutettua. Monimutkaisemmassa algoritmissa toki ylimääräisen koodin osuus jäisi hieman pienemmäksi. Listauksessa on oletettu, että virhetilanteita ei tapahdu ja niiden käsittely on jätetty pois. Liitteessä A on kuitenkin virhetilanteiden käsittely mukana.

Käytännössä rinnakkaistus tapahtui siten, että kuva renderöidään uudelleen käyttäen

OpenGL:ää ja renderöidessä fragment-varjostin muuntaa kuvan pikselit joko mustiksi tai valkeiksi riippuen pikseleiden arvoista. Vertex-varjostin vain laskee tekstuurin koordinaatit ja vastaa täysin varjostinta, jolla vain piirrettäisiin teksturoitu neliö. Kynnystysarvo on annettu uniform-tyyppisenä parametrinä fragment-varjostimille. Lopuksi kuva luetaan takaisin framebufferista. Grafiikkaprosessori suorittaa kuvan piirron automaattisesti rinnakkaistettuna.

Tähän tutkimukseen tehtiin myös toisenlainen toteutus, joka käytti piirtokohteena tekstuuria. Toteutus oli kuitenkin hiukan pidempi ja nopeus oli sama kuin esitetyssä toteutuksessa. Tekstuurin käyttämisessä piirtokohteena olisi ollut mahdollisesti hyötyä, jos kuvalle olisi tehty kynnystysoperaation jälkeen useampi toimenpide, koska kuva olisi ollut jo valmiina tekstuurissa seuraavaa vaihetta varten.

Seuraavassa listassa esitetään vaiheet, jotka OpenGL-toteutus kynnystysalgoritmistä suorittaa.

1. Generoidaan framebuffer.
2. Sidotaan luotu framebuffer.
3. Generoidaan renderbuffer.
4. Sidotaan luotu renderbuffer.
5. Asetetaan renderbufferin koko.
6. Asetetaan framebuffer käyttämään luotua renderbufferia.
7. Aktivoidaan GL_TEXTURE0.
8. Generoidaan tekstuuri alkuperäiselle kuvalle.
9. Sidotaan kyseinen tekstuuri.
10. Asetetaan tekstuurin suurennus- ja pienennysfiltterit.
11. Asetetaan tekstuurin data.
12. Luodaan vertex-varjostin ja fragment-varjostin.
13. Asetetaan vertex-varjostimen ja fragment-varjostimen lähdekoodi.
14. Käännetään vertex-varjostin ja fragment-varjostin.
15. Luodaan varjostinohjelma.
16. Kiinnitetään varjostimet varjostinohjelmaan.
17. Linkitetään varjostinohjelma.

18. Asetetaan vertex-varjostin ja fragment-varjostin tuhottavaksi.
19. Haetaan varjostinohjelman muuttujien paikat.
20. Asetetaan luotu varjostinohjelma käytettäväksi.
21. Asetetaan Viewport.
22. Asetetaan projektimatriisi ja modelviewmatriisi.
23. Sidotaan alkuperäisen kuvan sisältävä tekstuuri.
24. Asetetaan Varjostinohjelman muuttujat.
25. Piirretään teksturoitu neliö.
26. Luetaan framebufferista tuloskuva keskusmuistiin.
27. Siivotaan loput resurssit.

Jos tarkastellaan lähdekoodia,huomataan, että kynnystysalgoritmin toteutus käyttäen OpenGL:ää on huomattavasti monimutkaisempi kuin rinnakkaistamaton algoritmi. Koodi toimii ilman kontekstin luontia noin 6-10 kertaa nopeammin suurella kuvalla kuin rinnakkaistamaton algoritmi. Kontekstin luonti kestää taas hyvin kauan, mutta se tarvitsee tehdä vain ohjelman alussa. Kontekstin luontiin käytettiin tässä tutkimuksessa GLUT-kirjastoa, joka luo samalla ikkunan ikkunointijärjestelmään, jota ei tässä olisi tarvittu. OpenGL ES -toteutus on jätetty pois, koska se on huomattavan samankaltainen kuin OpenGL-toteutus, ja lisäksi käytettävä alusta ei ole vertailukelpoinen muihin alustoihin nähden.

3.1.3 Kynnystysalgoritmi käyttäen OpenCL:ää

Myös OpenCL-koodi kynnystysalgoritmille on niin pitkä, että se on esitetty kokonaisuudessaan vain liitteessä B. OpenCL-koodi on pienin muutoksin käytettävissä myös keskussuorittimella laskettavaksi. Tätä tutkimusta varten toteutettiin myös OpenCL-algoritmi, joka käytti sekä keskussuoritinta että yhtä tai useampaa grafiikkasuoritinta laskentaan. Kyseinen toteutus oli kuitenkin hitaampi kuin vain pelkkää CPU:ta tai GPU:ta käyttävä, ja siksi se jätettiin tästä tutkimuksesta pois.

Käytännössä rinnakkaistus on hoidettu yhdistämällä rinnakkaistamattoman version kaksi silmukkaa ja muuntamalla silmukoiden sisällä tehtävä laskenta kernel-

funktioksi, joka suoritetaan automaattisesti rinnakkain grafiikkaprosessorilla sopivina paloina. Palojen koko on optimoitu kysymällä OpenCL:ltä workgroupin kokoa ja käytetty sitä *local size* -parametrina kernel-funktiota suorittaessa. *Global size* parametrinä on käytetty kuvan pikselimäärää. Workgroupin koko vastaa grafiikkasuorittimella multiprosessorilla yhtä aikaa suoritettavia säikeitä ja on siksi optimaalinen.

Seuraavaksi esitetään listassa vaiheet, jotka tarvitsee suorittaa OpenCL:llä toteutetussa kynnystysalgoritmissa.

1. Haetaan alustatunnukset.
2. Haetaan laitetunnukset.
3. Valitaan käytettäväksi sama laite kuin muissa algoritmeissa.
4. Luodaan OpenCL-konteksti.
5. Luodaan Komentojono käytettävällä laitteelle.
6. Luodaan OpenCL-ohjelma kernel-funktion lähdekoodista.
7. Käännetään OpenCL-ohjelma.
8. Luodaan OpenCL-kernel OpenCL-ohjelmasta.
9. Luodaan puskurit laitteen muistiin lähde- ja kohdekuvalle.
10. Kopioidaan lähdekuva keskusmuistista laitteen muistiin.
11. Asetetaan OpenCL-kernelin parametrit.
12. Luetaan laitteen maksimi lokaali työryhmä koko.
13. Käynnistetään OpenCL-kernel laitteella.
14. Odotetaan kernelin ajon valmistuvan clFinish-käskyllä.
15. Luetaan kohdekuva laitteen muistista keskusmuistiin.
16. Vapautetaan käytetyt resurssit.

3.1.4 Kynnystysalgoritmi käyttäen CUDA:a

Kynnystysalgoritmin toteuttaminen CUDA:lla on lähes yhtä helppoa kuin rinnakkaistamaton versio. Ainoastaan alkuperäisen kuvan siirtäminen CUDA:n muistiin ja tuloksen lukeminen CUDA:n muistista on ylimääräistä verrattuna rinnakkaistamattamaan versioon nähden. Varsinainen suorituskoodi on lähes samankaltainen. Silmu-

kat on vain muodostettu uudelleen kernel-funktioksi, kuten aiemmin on esitetty, ja kernel-funktiossa pikselin indeksi on laskettu blokin ja säikeen indekseistä.

CUDA:n kääntäminen oli kuitenkin ongelmallista siksi, että CUDA käyttää omaa Nvidian nvcc-kääntäjää. Kääntäminen ja linkitys ei tapahdu yhtä helposti kuin muilla alustoilla. Lisäksi makefile-tiedoston tekeminen oli paljon monimutkaisempaa. Täydellinen CUDA-koodi ja kernel-funktio löytyvät liitteestä C. CUDA toteutus kuitenkin osoittautui kaikkein nopeimmaksi vertailtavista menetelmistä, mikä olikin odotettua , koska Karimi, Dickson ja Hamze (2010) olivat jo osoittanut CUDA:n nopeimmaksi alustaksi.

CUDA:lla rinnakkaistettu algoritmi etenee seuraavalla tavalla. Hilan ulottuvuudet on pyöristetty lähimmäksi seuraavaksi ylöspäin kuvan mitat jaettuna blokin dimensiolla. Jos kuva ei ole blokin ulottuvuuksilla jaettavissa, lasketaan vähän yli kuva-alueen ja säikeet uloimmissa blokeissa eivät laske mitään, jos kyseinen piste on kuvan ulkopuolella.

Algoritmi CUDA:lla suoritettuna etenee seuraavan listan mukaan.

1. Varaa muistia CUDA:lla lähdekuvalle.
2. Varaa muistia CUDA:lla kohdekuvalle.
3. Kopioi lähdekuva varattuun muistiin.
4. Asetetaan Blokin ja Hilan ulottuvuudet.
5. Ajetaan kernel-funktio tarvittavilla parametreilla.
6. Kopioidaan kohdekuva keskusmuistiin CUDA:n muistista.
7. Vapautetaan lähdekuva.
8. Vapautetaan kohdekuva.

3.1.5 Tulosten varmistaminen

Tätä tutkimusta varten tehtiin pieni aliohjelma, jota käytettiin kynnystysalgoritmien tulosten varmistamiseen. Ohjelma käy läpi kuvan kaikki pikselit ja vertaa niitä vertailukuvaan. Vertailukuva on tehty käyttäen rinnakkaistamatonta koodia ja se on

tulos johon tähdätään. Jos vertailualgoritmi löytää vääriä pikseleitä, se merkitsee ne punaisella virhekuvaan, johon muussa tapauksessa laitetaan oikea pikselin arvo. Lisäksi vertailualgoritmi ilmoittaa alueen, jossa on viallisia pikseleitä. Lopuksi aliohjelma tulostaa oliko kuva oikein ja palauttaa arvon, joka kertoo onko kuva sama vai eroaako se alkuperäisestä. Lähdekoodi vertailualgoritmiin löytyy liitteestä D.

Esimerkki algoritmin tuloksista löytyy kuvasta 11. Virheellisessä kuvassa oli käytetty kaikkien värikanavien sijalla vahingossa vain punaista kanavaa, josta laskettiin pikselin intensiteettiarvo ja tulos oli siksi virheellinen. Esimerkkinä on tässä virheellinen kuva, koska tarkastusalgoritmi tuotti virhekuvan vain, jos kuvassa oli virheellisiä pikseleitä. Rikkinäistä algoritmia korjattiin, kun oli huomattu, että tarkistusalgoritmi tuotti virhekuvan. Kaikki rinnakkaistetut algoritmit tuottivat testatuilla arvoilla oikean kuvan lopullisissa versioissaan. Kuvassa on myös oikea odotettu tulos oikealla ylhäällä.

3.2 Integraalikuvanlaskennan rinnakkaistaminen

Nelipuumetsäsegmentointialgoritmin rinnakkaistaminen näytönohjaimella kokonaisuudessaan olisi vaatinut enemmän työtä kuin tähän tutkimukseen oli mitoitettu. Siksi lähdimme liikkeelle rinnakkaistamalla vain integraalikuvan laskennan, mikä on on useissa tutkimuksissa suoritettu onnistuneesti (ks. Nguyen 2007; Bilgic, Horn ja Masaki 2010; Nehab ym. 2011).

Integraalikuvan laskennan rinnakkaistaminen kuitenkin osoittautui haastavammaksi kuin tutkimuksista olisi voinut päätellä. Koska käytettävissä oli rajallisesti aikaa ja kynnystysalgoritmin rinnakkaistamisen aikana oli selvinnyt, että CUDA oli nopein käytettävissä olevista alustoista, päätettiin integraalikuvan laskennan rinnakkaistaminen tehdä ainoastaan CUDA:lla.

Toimivin ratkaisu integraalikuvan laskemiseksi osoittautui olevan käyttää apuna cudpp-kirjastoa, joka perustuu Bilgic, Horn ja Masaki (2010) tutkimukseen. Cudpp-kirjastosta oli käytössä versio 2.2. Se tuki kuvan riveille suoritettavan kumulatiivisen summan laskentaa. Aikaisempien tutkimuksien suoria lähdekoodeja ei ollut käy-

(a) Alkuperäinen kuva

(b) Oikein kynnystetty kuva

(c) Väärin kynnystetty kuva

(d) Tarkistusalgoritmin tuottama virhekuva

Kuvio 11: Tarkistusalgoritmin esimerkkikuva; alkuperäinen kuva USC Image Database (2014)

tettävissä. Käytimme lähes samaa tapaa kuin Nguyen (2007) oli tekstimuodossa esitetty.

Nguyen (2007) käyttämä menetelmä sisälsi seuraavat vaiheet:

1. Poista kokonaisluku värikuvan lomitus, siten että saadaan 3 erillistä liukulukukuvaa.
2. Laske kumulatiiviset summat kullekin kuvalle jokaiselle riville rinnakkaisesti.
3. Transponoi kuvat, jotta voidaan laskea sarakkeiden summat.
4. Laske kumulatiiviset summat kullekin kuvalle jokaiselle sarakkeelle, jotka on nyt riveinä.
5. Lomita kuvat RGB-liukulukukuvaksi. Transponointia takaisin ei tarvita, koska kuvaan voidaan viitata transponoiduilla koordinaateilla.

Meidän käytössämme oli seuraavat vaiheet:

1. Nollaa memset-käskyllä keskusmuistissa olevat integraalikuvat.
2. Muunna harmaasävyinen kokonaislukukuva liukulukukuvaksi ja liukulukukuvaksi, jonka pikselit on korotettu toiseen potenssiin.
3. Laske kumulatiiviset summat molemmille kuville jokaiselle riville rinnakkaisesti käyttäen cudpp-kirjastoa.
4. Transponoi kuvat, jotta voidaan laskea sarakkeiden summat.
5. Laske kumulatiiviset summat molemmille kuville jokaiselle sarakkeelle, jotka ovat nyt riveinä.
6. Transponoi kuvat takaisin alkuperäiseen muotoon.
7. Kopioi kuvat käyttäen cudaMemcpy2D-käskyä siten, että kuva sijoittuu keskusmuistissa olevaan kuvan paikalle niin, että kuva alkaa toiselta riviltä toisesta pikselistä ja jokainen rivi alkaa rivin toisesta pikselistä.

Ohjelmakoodit löytyvät liitteestä E. Siellä on varsinainen käytetty ohjelmakoodi integraalikuvien laskentaan ja CUDA-kernelit transponointiin ja muunnokseen kokonaislukuharmaasävykuvasta kahdeksi liukulukukuvaksi, joista toinen on suora muunnos kokonaislukukuvasta ja toinen sisältää arvojen neliöt.

3.3 Nelipuumetsäsegmentoinnin rinnakkaistaminen

Varsinaista nelipuumetsäsegmentoinnin ydinosaa ei rinnakkaistettu tässä tutkimuksessa. Tässä luvussa kuitenkin mietitään mitä osia algoritmista olisi voinut rinnakkaistaa ja miten. Kovin syvällistä pohdintaa ei kuitenkaan tehdä.

Algoritmin eniten aikaa vievä osuus olisi ollut nelipuumetsän muodostamisen rinnakkaistaminen, mutta sen rinnakkaistaminen olisi vaatinut lähes kaiken muunkin rinnakkaistamista, koska siinä käsiteltiin osoittimia, jotka olisi täytyneet sijoittaa näytönohjaimen muistiin. Osoittimia tarvitaan viittaamaan nelipuiden solmuihin ja segmenttien naapureihin. Lisäksi kyseinen osuus täytyi tehdä esimerkiksi videokuvan käsittelyssä vain kerran.

Nelipuumetsäsegmentoinnin rinnakkaistaminen kokonaisuudessaan vaatisi kaikkien osuuksien siirtämistä näytönohjaimella, koska osassa algoritmia käytetään osoittimia. Osoittimia ei voi käyttää osittain keskusmuistissa ja osittain näytönohjaimen muistissa, koska niihin ei voi viitata kuin käytetystä kontekstista eli siis keskussuorittimelta tai grafiikkaprosessorilta. Koko algoritmin rinnakkaistamisessa taas on ongelmana osat joita on vaikea tai mahdoton rinnakkaistaa ja on suoritettava peräkkäislaskentana. Nämä osat on nopeampia laskea keskussuorittimella.

Esimerkiksi union-find-rakenteen kirjanpito kuuluu tähän kategoriaan. Ongelmaa olisi kyllä voinut kiertää jonkin verran tekemällä union-find rakenteen käsittelyyn funktiot, jotka olisi suojattu atomisilla operaatioilla ja prosessit olisivat voineet suorittaa muun laskennan ennen alueiden yhdistämistä. Tämä olisi kuitenkin voinut aiheuttaa pullonkaulan, joka olisi vaikuttanut negatiivisesti koko algoritmin nopeuteen.

Joitakin osia voitaisiin mahdollisesti rinnakkaistaa. Nelipuiden tasot voitaisiin generoida kuvina, jotka vastaisivat Gaussin pyramidin tasoja. Voitaisiin siis laskea auki keskiarvot ja keskihajonnat kaikille nelipuiden elementeille kerralla rinnakkain, koska näissä operaatioissa käsitellään vain yhtä nelipuun elementtia kerrallaan. Muistin käyttö lisääntyisi kuitenkin noin puolella. Lisäksi laskentaa tulisi paljon enemmän kuin nykyisessä toteutuksessa, koska nyt nämä arvot lasketaan vain, jos niitä tar-

vitaan. Olisi kyseenalaista nopeuttaisiko tämä algoritmia, mutta tätä olisi voinut kokeilla, jos aikaa olisi ollut käytettävissä enemmän.

4 Rinnakkaistamisella saatu hyöty

"Art is a harmony parallel with nature."

— Paul Cezanne

Tässä luvussa esitellään käytössä olleet vertailualustat ja kerrotaan miten algoritmien nopeuksia on mitattu. Saadut mittaustulokset raportoidaan ja niiden merkitystä analysoidaan.

4.1 Vertailualustat

Tässä tutkimuksessa käytettiin kahta erilaista vertailualustaa, joista toista käytettiin kahdella erilaisella näytönohjaimella. Alustoja ei valittu käytettäväksi tässä tutkimuksessa, vaan niitä alustoja käytettiin mitä oli saatavilla. Alustojen keskussuoritin oli suhteellisen nopea verrattuna grafiikkasuorittimeen. Erilaisilla kokoonpanoilla olisi mahdollisesti saatu erilaisia tuloksia. Linux-pöytäkoneessa oli kaksi eri näytönohjainta, joista toinen tuki kaksoistarkkuuden liukulukuja laskennassa.

Macbook Pro Retina 15 tuumaa kannettava:

- Käyttöjärjestelmä: OS X 10.9.5
- Prosessori: Intel®Core™i7 @ 2.7GHz x 4
- Muistia: 16 GB 1600 MHz DDR3
- Näytönohjaimet:
  - Intel HD Graphics 4000 1024 Mt ja
  - NVIDIA GeForce GT 650M @ 900 MHz PCIe x8 1024 Mt, josta testeissä käytettiin NVIDIAn ohjainta.

OpenCL-ominaisuudet Apple Macbook Pro Retina

Laite	Core^{TM}i7-3740QM	HD Graphics 400	Geforce GT 650M
Valmistaja	Intel	Intel	Nvidia
OpenCL:n versio	1.2	1.2	1.2
Laskentayksikköjä	8	16	2
Kellotaajuus	2700	1200	900
Globaali muisti tavua	17179869184	1073741824	1073741824

OpenGL-ominaisuudet Apple Macbook Pro Retina

Laite	Nvidia Geforce GT 650M
Käytetty OpenGL:n versio	2.1
Käytetty varjostinkielen versio	1.2

CUDA-ominaisuudet Apple Macbook Pro Retina

Laite	Nvidia Geforce GT 650M
Kellotaajuus	900 MHz
Globaali muisti tavua	1073741824
Vakiomuisti tavua	65536
Multiprosessorien määrä	2
Jaettua muistia per mp	49152
Rekistereitä per mp	65536
Säikeitä warpissa	32
Maksimi säikeitä per blokki	1024
Maksimi säikeiden ulottuvuudet	(1024, 1024, 64)
Maksimi hilan ulottuvuudet	(2147483647, 65535, 65535)

Linux-pöytäkone:

- Käyttöjärjestelmä: Ubuntu 14.04 LTS 64-bit
- Prosessori: Intel®Core^{TM}2 CPU 6400 @ 2.13GHz x 2
- Muistia: 4GB 800MHz DDR2

- näytönohjaimet:
  - GeForce 8600 GT/PCIe/SSE2 @ 1188 MHz 256 Mt
  - GeForce GT610 @ 810 MHz 1024 Mt,
    joista testeissä käytettiin molempia

OpenCL-ominaisuudet Linux-pöytäkoneella

Laite	Nvidia GeForce 8600 GT	Nvidia GeForce GT 610
OpenCL:n versio	1.1	1.1
Laskentayksikköjä	4	1
Kellotaajuus	1188	1620
Globaali muisti tavua	267714560	1072889856

OpenGL-ominaisuudet Linux-pöytäkoneella

Laite	Nvidia Geforce 8600 GT	Nvidia GeForce GT 610
Käytetty OpenGL:n versio	3.3	4.4
Tuettu varjostinkielen versio	3.3	4.4
Käytetty varjostinkielen versio	1.2	1.2

CUDA-ominaisuudet Linux-pöytäkoneella

Laite	Nvidia Geforce 8600 GT	Nvidia GeForce GT 610
Kellotaajuus	1188 MHz	1620 MHz
Globaali muisti tavua	267714560	1072889856
Vakiomuisti tavua	65536	65536
Multiprosessorien määrä	4	1
Jaettua muistia per mp	16384	49152
Rekistereitä per mp	8192	32768
Säikeitä warpissa	32	32
Maksimi säikeitä per blokki	512	1024
Maksimi säikeiden ulottuvuudet	(512, 512, 64)	(1024, 1024, 64)
Maksimi hilan ulottuvuudet	(65535, 65535, 1)	(65535, 65535, 65535)

4.2 Miten mitattiin

Eri alustoilla tutkittiin algoritmien nopeutta mittaamalla silmukassa suoritusnopeutta eri kokoisilla kuvilla. Koneet olivat mahdollisuuksien mukaan mahdollisimman vähäisellä kuormituksella. Vertailtavia kuvia oli 5 kappaletta ja jokaiselle kuvalle suoritusnopeus mitattiin peräkkäin eri alustoille.

Koska mittauskokemuksien mukaan ensimmäisen kerran suorituskerran vaihteli enemmän kuin myöhempien kertojen. Ensimmäinen suorituskerta jätettiin huomioimatta suorittamalla algoritmi kerran ennen jokaista mittaussarjaa. Mahdollinen vaihtelu saattoi johtua ainakin Macbook Pro -laitteella siitä, että laite käytti oletuksena toista näytönohjainta ja vaihtoi sen tarpeen tullen mittauksissa käytettyyn. Ohjaimen vaihtaminen saattoi kestää hetken.

Suoritusaika jokaiselle kuvalle mitattiin silmukassa 50 kertaa. Arvio suoritusajasta saatiin laskemalla miittaustulosten keskiarvo, joka on Taylor (s. 98 1997) mukaan paras arvio suoritusajalle tämän tyyppisissä mittauksissa. Erityisesti OpenGL-toteutuksella suoritusajassa oli välillä suuria vaihteluita. Yli 100% alkuperäisestä keskiarvosta eroavat tulokset hylättiin. Hylättyjä tuloksia oli kuitenkin vain muutama 50 mittauskerrasta. Keskiarvo saadaan jo aiemmin esitetyllä kaavalla 2.6.

Tämän lisäksi kuvaajiin on annettu virhearvioksi keskihajonta, joka on myös Taylor (1997) mukaan toistuvissa mittauksissa paras virhearvio. Keskihajonta saadaan laskettua kaavalla, jossa N on mittausten lukumäärä, x_i yksittäinen mittaus ja $\bar{x}$ keskiarvo.

$$\sigma_x = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(d_i)^2} = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \bar{x})^2} \tag{4.1}$$

4.3 Kynnystysalgoritmin mittausten tulokset

Kynnystysalgoritmin nopeusmittauksissa kahdella eri alustalla havaittiin seuraavaa. Kuvaajat 12a, 12b ja 12c sisältävät mittausten tulokset. Taulukoissa 1, 2 ja 3 on

Megapikseliä	CPU	σ	OpenGL	σ	CUDA	σ	OpenCL	σ
0.262144	3.31	0.63	3.87	0.48	1.61	0.55	10.06	0.51
1.048576	14.28	0.30	6.33	0.45	6.51	2.29	25.51	0.74
2.097152	40.42	1.71	11.95	2.95	6.49	1.26	45.31	0.37
3.145728	75.39	3.74	16.71	3.53	9.21	1.36	64.03	0.66
4.194304	114.62	2.56	21.15	4.24	10.93	2.37	82.24	0.99

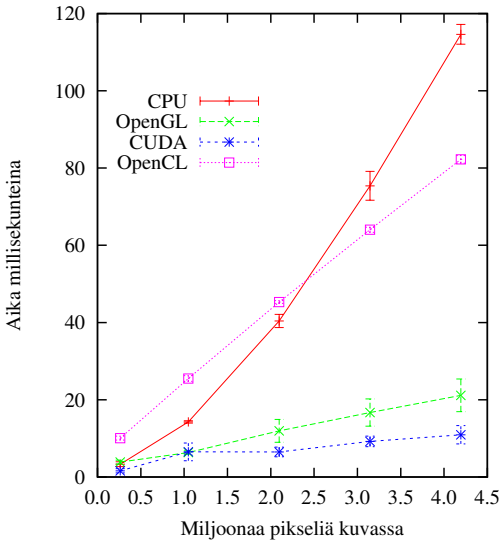
Taulukko 1: Kynnystysalgoritmin kesto millisekunteina käyttäen eri laskenta-alustoja Macbook Pro:lla.

Megapikseliä	CPU	σ	OpenGL	σ	CUDA	σ	OpenCL	σ
0.262144	8.42	1.22	6.54	1.67	4.97	0.02	40.24	2.24
1.048576	34.84	0.14	14.61	0.64	16.97	0.25	55.05	0.25
2.097152	163.13	1.84	28.90	0.88	32.83	0.26	69.86	1.31
3.145728	375.14	0.59	35.06	0.51	47.44	0.19	78.15	0.29
4.194304	507.01	1.21	45.08	0.06	63.28	0.10	93.68	0.24

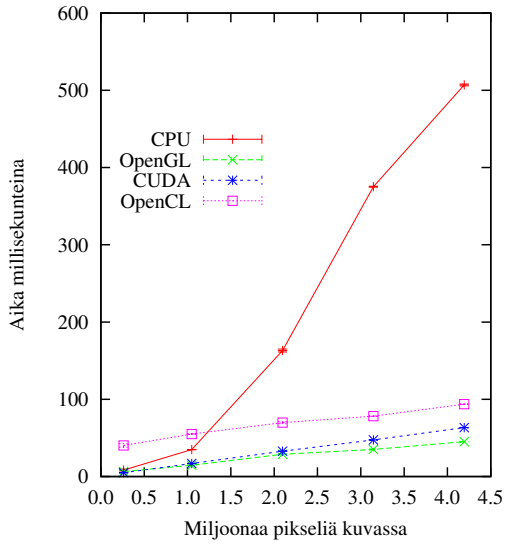
Taulukko 2: Kynnystysalgoritmin kesto käyttäen eri laskenta-alustoja Linux-pöytäkoneella käyttäen grafiikkakorttia Nvidia Geforce 8600 GT.

Megapikseliä	CPU	σ	OpenGL	σ	CUDA	σ	OpenCL	σ
0.262144	8.41	0.98	6.35	1.15	1.99	0.01	43.33	3.78
1.048576	62.46	0.34	17.72	0.99	5.49	0.23	43.03	0.58
2.097152	157.50	0.33	31.38	2.07	10.38	0.28	49.81	2.99
3.145728	383.42	0.30	44.33	0.12	14.11	0.43	52.52	0.56
4.194304	522.00	0.36	57.41	0.78	18.39	0.33	56.70	0.18

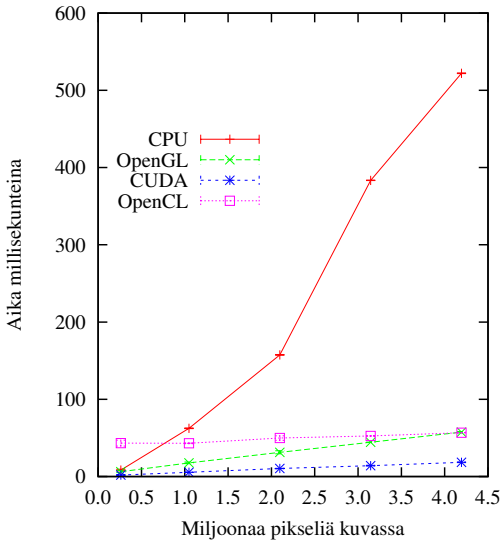
Taulukko 3: Kynnystysalgoritmin kesto millisekunteina käyttäen eri laskenta-alustoja Linux-pöytäkoneella käyttäen grafiikkakorttia Nvidia GeForce GT 610.



(a) Kynnystysalgoritmin kesto pikselimäärän funktiona eri laskenta-alustoilla Macbook Pro:lla.



(b) Kynnystysalgoritmin kesto pikselimäärän funktiona eri laskenta-alustoilla Linuxissa kortilla Nvidia Geforce 8600 GT.



(c) Kynnystysalgoritmin kesto pikselimäärän funktiona eri laskenta-alustoilla Linuxissa kortilla Nvidia GeForce GT 610.

kuvaajien pohjana olevat numeeriset arvot virhemarginaalien kanssa.

Tuloksista voidaan havaita, että Applen laitteella CUDA oli nopein. Toiseksi nopein oli OpenGL, ja OpenCL oli hiukan sen perässä. Hitain oli keskussuorittimilla tehty kynnystys lukuunottamatta aivan pieniä kuvia, joissa se oli nopeampi kuin OpenGL ja OpenCL. OpenGL-toteutuksen nopeuden vaihtelut olivat hyvin suuria. Suurilla kuvilla CUDA oli noin kymmenen kertaa nopeampi kuin keskussuorittimilla laskettu rinnakkaistamaton algoritmi.

Linux-pöytäkoneella taas OpenGL-toteutus oli nopein, jota seurasi vain pienellä erolla CUDA-toteutus. Niitä seurasi pienellä erolla OpenCL-toteutus, joka ei jäänyt jälkeen kuin pienen vakioajan myöskään isoilla kuvilla. Luultavasti ero näihin johtui siitä, että OpenCL-toteutuksessa käännetään kernel-ohjelma reaaliaikaisesti joka kuvan prosessoinnin yhteydessä. Sitä ei toisenlaisessa toteutuksessa luultavasti tarvittaisi, ja se voitaisiin jättää mittauksesta pois, kuten OpenGL-kontekstin luominen on jätetty. Tässä se on kuitenkin mukana, mutta se ei vaikuta kuvaajan kulmakertoimeen, joka on oleellisin tieto mittauksissa.

4.4 Integraalikuvan laskennan mittausten tulokset

Integraalikuvan laskentaa mitattiin vain Macbook Pro-alustalla ja toisella näytönohjaimella Linux-pöytäkoneella, koska käytössä ollut toinen Linux-pöytäkoneen näytönohjain tuki vain CUDA:n versiota 1.1. Tämä ei sisällä double-datatyyppiä ja lisäksi cudpp-kirjasto ei tue sitä. Mittausten suoritusten aikana selvisi, että rinnakkaistettu versio oli noin 1.1-1.2 kertaa hitaampi kuin rinnakkaistamaton versio. Eräs mahdollinen syy tälle olisi siirrettävä datamäärä suhteessa laskennan määrään: yhden kokonaislukukuvan siirtäminen näytönohjaimelle ja kahden kaksoistarkkuuden liukulukukuvan siirtäminen pois näytönohjaimelta vievät hyvin paljon aikaa suhteessa suoritettuun laskentaan. Lisäksi Macbook Pro:n keskussuoritin on hyvin nopea verrattuna näytönohjaimeen, joka joutuu pyörittämään myös korkean resoluution näyttökuvaa laskemisen lisäksi. Näytön tarkkuutta ei käytännössä voinut pudottaa asetuksista. Ilman kuvan siirtoa edestakaisin ja ylimääräistä transponointia rinnak-

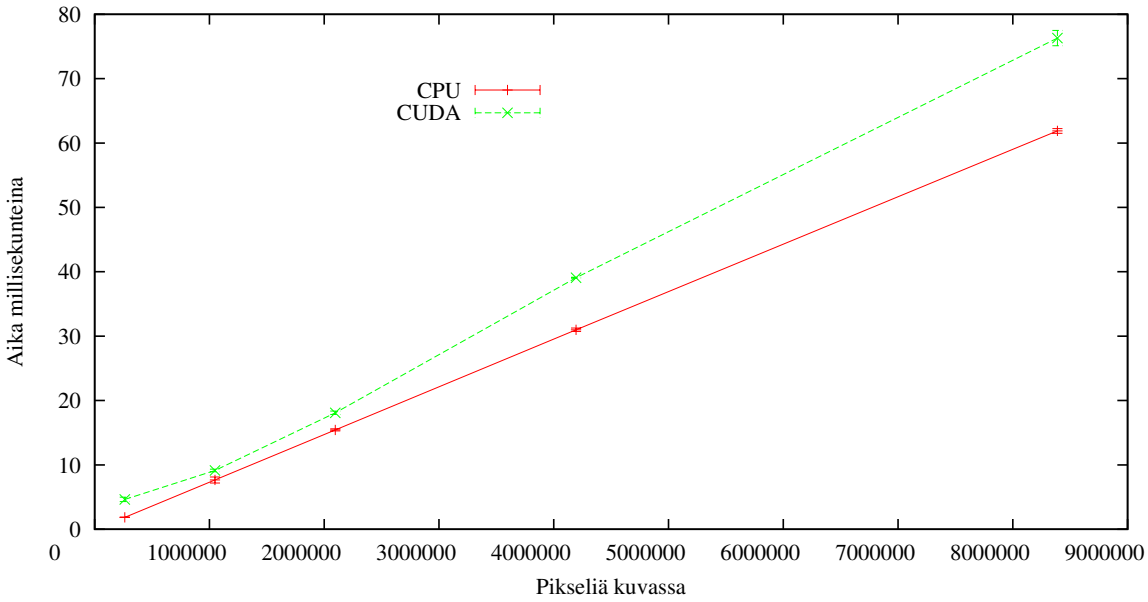
Pikseliä	CPU ms	σ	CUDA ms	σ
262,144	1.85	0.01	4.64	0.34
1,048,576	7.66	0.47	9.14	0.20
2,097,152	15.44	0.16	18.09	0.26
4,194,304	30.99	0.27	39.08	0.08
8,388,608	61.88	0.36	76.31	1.18

Taulukko 4: Integraalikuvan laskennan nopeus Macbook Pro:lla.

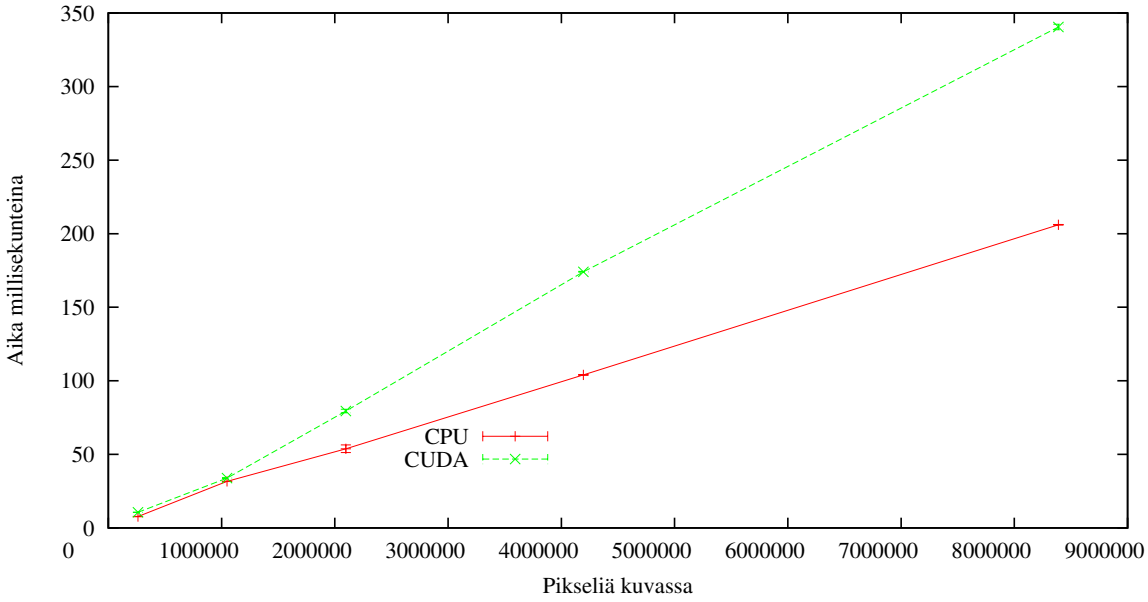
Pikseliä	CPU ms	σ	CUDA ms	σ
262,144	7.74	0.12	10.61	0.03
1,048,576	31.62	0.06	33.88	0.06
2,097,152	53.82	2.62	79.48	1.24
4,194,304	104.00	0.24	174.07	0.27
8,388,608	206.08	0.13	340.47	1.86

Taulukko 5: Integraalikuvan laskennan nopeus Linux-työasemalla.

kaistettu algoritmi olisi ollut hiukan nopeampi, mutta ei merkittävästi nopeampi, vaan samassa kertaluokassa.



(a) Integraalikuvan laskennan kesto Macbook Pro:lla pikselimäärän funktiona.



(b) Integraalikuvan laskennan kesto Linux-työasemalla pikselimäärän funktiona.

Kuvio 13: Integraalikuvan laskennan kesto eri alustoilla.

5 Yhteenveto

> *"To treat a 'big' subject in the intensely summarized fashion demanded by an evening's traffic of the stage when the evening, freely clipped at each end, is reduced to two hours and a half, is a feat of which the difficulty looms large."*
>
> *— Henry James*

Tässä työssä käsiteltiin heterogeenisten laskenta-alustojen käyttöä kuvien segmentoinnissa yleisesti ja erikseen kahdelle algoritmille. Ensimmäinen käsitelty algoritmi oli kynnystysalgoritmi, joka oli valittu siksi, että se oli helppo rinnakkaistaa. Kynnystysalgoritmi myös toteutettiin käyttäen CUDA:a, OpenCL:ää ja OpenGL:ää.

Toinen algoritmi oli Eskelinen, Tirronen ja Rossi (2013) esittämä nelipuumetsäsegmentointialgoritmi, jota ei saatu tämän tutkimuksen puitteissa rinnakkaistettua täysin. Osia kuitenkin rinnakkaistettiin ja lopun rinnakkaistamista analysoitiin. Rinnakkaistetuksi osaksi valikoitui alusta lähtien integraalikuvien laskenta, jonka rinnakkaistaminen onnistui, mutta tulokset jäivät laihoiksi. Integraalikuvan laskenta grafiikkasuorittimella oli hiukan hitaampi kuin keskussuorittimella. Syitä tähän analysoitiin.

Kahden algoritmin lisäksi käsiteltiin taustaosiossa yleisesti rinnakkaislaskentaa ja grafiikkasuoritinlaskentaa. Näistä esitettiin mahdollisimman kattavasti tutkimukselle keskeiset asiat. Kovin syvällistä kuvausta useimmista asioista ei kuitenkaan annettu, koska se olisi tehnyt tutkimuksesta laajemman kuin olisi ollut tarpeen. Lisäksi taustaosiossa käsiteltiin hiukan konenäköä tutkimuksen kannalta välttämättömiltä osin.

Algoritmien nopeuksia mitattiin laajasti käytettävissä olevilla alustoilla ja mittausten tuloksia analysoitiin. Tulokset olivat odotettuja kynnystysalgoritmille, mutta integraalikuvan laskennassa jäätiin heikompiin tuloksiin. Työn sisältö ei ole tieteellisesti merkittävä, mutta aikaisempien tutkimuksien tulokset alustojen nopeuksista varmistettiin. Erilaisten viitekehysten esittelystä ja kynnystysesimerkeistä voi olla jotain hyötyä eri viitekehyksiä harkitseville tahoille.

Tutkimuksessa olisi voitu rinnakkaistaa grafiikkaprosessorille myös kokonaan Eskelinen, Tirronen ja Rossi (2013) esittämä algoritmi, mutta valitettavasti koko algoritmin rinnakkaistaminen oli tähän tutkimukseen liian laaja tehtävä. Jos olisi keskitytty vain yhden alustan käyttöön ja rajattu rinnakkaistaminen vain sille, olisi tuloksia voitu saada käytettävissä olevassa ajassa. Algoritmin rinnakkaistaminen kokonaan jää kuitenkin jatkotutkimuksien aiheeksi.

Lähteet

Amdahl, Gene M. 1967. "Validity of the single processor approach to achieving large scale computing capabilities". Teoksessa *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, 483–485. AFIPS '67 (Spring). Atlantic City, New Jersey: ACM. doi:10.1145/1465482.1465560.

"Amdahl's law - Wikipedia, the free encyclopedia". 2013. Viitattu 12. marraskuuta 2013. http://en.wikipedia.org/wiki/Amdahl's_law.

Asanovic, Krste, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams ym. 2006. *The landscape of parallel computing research: A view from Berkeley.* Tekninen raportti. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley.

Barney, Blaise. 2010. "Introduction to parallel computing". Viitattu 16. lokakuuta 2014. https://computing.llnl.gov/tutorials/parallel_comp/.

Bernstein, A.J. 1966. "Analysis of programs for parallel processing". *Electronic Computers, IEEE Transactions on* EC-15 (5): 757–763. doi:10.1109/PGEC.1966.264565.

Bilgic, B., B. K P Horn ja I. Masaki. 2010. "Efficient integral image computation on the GPU". Teoksessa *Intelligent Vehicles Symposium (IV), 2010 IEEE*, 528–533. doi:10.1109/IVS.2010.5548142.

Bingley, F.J. 1954. "Colorimetry in Color Television-Part II". *Proceedings of the IRE* 42, numero 1 (tammikuuta): 48–51. doi:10.1109/JRPROC.1954.274606.

Blelloch, G.E. 1989. "Scans as primitive parallel operations". *Computers, IEEE Transactions on* 38, numero 11 (marraskuuta): 1526–1538. doi:10.1109/12.42122.

Blelloch, Guy E. 1990. *"Prefix sums and their applications"*. Tekninen raportti. Computer Science Department, Carnegie Mellon University.

Blythe, D. 2008. “Rise of the Graphics Processor”. *Proceedings of the IEEE* 96, numero 5 (toukokuuta): 761–778. doi:10.1109/JPROC.2008.917718.

Cohen, Jonathan, ja Michael Garland. 2009. “Solving Computational Problems with GPU Computing”. *Computing in Science and Engineering* (Los Alamitos, CA, USA) 11 (5): 58–63. doi:http://doi.ieeecomputersociety.org/10.1109/MCSE.2009.144.

Crow, Franklin C. 1984. “Summed-area tables for texture mapping”. *SIGGRAPH Comput. Graph.* (New York, NY, USA) 18, numero 3 (tammikuuta): 207–212. doi:10.1145/964965.808600.

“Data Reduction - Wikipedia, the free encyclopedia”. 2014. Viitattu 16. syyskuuta 2014. http://en.wikipedia.org/wiki/Data_reduction.

“Embarrassingly parallel - Wikipedia, the free encyclopedia”. 2014. Viitattu 17. tammikuuta 2014. http://en.wikipedia.org/wiki/Embarrassingly_parallel.

Eskelinen, Matti, Ville Tirronen ja Tuomo Rossi. 2013. “Fast and robust segmentation of low-contrast images”.

Felzenszwalb, P.F., ja D.P. Huttenlocher. 1998. “Image segmentation using local variation”. Teoksessa *Computer Vision and Pattern Recognition, 1998. Proceedings. 1998 IEEE Computer Society Conference on*, 98–104. doi:10.1109/CVPR.1998.698594.

Feng, Wu-chun, Heshan Lin, Thomas Scogland ja Jing Zhang. 2012. “OpenCL and the 13 Dwarfs: A Work in Progress”. Teoksessa *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*, 291–294. ICPE ’12. Boston, Massachusetts, USA: ACM. doi:10.1145/2188286.2188341.

Fiorio, Christophe, ja Jens Gustedt. 1996. “Two linear time union-find strategies for image processing”. *Theoretical Computer Science* 154 (2): 165–181.

Flynn, M. 1972. “Some Computer Organizations and Their Effectiveness”. *Computers, IEEE Transactions on* C-21 (9): 948–960. doi:10.1109/TC.1972.5009071.

Foster, Ian. 1995. *”Designing and building parallel programs”.* Addison Wesley Publishing Company.

Gonzales, Rafael C, ja Richard E Woods. 2002. *"Digital Image Processing, 2-nd Edition"*. Prentice Hall. ISBN: 0-13-094650-8.

Gustafson, John L. 1988. "Reevaluating Amdahl's law". *Commun. ACM* (New York, NY, USA) 31, numero 5 (toukokuuta): 532–533. doi:10.1145/42411.42415.

"Gustafson's law - Wikipedia, the free encyclopedia". 2013. Viitattu 12. marraskuuta 2013. http://en.wikipedia.org/wiki/Gustafson's_law.

Hensley, Justin, Thorsten Scheuermann, Greg Coombe, Montek Singh ja Anselmo Lastra. 2005. "Fast Summed-Area Table Generation and its Applications". *Computer Graphics Forum* 24 (3): 547–555. doi:10.1111/j.1467-8659.2005.00880.x.

Hillis, W. Daniel, ja Guy L. Steele Jr. 1986. "Data Parallel Algorithms". *Commun. ACM* (New York, NY, USA) 29, numero 12 (joulukuuta): 1170–1183. doi:10.1145/7902.7903.

Horowitz, Steven L., ja Theodosios Pavlidis. 1976. "Picture Segmentation by a Tree Traversal Algorithm". *J. ACM* (New York, NY, USA) 23, numero 2 (huhtikuuta): 368–388. doi:10.1145/321941.321956.

Karimi, Kamran, Neil G. Dickson ja Firas Hamze. 2010. "A Performance Comparison of CUDA and OpenCL". *Computer Research Repository* abs/1005.2581. http://arxiv.org/abs/1005.2581.

Knuth, Donald Erwin. 1997. *"The Art of Computer Programming second edition, Volume 3 Sorting and Searching"*. Addison-Wesley, Reading, Mass. ISBN: 978-0-201-89685-5.

Komatsu, Kazuhiko, Katsuto Sato, Yusuke Arai, Kentaro Koyama, Hiroyuki Takizawa ja Hiroaki Kobayashi. 2010. "Evaluating performance and portability of OpenCL programs". Teoksessa *The Fifth International Workshop on Automatic Performance Tuning*, 7.

Ladner, Richard E., ja Michael J. Fischer. 1980. "Parallel Prefix Computation". *J. ACM* (New York, NY, USA) 27, numero 4 (lokakuuta): 831–838. doi:10.1145/322217.322232.

Lee, Jaejin, Jungwon Kim, Sangmin Seo, Seungkyun Kim, Jungho Park, Honggyu Kim, Thanh Tuan Dao ym. 2010. "An OpenCL Framework for Heterogeneous Multicores with Local Memory". Teoksessa *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, 193–204. PACT '10. Vienna, Austria: ACM. ISBN: 978-1-4503-0178-7. doi:10.1145/1854273.1854301.

"Luma (video) - Wikipedia, the free encyclopedia". 2014. Viitattu 3. lokakuuta 2014. http://en.wikipedia.org/wiki/Luma_(video).

"MISD - Wikipedia, the free encyclopedia". 2013. Viitattu 23. lokakuuta 2013. http://en.wikipedia.org/wiki/MISD.

Moler, Cleve. 1986. "Matrix computation on distributed memory multiprocessors". *Hypercube Multiprocessors* 86:181–195.

Munshi, Aaftab, Benedict Gaster, Timothy G Mattson, James Fung ja Dan Ginsburg. 2012. *"OpenCL programming guide"*. Pearson Education. ISBN: 978-0-321-74964-2.

Nehab, Diego, André Maximo, Rodolfo S Lima ja Hugues Hoppe. 2011. "GPU-efficient recursive filtering and summed-area tables". *ACM Transactions on Graphics (TOG)* 30 (6): 176.

Nguyen, Hubert. 2007. *"GPU Gems 3"*. Addison-Wesley Professional. ISBN: 9780321515261.

Nickolls, John, Ian Buck, Michael Garland ja Kevin Skadron. 2008. "Scalable Parallel Programming with CUDA". *Queue* (New York, NY, USA) 6, numero 2 (maaliskuuta): 40–53. doi:10.1145/1365490.1365500.

OpenCL Working Group ym. 2012. *"The OpenCL specification, version 1.2, revision 17"*. https://www.khronos.org/registry/cl/specs/opencl-1.2.pdf.

Owens, J.D., M. Houston, D. Luebke, S. Green, J.E. Stone ja J.C. Phillips. 2008. "GPU Computing". *Proceedings of the IEEE* 96, numero 5 (toukokuuta): 879–899. doi:10.1109/JPROC.2008.917757.

Poil. 2005. Viitattu 14. toukokuuta 2005. http://commons.wikimedia.org/wiki/File:Fivestagespipeline.png.

Poil. 2010. Viitattu 15. maaliskuuta 2010. http://commons.wikimedia.org/wiki/File:Superscalarpipeline.png.

Poynton, Charles. 2003. "YUV and luminance considered harmful". *Digital Video and HDTV Algorithms and Interfaces:*595–600.

"Prefix sum - Wikipedia, the free encyclopedia". 2014. Viitattu 2. syyskuuta 2014. http://en.wikipedia.org/wiki/Prefix_sum.

Rost, Randi J, John M Kessenich, Barthold Lichtenbelt, Hugh Malan ja Mike Weiblen. 2007. *"OpenGL shading language"*. Second edition. Crawfordsville, Indiana: Addison-Wesley. ISBN: 0-321-33489-2.

Sanders, Jason, ja Edward Kandrot. 2010. *"CUDA by example: an introduction to general-purpose GPU programming"*. Addison-Wesley Professional. ISBN: 978-0-13-138768-3.

Sezgin, Mehmet, ja Bulent Sankur. 2004. "Survey over image thresholding techniques and quantitative performance evaluation". *Journal of Electronic Imaging* 13 (1): 146–168. doi:10.1117/1.1631315.

Shafait, Faisal, Daniel Keysers ja Thomas M Breuel. 2008. "Efficient implementation of local adaptive thresholding techniques using integral images." *DRR* 6815:681510.

Shreiner, Dave, Mason Woo, Jackie Neider, Tom Davis ym. 2007. *"OpenGL ®programming guide: The official guide to learning OpenGL ®, version 2.1"*. Sixth edition. Stoughton, Massachusetts: Addison-Wesley Professional. ISBN: 978-0-321-48100-9.

Stone, John E., David Gohara ja Guochun Shi. 2010. "OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems". *Computing in Science and Engineering* 12 (3): 66–73. doi:http://dx.doi.org/10.1109/MCSE.2010.69.

"Summed area table - Wikipedia, the free encyclopedia". 2014. Viitattu 9. syyskuuta 2014. http://en.wikipedia.org/wiki/Summed_area_table.

Tarjan, R., ja U. Vishkin. 1985. "An Efficient Parallel Biconnectivity Algorithm". *SIAM Journal on Computing* 14 (4): 862–874. doi:10.1137/0214061. eprint: http://dx.doi.org/10.1137/0214061.

Taylor, John R. 1997. *"Introduction to error analysis, the study of uncertainties in physical measurements"*. Second edition. Sausalito, California: University Science Books. ISBN: 0-935702-42-3.

"Thresholding (image processing)- Wikipedia, the free encyclopedia". 2014. Viitattu 10. maaliskuuta 2014. `http://en.wikipedia.org/wiki/Thresholding_(image_processing)`.

USC Image Database. 2014. "Girl (Lena, or Lenna)". Viitattu 3. joulukuuta. `http://sipi.usc.edu/database/download.php?vol=misc&img=4.2.04`.

Weisstein, Eric W. 2014. "Cumulative Sum – from Wolfram MathWorld". Viitattu 8. syyskuuta 2014. `http://mathworld.wolfram.com/CumulativeSum.html`.

"What Is Cuda". 2013. Viitattu 16. lokakuuta 2013. `https://developer.nvidia.com/what-cuda`.

Wilt, Nicholas. 2013. *"The CUDA Handbook: A Comprehensive Guide to GPU Programming"*. Addison-Wesley Professional. ISBN: 978-0-321-80946-9.

Viola, Paul, ja Michael Jones. 2001. "Rapid object detection using a boosted cascade of simple features". Teoksessa *Computer Vision and Pattern Recognition, 2001. CVPR 2001. Proceedings of the 2001 IEEE Computer Society Conference on*, 1:I–511. IEEE.

Liitteet

A Kynnystysalgoritmin toteutus käyttäen OpenGL:ää lähdekoodi

C- koodi

```
1  void thresholdopengl(pixel_image *src_image,
2                       pixel_image *dst_image,
3                       unsigned char threshold)
4  {
5      GLuint framebuffer,renderbuffer, originalImage;
6      GLenum status;
7
8      glGenFramebuffersEXT(1, &framebuffer);
9      glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, framebuffer);
10
11     glGenRenderbuffersEXT(1, &renderbuffer);
12     glBindRenderbufferEXT(GL_RENDERBUFFER_EXT, renderbuffer);
13     glRenderbufferStorageEXT(GL_RENDERBUFFER_EXT, GL_RGB,
14         (int)src_image->width,
15         (int)src_image->height);
16     glFramebufferRenderbufferEXT(GL_FRAMEBUFFER_EXT,
17         GL_COLOR_ATTACHMENT0_EXT,
18         GL_RENDERBUFFER_EXT, renderbuffer);
19     status = glCheckFramebufferStatusEXT(GL_FRAMEBUFFER_EXT);
20
21     if (status != GL_FRAMEBUFFER_COMPLETE_EXT)
22     {
23         printf("error\n");
24         return;
25     }
26     glActiveTexture(GL_TEXTURE0);
```

```
27      glGenTextures(1, &originalImage);
28      glBindTexture(GL_TEXTURE_2D, originalImage);
29      glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
30          GL_NEAREST);
31      glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
32          GL_NEAREST);
33      glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, (int)src_image->width,
34                   (int)src_image->height, 0, GL_RGB,
35                   GL_UNSIGNED_BYTE,  src_image->data);
36      GLuint vertexShaderObject, fragmentShaderObject;
37
38      vertexShaderObject = glCreateShader(GL_VERTEX_SHADER);
39      fragmentShaderObject = glCreateShader(GL_FRAGMENT_SHADER);
40
41      const GLchar * vertexShaderSource = &vertexshader[0];
42      const GLchar * fragmentShaderSource = &fragmentshader2[0];
43      glShaderSource(vertexShaderObject, 1, &vertexShaderSource,
44          NULL);
45      glShaderSource(fragmentShaderObject, 1, &fragmentShaderSource,
46          NULL);
47      GLint compiled;
48      glCompileShader(vertexShaderObject);
49      glGetObjectParameterivARB((void*)
50          (unsigned long)vertexShaderObject,
51          GL_COMPILE_STATUS, &compiled);
52      if(!compiled)
53      {
54          GLint blen = 0;
55          GLsizei slen = 0;
56
57          glGetShaderiv(vertexShaderObject, GL_INFO_LOG_LENGTH ,
```

```
58                          &blen);
59          if (blen > 1)
60          {
61              GLchar* compiler_log =
62                  (GLchar*)malloc((unsigned long)blen);
63              glGetInfoLogARB((void*)
64                  (unsigned long)vertexShaderObject,
65                  blen, &slen, compiler_log);
66              printf( "compiler_log:\n␣%s␣\n", compiler_log);
67              free (compiler_log);
68          }
69          return;
70      }
71
72      glCompileShader(fragmentShaderObject);
73
74      glGetObjectParameterivARB(
75          (void*)(unsigned long)fragmentShaderObject,
76          GL_COMPILE_STATUS, &compiled);
77      if (!compiled)
78      {
79          GLint blen = 0;
80          GLsizei slen = 0;
81
82          glGetShaderiv(fragmentShaderObject, GL_INFO_LOG_LENGTH ,
83                          &blen);
84          if (blen > 1)
85          {
86              GLchar* compiler_log =
87                  (GLchar*)malloc((unsigned long)blen);
88              glGetInfoLogARB(
```

```
89                  (void*)(unsigned long)fragmentShaderObject, blen,
90                  &slen, compiler_log);
91              printf( "compiler_log:\n␣%s␣\n", compiler_log);
92              free (compiler_log);
93          }
94          return;
95      }
96
97      GLuint ProgramObject = glCreateProgram();
98
99      glAttachShader(ProgramObject, vertexShaderObject);
100     glAttachShader(ProgramObject, fragmentShaderObject);
101     glLinkProgram(ProgramObject);
102     glDeleteShader(vertexShaderObject);
103     glDeleteShader(fragmentShaderObject);
104
105     GLint linked;
106     glGetProgramiv(ProgramObject, GL_LINK_STATUS, &linked);
107     if (!linked)
108     {
109         GLint blen = 0;
110         GLsizei slen = 0;
111
112         glGetShaderiv(ProgramObject, GL_INFO_LOG_LENGTH , &blen);
113         if (blen > 1)
114         {
115             GLchar* compiler_log =
116                 (GLchar*)malloc((unsigned long)blen);
117             glGetInfoLogARB((void*)(unsigned long)ProgramObject,
118                 blen, &slen, compiler_log);
119             printf( "compiler_log:\n␣%s␣\n", compiler_log);
```

```
120             free (compiler_log);
121         }
122         return;
123     }
124 
125     GLint thresholdLoc =
126         glGetUniformLocation(ProgramObject, "threshold");
127     GLint textureLoc =
128         glGetUniformLocation(ProgramObject, "originalImage");
129     glUseProgram(ProgramObject);
130 
131     glViewport(0, 0, (int)src_image->width, (int)src_image->height);
132     glMatrixMode(GL_PROJECTION);
133     glLoadIdentity();
134     glOrtho(0.0f, 1.0f, 0.0f, 1.0f, -1.0l, 1.0l);
135     glMatrixMode(GL_MODELVIEW);
136     glLoadIdentity();
137     glBindTexture(GL_TEXTURE_2D,originalImage);
138 
139     if (thresholdLoc != -1)
140     {
141         glUniform1f(thresholdLoc, (float)(threshold/255.0));
142 
143     }
144     else
145     {
146         printf("threshold␣location␣missing\n");
147         return;
148     }
149     if (textureLoc != -1)
150     {
```

```
151
152         glUniform1i(textureLoc, 0);
153     }
154     else
155     {
156         printf("originalImage location missing\n");
157         return;
158     }
159
160     glBegin(GL_QUADS);
161     glTexCoord2f(0.0f, 0.0f);
162     glVertex2i(0, 0);
163     glTexCoord2f(1.0f, 0.0f);
164     glVertex2f(1.0f, 0);
165     glTexCoord2f(1.0f, 1.0f);
166     glVertex2f(1.0f, 1.0f);
167     glTexCoord2f(0.0f, 1.0f);
168     glVertex2f(0, 1.0f);
169     glEnd();
170
171     glReadPixels(0,0,(int)src_image->width,(int)src_image->height,
172         GL_BGR,GL_UNSIGNED_BYTE,
173          (unsigned char*)(dst_image->data));
174     glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, 0);
175     glDeleteTextures(1, &originalImage);
176     glDeleteFramebuffersEXT(1,&renderbuffer);
177     glDeleteFramebuffersEXT(1, &framebuffer);
178     glDeleteProgram(ProgramObject);
179 }
```

vertex-varjostin

```
1  #version 120
2  void main()
3  {
4          gl_Position = gl_ProjectionMatrix *gl_Vertex;
5          gl_TexCoord[0] = gl_MultiTexCoord0;
6  }
```

fragment-varjostin

```
1  #version 120
2  uniform sampler2D originalImage;
3  uniform float threshold;
4  void main()
5  {
6          vec3 originalColor = texture2D(originalImage,
7                  gl_TexCoord[0].xy).rgb;
8          vec3 colorTransform = vec3(0.0722,0.7152,0.2126);
9          if(dot(originalColor,colorTransform) > threshold)
10         {
11                 gl_FragColor = vec4(1.0,1.0,1.0,1.0);
12         }
13         else
14         {
15                 gl_FragColor = vec4(0.0,0.0,0.0,1.0);
16         }
17 }
```

B Kynnystysalgoritmin toteutus käyttäen OpenCL:ää lähdekoodi

C- koodi

```
1   void thresholdopencl(pixel_image *src_image,
2                        pixel_image *dst_image,
```

```
3                       unsigned char threshold)
4 {
5     int err;                    // error code returned from api calls
6     size_t global;        // global domain size for our calculation
7     size_t local;          // local domain size for our calculation
8
9     cl_device_id device_id[2];              // compute device id
10    cl_context context;                   // compute context
11    cl_command_queue commands;            // compute command queue
12    cl_program program;                   // compute program
13    cl_kernel kernel;                     // compute kernel
14
15    cl_mem input;        // device memory used for the input array
16    cl_mem output;      // device memory used for the output array
17
18    unsigned int count = (unsigned int)(src_image->width*
19                src_image->height);
20    // Connect to a compute device
21    int gpu = 1;
22    cl_uint numdevices = 0;
23    cl_uint numplatforms = 0;
24    cl_platform_id platform_id[2];
25    err = clGetPlatformIDs(2,platform_id,&numplatforms);
26    err = clGetDeviceIDs(platform_id[0], gpu ? CL_DEVICE_TYPE_GPU :
27        CL_DEVICE_TYPE_CPU, 2, device_id, NULL);
28    if (err != CL_SUCCESS)
29    {
30        printf("Error:␣Failed␣to␣create␣a␣device␣group!\n");
31        return;
32    }
33    int deviceindex = 0;
```

```
34      // Create a compute context
35      cl_context_properties contextproperties[] = {
36          CL_CONTEXT_PLATFORM, (cl_context_properties)platform_id[0],
37          0};
38      context = clCreateContext(contextproperties, 1,
39          &device_id[deviceindex], &pfn_notify, NULL, &err);
40      if (!context)
41      {
42          printf("Error:␣Failed␣to␣create␣a␣compute␣context!\n");
43          return;
44      }
45
46      // Create a command commands
47      commands = clCreateCommandQueue(context, device_id[deviceindex],
48          numdevices, &err);
49       if (!commands)
50       {
51          printf("Error:␣Failed␣to␣create␣a␣command␣commands!\n");
52          return;
53       }
54
55
56      // Create the compute program from the source buffer
57      program = clCreateProgramWithSource(context, 1,
58          (const char **) & KernelSource, NULL, &err);
59      if (!program)
60      {
61          printf("Error:␣Failed␣to␣create␣compute␣program!\n");
62          return;
63      }
64
```

```
65      // Build the program executable
66      err = clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
67      if (err != CL_SUCCESS)
68      {
69          size_t len;
70          char buffer[2048];
71          printf("Error: Failed to build program executable!\n");
72          clGetProgramBuildInfo(program, device_id[deviceindex],
73                                CL_PROGRAM_BUILD_LOG, sizeof(buffer),
74                  buffer, &len);
75          printf("%s\n", buffer);
76          return;
77      }
78
79      // Create the compute kernel in the program we wish to run
80      kernel = clCreateKernel(program, "threshold", &err);
81      if (!kernel || err != CL_SUCCESS)
82      {
83          printf("Error: Failed to create compute kernel!\n");
84          return;
85      }
86
87      // Create the input and output arrays in device memory
88      // for our calculation
89      input = clCreateBuffer(context,  CL_MEM_READ_ONLY,
90          sizeof(unsigned char)* 3 * count, NULL, NULL);
91      output = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
92          sizeof(unsigned char)* 3 * count, NULL, NULL);
93      if (!input || !output)
94      {
95          printf("Error: Failed to allocate device memory!\n");
```

```
96          return;
97      }
98
99      // Write our data set into the input array in device memory
100     err = clEnqueueWriteBuffer(commands, input, CL_TRUE, 0,
101         sizeof(unsigned char)*3 * count,
102         ((unsigned char *)src_image->data), 0, NULL, NULL);
103     if (err != CL_SUCCESS)
104     {
105         printf("Error: Failed to write to source array!\n");
106         return;
107     }
108
109     // Set the arguments to our compute kernel
110     err = 0;
111     err  = clSetKernelArg(kernel, 0, sizeof(cl_mem), &input);
112     err |= clSetKernelArg(kernel, 1, sizeof(cl_mem), &output);
113     err |= clSetKernelArg(kernel, 2, sizeof(unsigned int), &count);
114     err |= clSetKernelArg(kernel, 3, sizeof(unsigned char),
115         &threshold);
116     if (err != CL_SUCCESS)
117     {
118         printf("Error: Failed to set kernel arguments! %d\n", err);
119         return;
120     }
121
122     // Get the maximum work group size for executing the kernel
123     //  on the device
124     err = clGetKernelWorkGroupInfo(kernel, device_id[deviceindex],
125         CL_KERNEL_WORK_GROUP_SIZE, sizeof(local),
126         &local, NULL);
```

```
127     if (err != CL_SUCCESS)
128     {
129         printf("Error: Failed to get kernel work group info! %d\n",
130                 err);
131         return;
132     }
133 
134     // Execute the kernel over the entire image
135     // using the maximum number of work group items for this device
136     global = count;
137     err = clEnqueueNDRangeKernel(commands, kernel, 1, NULL, &global,
138         &local, 0, NULL, NULL);
139     if (err)
140     {
141         printf("Error: Failed to execute kernel!\n");
142         return;
143     }
144 
145     // Wait for the command commands to get serviced before reading
146     // back results
147     clFinish(commands);
148 
149     // Read back the results from the device to verify the output
150     err = clEnqueueReadBuffer( commands, output, CL_TRUE, 0,
151         sizeof(unsigned char) * 3 * count,
152         ((unsigned char*)dst_image->data), 0, NULL, NULL );
153     if (err != CL_SUCCESS)
154     {
155         printf("Error: Failed to read output array! %d\n", err);
156         return;
157     }
```

```
158     clFinish(commands);
159     clReleaseMemObject(input);
160     clReleaseMemObject(output);
161     clReleaseProgram(program);
162     clReleaseKernel(kernel);
163     clReleaseCommandQueue(commands);
164     clReleaseContext(context);
165
166 }
```

OpenCL Kernel

```
1  __kernel void threshold(
2     __global unsigned char* input,
3     __global unsigned char* output,
4     const unsigned int count,
5     const unsigned char thresholdvalue)
6  {
7     unsigned int i = get_global_id(0);
8     if(i < count)
9     {
10        unsigned int offset = i*3;
11        float4 color = (float4)((float)input[offset],
12         (float)input[offset+1], (float)input[offset+2],0.0f);
13        float4 transform= (float4)(0.0722f,0.7152f,0.2126f,0.0f);
14        if(dot(color,transform)>(float)thresholdvalue)
15        {
16          output[offset] = 255;
17          output[offset+1] = 255;
18          output[offset+2] = 255;
19        }
20        else
```

```
21          {
22            output[offset] = 0;
23            output[offset+1] = 0;
24            output[offset+2] = 0;
25          }
26      }
27  }
```

C Kynnystysalgoritmin toteutus käyttäen CUDA:a lähdekoodi

C++- koodi

```
1   void thresholdcuda(pixel_image *src_image,
2                      pixel_image *dst_image,
3                      unsigned char threshold)
4   {
5       unsigned char *input;
6       unsigned char *output;
7       const long unsigned int size =
8           src_image->width*src_image->height*3;
9       cudaMalloc((void**)&input, size);
10      cudaMalloc((void**)&output, size);
11      cudaMemcpy( input, (unsigned char*)(src_image->data), size,
12          cudaMemcpyHostToDevice );
13      int threaddim = 32;
14      dim3 dimBlock( threaddim,threaddim);
15      dim3 dimGrid( (src_image->width+(threaddim-1))/threaddim,
16          (src_image->height+(threaddim-1))/threaddim);
17      thresholdcudakernel<<<dimGrid, dimBlock>>>(input, output,
18          src_image->width,src_image->height,threshold);
19      cudaMemcpy( (unsigned char*)(dst_image->data), output,
20          size, cudaMemcpyDeviceToHost );
```

```
21      cudaFree( input );
22      cudaFree( output );
23  }
```

Kernel-funktio

```
1    __global__
2  void thresholdcudakernel(unsigned char *input,
3                           unsigned char *output,
4                           const unsigned int width,
5                            const unsigned int height,
6                           const unsigned char thresholdvalue)
7  {
8      unsigned  int j = blockIdx.y*blockDim.y+ threadIdx.y;
9      unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
10     unsigned int pixel = i+j*blockDim.x*gridDim.x;
11     if(i<width && j < height)
12     {
13         unsigned int offset = pixel*3;
14         unsigned char b =input[offset];
15         unsigned char g =input[offset+1];
16         unsigned char r =input[offset+2];
17         float intensity = r*0.2126f +
18         g*0.7152f +
19         b*0.0722f;
20         if(intensity>(float)thresholdvalue)
21         {
22             output[offset] = 255;
23             output[offset+1] = 255;
24             output[offset+2] = 255;
25         }
26         else
```

```
27          {
28              output[offset] = 0;
29              output[offset+1] = 0;
30              output[offset+2] = 0;
31          }
32      }
33  }
```

D Kuvan oikeellisuuden varmistava ohjelma

```
1   int checker(pixel_image *ref_image,
2               pixel_image *test_image,
3               pixel_image *result_image)
4   {
5       uint32 width,height;
6       unsigned int x,y;
7
8       width = ref_image->width;
9       height = ref_image->height;
10      int correct = 1;
11      unsigned int minwidth = (unsigned int)width;
12      unsigned int maxwidth = 0;
13      unsigned int minheight = (unsigned int)height;
14      unsigned int maxheight = 0;
15      for(x =0; x < width; x++)
16      {
17          for(y =0 ; y< height; y++)
18          {
19              uint32 offset =3*(x+width*y);
20              unsigned char b = ((unsigned char*)ref_image->data)
21                  [offset];
```

```
22              unsigned char g = ((unsigned char*)ref_image->data)
23                  [offset+1];
24              unsigned char r = ((unsigned char*)ref_image->data)
25                  [offset+2];
26
27              unsigned char tb = ((unsigned char*)test_image->data)
28                  [offset];
29              unsigned char tg = ((unsigned char*)test_image->data)
30                  [offset+1];
31              unsigned char tr = ((unsigned char*)test_image->data)
32                  [offset+2];
33
34              if(r == tr && g == tg && b == tb)
35              {
36                  ((unsigned char*)result_image->data)[offset] = tb;
37                  ((unsigned char*)result_image->data)[offset+1] = tg;
38                  ((unsigned char*)result_image->data)[offset+2] = tr;
39              }
40              else
41              {
42                  ((unsigned char*)result_image->data)[offset] = 0;
43                  ((unsigned char*)result_image->data)[offset+1] = 0;
44                  ((unsigned char*)result_image->data)[offset+2] = 255;
45                  correct = 0;
46                  if(x<minwidth)
47                  {
48                      minwidth =x;
49                  }
50                  if(x>maxwidth)
51                  {
52                      maxwidth = x;
```

```
53                  }
54                  if(y<minheight)
55                  {
56                      minheight =y;
57                  }
58                  if(y>maxheight)
59                  {
60                      maxheight =y;
61                  }
62              }
63          }
64      }
65
66      if(!correct)
67      {
68          printf("Image␣has␣error␣at␣area␣%u␣,␣%u,␣%u,␣%u\n",
69                              minwidth,minheight,maxwidth,maxheight);
70          return 1;
71      }
72      else
73      {
74          printf("Image␣was␣correct\n");
75          return 0;
76      }
77  }
```

E Integraalikuvan laskenta CUDA:lla

Varsinainen ohjelmakoodi

```
1  result integral_image_updatecuda
2  (
```

```
3   integral_image *target
4   )
5  {
6
7       pixel_image *source;
8       source = target->original;
9
10      uint32 width, height;
11      width = source->width;
12      height = source->height;
13
14      unsigned char *input;
15      double *output[2][2];
16      cudaError_t error;
17      const long unsigned int inputsize = width*height;
18
19      error = cudaMalloc((void**)&input, inputsize);
20
21      if(error!= cudaSuccess)
22      {
23          printf("memory␣alloc␣input␣failed␣\n");
24          return FATAL;
25      }
26
27      CUDPPHandle theCudpp;
28      CUDPPHandle scanPlan;
29
30      CUDPPHandle theCudpp2;
31      CUDPPHandle scanPlan2;
32
33      size_t d_satPitch = width*sizeof(double);
```

```
34      size_t d_satPitchInElements = 0;
35      error=cudaMalloc( (void**) &output[0][0],
36          width*height*sizeof(double));
37      if(error!= cudaSuccess)
38      {
39          printf("memory␣alloc␣output00␣failed␣\n");
40          return FATAL;
41      }
42
43      error=cudaMalloc( (void**) &output[1][0],
44          width*height*sizeof(double));
45      if(error!= cudaSuccess)
46      {
47          printf("memory␣alloc␣output10␣failed␣\n");
48          return FATAL;
49      }
50      error=cudaMalloc( (void**) &output[0][1],
51          width*height*sizeof(double));
52      if(error!= cudaSuccess)
53      {
54          printf("memory␣alloc␣output01␣failed␣\n");
55          return FATAL;
56      }
57      error=cudaMalloc( (void**) &output[1][1],
58          width*height*sizeof(double));
59      if(error!= cudaSuccess)
60      {
61          printf("memory␣alloc␣output11␣failed␣\n");
62          return FATAL;
63      }
64      d_satPitchInElements = d_satPitch / sizeof(double);
```

```
65
66      CUDPPConfiguration config = { CUDPP_SCAN,
67          CUDPP_ADD,
68          CUDPP_DOUBLE,
69          CUDPP_OPTION_FORWARD | CUDPP_OPTION_INCLUSIVE };
70      cudppCreate(&theCudpp);
71
72      if (CUDPP_SUCCESS != cudppPlan(theCudpp, &scanPlan, config,
73          width, height, d_satPitchInElements))
74      {
75          printf("Error creating CUDPPPlan.\n");
76          return FATAL;
77      }
78
79      error = cudaMemcpy( input, (unsigned char*)(source->data),
80          inputsize, cudaMemcpyHostToDevice );
81      if(error!= cudaSuccess)
82      {
83          printf("cuda memcpy failed \n");
84          return FATAL;
85      }
86      dim3 block(512,1);
87      dim3 grid((width+(block.x-1)) / block.x,
88          (height+(block.y-1)) / block.y);
89      convertuchartodoubleintegralimagebase<<<grid, block>>>(
90                  output[0][0],
91                   output[0][1],
92                   input,
93                   width, height);
94      cudppMultiScan(scanPlan, output[1][0], output[0][0],
95          width, height);
```

```
96      cudppMultiScan(scanPlan, output[1][1], output[0][1],
97          width, height);
98      if (CUDPP_SUCCESS != cudppDestroyPlan(scanPlan))
99      {
100         printf("Error␣destroying␣CUDPPPlan.\n");
101         return FATAL;
102     }
103
104     // shut down CUDPP
105     if (CUDPP_SUCCESS != cudppDestroy(theCudpp))
106     {
107         printf("Error␣destroying␣CUDPP.\n");
108         return FATAL;
109     }
110     dim3 block2(16, 16);
111     dim3 grid2((width+(block2.x-1)) / block2.x,
112         (height+(block2.y-1)) / block2.y);
113     transpose<<<grid2, block2>>>
114     (output[0][0],output[1][0], width, height);
115     transpose<<<grid2, block2>>>
116     (output[0][1],output[1][1], width, height);
117     memset(target->I_1.data,0,(width+1)*(height+1)*sizeof(double));
118     memset(target->I_2.data,0,(width+1)*(height+1)*sizeof(double));
119     CUDPPConfiguration config2 = { CUDPP_SCAN,
120         CUDPP_ADD,
121         CUDPP_DOUBLE,
122         CUDPP_OPTION_FORWARD | CUDPP_OPTION_INCLUSIVE };
123     cudppCreate(&theCudpp2);
124
125     if (CUDPP_SUCCESS != cudppPlan(theCudpp2, &scanPlan2,
126         config2, height, width, height))
```

```
127     {
128         printf("Error␣creating␣CUDPPPlan.\n");
129         return FATAL;
130     }
131     dim3 block3(16, 16);
132     dim3 grid3((height+(block3.x-1)) / block3.x,
133         (width+(block3.y-1)) / block3.y);
134
135     cudppMultiScan(scanPlan2, output[1][0], output[0][0],
136         height, width);
137     cudppMultiScan(scanPlan2, output[1][1], output[0][1],
138         height, width);
139
140     transpose<<<grid3, block3>>>
141     (output[0][0],output[1][0], height, width);
142
143     transpose<<<grid3, block3>>>
144     (output[0][1],output[1][1], height, width);
145
146     if (CUDPP_SUCCESS != cudppDestroyPlan(scanPlan2))
147     {
148         printf("Error␣destroying␣CUDPPPlan.\n");
149         return FATAL;
150     }
151
152     if (CUDPP_SUCCESS != cudppDestroy(theCudpp2))
153     {
154         printf("Error␣destroying␣CUDPP.\n");
155         return FATAL;
156     }
157
```

```
158     d_satPitch = width*sizeof(double);
159     d_satPitchInElements = width;
160
161     cudaMemcpy2D((double*)((char*)target->I_1.data+
162         ((1+width+1)*sizeof(double))), (width+1)*sizeof(double),
163         output[0][0], d_satPitch, (width)*sizeof(double),
164         height, cudaMemcpyDeviceToHost);
165
166     cudaMemcpy2D((double*)((char*)target->I_2.data+
167         ((1+width+1)*sizeof(double))), (width+1)*sizeof(double),
168         output[0][1], d_satPitch, (width)*sizeof(double),
169         height, cudaMemcpyDeviceToHost);
170
171     cudaFree(input);
172     cudaFree(output[0][0]);
173     cudaFree(output[1][0]);
174     cudaFree(output[0][1]);
175     cudaFree(output[1][1]);
176
177     return SUCCESS;
178
179 }
```

Kuvan muuntamiseen käytetty kerneli

```
1 __global__
2 void convertuchartodoubleintegralimagebase(double *out,
3                                              double *out2,
4                                              unsigned char *in,
5                                              size_t width,
6                                              size_t height)
7 {
```

```
8       unsigned int xIndex = __mul24(blockDim.x, blockIdx.x) +
9           threadIdx.x;
10      unsigned int yIndex = __mul24(blockDim.y, blockIdx.y) +
11          threadIdx.y;
12
13      unsigned int index  = __mul24(width,yIndex) + xIndex;
14      unsigned char pixel_char;
15
16      if (xIndex < width && yIndex < height)
17      {
18          pixel_char = in[index];
19
20      }
21      __syncthreads();
22      if (xIndex < width && yIndex < height)
23      {
24          out[index] = (double)(pixel_char);
25      }
26      __syncthreads();
27
28      if (xIndex < width && yIndex < height)
29      {
30          out2[index] = (double)(pixel_char)*(double)(pixel_char);
31      }
32  }
```

Transponointiin käytetty kerneli

```
1   __global__ void transpose(double *output, double *input,
2           size_t width, size_t height)
3   {
4
```

```
5       double temp;
6
7       int xIndex = blockIdx.x*blockDim.x + threadIdx.x;
8       int yIndex = blockIdx.y*blockDim.y + threadIdx.y;
9
10      if((xIndex < width) && (yIndex <height))
11      {
12          int id_in = yIndex * width +xIndex;
13          temp = input[id_in];
14      }
15
16      __syncthreads();
17      xIndex = blockIdx.y * blockDim.y + threadIdx.y;
18      yIndex = blockIdx.x * blockDim.x + threadIdx.x;
19
20      if((xIndex <height) && (yIndex <width))
21      {
22          int id_out = yIndex * height +xIndex;
23          output[id_out] = temp;
24      }
25  }
```